Alexandra Silva
K. Rustan M. Leino (Eds.)

LNCS 12760

Computer Aided Verification

33rd International Conference, CAV 2021
Virtual Event, July 20–23, 2021
Proceedings, Part II

2 Part II

Springer

OPEN ACCESS

Alexandra Silva · K. Rustan M. Leino (Eds.)

Computer Aided Verification

33rd International Conference, CAV 2021
Virtual Event, July 20–23, 2021
Proceedings, Part II

Springer

Editors
Alexandra Silva
University College London
London, UK

K. Rustan M. Leino
Automated Reasoning Group | AWS
Seattle, WA, USA

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-030-81687-2 ISBN 978-3-030-81688-9 (eBook)
https://doi.org/10.1007/978-3-030-81688-9

LNCS Sublibrary: SL1 – Theoretical Computer Science and General Issues

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

It was our privilege to serve as the program chairs for CAV 2021, the 33rd International Conference on Computer-Aided Verification. CAV 2021 was held as a virtual conference during July 20–23, 2021. The tutorial days were on July 19 and July 24, 2021, and the pre-conference workshops were held during July 18–19, 2021. Due to the COVID-19 outbreak, all events took place online.

CAV is an annual conference dedicated to the advancement of the theory and practice of computer-aided formal analysis methods for hardware and software systems. The primary focus of CAV is to extend the frontiers of verification techniques by expanding to new domains such as security, quantum computing, and machine learning. This puts CAV at the cutting edge of formal methods research, and this year's program is a reflection of this commitment.

CAV 2021 received a very high number of submissions (290). We accepted 16 tool papers, 3 case studies, and 60 regular papers, which amounts to an acceptance rate of roughly 27%. The accepted papers cover a wide spectrum of topics, from theoretical results to applications of formal methods. These papers apply or extend formal methods to a wide range of domains such as concurrency, machine learning, and industrially deployed systems. The program featured keynote talks by Loris D'Antoni (UW-Madison), Corina Pasareanu (NASA), and Anna Slobodova (Centaur Technology, Inc.) as well as invited tutorials by Nate Foster (Cornell University), Zak Kincaid (Princeton) together with Tom Reps (UW-Madison), and Nadia Polikarpova (UC San Diego). Furthermore, we continued the tradition of Logic Lounge, a series of discussions on computer science topics targeting a general audience.

In addition to the main conference, CAV 2021 hosted the following workshops: Formal Approaches to Certifying Compliance (FACC), Formal Methods for ML-Enabled Autonomous Systems (FoMLAS), Formal Methods for Blockchains (FMBC), Numerical Software Verification (NSV), Theory and Practice of String Solving (TPSS), Verifying Probabilistic Programs (VeriProP), Synthesis (SYNT), Satisfiability Modulo Theories (SMT), and Verification Mentoring Workshop (VMW).

Organizing a flagship conference like CAV requires a great deal of effort from the community. The Program Committee for CAV 2021 consisted of 79 members — a committee of this size ensures that each member has to review only a reasonable number of papers in the allotted time. In all, the committee members wrote over 900 reviews while investing significant effort to maintain and ensure the high quality of the conference program. We are grateful to the CAV 2021 Program Committee for their outstanding efforts in evaluating the submissions and making sure that each paper got a fair chance. Like last year's CAV, we made the artifact evaluation mandatory for tool paper submissions and optional, but encouraged, for the rest of the accepted papers. This year saw an unprecedented number of 66 artifact submissions. The Artifact Evaluation Committee consisted of 72 members who put in significant effort to evaluate each artifact. The goal of this process was to provide constructive feedback to tool developers and help make the research published in CAV more reproducible. We are also very grateful to the Artifact Evaluation Committee for their hard work and dedication in evaluating the submitted artifacts.

CAV 2021 would not have been possible without the tremendous help we received from several individuals, and we would like to thank everyone who helped make CAV 2021 a success. First, we would like to thank Clément Pit-Claudel and Maria Schett for chairing the Artifact Evaluation Committee and John Cyphert for putting together the proceedings. We also thank Arie Gurfinkel for chairing the workshop organization, Bor-Yuh Evan Chang for managing sponsorship, Thomas Wies for arranging student fellowships, Norine Coenen for handling publicity, Leopold Haller for organising the Logic Lounge, and Peter Müller for putting together the *Ask me Anything* program. We also thank Jean-Baptiste Jeannin and Arjun Radhakrishna for chairing the Mentoring Committee. Putting together an online conference is a complex task and we are grateful to the virtualization chair Tiago Ferreira, the student volunteer coordinators Tobias Kappé and Tao Gu, the local organizers for the Asia timezone, Ichiro Hasuo and Krishna S, and the team at Slides Live for all their efforts. Last but not least, we would like to thank the members of the CAV Steering Committee (Kenneth McMillan, Aarti Gupta, Orna Grumberg, and Daniel Kroening) for helping us with several important aspects of organizing CAV 2021.

We hope that you will find the proceedings of CAV 2021 scientifically interesting and thought-provoking!

June 2021

Alexandra Silva
Rustan Leino

Organization

Steering Committee

Ornal Grumberg	Technion, Israel
Aarti Gupta	Princeton University, USA
Daniel Kroening	Amazon, USA
Kenneth Mcmillan	University of Texas at Austin, USA

Conference Co-chairs

K. Rustan M. Leino	Amazon, USA
Alexandra Silva	University College London, UK

Artifact Co-chairs

Clément Pit-Claudel	Massachusetts Institute of Technology, USA
Maria Schett	University College London, UK

Workshop Chair

Arie Gurfinkel	University of Waterloo, Canada

Verification Mentoring Workshop Organizing Committee

Jean-Baptiste Jeannin (Co-chair)	University of Michigan, USA
Arjun Radhakrishna (Co-chair)	Microsoft Research, USA
Suguman Bansal	University of Pennsylvania, USA
Roopsha Samanta	Purdue University, USA
Caterina Urban	Inria and École Normale Supérieure, France

Logic Lounge Organizer

Leopold Haller	Google Inc., USA

Ask Me Anything Organizer

Peter Müller	ETH Zürich, Switzerland

Publicity Chair

Norine Coenen	CISPA Helmholtz Center for Information Security, Germany

Sponsorship Chair

Bor-Yuh Evan Chang	University of Colorado Boulder, USA

Fellowship Chair

Thomas Wies	New York University, USA

Student Volunteer Coordinators

Tao Gu	University College London, UK
Tobias Kappé	Cornell University, USA

Proceedings and Talks Chair

John Cyphert	University of Wisconsin–Madison, USA

Virtualization Chair

Tiago Ferreira	University College London, UK

Local Organization Chairs

Ichiro Hasuo	National Institute of Informatics, Japan
Krishna S.	IIT Bombay, India

Program Committee

Erika Abraham	RWTH Aachen University, Germany
Elvira Albert	Universidad Complutense de Madrid, Spain
Christel Baier	TU Dresden, Germany
Clark Barrett	Stanford University, USA
Ezio Bartocci	TU Wien, Austria
Josh Berdine	Facebook, UK
Armin Biere	Johannes Kepler University Linz, Austria
Sam Blackshear	Novi, USA
Jasmin Blanchette	Vrije Universiteit Amsterdam, Netherlands
Roderick Bloem	Graz University of Technology, Austria
Borzoo Bonakdarpour	Michigan State University, USA
Ahmed Bouajjani	Université de Paris, France
Tevfik Bultan	University of California, Santa Barbara, USA

Sagar Chaki	Mentor Graphics, USA
Bor-Yuh Evan Chang	University of Colorado Boulder and Amazon, USA
Hana Chockler	King's College London, UK
Cristina David	University of Bristol, UK
Jennifer Davis	Collins Aerospace, USA
Yuxin Deng	East China Normal University, China
Rayna Dimitrova	CISPA Helmholtz Center for Information Security, Germany
Alastair Donaldson	Imperial College London, UK
Constantin Enea	Université de Paris, France
Joao Fernandes	University of Porto, Portugal
Bernd Finkbeiner	CISPA Helmholtz Center for Information Security, Germany
Vijay Ganesh	University of Waterloo, Canada
Pierre Ganty	IMDEA Software Institute, Spain
Aarti Gupta	Princeton University, USA
Arie Gurfinkel	University of Waterloo, Canada
Ichiro Hasuo	National Institute of Informatics, Japan
Marieke Huisman	University of Twente, Netherlands
David N. Jansen	Institute of Software, Chinese Academy of Sciences, China
Jean-Baptiste Jeannin	University of Michigan, USA
Ranjit Jhala	University of California, San Diego, USA
Rajeev Joshi	Amazon, USA
Temesghen Kahsai	The University of Iowa, USA
Benjamin Lucien Kaminski	University College London, UK
Joost-Pieter Katoen	RWTH Aachen University, Germany
Guy Katz	The Hebrew University of Jerusalem, Israel
Laura Kovacs	Vienna University of Technology, Austria
Mitja Kulczynski	Kiel University, Germany
Mohit Kumar Tekriwal	University of Michigan, USA
Orna Kupferman	The Hebrew University of Jerusalem, Israel
Marta Kwiatkowska	University of Oxford, UK
Shuvendu Lahiri	Microsoft Research, USA
Akash Lal	Microsoft Research, India
Kim Larsen	Aalborg University, Denmark
Marijana Lazic	Technical University of Munich, Germany
Owolabi Legunsen	University of Illinois at Urbana-Champaign, USA
K. Rustan M. Leino (Co-chair)	Amazon, USA
Rupak Majumdar	Max Planck Institute for Software Systems, Germany
Ruben Martins	Carnegie Mellon University, USA
Ken McMillan	University of Texas at Austin, USA
Aina Niemetz	Stanford University, USA
Ruzica Piskac	Yale University, USA
Sylvie Putot	Ecole Polytechnique, France

Markus N. Rabe	Google, USA
Talia Ringer	University of Washington, USA
Kristin Yvonne Rozier	Iowa State University, USA
Philipp Ruemmer	Uppsala University, Sweden
Krishna S.	IIT Bombay, India
Roopsha Samanta	Purdue University, USA
Sanjit A. Seshia	University of California, Berkeley
Natarajan Shankar	SRI International, USA
Natasha Sharygina	Università della Svizzera italiana, Switzerland
Sharon Shoham	Tel Aviv University, Israel
Alexandra Silva (Co-chair)	University College London, UK
Tachio Terauchi	Waseda University, Japan
Cesare Tinelli	The University of Iowa, USA
Aaron Tomb	Galois, Inc., USA
Ashutosh Trivedi	University of Colorado Boulder, USA
Caterina Urban	Inria, France
Margus Veanes	Microsoft, USA
Jules Villard	Facebook, UK
Yakir Vizel	Technion, Israel
Chao Wang	University of Southern California, USA
Wang Yi	Uppsala University, Sweden
Mingsheng Ying	University of Technology Sydney, Australia
Nobuko Yoshida	Imperial College London, UK
Lijun Zhang	Institute of Software, Chinese Academy of Sciences, China

Artifact Evaluation Committee

Rosa Abbasi Boroujeni	Max Planck Institute for Software Systems, Germany
Guy Amir	The Hebrew University of Jerusalem, Israel
Vincent Archambault	University of Montreal, Canada
M. Fareed Arif	The Unviersity of Iowa, USA
Filipe Arruda	Universidade Federal de Pernambuco, Brazil
Kshitij Bansal	Facebook, USA
Suguman Bansal	Rice University, USA
Shraddha Barke	University of California, San Diego, USA
Kevin Batz	RWTH Aachen University, Germany
Heiko Becker	Max Planck Institute for Software Systems, Germany
Julia Belyakova	Southern Federal University, Russia
Murphy Berzish	University of Waterloo, Canada
Ranadeep Biswas	Université de Paris, France
Alexandra Bugariu	ETH Zurich, Switzerland
Katherine Cordwell	Carnegie Mellon University, USA
Martin Desharnais	Bundeswehr University Munich, Germany
Zafer Esen	Uppsala University, Sweden
Mathias Fleury	Johannes Kepler University Linz, Austria

Masaki Waga	Kyoto University, Japan
Peixin Wang	Shanghai Jiao Tong University, China
Sarah Winkler	Free University of Bozen-Bolzano, Italy
Tobias Winkler	RWTH Aachen University, Germany
Ali Younes	Bauman Moscow State University, Russia
Xiao-Yi Zhang	National Institute of Informatics, Japan
Yuhao Zhang	University of Wisconsin-Madison, USA

Additional Reviewers

Ahmad, Hammad
An, Jie
Armborst, Lukas
Almagor, Shaull
Arenas, Puri
Asadi, Sepideh
Amir, Guy
Arif, Fareed
Asarin, Eugene
Baanen, Anne
Batz, Kevin
Berzish, Murphy
Bacci, Giovanni
Baumeister, Jan
Blicha, Martin
Balasubramanian, A. R.
Belo Lourenço, Cláudio
Boker, Udi
Barbosa, Haniel
Bentkamp, Alexander
Bønneland, Frederik M.
Barwell, Adam
Berger, Jana
Brain, Martin
Castellano, Ezequiel
Chen, Mingshuai
Coenen, Norine
Castro-Pérez, David
Chida, Nariyoshi
Cogumbreiro, Tiago
Cetinkaya, Ahmet
Chipara, Octav
Correas Fernández, Jesús
Cheang, Kevin
Dai, Gaoyang
Defourné, Antoine
Downing, Mara
Darwin, Oscar
Dill, David
Dunn, Isaac
Dave, Vrunda
Dohmen, Taylor
Dureja, Rohit
De Masellis, Riccardo
Doveri, Kyveli
Eberhart, Clovis
Eiers, William
Esen, Zafer
Ebrahimi, Masoud
Farzan, Azadeh
Feng, Yuan
Fleury, Mathias
Fedyukovich, Grigory
Ferraiuolo, Andrew
Gardy, Patrick
Godefroid, Patrice
Graham-Lengrand, Stéphane
Gehani, Ashish
Gomez-Zamalloa, Miguel
Grumberg, Orna
Genaim, Samir
Goorden, Martijn
Guan, Ji
Georgiou, Pamina
Gordillo, Pablo
Guha, Shibashis
Giacobbe, Mirco
Graf, Susanne
Gupta, Ashutosh
Giesl, Jürgen

Habermehl, Peter
Helfrich, Martin
Huang, Chengchao
Hadzic, Vedad
Hofmann, Jana
Huber, Nikolaus
Hark, Marcel
Holík, Lukáš
Hyvärinen, Antti
Hecking-Harbusch, Jesko
Hozzova, Petra
Irfan, Ahmed
Isabel, Miguel
Jaber, Nouraldin
Jha, Susmit
Jovanović, Dejan
Jensen, Mathias Claus
Jiang, Xu
Junges, Sebastian
Jensen, Peter Gjøl
Kadron, Burak
Klikovits, Stefan
Koenighofer, Bettina
Kempa, Brian
Klinkenberg, Lutz
Kremer, Gereon
Kheterpal, Nishant
Klüppelholz, Sascha
Kura, Satoshi
Kim, Edward
La Malfa, Emanuele
Li, Jianlin
Lin, Shaokai
Lachnitt, Hanna
Li, Yangjia
Lorber, Florian
Larraz, Daniel
Li, Yong
Lukina, Anna
Lathouwers, Sophie
Limperg, Jannis
Luppen, Zachary
Lee, Sang-Hwa
Maderbacher, Benedikt
Merayo, Alicia
Mora, Federico
Madnani, Khushraj
Metzger, Niklas
Mueller, Peter
Mallik, Kaushik
Michelmore, Rhiannon
Mundkur, Prashanth
Mann, Makai
Mohaqeqi, Morteza
Murali, Vishnu
Martin-Martin, Enrique
Monti, Raul
Möhle, Sibylle
Mazzucato, Denis
Moosbrugger, Marcel
Nagisetty, Vineel
Nenzi, Laura
Noll, Thomas
Narodytska, Nina
Nikšić, Filip
Nummelin, Visa
Nejati, Saeed
Otoni, Rodrigo
Ozdemir, Alex
Özkan, Burcu
Overbeek, Roy
Pant, Yash Vardhan
Perez, Mateo
Polgreen, Elizabeth
Passing, Noemi
Philipoom, Jade
Poulsen, Danny Bøgsted
Patane, Andrea
Pick, Lauren
Preiner, Mathias
Pereira, Mário
Piribauer, Jakob
Purser, David
Quatmann, Tim
Reynolds, Andrew
Rubbens, Bob
Ryan, Megan
Rowe, Reuben
Sato, Sota
Sebastiani, Roberto
Stanford, Caleb
Schupp, Stefan

Shah, Ameesh
Stankovic, Miroslav
Schurr, Hans-Jörg
Solovyev, Alexey
Stein, Benno
Schwenger, Maximilian
Spel, Jip
Tabar, Asmae
Torfah, Hazem
Tsiskaridze, Nestan
Tekriwal, Mohit
Tschaikowski, Max
Turrini, Andrea
Tibo, Alessandro
Unno, Hiroshi
Vasconcelos, Vasco
Vediramana Krishnan, Hari Govind
Vukmirović, Petar
Vazquez-Chanlatte, Marcell
Venkatesan, Abinaya
Waga, Masaki
Wang, Qisheng
Wilson, Amalee
Wagner, Christopher
Weil-Kennedy, Chana
Winkler, Tobias
Wang, Benjie
Welzel, Christoph
Wu, Haoze
Wang, Fang
Wicker, Matthew
Wu, Min
Wang, Peixin
Xue, Bai
Yu, Emily
Zeljić, Aleksandar
Zhang, Linpeng
Zhou, Mengchu
Zhang, Hanwei
Zhao, Hengjun
Zuleger, Florian
Zhang, Hengjun
Zhou, Li

Contents – Part II

Contents – Part I

Complexity and Termination

Learning Probabilistic Termination Proofs

Alessandro Abate(✉), Mirco Giacobbe(✉), and Diptarko Roy(✉)

University of Oxford, Oxford, UK
{alessandro.abate,mirco.giacobbe, diptarko.roy}@cs.ox.ac.uk

Abstract. We present the first machine learning approach to the termination analysis of probabilistic programs. Ranking supermartingales (RSMs) prove that probabilistic programs halt, in expectation, within a finite number of steps. While previously RSMs were directly synthesised from source code, our method learns them from sampled execution traces. We introduce the *neural ranking supermartingale*: we let a neural network fit an RSM over execution traces and then we verify it over the source code using satisfiability modulo theories (SMT); if the latter step produces a counterexample, we generate from it new sample traces and repeat learning in a counterexample-guided inductive synthesis loop, until the SMT solver confirms the validity of the RSM. The result is thus a sound witness of probabilistic termination. Our learning strategy is agnostic to the source code and its verification counterpart supports the widest range of probabilistic single-loop programs that any existing tool can handle to date. We demonstrate the efficacy of our method over a range of benchmarks that include linear and polynomial programs with discrete, continuous, state-dependent, multi-variate, hierarchical distributions, and distributions with undefined moments.

1 Introduction

Probabilistic programs are programs whose execution is affected by random variables [17,19,23,29,36]. Randomness in programs may emerge from numerous sources, such as uncertain external inputs, hardware random number generators, or the (probabilistic) abstraction of pseudo-random generators, and is intrinsic in quantum programs [34]. Notable exemplars are randomised algorithms, cryptographic protocols, simulations of stochastic processes, and Bayesian inference [7,33]. Verification questions for probabilistic programs require reasoning about the probabilistic nature of their executions in order to appropriately characterise properties of interest. For instance, consider the following question, corresponding to the program in Fig. 1: will an ambitious marble collector eventually gather any arbitrarily large amounts of red and blue marbles? Intuitively, the question has an affirmative answer regardless of the initially established target amounts, since there is always a chance of collecting a marble of either color. Notice that, if the probabilistic choice is replaced with non-determinism, as often happens in software verification, an adversary may exclusively draw one color of marble

A. Silva and K. R. M. Leino (Eds.): CAV 2021, LNCS 12760, pp. 3–26, 2021.
https://doi.org/10.1007/978-3-030-81688-9_1

and make the program run forever. The question that matches the original intuition is whether the expected number of steps to termination is finite; this is the *positive almost-sure termination* (PAST) question [8,10,13,19,27].

```
while (red > 0 || blue > 0) do
  p ∼ Bernoulli(.01);
  if p == 1 then
    red = red - 1
  else
    blue = blue - 1
  fi
od
```

Fig. 1. The ambitious marble collector (the variables `red` and `blue` are initialised non-deterministically).

Probabilistic termination analysis is typically mechanised through the automated synthesis of *ranking supermartingales* (RSMs), which are functions of the program variables whose value (i) decreases in expectation by a discrete amount across every loop iteration and (ii) is always bounded from below; an RSM formally witnesses that a program is PAST [10,13]. Early techniques for discovering RSMs reduced the synthesis problem from the source code of the program into constraint solving [10]. These methods have lent themselves to various generalisations, including polynomial programs, programs with non-determinism, lexicographic and modular termination arguments, and persistence properties [2,14–16,20,25]. Recently, for special classes of probabilistic programs or term rewriting systems, novel automated proof techniques that leverage computer algebra systems and satisfiability modulo theories (SMT) have been introduced [5,6,38,39,41]. All the above methods are sound and, under specific assumptions, complete; they represent the state of the art for the class of programs they have been designed for. However, their assumptions are often too restrictive for the analysis of many simple programs. In particular, to the best of our knowledge, none can identify an RSM for the program in Fig. 1. For this simple program, it is easy to argue that the expected output of the *neural network* depicted in Fig. 2 decreases after every iteration of the loop and that it is always non-negative (see Ex. 1). As such, this neural network is an appropriate RSM for the program.

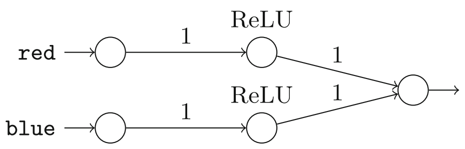


Fig. 2. A neural ranking supermartingale for the program in Fig. 1.

We present a novel method for discovering RSMs using machine learning together with SMT solving. We introduce the *neural ranking supermartingale* (NRSM) model, which lets a neural network mimic a supermartingale over sampled execution traces from a program. We train an NRSM using standard optimisation algorithms over a loss function that makes the neural network decrease—in average—across sampled iterations. We phrase the certification problem into that of computing a counterexample for the NRSM. To do so, we encode the neural network together with the expected value of the program variables; then, we use an SMT solver for verifying that the expected output of the network decreases along every execution. If the solver falsifies the NRSM, then it provides a counterexample that we use to guide a resampling of the execution traces; with this new data we retrain the neural network and repeat verification in a *counterexample-guided inductive synthesis* (CEGIS) fashion, until the SMT solver determines that no counterexample exists [4,44]. In the latter case, the solver has certified the generated NRSM; our method thus produces a *sound* PAST proof or runs indefinitely. Our procedure does not return for programs that are not PAST and may, in general, not return for some PAST instances. However, we experimentally demonstrate that, in practice, our method succeeds over a broad range of PAST benchmarks within a few CEGIS iterations. Previously, machine learning has been applied to the termination analysis of deterministic programs and to the stability analysis of dynamical systems [1,12,21,24,28,30–32,42,43,45]; our method is the first machine learning approach for probabilistic termination analysis.

Our approach builds upon two key observations. First, the average of expressions along execution traces statistically approximates their true expected value. Thanks to this, we obtain a machine learning model for *guessing* RSM candidates that only requires execution traces and is thus agnostic to the source code. Second, solving the problem of *checking* an RSM is simpler than solving the entire termination analysis problem. Reasoning about source code is entirely delegated to the checking phase which, as such, supports programs that are out of reach to the available probabilistic termination analysers.

We experimentally demonstrate that our method is effective over many programs with linear and polynomial expressions, with both discrete and continuous distributions. This includes joint distributions, state-dependent distributions, distributions whose parameters are in turn random (hierarchical models), and distributions with undefined moments (e.g., the Cauchy distribution). We compare our method with a tool based on Farkas' lemma and with the tools Amber and Absynth [2,39,41]; whilst our software prototype is slower than these alternatives, it covers the widest range of benchmark single-loop programs.

Summarising, our contribution is fivefold. First, we present the first machine learning method for the termination analysis of probabilistic programs. Second, we introduce a loss function for training neural networks to behave as ranking supermartingales over execution traces. Third, we show an approach to verify the validity of ranking supermartingales using SMT solving, which applies to a wide variety of single-loop probabilistic programs. Fourth, we experimentally demonstrate over multiple baselines and newly-defined benchmarks the practical efficacy of our method. Fifth, we built a software prototype for evaluating our method.

$$
\begin{array}{rll}
x \in \mathrm{Vars} & & \text{(variables)} \\
N \in \mathbb{R} & & \text{(numerals)} \\
\mathrm{op}_2 ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{\&\&} \mid \texttt{||} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{==} \mid \ldots & \text{(binary operators)} \\
E ::= & x \mid N \mid E \ \mathrm{op}_2 \ E \mid \texttt{-}E & \text{(arithmetic expressions)} \\
D ::= & \texttt{Bernoulli(} \ E \ \texttt{)} \mid \texttt{Gaussian(} \ E\texttt{,} \ E \ \texttt{)} \mid \ldots & \text{(probability distributions)} \\
B ::= & B \ \mathrm{op}_2 \ B \mid \texttt{!} \ B \mid E \ \mathrm{op}_2 \ E \mid \texttt{true} \mid \texttt{false} & \text{(Boolean expressions)} \\
C ::= & \texttt{skip} & \text{(commands)} \\
 & \mid x \ \texttt{=} \ E & \text{(deterministic assignment)} \\
 & \mid x \sim D & \text{(probabilistic assignment)} \\
 & \mid C \ ; \ C & \text{(sequential composition)} \\
 & \mid \texttt{if} \ B \ \texttt{then} \ C \ \texttt{else} \ C \ \texttt{fi} & \text{(conditional composition)}
\end{array}
$$

Fig. 3. Syntax of loop-free probabilistic programs.

2 Termination Analysis of Probabilistic Programs

We treat the termination analysis of single-loop probabilistic programs. We consider an imperative language that includes C-like arithmetic and Boolean expressions, and sequential and conditional composition of commands [13,17,19,23].

Syntax. A grammar for this language is shown in Fig. 3. We analyse single-loop programs of the form

```
while G do
  U
od
```

where the loop guard G is a Boolean expression and the update statement U is a command. Variables are real-valued and can be either assigned to arithmetic expressions using the usual = operator, or sampled from probability distributions using the $\sim$ operator. Probability distributions, which can be either discrete or continuous, take not only parameters that are constant, and thus known at compile time, but also parameters that depend on other variables, and thus determined only at run time. In other words, distributions may depend on the current state of the program, which is a random variable. Also, they may depend on other random variables; as such, distributions may be multi-variate, resulting from models with coupled and hierarchically-structured variables.

Semantics. The operational semantics of a probabilistic program induces a probability space over runs, together with a stochastic process [13]. A state of the process is an element of $\mathbb{R}^n$ with $n = |\text{Vars}|$, that is, a valuation of the variables in the program. The space of outcomes Ω_{run} of a program is the set of runs. A run is a possibly infinite sequence of variable valuations (taken at the beginning of every loop iteration). This comes with a σ-algebra $\mathcal{F}$ of measurable subsets of Ω_{run}. Initial states are chosen non-deterministically and, thereafter, the process is purely probabilistic. Every initial state $x_0 \in \mathbb{R}^n$ determines a unique probability measure $\mathbb{P}^{(x_0)}\colon \mathcal{F} \to [0,1]$, namely a probability measure conditional on the state x_0. The associated stochastic process is $X^{(x_0)} = \{X_t^{(x_0)}\}_{t\in\mathbb{N}}$, where $X_t^{(x_0)}$ is a random vector representing the state at the t-th step, initialised as $X_0^{(x_0)} = x_0$. Given an initial condition x_0 and a solution process $X^{(x_0)}$, the associated termination time is a random variable $T^{(x_0)}$ denoting the length of an execution, which takes values in $\mathbb{N} \cup \{\infty\}$.

Positive Almost-Sure Termination. Runs are probabilistic and thus also the notion of termination requires a quantitative semantics. The termination question is generalised to the notions of *almost-sure* and *positive almost-sure* termination. Almost-sure termination (AST) indicates whether the joint probability of all runs that do not terminate is zero; positive almost-sure termination (PAST), which is stronger, indicates whether the expected number of steps to termination is finite. Formally, a probabilistic program terminates positively almost-surely if $\mathbb{E}[T^{(x_0)}] < \infty$ for all $x_0 \in \mathbb{R}^n$. Notably, this implies that the program also terminates almost-surely, that is, $\mathbb{P}[T^{(x_0)} < \infty] = 1$ for all $x_0 \in \mathbb{R}^n$. We provide conditions ensuring that probabilistic programs are PAST and, consequently, that they are AST. Notice that the converse may not be true, that is, there exist programs that are AST but not PAST. Our method addresses the PAST question only, by building upon the theory of ranking supermartingales [10].

Ranking Supermartingales. A scalar stochastic process $\{M_t\}$ is an RSM if, for some $\epsilon > 0$ and lower bound $K \in \mathbb{R}$,

$$\mathbb{E}\left[M_{t+1} \mid M_t = m_t, \ldots, M_0 = m_0\right] \le m_t - \epsilon \tag{1}$$

and $M_t \ge K$ for all $t \ge 0$. In other words, this a process whose values are bounded from below and whose expected value decreases by a discrete amount at each step of the program. We prove that a program is PAST by mapping $X^{(x_0)}$ into an RSM. Our goal is finding a function $\eta\colon \mathbb{R}^n \to \mathbb{R}$ such that, for every initial condition x_0, it satisfies the following two properties:

(i) $\mathbb{E}[\eta(X_{t+1}^{(x_0)}) \mid X_t^{(x_0)} = x] \le \eta(x) - \epsilon$ for all $x \in I$ and
(ii) $\eta(x) \ge K$ for all $x \in I$,

where $I \subseteq \mathbb{R}^n$ is some sufficiently strong loop invariant that can be the loop guard or, possibly, a stronger condition. Function η maps the entire stochastic process into an RSM. For this reason, we call η an RSM for the program.

```
**Input**: Single-loop probabilistic program $(G, U)$,
           Initial state $x_0 \in \mathbb{R}^n$
**Output**: Transition samples $S \subset \mathbb{R}^n \times \mathcal{P}(\mathbb{R}^n)$
1  $S \leftarrow \emptyset$;
2  $P' \leftarrow \{x_0\}$;
3  for $i \leftarrow 1$ to $k$ do                          // $k$ = path length
4      $P \leftarrow P'$;
5      $P' \leftarrow \emptyset$;
6      $p \leftarrow$ pick arbitrary element from $P$;
7      if eval($G$,$p$) $=$ True then
8          for $j \leftarrow 1$ to $m$ do                  // $m$ = branching factor
9              $P' \leftarrow P' \cup \{$exec($U$,$p$)$\}$
10         $S \leftarrow S \cup \{(p, P')\}$;
11 return $S$
```

Algorithm 1: Interpreter

Example 1. Consider the ambitious marble collector problem from Fig. 1. An RSM for this program is a function η mapping variables `red` and `blue` to $\mathbb{R}$. Rephrasing condition (i) over this program, η is required to satisfy

$$0.01 \cdot \eta(\texttt{red} - 1, \texttt{blue}) + 0.99 \cdot \eta(\texttt{red}, \texttt{blue} - 1) \leq \eta(\texttt{red}, \texttt{blue}) - \epsilon, \qquad (2)$$

for all $\texttt{red}, \texttt{blue} \in \mathbb{Z}$ that satisfy $\texttt{red} > 0 \vee \texttt{blue} > 0$, that is, the loop guard. So, for example, function $\eta(\texttt{red}, \texttt{blue}) = \texttt{red} + \texttt{blue}$ satisfies this condition; however, it may take any negative value over the arguments `red` and `blue` such that $\texttt{red} > 0 \vee \texttt{blue} > 0$, thus violating condition (ii). By contrast, the neural network in Fig. 2 succeeds at satisfying both conditions. In fact, the network realises function $\eta(\texttt{red}, \texttt{blue}) = \max\{\texttt{red}, 0\} + \max\{\texttt{blue}, 0\}$, which satisfies Eq. (2) and is bounded from below by zero. □

3 Training Neural Ranking Supermartingales

Our framework synthesises RSMs by learning from program execution traces. We define a loss function, that measures the number of sampled program transitions that do not satisfy the RSM conditions. Applying gradient-descent optimisation to the loss function guides the parameters to values at which the candidate's value decreases, on average, across sampled program transitions. Since the learner does not require the underlying program (only execution traces), the learner is agnostic to the structure of program expressions, and the cost of evaluating the loss function does not scale with the size of the program.

A dataset of sampled transitions is produced using an instrumented program interpreter (Algorithm 1). At a program state p, the interpreter runs the loop body m times to sample successor states P', where m is a branching factor hyperparameter, before resuming execution from an arbitrarily chosen successor. The dataset S consists of the union of pairs (p, P') generated by the interpreter.

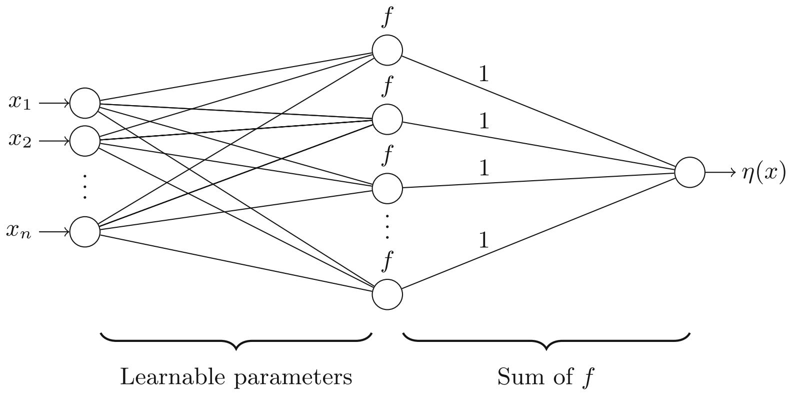


Fig. 4. Neural ranking supermartingale architecture.

The loss function is used to optimise the parameters of an NRSM, whose architecture is shown in Fig. 4. This is a neural network with n inputs, one output neuron, and one hidden layer. The hidden layer has h neurons, each of which applies an activation function f to a weighted sum of its inputs. In our experiments, the activation function f is either $f(x) = x^2$ or $f(x) = \text{ReLU}(x)$, where $\text{ReLU}(x) = \max\{x, 0\}$.

Therefore, we employ either of the two following functional templates, defined over the learnable parameters $w_{i,j}$ and b_i:

– Sum of ReLU (SOR):

$$\eta(x_1, \ldots, x_n) = \sum_{i=1}^{h} \text{ReLU}\left(\sum_{j=1}^{n} w_{i,j}x_j + b_i\right); \tag{3}$$

– Sum of Squares (SOS):

$$\eta(x_1, \ldots, x_n) = \sum_{i=1}^{h} \left(\sum_{j=1}^{n} w_{i,j}x_j + b_i\right)^2. \tag{4}$$

These choices of activation mean that our NRSMs are restricted to non-negative outputs, and therefore satisfy condition (ii) by construction. The learner therefore needs to find parameters that satisfy condition (i), which requires η to decrease in expectation by at least some positive constant $\epsilon > 0$.

The role of the loss function is to allow the learner parameters to be optimised such that the NRSM decreases, on average, across sampled transitions. That is, the loss function evaluates the number of sampled transitions for which the NRSM does not satisfy the RSM condition (i), and the lower its value, the more the neural network behaves like an RSM.

Concretely, the loss associated with a state p and its successors P' is:

$$L(p, P') = \text{softplus}\left(\mathbb{E}_{p' \sim P'}[\eta(p')] - \eta(p) + \epsilon\right), \tag{5}$$

where $\text{softplus}(x) = \ln(1 + e^x)$, and $\mathbb{E}_{p' \sim P'}[\eta(p')]$ is the average of η over the sampled successor states p' from P'.

We then train an NRSM by solving the following optimisation problem:

$$\min \frac{1}{|S|} \sum_{(p,P') \in S} L(p, P'), \tag{6}$$

which aims to minimise the average loss over all sampled transitions in the dataset S, over the trainable weights $w_{1,1}, \ldots, w_{h,n} \in \mathbb{R}$ and biases $b_1, \ldots, b_h \in \mathbb{R}$. This objective is non-convex and non-linear, and we resort to gradient-based optimisation (see Sect. 6).

The softplus in Eq. (5) forces the parameters to satisfy condition (i) uniformly across all sampled transitions in the dataset, rather than decreasing by a large amount in expectation over some transitions at the expense of failing to decrease sufficiently quickly for others. Furthermore, for NRSMs of SOR form we replace the ReLU activation function by softplus, to help gradient descent converge faster. Softplus approximates the ReLU function, and has the same asymptotic behaviour, but results in an NRSM that is differentiable w.r.t. the network parameters at all inputs, unlike ReLU [22, p.193]. However, since softplus is a transcendental function, we revert back to using a simpler ReLU activation when verifying an SOR candidate.

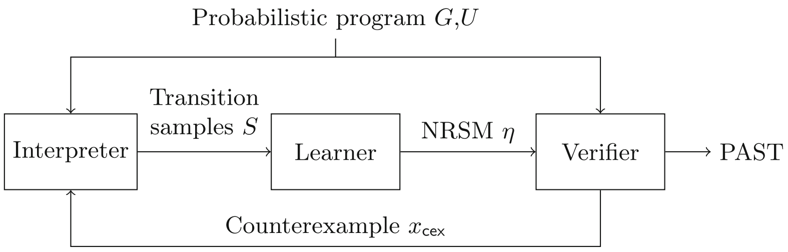


Fig. 5. CEGIS architecture for the adversarial training of NRSM.

A CEGIS loop integrates the learner and verifier (Fig. 5). The dataset S sampled by the interpreter is used to train an NRSM candidate η according to Eq. (6). The verifier checks whether η satisfies condition (i), concluding either that the program is PAST, or producing a counterexample program state x_{cex} for which η does not satisfy (i). The interpreter generates new traces, starting at x_{cex}, forcing it to explore parts of the state space over which the NRSM fails to decrease sufficiently in expectation.

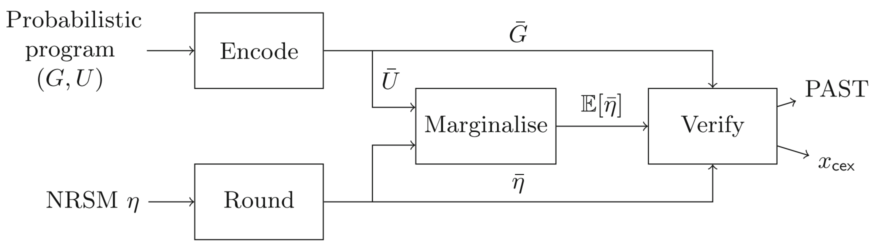


Fig. 6. Verifier architecture.

4 Verifying Ranking Supermartingales by SMT Solving

To verify an NRSM we must check that it decreases in expectation by at least some constant (condition (i)). Condition (ii) is satisfied by construction because the network's output is non-negative for every input, leaving only condition (i) to verify. The architecture of the verifier is depicted in Fig. 6. First, a program (G, U) is translated into an equivalent logical formulation denoted by $\bar{G}$ and $\bar{U}$ ('Encode' block), which are used to construct a closed-form term $\mathbb{E}[\bar{\eta}]$ for the NRSM's expected value at the end of the loop body ('Marginalise' block). Secondly, given an NRSM η, its parameters are rounded and encoded as a logical term $\bar{\eta}$ ('Round' block). Then, the satisfiability of the following formula is decided using SMT solving:

$$\bar{G}(x_1 \dots x_n) \wedge \mathbb{E}[\bar{\eta}](x_1 \dots x_n) > \bar{\eta}(x_1 \dots x_n) - \epsilon. \tag{7}$$

This is the dual satisfiability problem for the validity problem associated with condition (i) on page 5. If Eq. (7) is unsatisfiable, then $\bar{\eta}$ is a valid RSM and we conclude the program is PAST. Otherwise, the solver yields a counterexample state $x_{\mathsf{cex}} \in \mathbb{R}^n$.

The rounding strategy ('Round' block) provides multiple candidates to the verifier by adding i.i.d. noise to parameters and rounding them to various precisions. Setting parameters that are numerically very small to zero is useful since learning that a parameter should be exactly zero could require an unbounded number of samples; rounding provides a pragmatic way of making this work in practice. If none of the generated candidates are valid NRSMs, all counterexamples are passed back to the interpreter which generates more transition samples for the learner (Fig. 5).

$$\begin{aligned}
&x \in \text{Vars} && \text{(variables)}\\
&N \in \mathbb{R} && \text{(numerals)}\\
&\tau ::= x \mid N \mid \tau + \tau \mid \tau - \tau \mid \dots && \text{(terms)}\\
&\phi ::= \top \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \tau \leq \tau \mid \tau = \tau \mid \dots && \text{(formulae)}
\end{aligned}$$

Fig. 7. Quantifier-free first-order logic formulae.

Notice that, if a program's guard predicate is not strong enough to allow a valid RSM to be verified as such, the CEGIS loop will run indefinitely. In general, stronger supporting loop invariants may need to be provided.

4.1 From Programs to Symbolic Store Trees

We now introduce a translation from a loop-free probabilistic program to a *symbolic store tree* (Fig. 8), a datastructure representing the distribution over program states at the end of a loop iteration as a function of the variable valuation at its start. Marginalising out the probabilistic choices made in the loop yields the NRSM expectation $\mathbb{E}[\bar{\eta}]$.

$$\begin{aligned}
&\pi ::= \tau \mid \texttt{Bernoulli}(\tau) \mid \texttt{Gaussian}(\tau, \tau) \mid \ldots && \text{(probabilistic terms)}\\
&\Sigma = \{x_1 \mapsto \pi_1, \ldots, x_n \mapsto \pi_n\} && \text{(symbolic store)}\\
&\sigma ::= \texttt{node}(\phi, \sigma, \sigma) \mid \Sigma && \text{(symbolic store tree)}
\end{aligned}$$

Fig. 8. Symbolic store tree.

This requires a form of symbolic execution. We represent program states symbolically using *symbolic stores*, denoted Σ (Fig. 8), which map program variables to *probabilistic terms*. A probabilistic term π can be either a first-order logic term (Fig. 7) representing an arithmetic expression, or a placeholder for a probability distribution whose parameters are terms (allowing them to be functions of the program state). Finally, *symbolic store trees* σ (Fig. 8) represent the set of control-flow paths through the loop body, arising from if-statements; it is a binary tree with symbolic stores at the leaves, and internal nodes labelled by logical formulae over program variables.

$$\begin{aligned}
&\text{enc}(\Sigma, x) = \Sigma(x)\\
&\text{enc}(\Sigma, -O) = -\text{enc}(\Sigma, O) \qquad \text{enc}(\Sigma, \texttt{!}\ O) = \neg\text{enc}(\Sigma, O)\\
&\text{enc}(\Sigma, O_1 \text{ op}_2\ O_2) = \text{enc}(\Sigma, O_1)\ \boxed{\text{op}_2}\ \text{enc}(\Sigma, O_1)
\end{aligned}$$

$$\begin{aligned}
&\text{enc}(\Sigma, \texttt{skip}) = \Sigma\\
&\text{enc}(\Sigma, x \texttt{ = } E) = \Sigma[x' \mapsto \text{enc}(\Sigma, E)]\\
&\text{enc}(\Sigma, C_1\ ;\ C_2) = \text{enc}(\text{enc}(\Sigma, C_1), C_2)\\
&\text{enc}(\Sigma, \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2\ \texttt{fi}) = \texttt{node}(\text{enc}(\Sigma, B), \text{enc}(\Sigma, C_1), \text{enc}(\Sigma, C_2))
\end{aligned}$$

$$\text{enc}(\texttt{node}(\phi, \sigma_1, \sigma_2), C) = \texttt{node}\,(\phi, \text{enc}(\sigma_1, C), \text{enc}(\sigma_2, C))$$

$$\begin{aligned}
&\text{enc}(\Sigma, x \sim \texttt{Bernoulli}(E)) = \Sigma[x' \mapsto \nu, \nu \mapsto \texttt{Bernoulli}(\text{enc}(\Sigma, E))]\\
&\text{enc}(\Sigma, x \sim \texttt{Gaussian}(E_1, E_2)) = \Sigma[x' \mapsto \nu, \nu \mapsto \texttt{Gaussian}(\text{enc}(\Sigma, E_1), \text{enc}(\Sigma, E_2))]\\
&\quad\vdots \qquad\qquad\qquad \text{where every} \sim \text{command creates a fresh } \nu \text{ variable.}
\end{aligned}$$

Fig. 9. Translation from a loop-free command to a symbolic store tree.

Figure 9 defines a translation from an initial symbolic store tree and command to a new symbolic store tree characterising the distribution over states after executing the command. At the top level, we provide the command G (the loop body) and the initial symbolic store $\{x'_1 \mapsto x_1, \ldots, x'_n \mapsto x_n\}$, where primed variables represent the variable valuation at the end of the iteration, whereas unprimed variables represent the variable valuation at the beginning of the loop.

The first four cases of Fig. 9 define the translation of arithmetic expressions (to terms) and Boolean expressions (to formulae), by replacing program syntax with the corresponding logical operators.

The next four cases define the translation of commands. `skip` leaves the symbolic store unchanged. For deterministic assignments, the right hand side of the assignment is translated in the current symbolic store and bound to the variable. Sequential composition involves translating the first command, and translating the second command in the resulting store tree. A conditional statement creates a new `node` in the symbolic store tree that selects between the two recursively-translated branches, based on the formula derived from the guard predicate. These rules assume the store tree to be a leaf-level symbolic store, because the next rule handles the case where the initial symbolic store tree is a `node`. Finally, if the command is a probabilistic assignment, we translate the parameters to terms, and bind the resulting probabilistic term to a freshly generated symbol. This allows variables to be overwritten by multiple probabilistic sampling operations in the body of the loop. The mapping of variables to distributions in leaf-level stores defines the probability density over particular probabilistic choices.

Example 2. Figure 10 is the store tree produced for the ambitious marble collector program (Fig. 1). Each leaf-level store in the program's store tree corresponds to a particular control-flow path through the loop body. The interpretation of a symbolic store tree is that if we fix the outcomes of the probabilistic sampling operations performed by the loop body, then the state of the variables at the end of the iteration is determined by the predicates labelling the internal nodes.

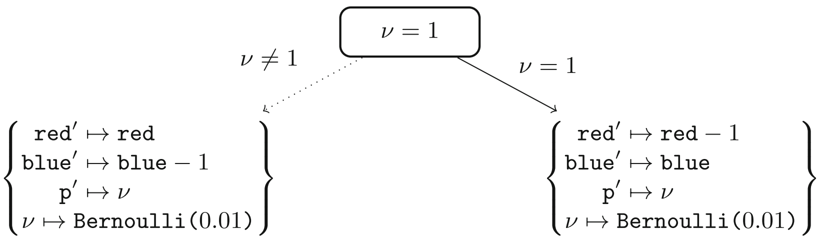

$\nu = 1$
$\nu \neq 1$
$\nu = 1$
$\mathtt{red}' \mapsto \mathtt{red}$
$\mathtt{blue}' \mapsto \mathtt{blue} - 1$
$\mathtt{p}' \mapsto \nu$
$\nu \mapsto \mathtt{Bernoulli}(0.01)$
$\mathtt{red}' \mapsto \mathtt{red} - 1$
$\mathtt{blue}' \mapsto \mathtt{blue}$
$\mathtt{p}' \mapsto \nu$
$\nu \mapsto \mathtt{Bernoulli}(0.01)$

Fig. 10. A store tree for the program in Fig. 1.

4.2 Marginalisation

To construct the closed-form logical term representing the NRSM's expected value at the end of an iteration, the probabilistic choices in the symbolic store tree must be marginalised out. If the program is limited to discrete random variables with finite support, we automatically marginalise the random choices by enumeration (for both SOR- and SOS-form NRSMs), as illustrated by Ex. 3.

Example 3. The ambitious marble collector program of Fig. 1, yields the symbolic store tree of Fig. 10. Suppose we want to marginalise the NRSM:

$$\begin{aligned}\eta(\mathtt{red}, \mathtt{blue}) = \mathrm{ReLU}(w_{1,1} \cdot \mathtt{red} + w_{1,2} \cdot \mathtt{blue} + b_1) \\ + \mathrm{ReLU}(w_{2,1} \cdot \mathtt{red} + w_{2,2} \cdot \mathtt{blue} + b_2)\end{aligned} \tag{8}$$

with respect to this symbolic store tree. We first apply the encoding of the NRSM to each leaf-level symbolic store of Fig. 10, and enumerate the possible choices for the probabilistic choices (which in this example is limited to $\nu \in \{0, 1\}$), using the bindings of ν to distributions in leaf-level stores to compute the probability mass of each choice. After resolving the predicates for each choice of ν, this yields:

$$0.01 \cdot \eta(\mathtt{red} - 1, \mathtt{blue}) + 0.99 \cdot \eta(\mathtt{red}, \mathtt{blue} - 1). \tag{9}$$

The term (9) is then provided as the value of the NRSM's expectation to the verifier. □

If the program samples from continuous distributions, we marginalise SOS-form NRSMs (but not SOR-form NRSMs) by substituting symbolic moments for a set of supported built-in distributions, including `Gaussian`, `Multivari-ateGaussian`, and `Exponential`, though could include any distribution whose closed-form symbolic moments are available. Example 4 provides an example. This strategy is general enough to support a wide variety of programs, including those of Sect. 5. If a sampling distribution lacks symbolic moments, the cumulative distribution function can also be utilised, which is illustrated in the `slicedcauchy` case study (Fig. 15).

Example 4. Consider an NRSM $\eta(\mathtt{x}) = (w\mathtt{x} + b)^2$ and a symbolic store tree $\mathtt{node}(p = 1, \sigma_1, \sigma_2)$ where $\sigma_1 = \{x \mapsto x + v, v \mapsto \mathtt{Exp}(\lambda), p \mapsto \mathtt{Bernoulli}(3/4)\}$ and $\sigma_2 = \{x \mapsto x - v, v \mapsto \mathtt{Exp}(\lambda), p \mapsto \mathtt{Bernoulli}(3/4)\}$. $\mathtt{Exp}(\lambda)$ denotes the exponential distribution with parameter λ, with pdf denoted $p_{\mathtt{Exp}(\lambda)}(v)$. We apply η to each leaf-level symbolic store, and marginalise the probabilistic choices. We marginalise p first by enumerating over its possible values, and then marginalise v. There are no dependencies between the distributions in this example, so the order in which they are marginalised does not matter.

$$\int_0^\infty \left(\frac{3}{4}\eta(x + v) + \frac{1}{4}\eta(x - v)\right) p_{\mathtt{Exp}(\lambda)}(v)\mathrm{d}v. \tag{10}$$

The result of marginalisation is a closed-form expression for Eq. (10). Note that since

$$\eta(x+v) = w^2v^2 + 2(wx+b)wv + (wx+b)^2 \tag{11}$$

and $\int_0^\infty v^n p_{\texttt{Exp}(\lambda)}(v)\mathrm{d}v = \frac{n!}{\lambda^n}$, we use linearity of integration to perform the following simplification, by substituting expressions for the moments of `v` in terms of the parameter λ:

$$\int_0^\infty \eta(x+v)p_{\texttt{Exp}(\lambda)}(v)\mathrm{d}v = \frac{2w^2}{\lambda^2} + \frac{2(wx+b)w}{\lambda} + (wx+b)^2, \tag{12}$$

which is used to reduce Eq. (10) to a closed form. This is the method used to perform marginalisation for several case studies, including `crwalk`, `gaussrw` and `expdistrw`. □

Notably, our verifier requires the expected value of the RSM to be computed (or soundly approximated) in closed form. We automate marginalisation for discrete distributions of finite support, but require manual intervention for continuous distributions. Nevertheless, our learning component is automated in both cases. Characterising the space of programs with continuous distributions that admit fully automated verification of an RSM is an open question.

5 Case Studies

Existing tools for synthesising RSMs reduce the problem to constraint-solving [2,10,11,14], which can limit the generality of the synthesis framework. For instance, methods that convert the RSM constraints into a linear program using Farkas' lemma can only handle programs with affine arithmetic, and can only synthesise linear/affine (lexicographic) RSMs [2,10]. A second restriction of existing approaches is that they typically require the moments of distributions to be compile-time constants. This rules out programs whose distributions are determined at runtime, such as hierarchical and state-dependent distributions. Since the loss function of Eq. (6) only requires execution traces, our learner is agnostic to the structure of program expressions, imposing minimal restrictions on the kinds of expressions that can occur, or the kinds of distributions that can be sampled from. This allows us to learn RSMs for a wider class of programs compared to existing tools, as we will illustrate in this section using a number of case studies.

5.1 Non-linear Program Expressions and NRSMs

Many simple programs do not admit linear or polynomial RSMs, such as Fig. 1. Since the program cannot be encoded as a prob-solvable loop (due to the disjunctive guard predicate which cannot be replaced by a polynomial inequality), it cannot be handled by another recent tool, Amber [39]. However, this program admits the following piecewise-linear NRSM:

$$\mathrm{ReLU}(0 \cdot \texttt{red} + 1 \cdot \texttt{blue} + 11) + \mathrm{ReLU}(1 \cdot \texttt{red} + 0 \cdot \texttt{blue} + 11), \tag{13}$$

whose parameters are learnt by our method, within the first CEGIS iteration.

```
 1  while (i <= 10 && s > 0) do
 2      r ~ DiscreteUniform({-2, 2});
 3      s = r + s * i;
 4      p ~ Bernoulli(3/4);
 5      if (p == 1) then
 6          i = i + 1
 7      else
 8          i = i - 1
 9      fi
10  od
```

Fig. 11. Probabilistic factorial (`probfact`).

Similarly, we learn the piecewise-linear NRSM:

$$\mathrm{ReLU}(-1 \cdot \texttt{i} + 0 \cdot \texttt{s} + 12) + \mathrm{ReLU}(0 \cdot \texttt{i} + 0 \cdot \texttt{s} + 9) \tag{14}$$

for the program in Fig. 11, which contains a bilinear assignment (cf. multiplication of `s` and `i` on line 3), so this program is not supported by [2]. The conjunction in the guard means it is not supported by Amber, either.

```
1  while (x < 10) do
2      rho ~ ContinuousUniform(-0.5, 1);
3      covM = [[1, rho], [rho, 1]];
4      w1, w2 ~ MultivariateGaussian([0, 0], covM);
5      x = x + power((w1 + w2), 2) - 2
6  od
```

Fig. 12. Random walk with correlated variables (`crwalk`).

5.2 Multivariate and Hierarchical Distributions

Figure 12 is a random walk that samples from a multivariate Gaussian distribution, with zero mean, unit variances, and correlation sampled uniformly in the range $\left[-\frac{1}{2}, 1\right]$. The `MultivariateGaussian` of line 4 is an instance of a hierarchical distribution, having parameters that are random variables. This program also contains a non-linear (polynomial) expression that updates the value of `x`. For `crwalk` we learn an SOS-form NRSM:

$$(0.1 \cdot \texttt{x} - 47.2)^2, \tag{15}$$

proving this program is PAST. To verify this, the NRSM expectation is computed via the symbolic moments of the multivariate Gaussian distribution, given its covariance matrix (line 3), and then marginalising w.r.t. `rho` (again, using the moments of the uniform distribution over $\left[-\frac{1}{2}, 1\right]$). Unfortunately, it is challenging to translate many simple programs containing hierarchical distributions into ones that can be handled by existing tools. For instance, although it is possible to simulate sampling from a bivariate Gaussian of arbitrary correlation by sampling from independent standard Gaussian distributions, this would involve computing a non-polynomial function of the correlation. Similarly, for the program in Fig. 14 (further discussed below), if a variable is exponentially distributed, $X \sim \texttt{Exponential}(1)$, then $\frac{X}{\lambda} \sim \texttt{Exponential}(\lambda)$, providing a way of simulating an exponential distribution with arbitrary parameter λ. However, this again requires a non-polynomial program expression (i.e. the reciprocal of λ) when λ is part of the program state and not a constant, and therefore out of scope for methods that restrict program expressions to being linear/polynomial.

5.3 State-Dependent Distributions and Non-Linear Expectations

```
while (x < 0 && y < 0) do
    s1 ~ Gaussian(0, 1/4);
    vx = min(2, max(0.1, vx + s1));
    s2 ~ Gaussian(0, 1/4);
    vy = min(2, max(0.1, vx + s2));
    s3 ~ Gaussian(0, 1/4);
    rho = min(1, max(-1, rho + s3));
    mean = [sqrt(1+power(x, 2)),sqrt(1+power(y, 2))];
    cov = rho * sqrt(vx * vy);
    covM = [[vx cov], [cov vy]];
    w1, w2 ~ MultivariateGaussian(mean, covM);
    x = x + w1;
    y = y + w2
od
```

Fig. 13. Gaussian random walk with time-varying and coupled noise (`gaussrw`).

Once we allow hierarchical distributions, it is natural to consider *state-dependent* distributions, i.e. distributions whose parameters depend on the program state rather than being sampled from other distributions. As an example, consider the program in Fig. 13 (a 2-dimensional Gaussian random walk with state-dependent moments). This is unsupported by existing tools because the mean of the Gaussian is a non-polynomial function of the program state. However, after defining the function $\sqrt{1+\mathtt{x}^2}$ by means of the following *polynomial* logical inequalities:

$$\mathtt{mu\_x}^2 = 1 + \mathtt{x}^2 \tag{16}$$

$$\mathtt{mu\_x} \geq 1 \tag{17}$$

(similarly for `mu_y`), we express the expected value of an SOS-form NRSM in terms of symbolic moments `mu_x`, etc. Since these moments are state-dependent, we cannot marginalise them out as in the hierarchical case. Instead we perform non-deterministic abstraction, providing inequalities $\frac{1}{10} \leq \texttt{vx}, \texttt{vy} \leq 2$ and $-1 \leq \texttt{rho} \leq 1$ as further verifier assumptions.

```
while (x < 10) do
    s ∼ Gaussian(0, 1);
    lambda = min(10, max(1, lambda + s);
    step ∼ Exponential(lambda);
    p ∼ Bernoulli(3/4);
    if (p == 1) then
        x = x + step
    else
        x = x - step
    fi
od
```

Fig. 14. State-dependent exponential random walk (`expdistrw`).

Even if program expressions are linear, the presence of state-dependent distributions can result in a non-linear verification problem, if the moments are themselves non-linear functions of the program variables. For instance, the program in Fig. 14 represents a 1-dimensional random walk, with steps sampled from an exponential distribution. Since the n^{th} moment of $\texttt{Exponential}(\lambda)$ is $\frac{n!}{\lambda^n}$, the expectation of an SOS-form NRSM is non-polynomial but still expressible in the theory of non-linear real arithmetic (see Ex. 4). For `expdistrw` we learn

$$(0.1 \cdot \texttt{x} - 3.3)^2, \tag{18}$$

whereas for `gaussrw` in Fig. 13 we learn

$$(0 \cdot \texttt{x} - 1 \cdot \texttt{y} + 11)^2 + (0 \cdot \texttt{x} + 0 \cdot \texttt{y} + 8)^2. \tag{19}$$

We translate the program in Fig. 14 for Amber by replacing the update for λ by instead sampling it uniformly from $[1, 10]$. Amber correctly identifies the program is AST, and that $(10 - \texttt{x})$ is a supermartingale expression (note, not an RSM), though does not report that the program is PAST (answering "maybe").

5.4 Undefined Moments

The ability to evaluate the cumulative distribution function (CDF) of a sampled distribution could be useful in marginalisation, even if the moments of the sampled distribution are undefined or not known analytically to infinite precision. An example is Fig. 15: the program samples from the standard Cauchy distribution, for which all moments are undefined. Since the sampled value is *only* used to determine which branch of a conditional is taken, the RSM expectation is well defined, and can be expressed in terms of the standard Cauchy CDF. Namely, the if-branch is taken with probability $q = 1 - \left(\frac{1}{\pi}\arctan(10) + \frac{1}{2}\right)$. This equation is not expressible using polynomials; so we perform a sound approximation by introducing a new variable that is quantified over a small interval surrounding a finite precision approximation to q. This allows us to learn and verify the SOR-form NRSM:

$$\mathrm{ReLU}(1.2 \cdot \mathtt{x} + 9.1). \tag{20}$$

```
while (x > 0) do
    p ~ StandardCauchy();
    if (p > 10) then
        x = x + 2
    else
        x = x - 1
    fi
od
```

Fig. 15. Sliced Cauchy distribution (`slicedcauchy`).

For our experimental evaluation (Sect. 6) we create a modified version of each of the six case studies described in this section, as follows:

- program `marbles3` is a generalisation of `marbles` to three marble types, instead of two;
- `probfact2` uses 5/8 as the Bernoulli parameter, rather than 3/4;
- `crwalk2` samples `rho` from a Beta(1, 3) distribution, instead of a uniform distribution over $\left[-\frac{1}{2}, 1\right]$;
- `expdistrw2` samples from an exponential distribution, where parameter `lambda` is replaced by `lambda*lambda`;
- `gaussrw2` uses $[3 + 1/(1 - \mathtt{x}), 3 + 1/(1 - \mathtt{y})]^T$ for its mean vector, instead of $[\sqrt{1 + \mathtt{x}^2}, \sqrt{1 + \mathtt{y}^2}]^T$; and
- `slicedcauchy2` has a loop guard of $\mathtt{x} < 10$, instead of $\mathtt{x} > 0$, and swaps the two branches of the conditional.

5.5 Rare Transitions

A limitation of relying on a sampled transition dataset to learn NRSM parameters is we rely on the average $\mathbb{E}_{p' \sim P'}[\eta(p')]$ in Eq. (5) being accurate (see Sect. 3). This assumption is challenged by programs that have certain control-flow paths of very low probability, which are unlikely to be sampled by the interpreter. For example, in the context of the ambitious marble collector (Fig. 1), Fig. 16 shows that when the probability of obtaining a red marble decreases below 2^{-7}, our success rate drops. This is because a lower probability makes the corresponding control-flow path rarer in the dataset, to the point where the expected value of the NRSM cannot be estimated accurately.

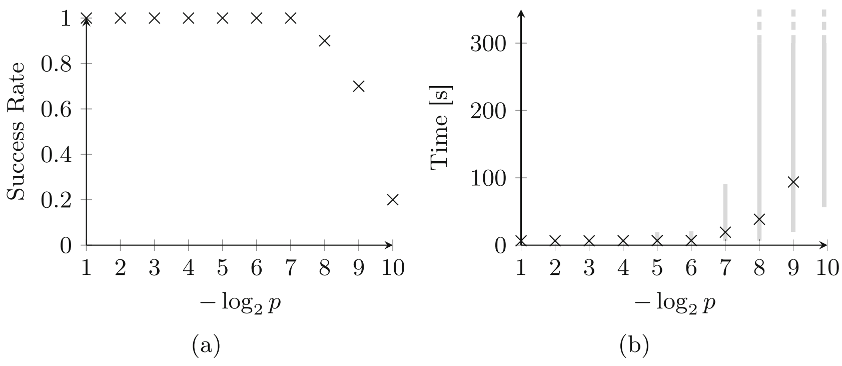


Fig. 16. Success rate and execution times for the ambitious marble collector program (Fig. 1), where p is the probability of taking the if-branch. Success rate refers to the fraction of 10 executions that succeeded in finding an NRSM before a timeout of 300 s. Execution times show the median time with the error bar ranging between the minimum and maximum times of the 10 executions.

6 Experimental Results

We built a prototype implementation of our framework (in Python) and present experimental results for benchmarks adapted from previous work, as well as our own case studies (from Sect. 5). The case studies illustrate programs for which our framework synthesises an RSM, yet existing tools cannot prove to be PAST.

The learner is implemented with JAX [9]. To train NRSMs, we use AdaGrad [18] for gradient-based optimisation, with a learning rate of 10^{-2}. Parameters are initialised by sampling from Gaussian distributions: weight parameters are sampled from a zero-mean Gaussian, whereas the bias parameters are sampled either from a Gaussian with mean 10 (for SOR candidates) or mean 0 (for SOS candidates). We verify the NRSMs using the SMT solver Z3 [26,40]. The outcomes are obtained on the following platform: macOS Catalina version 10.15.4, 8 GB RAM, Intel Core i5 CPU 2.4 GHz QuadCore, 64-bit.

As mentioned in Sect. 4, the verifier checks a candidate NRSM over states satisfying the loop predicate, which characterises the set of reachable states. For our experiments, we manually provide the NRSM expectation, and augment the guard predicate with additional invariants where necessary. We generate outcomes using two different rounding strategies (Sect. 4): an "aggressive" rounding strategy which generated between 80 and 120 candidates per CEGIS iteration, and a "weaker" rounding strategy producing between 15 to 25 candidates per CEGIS iteration. The outcomes in Table 1 used the aggressive rounding strategy.

Table 1. Experimental results over existing (top section) and newly added benchmarks (bottom section); (c) indicates the benchmark uses continuous distributions, (d) indicates it only uses discrete distributions. All reported times are in seconds, oot indicates time-out after 300 s, n/a indicates the tool terminated without definite answer, and—indicates the benchmark is unsupported. Our method is run 10 times with different seeds; the overall success rate is reported. Runtimes of interpretation, training, verification phases, and # of CEGIS iterations refer to the run with median total runtime.

Program	Amber [39]	Farkas' lemma [2]	Absynth [41]	Succ. rate	Inter.	Train.	Verif.	#iter	NRSM
`Hare & Tortoise` (d)	0.04	≈0	0.09	10/10	0.61	3.86	0.70	0	SOR
`exmini/terminate` (d)	—	0.02	oot	10/10	1.75	29.35	7.67	2	SOR
`aaron2` (d)	0.03	0.02	0.02	10/10	0.04	2.27	0.01	0	SOR
`catmouse` (c)	0.03	0.02	—	9/10	0.39	12.41	3.68	1	SOS
`counterex1c` (d)	—	0.02	0.22	8/10	1.00	6.71	0.02	0	SOR
`easy1` (d)	0.12	0.01	0.05	10/10	1.12	5.55	1.27	0	SOR
`easy2` (c)	0.04	0.02	—	10/10	1.55	6.79	0.18	0	SOS
`ndecr` (d)	0.04	0.02	0.03	10/10	1.18	5.63	0.02	0	SOR
`random1d` (c)	0.05	0.02	—	10/10	1.14	4.86	0.79	0	SOS
`rsd` (d)	error	0.01	oot	10/10	1.14	6.18	2.04	0	SOR
`speedFails1` (d)	0.07	0.01	0.04	10/10	0.45	4.09	0.67	0	SOR
`speedPldi2` (d)	—	0.02	0.40	9/10	1.36	7.85	0.02	0	SOR
`speedPldi3` (d)	—	0.02	0.36	8/10	2.58	30.70	2.12	1	SOR
`speedPldi4` (d)	—	0.02	0.17	10/10	0.68	5.07	0.04	0	SOR
`speedSingleSingle` (c)	0.03	0.02	—	10/10	0.39	2.85	0.51	0	SOS
`speedSingleSingle2` (d)	—	0.02	0.15	10/10	0.83	7.30	0.04	0	SOR
`wcet0` (d)	—	0.02	0.10	10/10	1.45	5.64	0.09	0	SOR
`wcet1` (d)	—	0.02	0.10	10/10	0.85	4.31	0.09	0	SOR
`probfact` (d)	—	—	n/a	10/10	0.49	6.12	0.16	0	SOR
`probfact2` (d)	—	—	n/a	10/10	0.45	5.89	0.23	0	SOR
`marbles` (d)	—	—	n/a	10/10	0.84	10.83	0.91	0	SOR
`marbles3` (d)	—	—	n/a	10/10	0.40	70.14	7.87	2	SOR
`crwalk` (c)	—	—	—	10/10	0.53	3.06	1.56	1	SOS
`crwalk2` (c)	—	—	—	10/10	1.32	3.11	0.75	1	SOS
`expdistrw` (c)	n/a	—	—	10/10	0.05	1.53	0.01	0	SOS
`expdistrw2` (c)	n/a	—	—	10/10	4.92	3.15	1.03	1	SOS
`gaussrw` (c)	—	—	—	10/10	10.30	3.45	0.75	0	SOS
`gaussrw2` (c)	—	—	—	9/10	15.46	4.91	5.33	0	SOS
`slicedcauchy` (c)	—	—	—	10/10	0.02	3.31	0.01	0	SOR
`slicedcauchy2` (c)	—	—	—	10/10	0.01	2.16	0.03	0	SOR

Benchmarks from Previous Work. We run our prototype on single-loop programs from the WTC benchmark suite [3], augmented with probabilistic branching and assignments [2]. These correspond to the programs in the first section of Table 1. We perturb assignment statements by adding noise sampled from a discrete uniform distribution of support $\{-2, 2\}$, or a continuous uniform distribution on the interval $[-2, 2]$. The *while* loops are also made probabilistic; with probability 1/2 the loop is executed, and with the remaining probability a `skip` command is executed.

We compare our framework against three existing tools. The first is AMBER [39]: where possible, we translate instances from the WTC suite into the language of AMBER, but this is not possible for some programs where the loop predicate is a logical conjunction or disjunction of predicates (indicated by dashes in Table 1). Second, we compare against a tool for synthesising affine lexicographic RSMs (referred to as Farkas' lemma) for affine programs (i.e. containing only linear expressions), based on reduction to linear programming via Farkas' lemma [2]. This is applicable to probabilistic programs with nested-loops, unlike our method. However, since it is limited to affine programs and affine lexicographic RSMs, it is not able to analyse all the programs we consider (again, indicated by dashes in Table 1). The third tool is ABSYNTH [41], for which we are able to encode all programs that were limited to discrete random variables.

The experimental results (Table 1) show that for all the WTC benchmarks our approach has a success rate of at least 8/10, and is able to synthesise an RSM within 2 iterations (for the seed that results in median total execution time). For 15 of the 18 WTC benchmarks no full CEGIS iterations are required. As expected our approach, particularly the learning component, is much slower than all three tools. However, our framework has broader applicability, as illustrated with the next set of experiments.

Newly Defined Case Studies. The examples in the second section of Table 1 (from Sect. 5) are not proven PAST by any of the three tools. Our approach is able to do so with a success rate of at least 9/10, under the "aggressive" rounding strategy. Of the new examples, `marbles3` (Sect. 5) requires the longest time, since we use an NRSM with $h = 3$ ReLU nodes (see Sect. 3), and six of the nine parameters must be brought sufficiently close to zero to learn a valid RSM. For `gaussrw`/`gaussrw2`, we find it necessary to set an SMT solver time limit within the CEGIS loop (of 200 ms for `gaussrw`, and 5 s for `gaussrw2`), such that candidates taking longer than this to verify are skipped. The fact that these examples are harder to verify is unsurprising, given that they give rise to non-polynomial decision problems, containing equationally defined rational expressions. In comparing the two rounding strategies, we find that using the "aggressive" strategy tends to result in fewer CEGIS iterations, reducing the learner time, while increasing the verifier time: this is to be expected, since a larger number of candidates needs to be checked in each CEGIS iteration.

7 Conclusion

We have presented the first machine learning method for the termination analysis of probabilistic programs. We have introduced a loss function for training neural networks so that they behave as RSMs over sampled execution traces; our training phase is agnostic to the program and thus easily portable to different programming languages. Reasoning about the program code is entirely delegated to our checking phase which, by SMT solving over a symbolic encoding of program and neural network, verifies whether the neural network is a sound RSM. Upon a positive answer, we have formally certified that the program is PAST; upon a negative answer, we obtain a counterexample that we use to resample traces and repeat training in a CEGIS loop. Our procedure runs indefinitely for programs that are not PAST, as these necessarily lack a ranking supermartingale, and may run indefinitely for some PAST programs. Nevertheless, we have experimentally demonstrated over several PAST benchmarks that our method is effective in practice and covers a broad range of programs w.r.t. existing tools.

Our method naturally generalises to deeper networks, but whether these are necessary in practice remains an open question; notably, neural networks with one hidden layer were sufficient to solve our examples. We have exclusively tackled the PAST question, and techniques for almost-sure (but not necessarily PAST) termination and non-termination exist [16,37,39]. Our results pose the basis for future research in machine learning (and CEGIS) for the formal verification of probabilistic programs. Different verification questions will require different learning models. Our approach lends itself to extensions toward probabilistic safety, exploiting supermartingale inequalities, and towards the non-termination question, using repulsing supermartingales [16]. Adapting our method to termination analysis with infinite expected time is also a matter for future investigation [37]. Moreover, we have exclusively considered purely probabilistic single-loop programs: generalisations to programs with non-determinism, arbitrary control-flow, and concurrency are material for future work [15,20,35].

Acknowledgments. This work was in part supported by a partnership between Aerospace Technology Institute (ATI), Department for Business, Energy & Industrial Strategy (BEIS) and Innovate UK under project HICLASS (113213), by the Engineering and Physical Sciences Research Council (EPSRC) Doctoral Training Partnership, by the Department of Computer Science Scholarship, University of Oxford, and by the DeepMind Computer Science Scholarship.

References

1. Abate, A., Ahmed, D., Giacobbe, M., Peruffo, A.: Formal synthesis of Lyapunov neural networks. IEEE Control. Syst. Lett. **5**(3), 773–778 (2021)
2. Agrawal, S., Chatterjee, K., Novotný, P.: Lexicographic ranking supermartingales: an efficient approach to termination of probabilistic programs. Proc. ACM Program. Lang. **2(POPL)**, 34:1–34:32 (2018)
3. Alias, C., Darte, A., Feautrier, P., Gonnord, L.: Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In: Cousot, R., Martel, M. (eds.) Static Analysis, pp. 117–133. Springer, Berlin, Heidelberg (2010)
4. Alur, R., et al.: Syntax-guided synthesis. In: FMCAD, pp. 1–8. IEEE (2013)
5. Avanzini, M., Dal Lago, U., Yamada, A.: On probabilistic term rewriting. Sci. Comput. Program. **185**, 102338 (2020)

6. Avanzini, M., Moser, G., Schaper, M.: A modular cost analysis for probabilistic programs. Proc. ACM Program. Lang. **4(OOPSLA)**, 172:1–172:30 (2020)
7. Batz, K., Kaminski, B.L., Katoen, J.-P., Matheja, C.: How long, O Bayesian network, will I sample thee? In: Ahmed, A. (ed.) ESOP 2018. LNCS, vol. 10801, pp. 186–213. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89884-1_7
8. Bournez, O., Garnier, F.: Proving positive almost-sure termination. In: Giesl, J. (ed.) RTA 2005. LNCS, vol. 3467, pp. 323–337. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-32033-3_24
9. Bradbury, J., et al.: JAX: composable transformations of Python+NumPy programs (2018). http://github.com/google/jax
10. Chakarov, A., Sankaranarayanan, S.: Probabilistic program analysis with martingales. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 511–526. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_34
11. Chakarov, A., Voronin, Y.-L., Sankaranarayanan, S.: Deductive proofs of almost sure persistence and recurrence properties. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 260–279. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_15
12. Chang, Y., Roohi, N., Gao, S.: Neural Lyapunov control. In: NeurIPS, pp. 3240–3249 (2019)
13. Chattenjee, K., Fu, H., Novotný, P.: Termination analysis of probabilistic programs with martingales. In: Barthe, G., Katoen, J.P., Silva, A. (eds.) Foundations of Probabilistic Programming, p. 221–258. Cambridge University Press (2020)
14. Chatterjee, K., Fu, H., Goharshady, A.K.: Termination analysis of probabilistic programs through Positivstellensatz's. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 3–22. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_1
15. Chatterjee, K., Fu, H., Novotný, P., Hasheminezhad, R.: Algorithmic analysis of qualitative and quantitative termination problems for affine probabilistic programs. ACM Trans. Program. Lang. Syst. **40**(2), 7:1–7:45 (2018)
16. Chatterjee, K., Novotný, P., Zikelic, D.: Stochastic invariants for probabilistic termination. In: POPL, pp. 145–160. ACM (2017)
17. Dahlqvist, F., Silva, A.: Semantics of probabilistic programming: a gentle introduction. In: Barthe, G., Katoen, J.P., Silva, A. (eds.) Foundations of Probabilistic Programming, pp. 1–42. Cambridge University Press (2020)
18. Duchi, J.C., Hazan, E., Singer, Y.: Adaptive subgradient methods for online learning and stochastic optimization. In: COLT, pp. 257–269. Omnipress (2010)
19. Fioriti, L.M.F., Hermanns, H.: Probabilistic termination: Soundness, completeness, and compositionality. In: POPL, pp. 489–501. ACM (2015)
20. Fu, H., Chatterjee, K.: Termination of nondeterministic probabilistic programs. In: Enea, C., Piskac, R. (eds.) VMCAI 2019. LNCS, vol. 11388, pp. 468–490. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-11245-5_22
21. Giacobbe, M., Kroening, D., Parsert, J.: Neural termination analysis. CoRR abs/2102.03824 (2021)
22. Goodfellow, I., Bengio, Y., Courville, A.: Deep learning. MIT Press (2016)
23. Gordon, A.D., Henzinger, T.A., Nori, A.V., Rajamani, S.K.: Probabilistic programming. In: FOSE, pp. 167–181. ACM (2014)
24. Heizmann, M., Hoenicke, J., Podelski, A.: Termination analysis by learning terminating programs. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 797–813. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_53

25. Huang, M., Fu, H., Chatterjee, K., Goharshady, A.K.: Modular verification for almost-sure termination of probabilistic programs. Proc. ACM Program. Lang. **3(OOPSLA)**, 129:1–129:29 (2019)
26. Jovanović, D., de Moura, L.: Solving non-linear arithmetic. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS (LNAI), vol. 7364, pp. 339–354. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31365-3_27
27. Kaminski, B.L., Katoen, J.-P., Matheja, C.: On the hardness of analyzing probabilistic programs. Acta Informatica **56**(3), 255–285 (2018). https://doi.org/10.1007/s00236-018-0321-1
28. Kapinski, J., Deshmukh, J.V., Sankaranarayanan, S., Aréchiga, N.: Simulation-guided Lyapunov analysis for hybrid dynamical systems. In: HSCC, pp. 133–142. ACM (2014)
29. Kozen, D.: Semantics of probabilistic programs. J. Comput. Syst. Sci. **22**(3), 328–350 (1981)
30. Kura, S., Unno, H., Hasuo, I.: Decision tree learning in CEGIS-based termination analysis. In: CAV (2021)
31. Le, T.C., Antonopoulos, T., Fathololumi, P., Koskinen, E., Nguyen, T.: DynamiTe: Dynamic termination and non-termination proofs. Proc. ACM Program. Lang. **4(OOPSLA)**, 189:1–189:30 (2020)
32. Lee, W., Wang, B.-Y., Yi, K.: Termination analysis with algorithmic learning. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 88–104. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31424-7_12
33. Lee, W., Yu, H., Rival, X., Yang, H.: Towards verified stochastic variational inference for probabilistic programs. Proc. ACM Program. Lang. **4(POPL)**, 16:1–16:33 (2020)
34. Li, Y., Ying, M.: Algorithmic analysis of termination problems for quantum programs. Proc. ACM Program. Lang. **2(POPL)**, 35:1–35:29 (2018)
35. Lin, A.W., Rümmer, P.: Liveness of randomised parameterised systems under arbitrary schedulers. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 112–133. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_7
36. McIver, A., Morgan, C.: Abstraction, Refinement and Proof for Probabilistic Systems. Monographs in Computer Science. Springer, Berlin (2005)
37. McIver, A., Morgan, C., Kaminski, B.L., Katoen, J.: A new proof rule for almost-sure termination. Proc. ACM Program. Lang. **2(POPL)**, 33:1–33:28 (2018)
38. Meyer, F., Hark, M., Giesl, J.: Inferring expected runtimes of probabilistic integer programs using expected sizes. In: TACAS 2021. LNCS, vol. 12651, pp. 250–269. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72016-2_14
39. Moosbrugger, M., Bartocci, E., Katoen, J.-P., Kovács, L.: Automated termination analysis of polynomial probabilistic programs. In: ESOP 2021. LNCS, vol. 12648, pp. 491–518. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72019-3_18
40. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
41. Ngo, V.C., Carbonneaux, Q., Hoffmann, J.: Bounded expectations: resource analysis for probabilistic programs. In: PLDI, pp. 496–512. ACM (2018)
42. Nori, A.V., Sharma, R.: Termination proofs from tests. In: ESEC/SIGSOFT FSE, pp. 246–256. ACM (2013)
43. Richards, S.M., Berkenkamp, F., Krause, A.: The Lyapunov neural network: Adaptive stability certification for safe learning of dynamical systems. In: CoRL. Proceedings of Machine Learning Research, vol. 87, pp. 466–476. PMLR (2018)

44. Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: ASPLOS, pp. 404–415. ACM (2006)
45. Urban, C., Gurfinkel, A., Kahsai, T.: Synthesizing ranking functions from bits and pieces. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 54–70. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_4

Ghost Signals: Verifying Termination of Busy Waiting

Tobias Reinhard(✉) and Bart Jacobs

imec-DistriNet Research Group, KU Leuven, Leuven, Belgium
{tobias.reinhard,bart.jacobs}@kuleuven.be

Abstract. Programs for multiprocessor machines commonly perform busy waiting for synchronization. We propose the first separation logic for modularly verifying termination of such programs under fair scheduling. Our logic requires the proof author to associate a *ghost signal* with each busy-waiting loop and allows such loops to iterate while their corresponding signal s is not set. The proof author further has to define a well-founded order on signals and to prove that if the looping thread holds an obligation to set a signal s', then s' is ordered above s. By using conventional shared state invariants to associate the state of ghost signals with the state of data structures, programs busy-waiting for arbitrary conditions over arbitrary data structures can be verified.

1 Introduction

Programs for multiprocessor machines commonly perform busy waiting for synchronization [22,23]. In this paper, we propose a separation logic [24,31] to modularly verify termination of such programs under fair scheduling. Specifically, we consider programs where some threads busy-wait for a certain condition C over a shared data structure to hold, e.g., a memory flag being set by other threads. By modularly, we mean that we reason about each thread and each function in isolation. That is, we do not reason about thread scheduling or interleavings. We only consider these issues when proving the soundness of our logic. Assuming fair scheduling is necessary since busy-waiting for a condition C only terminates if the thread responsible for establishing the condition is sufficiently often scheduled to establish C.

Busy waiting is an example of *blocking* behaviour, where a thread's progress *requires interference* from other threads. This is not to be confused with *non-blocking* concurrency, where a thread's progress does not rely on—and may in fact be *impeded* by—interference from other threads. Existing proposed approaches for verifying termination of concurrent programs consider only programs that only involve non-blocking concurrent objects [32], or *primitive blocking constructs* of the programming language, such as acquiring built-in mutexes, receiving from built-in channels, joining threads, or waiting for built-in monitor condition variables [2,5,19], or both [11]. Existing techniques that do support busy waiting are not Hoare logics; instead, they verify termination-preserving

https://doi.org/10.1007/978-3-030-81688-9_2

contextual refinements between more concrete and more abstract implementations of busy-waiting concurrent objects [15,21]. In contrast, we here propose the first conventional program logic for modular verification of termination of programs involving busy waiting, using Hoare triples as module specifications.

In order to prove that a busy-waiting loop terminates, we have to prove that it performs only finitely many iterations. To do this we introduce a special form of *ghost resources* [13] which we call *ghost signals*. As ghost resources they only exist on the verification level and hence do not affect the program's runtime behaviour. Signals are initially unset and come with an obligation to set them. Setting a signal does not by definition correspond to any runtime condition. So, in order to use a signal s effectively, anyone using our approach has to prove an invariant stating that s is set if and only if the condition of interest holds. Further, the proof author must prove that every thread discharges all its obligations by performing the corresponding actions, e.g., by setting a signal and establishing the corresponding condition by setting the memory flag.

In our verification approach we tie every busy-waiting loop to a finite set of ghost signals S that correspond to the set of conditions the loop is waiting for. Every iteration that does not terminate the loop must be justified by the proof author proving that some signal $s \in S$ has indeed not been set, yet. This way, we reduce proving termination to proving that no signal is waited for infinitely often.

Our approach ensures that no thread directly or indirectly waits for itself by requiring the proof author (i) to choose a well-founded and partially ordered set of levels $\mathcal{L}evs$ and (ii) to assign a level to every signal and by (iii) only allowing a thread to wait for a signal if the signal's level is lower than the level of each held obligation. This guarantees that every signal is waited for only finitely often and hence that every busy-waiting loop terminates. We use this to prove that every program that is verified using our approach indeed terminates.

We start by gradually introducing the intuition behind our verification approach and the concepts we use. In Sect. 2.1 and Sect. 2.2 we present the main aspects of using signals to verify termination. We start by treating them as physical thread-safe resources and only consider busy waiting for a signal to be set. Then, we drop thread-safety and explain how to prove data-race- and deadlock-freedom. In Sect. 2.3 and Sect. 2.4 we generalize our approach to busy waiting for arbitrary conditions over arbitrary data structures and then lift signals to the verification level by introducing ghost signals.

In Sect. 3 we sketch the verification of a realistic producer-consumer example involving a bounded FIFO to demonstrate our approach's usability and address fine-grained concurrency in Sect. 4. Further, we describe the available tool support in Sect. 5 and discuss integrating higher-order features in Sect. 6. We conclude by comparing our approach to related work and reflecting on it in Sect. 7 and Sect. 8.

We formally define our logic and prove its soundness in the extended version of this paper [28]. To keep the presentation in this paper simple, we assume busy-waiting loops to have a certain syntactical form. In our technical report [29] we present a generalised version of our logic and its soundness proof. Further, we verify the realistic example presented in Sect. 3 in full detail in the extended version of this paper and in the technical report, using the respective version of our logic. We used our tool support to verify C versions of the bounded FIFO example and the CLH lock. The tool we used and the annotated .c files can be found at [10,26,27].

2 A Guide on Verifying Termination of Busy Waiting

When we try to verify termination of busy-waiting programs, multiple challenges arise. Throughout this section, we describe these challenges and our approach to overcome them. In Sect. 2.1 we start by discussing the core ideas of our logic. In order to simplify the presentation we initially consider a simple language with built-in thread-safe *signals* and a corresponding minimal example where one thread busy-waits for such a signal. Signals are heap cells containing boolean values that are specially marked as being solely used for busy waiting. Throughout this section, we generalize our setting as well as our example towards one that allows to verify programs with busy waiting for arbitrary conditions over arbitrary shared data structures. In Sect. 2.2 we present the concepts necessary to verify data-race-, deadlock-freedom and termination in the presence of built-in signals that are not thread safe. In Sect. 2.3 we explain how to use these non-thread-safe signals to verify programs that wait for arbitrary conditions over shared data structures. We illustrate this by an example waiting for a shared heap cell to be set. In Sect. 2.4 we erase the signals from our program and lift them to the verification level in the form of a concept we call *ghost signals*.

2.1 Simplest Setting: Thread-Safe Physical Signals

We want to verify programs that busy-wait for arbitrary conditions over arbitrary shared data structures. As a first step towards achieving this, we first consider programs that busy-wait for simple boolean flags, specially marked as being used for the purpose of busy waiting. We call these flags *signals*. For now, we assume that read and write operations on signals are thread-safe. Consider a simple programming language with built-in signals and with the following commands: (i) **new_signal** for creating a new unset signal, (ii) **set_signal**(x) for setting x and (iii) **await is_set**(x) for busy-waiting until x is set. Figure 1 presents a minimal example where two threads communicate via a shared signal sig. The main thread creates the signal sig and forks a new thread that busy-waits for sig to be set. Then, the main thread sets the signal. As we assume signal operations to be thread-safe in this example, we do not have to care about potential data races. Notice that like all busy-waiting programs, this program is guaranteed to terminate only under fair thread scheduling: Indeed, it does not terminate if the main thread is never scheduled after it forks the new thread. In this paper we verify termination under fair scheduling.

```
let sig := new_signal in
fork await is_set(sig);
set_signal(sig)
```

Fig. 1. Minimal example with two threads communicating via a physical thread-safe signal.

Augmented Semantics

Obligations. The only construct in our language that can lead to non-termination are busy-waiting loops of the form **await is_set**(sig). In order to prove that programs terminate it is therefore sufficient to prove that all created signals are eventually set. We use so-called *obligations* [5,6,16,19] to ensure this. These are *ghost resources* [13], i.e., resources that do not exist during runtime and can hence not influence a program's runtime behaviour. They carry, however, information relevant to the program's verification. Generally, holding an obligation requires a thread to discharge it by performing a certain action. For instance, when the main thread in our example creates signal sig, it simultaneously creates an obligation to set it. The only way to discharge this obligation is to set sig.

We denote thread IDs by θ and describe which obligations a thread θ holds by bundling them into an obligations chunk $\theta.\mathsf{obs}(O)$, where O is a multiset of signals. We denote multisets by double braces $\{\!\{\ldots\}\!\}$ and multiset union by $\uplus$. Each occurrence of a signal s in O corresponds to an obligation by thread θ to set s. Consequently, $\theta.\mathsf{obs}(\emptyset)$ asserts that thread θ does not hold any obligations.

Augmented Semantics. In the *real* semantics of the programming language we consider here, ghost resources such as obligations do not exist during runtime. To prove termination, we consider an *augmented* version of it that keeps track of ghost resources during runtime. In this semantics, we maintain the invariant that every thread holds exactly one obs chunk. That is, for every running thread θ, our heap contains a unique heap cell $\theta.\mathsf{obs}$ that stores the thread's bag of obligations. Further, we let a thread get stuck if it tries to finish while it still holds undischarged obligations. Note that we use the term *finish* to refer to thread-local behaviour while we write *termination* to refer to program-global behaviour, i.e., meaning that every thread finishes. For every augmented execution there trivially exists a corresponding execution in the real semantics.

Figure 2 presents some of the reduction rules we use to define the augmented semantics. We use $\widehat{h}$ to refer to augmented heaps, i.e., heaps that can contain ghost resources. A reduction step has the form $\widehat{h}, c \overset{\theta}{\rightsquigarrow}_{\mathsf{aug}} \widehat{h}', c', T$ expresses that thread θ reduces heap $\widehat{h}$ (which is shared by all threads) and command c to heap $\widehat{h}'$ and command c'. Further, T represents the set of threads forked during this step. It is either empty or a singleton containing the new thread's ID and the command it is going to execute, i.e., $\{(\theta_f, c_f)\}$. We omit it whenever it is clear from the context that no thread is forked. Further, we denote disjoint union of sets by $\sqcup$.

Our reduction rules comply with the intuition behind obligations we outlined above. Aug-Red-NewSignal creates a new signal and simultaneously a corresponding obligation. The only way to discharge it is by setting the signal using Aug-Red-SetSignal.

Aug-Red-NewSignal

$$\frac{id \notin \mathsf{ids}(\widehat{h}) \qquad L \in \mathcal{L}evs}{\widehat{h} \sqcup \{\theta.\mathsf{obs}(O)\}, \textbf{new\_signal} \overset{\theta}{\rightsquigarrow}_{\mathsf{aug}} \widehat{h} \sqcup \{\theta.\mathsf{obs}(O \uplus \{\!\!\{(id, L)\}\!\!\}), \mathsf{signal}((id, L))\}, id}$$

Aug-Red-SetSignal

$$\widehat{h} \sqcup \{\theta.\mathsf{obs}(O \uplus \{\!\!\{s\}\!\!\})\}, \textbf{set\_signal}(s.\mathsf{id}) \overset{\theta}{\rightsquigarrow}_{\mathsf{aug}} \widehat{h} \sqcup \{\theta.\mathsf{obs}(O), \mathsf{signalSet}(s)\}, \mathsf{tt}$$

Aug-Red-Fork

$$\frac{\theta_f \notin \mathsf{thIds}(\widehat{h})}{\widehat{h} \sqcup \{\theta.\mathsf{obs}(O \uplus O_f)\}, \textbf{fork } c \overset{\theta}{\rightsquigarrow}_{\mathsf{aug}} \widehat{h} \sqcup \{\theta.\mathsf{obs}(O), \theta_f.\mathsf{obs}(O_f)\}, \mathsf{tt}, \{(\theta_f, c)\}}$$

Aug-Red-Await

$$\frac{\theta.\mathsf{obs}(O) \in \widehat{h} \qquad \mathsf{signal}(s) \in \widehat{h} \qquad \mathsf{signalSet}(s) \notin \widehat{h} \qquad s.\mathsf{lev} \prec_{\mathsf{L}} O}{\widehat{h}, \textbf{await is\_set}(s.\mathsf{id}) \overset{\theta}{\rightsquigarrow}_{\mathsf{aug}} \widehat{h}, \textbf{await is\_set}(s.\mathsf{id})}$$

Fig. 2. Reduction rules for augmented semantics.

Forking. Whenever a thread forks a new thread, it can pass some of its obligations to the newly forked thread, cf. Aug-Red-Fork. Forking a new thread with ID θ_f also allocates a new heap cell $\theta_f.\mathsf{obs}$ to store its bag of obligations. Since this is the only way to allocate a new obs heap cell, we will never run into a heap $\widehat{h} \sqcup \{\theta.\mathsf{obs}(O)\} \sqcup \{\theta.\mathsf{obs}'(O')\}$ that contains multiple obligations chunks belonging to the same thread θ. Remember that threads cannot finish while holding obligations. This prevents them from dropping obligations via dummy forks.

Levels. In order to prove that a busy-waiting loop **await is_set**(sig) terminates, we must ensure that the waiting thread does not directly or indirectly wait for itself. We could just check that it does not hold an obligation for the signal it is waiting for, but that is not sufficient as the following example demonstrates: Consider a program with two signals $\mathsf{sig}_1, \mathsf{sig}_2$ and two threads. Let one thread hold the obligation for sig_2 and execute **await is_set**(sig_1); **set_signal**(sig_2). Likewise, let the other thread hold the obligation for sig_1 and let it execute **await is_set**(sig_2); **set_signal**(sig_1).

To prevent such *wait cycles* modularly, we apply the usual approach [3,4,19]. For every program that we want to execute in our augmented semantics, we choose a partially ordered set of levels $\mathcal{L}evs$. Further, during every reduction step in the augmented semantics that creates a signal s, we pick a level $L \in \mathcal{L}evs$ and associate it with s. Note that much like obligations, levels do not exit during runtime in the real semantics. Signal chunks in the augmented semantics have the form $\mathsf{signal}((id, L))$ where id is the unique signal identifier returned by **new_signal**. The level assigned to any signal can be chosen freely, cf. Aug-Red-NewSignal. In practice, determining levels boils down to solving a set of constraints that reflect the dependencies. In our example, however, the choice is trivial as it only involves a single signal. We choose $\mathcal{L}evs = \{0\}$ and 0 as level for sig and thereby get $\mathsf{signal}((\mathsf{sig}, 0))$. Generally, we denote signal tuples by $s = (id, L)$. Now we can rule out cyclic wait dependencies by only allowing a thread to busy-wait for a signal s if its level $s.\mathsf{lev}$ is smaller than the level of each held obligation, cf. Aug-Red-Await[1]. Given a bag of obligations O, we denote this by $s.\mathsf{lev} \prec_{\mathsf{L}} O$.

Proving Termination. As we will explain below, the augmented semantics has no fair infinite executions. We can use this as follows to prove that a program c terminates under fair scheduling: For every fair infinite execution of c, show that we can construct a corresponding augmented execution. (This requires that each step's side conditions in the augmented semantics are satisfied. Note that we thereby prove certain properties for the real execution, like absence of cyclic wait dependencies.) As there are no fair infinite executions in the augmented semantics, we get a contradiction. It follows that c has no fair infinite executions in the real semantics.

Soundness. In order to prove soundness of our approach, we must prove that there indeed are no fair infinite executions in the augmented semantics. This boils down to proving that no signal can be waited for infinitely often. Consider any program and any fair augmented execution of it. Consider the execution's *program order graph*, (i) whose nodes are the execution steps and (ii) which has an edge from a step to the next step of the same thread and to the first step of the forked thread, if it is a fork step. Notice that for each obligation created during the execution, the set of nodes corresponding to a step made by a thread while that thread holds the obligation constitutes a path that ends when the obligation is discharged. We say that this path *carries* the obligation.

It is not possible that a signal is waited for infinitely often. Indeed, suppose some signals S^∞ are. Take $s_{\min} \in S^\infty$ with minimal level. Since $s_{\min}$ is never set, the path in the program order graph that carries the obligation must be infinite as well. Indeed, suppose it is finite. The final node N of the path cannot discharge the obligation without setting the signal, so it must pass the obligation on either to the next step of the same thread or to a newly forked thread. By fairness of the scheduler, both of these threads will eventually be scheduled. This contradicts N being the final node of the path.

The path carrying the obligation for $s_{\min}$ waits only for signals that are waited for finitely often. (Remember that Aug-Red-Await requires the signal waited for to be of a lower level than all held obligations, i.e., a lower level than that of $s_{\min}$.) It is therefore a finite path. A contradiction.

[1] For simplicity, our augmented semantics assumes that the level order and the level associated with any object remains fixed for the entire execution. However, following the approach presented in [18], it would be sound to add a step rule that allows a thread to change the level of an object it has exclusive access to (cf. Sect. 2.2).

Notice that the above argument relies on the property that every non-empty set of levels has a minimal element. For this reason, for termination verification we require that $\mathcal{L}evs$ is not just partially ordered, but also well-founded.

Program Logic

Directly using the augmented semantics to prove that our example program terminates is cumbersome. In the following, we present a separation logic that simplifies this task.

Safety. We call a program c *safe* under a (partial) heap $\widehat{h}$ if it provides all the resources necessary such that both c and any threads it forks can execute without getting stuck in the augmented semantics. (This depends on the angelic choices.) We denote this by $\mathsf{safe}(\widehat{h}, c)$ [33][2].

Consider a program c that is safe under an augmented heap $\widehat{h}$. Let h be the real heap that matches $\widehat{h}$ apart from the ghost resources. Then, for every real execution that starts with h we can construct a corresponding augmented execution.

Specifications. We use Hoare triples $\{A\}\ c\ \{\lambda r.\, B(r)\}$ [8] to specify the behaviour of a program c. Such a triple expresses the following: Consider any evaluation context E, such that for every return value v, running $E[v]$ from a state that satisfies $B(v)$ is safe. Then, running $E[c]$ from a state that satisfies A is safe.

Proof System. We define a proof relation $\vdash$ which ensures that whenever we can prove $\vdash\ \{A\}\ c\ \{\lambda r.\, B(r)\}$, then c complies with the specification $\{A\}\ c\ \{\lambda r.\, B(r)\}$. Figure 3b presents some of the proof rules we use to define $\vdash$. As we evolve our setting throughout this section, we also adapt our proof rules. Rules that will be changed later are marked with a prime in their name. The full set of rules is presented in the extended version of this paper [28]. Our proof rules PR-SetSignal' and PR-Await' are similar to the rules for sending and receiving on a channel presented in [19].

Notice how the proof rules enforce the side-conditions of the augmented semantics. Hence, all we have to do to prove that a program c terminates is to prove that every thread eventually discharges all its obligations. That is, we have to prove $\vdash\ \{\mathsf{obs}(\emptyset)\}\ c\ \{\mathsf{obs}(\emptyset)\}$. Figure 3a illustrates how we can apply our rules to verify that our minimal example terminates.

2.2 Non-Thread-Safe Physical Signals

As a step towards supporting waiting for arbitrary conditions over shared data structures, including non-thread-safe ones, we now move to non-thread-safe signals. For simplicity, in this paper we consider programs that use mutexes to synchronize concurrent accesses to shared data structures. (Our ideas apply equally to programs that use other constructs, such as atomic machine instructions.) Figure 4 presents our updated example.

[2] For a formal definition see this paper's extended version [28] and the technical report [29].

```
{obs(∅)}
let sig := new_signal in                       PR-NewSignal' with $L = 0$
{obs({[(sig, 0)]}) ∗ signal((sig, 0))}         $s :=$ (sig, 0)
fork ({obs(∅) ∗ signal($s$)}
        await is_set(sig)                      $s$.lev $= 0 \prec_L \emptyset$
      {obs(∅) ∗ signal($s$)});
{obs({[$s$]})}
set_signal(sig)
{obs(∅)}
```

(a) Proof outline for program from Fig. 1. Applied proof rule marked in purple. Abbreviation marked in brown. General hint marked in red.

PR-NewSignal'

$$\frac{L \in \mathcal{L}evs}{\vdash \{\mathsf{obs}(O)\}\ \textbf{new\_signal}\ \{\lambda r.\, \mathsf{obs}(O \uplus \{\!\![(r, L)]\!\!\}) * \mathsf{signal}((r, L))\}}$$

PR-SetSignal'

$$\vdash \{\mathsf{obs}(O \uplus \{\!\![s]\!\!\})\}\ \textbf{set\_signal}(s.\mathsf{id})\ \{\mathsf{obs}(O)\}$$

PR-Fork'

$$\frac{\vdash \{\mathsf{obs}(O_f) * A\}\ c\ \{\mathsf{obs}(\emptyset) * B\}}{\vdash \{\mathsf{obs}(O_m \uplus O_f) * A\}\ \textbf{fork}\ c\ \{\mathsf{obs}(O_m)\}}$$

PR-Await'

$$\frac{s.\mathsf{lev} \prec_{\mathsf{L}} O}{\vdash \{\mathsf{obs}(O) * \mathsf{signal}(s)\}\ \textbf{await is\_set}(s.\mathsf{id})\ \{\mathsf{obs}(O) * \mathsf{signal}(s)\}}$$

PR-Let

$$\frac{\vdash \{A\}\ c\ \{\lambda r.\, C(r)\} \qquad \forall v.\ \vdash \{C(v)\}\ c'[v/x]\ \{B\}}{\vdash \{A\}\ \textbf{let}\ x := c\ \textbf{in}\ c'\ \{B\}}$$

(b) Proof rules. Rules only used in this section marked with '.

Fig. 3. Verifying termination of minimal example with physical thread-safe signal. (Color figure online)

```
let sig := new_signal in
let mut := new_mutex in
fork with mut await is_set(sig);
acquire mut;
set_signal(sig);
release mut
```

(a) Code.

```
with mut await $c$ := (while acquire mut;
                             let $r$ := $c$ in
                             release mut;
                             $\neg r$
                         do skip)
```

(b) Syntactic sugar. r not free in mut.

Fig. 4. Minimal example with two threads communicating via a physical non-thread-safe signal protected by a mutex.

As signal sig is no longer thread-safe, the two threads can no longer use it directly to communicate. Instead, we have to synchronize accesses to avoid data races. Hence, we protect the signal by a mutex mut created by the main thread. In each iteration, the forked thread acquires the mutex, checks whether sig has been set and releases it again. After forking, the main thread acquires the mutex, sets the signal and releases it again.

Exposing Signal Values. Signals are specially marked heap cells storing boolean values. We make this explicit by extending our signal chunks from $\mathsf{signal}(s)$ to $\mathsf{signal}(s, b)$ where b is the current value of s and by updating our proof rules accordingly. Upon creation, signals are unset. Hence, creating a signal sig now spawns an *unset* signal chunk $\mathsf{signal}((\mathsf{sig}, L), \mathsf{False})$ for some freely chosen level L and an obligation for (sig, L), cf. PR-NewSignal”. We present our new proof rules in Fig. 6 and demonstrate their application in Fig. 5.

```
$\{\mathsf{obs}(\emptyset)\}$
let sig := new_signal in                                        PR-NewSignal” with $L = 1$
$\{\mathsf{obs}(\{\!\{(\mathsf{sig},1)\}\!\}) * \mathsf{signal}((\mathsf{sig},1),\mathsf{False})\}$            PR-ViewShift & VS-SemImp
$\{\mathsf{obs}(\{\!\{(\mathsf{sig},1)\}\!\}) * \exists b.\ \mathsf{signal}((\mathsf{sig},1),b)\}$             $s := (\mathsf{sig},1),\ \ P := \exists b.\ \mathsf{signal}(s,b)$
let mut := new_mutex in                                         PR-NewMutex” with $L = 0$
$\{\mathsf{obs}(\{\!\{s\}\!\}) * \mathsf{mutex}(m,P)\}$                                     PR-ViewShift
$\{\mathsf{obs}(\{\!\{s\}\!\}) * \mathsf{mutex}(m,P) * \mathsf{mutex}(m,P)\}$                       & VS-CloneMut”
fork ($\{\mathsf{obs}(\emptyset) * \mathsf{mutex}(m,P)\}$
        with $m$ await                                          $m.\mathsf{lev}, s.\mathsf{lev} \prec_{\mathsf{L}} \emptyset$
            $\{\mathsf{obs}(\{\!\{m\}\!\}) * P\}$                                      PR-Exists
            $\forall b.\ \{\mathsf{obs}(\{\!\{m\}\!\}) * \mathsf{signal}(s,b)\}$
                is_set(sig)
                $\{\lambda r.\, \mathsf{obs}(\{\!\{m\}\!\}) * \mathsf{signal}(s,b) \wedge r = b\}$         PR-ViewShift & VS-SemImp
                $\{\lambda r.\, \mathsf{obs}(\{\!\{m\}\!\})$
                     $* \text{ if } r \text{ then } P \text{ else } \mathsf{signal}(s,\mathsf{False})\}$
        $\{\mathsf{obs}(\emptyset) * \mathsf{mutex}(m,P)\}$                                PR-ViewShift & VS-SemImp
        $\{\mathsf{obs}(\emptyset)\}$);
$\{\mathsf{obs}(\{\!\{s\}\!\}) * \mathsf{mutex}(m,P)\}$
acquire mut;                                                    $m.\mathsf{lev} = 0 < 1 = s.\mathsf{lev}$
$\{\mathsf{obs}(\{\!\{s,m\}\!\}) * \mathsf{locked}(m,P) * \exists b.\ \mathsf{signal}(s,b)\}$          PR-Exists
$\forall b.\ \{\mathsf{obs}(\{\!\{s,m\}\!\}) * \mathsf{locked}(m,P) * \mathsf{signal}(s,b)\}$
    set_signal(sig);
    $\{\mathsf{obs}(\{\!\{m\}\!\}) * \mathsf{locked}(m,P) * \mathsf{signal}(s,\mathsf{True})\}$           PR-ViewShift & VS-SemImp
    $\{\mathsf{obs}(\{\!\{m\}\!\}) * \mathsf{locked}(m,P) * P\}$
    release mut
    $\{\mathsf{obs}(\emptyset) * \mathsf{mutex}(m,P)\}$                                    PR-ViewShift & VS-SemImp
    $\{\mathsf{obs}(\emptyset)\}$
```

Fig. 5. Proof outline for program Fig. 4, verifying termination with mutexes & non-thread safe signals. Applied proof and view shift rules marked in purple. Abbreviations marked in brown. General hints marked in red. (Color figure online)

PR-NewSignal”

$$\frac{L \in \mathcal{L}evs}{\vdash \{\mathsf{obs}(O)\}\ \mathbf{new\_signal}\ \{\lambda id.\, \mathsf{obs}(O \uplus \{\!\{(id, L)\}\!\}) * \mathsf{signal}((id, L), \mathsf{False})\}}$$

PR-SetSignal”

$$\vdash \{\mathsf{obs}(O \uplus \{\!\{s\}\!\}) * \mathsf{signal}(s, \_)\}\ \mathbf{set\_signal}(s.\mathsf{id})\ \{\mathsf{obs}(O) * \mathsf{signal}(s, \mathsf{True})\}$$

PR-IsSignalSet”

$$\vdash \{\mathsf{signal}(s, b)\}\ \mathbf{is\_set}(s.\mathsf{id})\ \{\lambda r.\, \mathsf{signal}(s, b) \wedge r = b\}$$

PR-Await”

$$\frac{\begin{array}{c} m.\mathsf{lev}, s.\mathsf{lev} \prec_{\mathsf{L}} O \qquad \mathsf{signal}(s, \mathsf{False}) * R \Rrightarrow P \\ \vdash \{\mathsf{obs}(O \uplus \{\!\{m\}\!\}) * P\}\ c\ \{\lambda r.\, \mathsf{obs}(O \uplus \{\!\{m\}\!\}) * \mathsf{if}\ r\ \mathsf{then}\ P\ \mathsf{else}\ \mathsf{signal}(s, \mathsf{False}) * R\} \end{array}}{\vdash \{\mathsf{obs}(O) * \mathsf{mutex}(m, P)\}\ \mathbf{with}\ m.\mathsf{loc}\ \mathbf{await}\ c\ \{\mathsf{obs}(O) * \mathsf{mutex}(m, P)\}}$$

(a) Signals & busy waiting.

PR-NewMutex”

$$\frac{L \in \mathcal{L}evs}{\vdash \{P\}\ \mathbf{new\_mutex}\ \{\lambda \ell.\, \mathsf{mutex}((\ell, L), P)\}}$$

PR-Acquire”

$$\begin{array}{ll} & \{\mathsf{obs}(O) * \mathsf{mutex}(m, P) \wedge m.\mathsf{lev} \prec_{\mathsf{L}} O\} \\ \vdash & \mathbf{acquire}\ m.\mathsf{loc} \\ & \{\mathsf{obs}(O \uplus \{\!\{m\}\!\}) * \mathsf{locked}(m, P) * P\} \end{array}$$

PR-Release”

$$\begin{array}{ll} & \{\mathsf{obs}(O \uplus \{\!\{m\}\!\}) * \mathsf{locked}(m, P) * P\} \\ \vdash & \mathbf{release}\ m.\mathsf{loc} \\ & \{\mathsf{obs}(O) * \mathsf{mutex}(m, P)\} \end{array}$$

(b) Mutexes.

PR-Frame

$$\frac{\vdash \{A\}\ c\ \{B\}}{\vdash \{A * F\}\ c\ \{B * F\}}$$

PR-Exists

$$\frac{\forall a \in A.\ \vdash \{a\}\ c\ \{B\}}{\vdash \{\bigvee A\}\ c\ \{B\}}$$

PR-Fork

$$\frac{\vdash \{\mathsf{obs}(O_f) * A\}\ c\ \{\mathsf{obs}(\emptyset)\}}{\vdash \{\mathsf{obs}(O_m \uplus O_f) * A\}\ \mathbf{fork}\ c\ \{\mathsf{obs}(O_m)\}}$$

PR-ViewShift

$$\frac{A \Rrightarrow A' \qquad \vdash \{A'\}\ c\ \{B'\} \qquad B' \Rrightarrow B}{\vdash \{A\}\ c\ \{B\}}$$

(c) Standard rules.

VS-SemImp

$$\frac{\forall H.\ \mathsf{consistent_{lh}}(H) \wedge H \vDash_{\mathsf{A}} A \Rightarrow H \vDash_{\mathsf{A}} B}{A \Rrightarrow B}$$

VS-Trans

$$\frac{A \Rrightarrow C \qquad C \Rrightarrow B}{A \Rrightarrow B}$$

VS-CloneMut”

$$\mathsf{mutex}(m, P) \Rrightarrow \mathsf{mutex}(m, P) * \mathsf{mutex}(m, P)$$

(d) View shifts.

Fig. 6. Proof rules and view shift rules for mutexes and non-thread safe signals. Rules only used in this section marked with ”.

Data Races. As read and write operations on signals are no longer thread-safe, our logic has to ensure that two threads never try to access sig at the same time. Hence, in our logic possession of a signal chunk $\mathsf{signal}(s, b)$ expresses (temporary) *exclusive ownership* of s. Further, our logic requires threads to own any signal they are trying to access. Specifically, when a thread wants to set sig, it must hold a chunk of the form $\mathsf{signal}((\mathsf{sig}, L), b)$, cf. PR-SetSignal”. The same holds for reading a signal's value, cf. PR-IsSignalSet”. Note that signal chunks are not duplicable and only created upon creation of the signal they refer to. Therefore, holding a signal chunk for sig indeed guarantees that the holding thread has the exclusive right to access sig (while holding the signal chunk).

Synchronization and Lock Invariants. After the main thread creates sig, it exclusively owns the signal. The main thread can transfer ownership of this resource during forking, cf. PR-Fork', and thereby allow the forked thread to busy-wait for sig. This would, however, leave the main thread without any permission to set the signal and thereby discharge its obligation.

We use mutexes to let multiple threads share ownership of a common set of resources in a synchronized fashion. Every mutex is associated with a *lock invariant* P, an assertion chosen by the proof author that specifies which resources the mutex protects. In our example, we want both threads to share sig. To reflect the fact that the signal's value changes over time, we choose a lock invariant that abstracts over its concrete value. We choose $P := \exists b.\ \mathsf{signal}((\mathsf{sig}, L), b)$. Let us ignore the chosen signal level L for now. Creating the mutex mut consumes this lock invariant and binds it to mut by creating a mutex chunk $\mathsf{mutex}((\mathsf{mut}, \dots), P)$, cf. PR-NewMutex”. Thereby, the main thread loses access to sig. The only way to regain access is by acquiring mut, cf. PR-Acquire”. Once the thread releases mut, it again loses access to all resources protected by the mutex, cf. PR-Release”.

Deadlocks. We have to ensure that any acquired mutex is eventually released, again. Hence, acquiring a mutex spawns a release obligation for this mutex and the only way to discharge this obligation is indeed by releasing it, cf. PR-Acquire” and PR-Release”.

Any attempt to acquire a mutex will block until the mutex becomes available. In order to prove that our program terminates, we have to prove that it does not get stuck during an acquisition attempt. To prevent wait cycles involving mutexes, we require the proof author to associate every mutex as well (just like signals) with a level L. This level can be freely chosen during the mutex' creation, cf. PR-NewMutex”. Mutex chunks therefore have the form $\mathsf{mutex}((\ell, L), P)$ where ℓ is the heap location the mutex is stored at. Their only purpose is to record the level and lock invariant a mutex is associated with. Hence, these chunks can be freely duplicated as we will see later. Generally, we denote mutex tuples by $m = (\ell, L)$. We only allow to acquire a mutex if its level is lower than the level of each held obligation, cf. PR-Acquire”. This also prevents any thread from attempting to acquire mutexes twice, e.g., **acquire** mut; **acquire** mut or **with** mut **await acquire** mut.

View Shifts. When verifying a program, it can be necessary to reformulate the proof state and to draw semantic conclusions. To allow this we introduce a so-called *view shift* relation $\Rrightarrow$ [14]. By applying proof rule PR-ViewShift and VS-SemImp we can strengthen the precondition and weaken the postcondition. In our example, we use this to convert the unset signal chunk into the lock invariant which abstracts over the signal's value, i.e., $\mathsf{signal}(s, \mathsf{False}) \Rrightarrow \exists b.\ \mathsf{signal}(s, b)$.

The logic we present in this work is an intuitionistic separation logic that allows us to drop chunks.[3] This allows us to simplify the postcondition of our fork proof rule's premise from $\mathsf{obs}(\emptyset) * B$ to $\mathsf{obs}(\emptyset)$, cf. PR-Fork, and drop all unneeded chunks via a semantic implication $\mathsf{obs}(\emptyset) * B \Rrightarrow \mathsf{obs}(\emptyset)$.

We also allow to clone mutex chunks via view shifts, cf. VS-CloneMut". In our example, this is necessary to inform both threads which level and lock invariant mutex mut is associated with. That is, the main thread clones the mutex chunk $\mathsf{mutex}(m, P)$ and passes one chunk on when it forks the busy-waiting thread.

In Sect. 2.4 we extend our view shift relation and revisit our interpretation of what a view shift expresses. The full set of rules we use to define $\Rrightarrow$ is presented in the extended version of this paper [28].

Busy Waiting. In the approach presented in this paper, for simplicity we only support busy-waiting loops of the form **with** mut **await** c, which is syntactic sugar for **while acquire** mut; **let** $r := c$ **in release** $\mathsf{mut}; \neg r$ **do skip** where r denotes a fresh variable.[4] In each iteration, the loop tries to acquire mut, executes c, releases mut again and lets the result returned by c determine whether the loop continues. Such loops can fail to terminate for two reasons: (i) Acquiring mut can get stuck and (ii) the loop could diverge.

We prevent the loop from getting stuck by requiring mut's level to be lower than the level of each held obligation, cf. PR-Await". Further, we enforce termination by requiring the loop to wait for a signal. That is, when verifying a busy-waiting loop using our approach, the proof author must choose a fixed signal and prove that this signal remains unset at the end of every non-finishing iteration. This way, we can prove that the loop terminates by proving that every signal is eventually set, just as in Sect. 2.1. And just as before, our logic requires the level of the waited-for signal to be lower than the level of each held obligation.

Acquiring the mutex in every iteration makes the lock invariant available during the verification of the loop body c. This lock invariant has to be restored at the end of the iteration such that it can be consumed during the mutex's release. PR-Await" allows for an additional view shift to restore the invariant. In our example, we end our busy-waiting loop's non-finishing iterations with the assertion $\mathsf{signal}(s, \mathsf{False})$. We use a semantic implication view shift to convert the signal chunk into the mutex invariant $\exists b.\ \mathsf{signal}(s, b)$.

[3] This allows a thread to drop its obligations chunk $\mathsf{obs}(O)$. Note, however, that by dropping this chunk the thread does not drop its obligations, but only its ability to show what its obligations are. In particular the thread would be unable to present an empty obligations chunk upon termination.

[4] As we discuss in Sect. 5, in the technical report accompanying this paper we present a more general logic that imposes no such syntactic restrictions.

Choosing Levels. In our example, we have to assign levels to the mutex mut and to the signal sig. Our proof rules for mutex acquisition and busy waiting impose some restrictions on the levels of the involved mutexes and signals. By analysing the corresponding rule applications that occur in our proof, we can derive which constraints our level choice must comply with. Our example's verification involves one application of PR-ACQUIRE" and one application of PR-AWAIT": (i) Our main thread tries to acquire mut while holding an obligation to set sig. (ii) The forked thread busy-waits for sig while not holding any obligations. Our assignment of levels must therefore satisfy the single constraint $m.\mathsf{lev} <_{\mathsf{L}} s.\mathsf{lev}$. So, we choose $\mathcal{L}evs = \{0, 1\}$, $m.\mathsf{lev} = 0$ and $s.\mathsf{lev} = 1$.

2.3 Arbitrary Data Structures

The proof rules we introduced in Sect. 2.2 allow us to verify programs busy-waiting for arbitrary conditions over arbitrary shared data structures as follows: For every condition C the program waits for, the proof author inserts a signal s into the program. They ensure that s is set at the same time the program establishes C and prove an invariant stating that the signal's value expresses whether C holds. Then, the waiting thread can use s to wait for C. We illustrate this here for the simplest case of setting a single heap cell in Fig. 7a.

```
let x := cons(0) in
let mut := new_mutex in
fork with mut await [x] = 1;
acquire mut;
[x] := 1;
release mut
```

(a) Example program with busy waiting for heap cell x to be set.

```
let x := cons(0) in
let sig := new_signal in
let mut := new_mutex in
fork with mut await [x] = 1;
acquire mut;
[x] := 1;
set_signal(sig);
release mut
```

(b) Example program 7a with additional signal sig inserted, marked in green. sig and x are kept in sync.

$$[e] = e' \quad := \quad (\textbf{let } r := [e] \textbf{ in } r = e')$$

(c) Syntactic sugar. r free in e'.

Fig. 7. Minimal example illustrating busy waiting for condition over heap cell. (Color figure online)

The program involves three new non-thread-safe commands: (i) **cons**(v) for allocating a new heap cell and initializing it with value v, (ii) $[\ell] := v$ for assigning value v to heap location ℓ, (iii) $[\ell]$ for reading the value stored in heap location ℓ. We use $[\ell] = v$ as syntactic sugar for **let** $r := [e]$ **in** $r = e'$.

In our example, the main thread allocates x, initializes it with the value 0 and protects it using mutex mut. It forks a new thread busy-waiting for x to be set. Afterwards, the main thread sets x. As explained above, we verify the program by inserting a signal sig that reflects whether x has been set, yet. Figure 7b presents the resulting code. The main thread creates the signal and sets it when it sets x.

```
{obs(∅)}
let x := cons(0) in
{obs(∅) ∗ x ↦ 0}
let sig := new_signal in                                  PR-NewSignal" with L = 1
let mut := new_mutex in                                   PR-NewMutex" with L = 0
s := (sig, 1),  m := (mut, 0)
P := ∃v. x ↦ v ∗ signal(s, v = 1)
{obs({[s]}) ∗ mutex(m, P) ∗ mutex(m, P)}
fork ({obs(∅) ∗ mutex(m, P)}
        with m await                                      m.lev, s.lev ≺_L ∅
           {obs({[m]}) ∗ P}
           ∀v. {obs({[m]}) ∗ x ↦ v ∗ signal(s, v = 1)}
              [x] = 1
              { λr. obs({[m]})
                   ∗ if r then P
                     else x ↦ v ∧ v ≠ 1 ∗ signal(s, False) }
      {obs(∅)});
{obs({[s]}) ∗ mutex(m, P)}
acquire mut;                                              m.lev = 0 < 1 = s.lev
∀v. {obs({[s, m]}) ∗ locked(m, P) ∗ x ↦ v ∗ signal(s, v = 1)}
    [x] := 1;
    {obs({[s, m]}) ∗ locked(m, P) ∗ x ↦ 1 ∗ signal(s, v = 1)}
    set_signal(sig);
    {obs({[m]}) ∗ locked(m, P) ∗ x ↦ 1 ∗ signal(s, True)}
    release mut
    {obs(∅)}
```

(a) Proof outline for program 7b. Applied proof rules marked in purple. Abbreviations marked in brown. General hints marked in red.

PR-Cons
$\vdash \{\mathsf{True}\}\ \mathbf{cons}(v)\ \{\lambda\ell.\, \ell \mapsto v\}$

PR-AssignToHeap
$\vdash \{\ell \mapsto \_\}\ [\ell] := v\ \{\ell \mapsto v\}$

PR-ReadHeapLoc"
$\vdash \{\ell \mapsto v\}\ [\ell]\ \{\lambda r.\, r = v * \ell \mapsto v\}$

PR-Exp
$$\frac{[\![e]\!] \in \mathit{Values}}{\vdash \{\mathsf{True}\}\ e\ \{\lambda r.\, r = [\![e]\!]\}}$$

(b) Proof rules. Evaluation function $[\![\cdot]\!]$. Rules only used in this section marked with ".

Fig. 8. Verifying termination of busy waiting for condition over heap cell. (Color figure online)

Heap Cells. Verifying this example does not conceptually differ from the example we presented in Sect. 2.2. Figure 8b presents the new proof rules we need and Fig. 8a sketches our example's verification. As with non-thread-safe signals, we have to prevent multiple threads from trying to access x at the same time in order to prevent data races. For this we use so-called *points-to* chunks [24,31]. They have the form $\ell \mapsto v$ and express that heap location ℓ stores the value v. When a thread holds such a chunk, it exclusively owns the right to access heap location ℓ.

Heap locations are unique and the only way to create a new points-to chunk is to allocate and initialize a new heap cell via **cons**(v), cf. PR-Cons. Hence, there will never be two points-to chunks involving the same heap location. In order to read or write a heap cell via $[\ell]$ or $[\ell] := e$, the acting thread must first acquire possession of the corresponding points-to chunk, cf. PR-AssignToHeap and PR-ReadHeapLoc” ’.

Relating Signals to Conditions. In our example, the forked thread busy-waits for x to be set while our proof rules require us to justify each iteration by showing an unset signal. That is, we must prove an invariant stating that the value of x matches sig. As this invariant must be shared between both threads, we encode it in the lock invariant: $P := \exists v.\ \mathsf{x} \mapsto v * \mathsf{signal}(s, v = 1)$. This does not only allow both threads to share the heap cell and the signal but it also automatically enforces that they maintain the invariant whenever they acquire and release the mutex.

2.4 Signal Erasure

In the program from Fig. 7b signal sig is never read and does hence not influence the waiting thread’s runtime behaviour. Therefore, we can verify the original program presented in Fig. 7a by erasing the physical signal and treating it as ghost code.

Ghost Signals. Central aspects of the proof sketch we presented in Fig. 8a are that (i) the main thread was obliged to set sig and that (ii) the value of sig reflected whether x was already set. *Ghost signals* allow us to keep this information but at the same to remove the physical signals from the code. Ghost signals are essentially identical to the physical non-thread-safe signals we used so far. However, as ghost resources they cannot influence the program’s runtime behaviour. They merely carry information we can use during the verification process.

View Shifts Revisited. We implement ghost signals by extending our view shift relation. In particular, we introduce two new view shift rules: VS-NewSignal and VS-SetSignal presented in Fig. 9b. The former creates a new unset signal and simultaneously spawns an obligation to set it. The latter can be used to set a signal and thereby discharge a corresponding obligation. We say that these rules change the *ghost state* and therefore call their application a *ghost proof step*. With this extension, a view shift $A \Rrightarrow B$ expresses that we can reach postcondition B from precondition A by (i) drawing semantic conclusions or by (ii) manipulating the ghost state. In Fig. 9a we use ghost signals to verify the program from Fig. 7a.

Note that lifting signals to the verification level does not affect the soundness of our approach. The argument we presented in Sect. 2.1 still holds. We formalize our logic and provide a formal soundness proof in the extended version of this paper [28] and in the technical report [29]. The latter contains a more general version of the presented logic that (i) is not restricted to busy-waiting loops of the form **with** mut **await** c and that (ii) is easier to integrate into existing tools like VeriFast [12], as explained in Sect. 5.

```
$\{\mathsf{obs}(\emptyset)\}$
let x := cons(0) in
$\{\mathsf{obs}(\emptyset) * \mathsf{x} \mapsto 0\}$
new_ghost_signal;                                                  VS-NewSignal with $L = 1$.
$\{\exists \mathsf{sig}.\ \mathsf{obs}(\{\!\{(\mathsf{sig},1)\}\!\}) * \mathsf{x} \mapsto 0 * \mathsf{signal}((\mathsf{sig},1),\mathsf{False})\}$    $s := (\mathsf{sig}, 1)$
$\forall \mathsf{sig}.\ \{\mathsf{obs}(\{\!\{s\}\!\}) * \mathsf{x} \mapsto 0 * \mathsf{signal}(s,\mathsf{False})\}$             $P := \exists v.\ \mathsf{x} \mapsto v * \mathsf{signal}(s, v = 1)$
    let mut := new_mutex in                                        PR-NewMutex" with $L = 0$
    $\{\mathsf{obs}(\{\!\{s\}\!\}) * \mathsf{mutex}((\mathsf{mut},0),P)$
    $* \mathsf{mutex}((\mathsf{mut},0),P)\}$                                       $m := (\mathsf{mut}, 0)$
    fork ($\{\mathsf{obs}(\emptyset) * \mathsf{mutex}(m,P)\}$
        with $m$ await                                             $m.\mathsf{lev}, s.\mathsf{lev} \prec_{\mathsf{L}} \emptyset$
          $\{\mathsf{obs}(\{\!\{m\}\!\}) * P\}$
          $\forall v.\ \{\mathsf{obs}(\{\!\{m\}\!\}) * \mathsf{x} \mapsto v * \mathsf{signal}(s, v = 1)\}$
              [x] = 1
              $\{\lambda r.\, \mathsf{obs}(\{\!\{m\}\!\}) *$
                  if $r$ then $P$
                  else $\mathsf{x} \mapsto v \wedge v \neq 1 * \mathsf{signal}(s,\mathsf{False})\}$
        $\{\mathsf{obs}(\emptyset)\}$);
    $\{\mathsf{obs}(\{\!\{s\}\!\}) * \mathsf{mutex}(m,P)\}$
    acquire mut;                                                   $m.\mathsf{lev} = 0 < 1 = s.\mathsf{lev}$
    $\forall v.$ $\{\mathsf{obs}(\{\!\{s, m\}\!\}) * \mathsf{locked}(m,P)$
         $* \mathsf{x} \mapsto v * \mathsf{signal}(s, v = 1)\}$
       [x] := 1;
       set_ghost_signal($s$);
       $\{\mathsf{obs}(\{\!\{m\}\!\}) * \mathsf{locked}(m,P)$
       $* \mathsf{x} \mapsto 1 * \mathsf{signal}(s,\mathsf{True})\}$
       release mut
       $\{\mathsf{obs}(\emptyset)\}$
```

(a) Proof outline for the program presented in Fig. 7a. Auxiliary commands hinting at view shifts and general hints marked in red. Applied proof and view shift rules marked in purple. Abbreviations marked in brown.

VS-NewSignal

$$\frac{L \in \mathcal{L}evs}{\mathsf{obs}(O) \Rrightarrow \exists id.\ \mathsf{obs}(O \uplus \{\!\{(id, L)\}\!\}) * \mathsf{signal}((id, L), \mathsf{False})}$$

VS-SetSignal

$$\mathsf{obs}(O \uplus \{\!\{s\}\!\}) * \mathsf{signal}(s, \_) \Rrightarrow \mathsf{obs}(O) * \mathsf{signal}(s, \mathsf{True})$$

(b) Proof rules.

Fig. 9. Verifying termination with ghost signals. (Color figure online)

3 A Realistic Example

To demonstrate the expressiveness of the presented verification approach, we verified the termination of the program presented in Fig. 10a. It involves two threads, a consumer and a producer, communicating via a shared bounded FIFO with a maximal capacity of 10. The producer enqueues numbers 100, . . . , 1 into the FIFO and the consumer dequeues those. Whenever the queue is full, the producer busy-waits for the consumer to dequeue an element. Likewise, whenever the queue is empty, the consumer busy-waits for the producer to enqueue the next element. Each thread's finishing depends on the other thread's productivity. This is, however, no cyclic dependency. For instance, in order to prove that the producer eventually pushes number i into the queue, we only need to rely on the consumer to pop $i + 10$. A similar property holds for the consumer.

```
$\text{alloc\_ghost\_signal\_IDs}(id^i_{\text{pop}}, id^i_{\text{push}})$   for  $1 \le i \le 100$;
$L^i_{\text{pop}} := 102 - i$,  $L^i_{\text{push}} := 101 - i$,  $s^i_x := (id^i_x, L^i_x)$  for  $1 \le i \le 100$
$\text{init\_ghost\_signals}(s^{100}_{\text{pop}}, s^{100}_{\text{push}})$;
$\{\mathsf{obs}(\{\!\{s^{100}_{\text{pop}}, s^{100}_{\text{push}}\}\!\}) * \ldots\}$
let $\mathsf{fifo}_{10}$ := cons(nil) in let mut := new_mutex in
let $\mathsf{c_p}$ := cons(100) in let $\mathsf{c_c}$ := cons(100) in
fork (while (                                              $\mathsf{c_p}$ decreases in each iteration.
        with mut await (                                   Busy-wait for $\mathsf{fifo}_{10}$ not being full.
          $\{\mathsf{obs}(\{\!\{s^{\mathsf{c_p}}_{\text{push}}, (\mathsf{mut}, 0)\}\!\}) * \ldots\}$            $\rightarrow$ Wait for consumer to pop.
          let f := [$\mathsf{fifo}_{10}$] in
          if size(f) < 10 then (                           If $\mathsf{fifo}_{10}$ not full, push next element.
            let c := [$\mathsf{c_p}$] in [$\mathsf{fifo}_{10}$] := $\mathsf{f} \cdot \langle \mathsf{c} \rangle$;  [$\mathsf{c_p}$] := c − 1;
            $\text{set\_ghost\_signal}(s^{\mathsf{c}}_{\text{push}})$;
            if c − 1 ≠ 0 then $\text{init\_ghost\_signal}(s^{\mathsf{c}-1}_{\text{push}})$);
          size(f) ≠ 10);                                   if size(f) = 10 then wait for $s^{\mathsf{c_p}+10}_{\text{pop}}$
        [$\mathsf{c_p}$] ≠ 0)                                        $L^{\mathsf{c_p}+10}_{\text{pop}} = 92 - \mathsf{c_p} < 101 - \mathsf{c_p} = L^{\mathsf{c_p}}_{\text{push}}$
      do skip);
while (                                                    $\mathsf{c_c}$ decreases in each iteration.
  with mut await (                                         Busy-wait for $\mathsf{fifo}_{10}$ not being empty.
    $\{\mathsf{obs}(\{\!\{s^{\mathsf{c_c}}_{\text{pop}}, (\mathsf{mut}, 0)\}\!\}) * \ldots\}$                    $\rightarrow$ Wait for producer to push.
    let f := [$\mathsf{fifo}_{10}$] in
    if size(f) > 0 then (                                  If $\mathsf{fifo}_{10}$ not empty, pop next element.
      let c := [$\mathsf{c_c}$] in [$\mathsf{fifo}_{10}$] := tail(f);  [$\mathsf{c_c}$] := c − 1;
      $\text{set\_ghost\_signal}(s^{\mathsf{c}}_{\text{pop}})$;
      if c − 1 ≠ 0 then $\text{init\_ghost\_signal}(s^{\mathsf{c}-1}_{\text{pop}})$);
    size(f) > 0);                                          if size(f) = 0 then wait for $s^{\mathsf{c_c}}_{\text{push}}$
  [$\mathsf{c_c}$] ≠ 0)                                              $L^{\mathsf{c_c}}_{\text{push}} = 101 - \mathsf{c_c} < 102 - \mathsf{c_c} = L^{\mathsf{c_c}}_{\text{push}}$
do skip);
```

(a) Example program with two threads communicating via a shared bounded FIFO with maximal size 10. Auxiliary commands hinting at view shifts and general hints marked in red. Abbreviations marked in brown. Hints on proof state marked in blue.

VS-AllocSigID
$\mathsf{True} \Rrightarrow \exists id.\ \mathsf{uninitSig}(id)$

VS-SigInit
$\mathsf{obs}(O) * \mathsf{uninitSig}(id)$
$\Rrightarrow \mathsf{obs}(O \uplus \{\!\{(id, L)\}\!\}) * \mathsf{signal}((id, L), \mathsf{False})$

(b) Fine-grained view shift rules for signal creation.

Fig. 10. Realistic example program. (Color figure online)

Fine-Tuning Signal Creation. To simplify complex proofs involving many signals we refine the process of creating a new ghost signal. For simplicity, we combined the allocation of a new signal ID and its association with a level and a boolean in one step. For some proofs, such as the one we outline in this section, it can be helpful to fix the IDs of all signals that will be created throughout the proof already at the beginning. To realize this, we replace view shift rule VS-NewSignal by the rules presented in Fig. 10b and adapt our signal chunks accordingly. With these more fine-grained view shifts, we start by allocating a signal ID, cf. VS-AllocSigID. Thereby we obtain an *uninitialized* signal $\mathsf{uninitSig}(id)$ that is not associated with any level or boolean, yet. Also, allocating a signal ID does not create any obligation because threads can only wait for *initialized* (and unset) signals. When we initialize a signal, we bind its already allocated ID to a level of our choice and associate the signal with False, cf. VS-SigInit. This creates an obligation to set the signal.

Loops and Signals. In our program, both threads have a local counter initially set to 100 and run a nested loop. The outer loops are controlled by their thread's counter, which is decreased in each iteration until it reaches 0 and the loop stops. For such loops, we introduce a conventional proof rule for total correctness of loops, cf. this paper's extended version [28]. Verifying termination of the inner loops is a bit more tricky and requires the use of ghost signals.

So far, we had to fix a single signal for the verification of every **await** loop. We can relax this restriction to considering a finite set of signals the loop may wait for, cf. PR-Await presented in [28]. Apart from being a generalisation, this rule does not differ from PR-Await'' introduced in Sect. 2.2.

Initially, we allocate 200 signal IDs $id^{100}_{\text{push}}, \ldots, id^{1}_{\text{push}}, id^{100}_{\text{pop}}, \ldots, id^{1}_{\text{pop}}$. We are going to ensure that always at most one push signal and at most one pop signal are initialized and unset. The producer and consumer are going to hold the obligation for the push and pop signal, respectively. The producer will hold the obligation for s^i_{push} while i is the next number to be pushed into the FIFO and it will set s^i_{push} when it pushes the number i into the FIFO. Meanwhile, the consumer will use s^i_{push} to wait for the number i to arrive in the queue when it is empty. Similarly, the consumer will hold the obligation for s^i_{pop} while number i is the next number to be popped from the FIFO and will set s^i_{pop} when it pops the number i. The producer uses s^i_{pop} to wait for the consumer to pop i from the queue when it is full. At any time, we let the mutex mut protect the two active signals and thereby make them accessible to both threads.

Choosing the Levels. Note that we ignored the levels so far. The producer and the consumer both acquire the mutex while holding an obligation for a signal. Hence, we choose $\mathcal{L}evs = \mathbb{N}$, $\mathsf{m.lev} = 0$ and $s.\mathsf{lev} > 0$ for every signal s. Both threads will justify iterations of their respective **await** loop by using an unset signal at the end of such an iteration. Our proof rules allow us to ignore the mutex obligation during this step. Hence, the mutex level does not interfere with the level of the unset signal. Whenever the queue is full, the producer waits for the consumer to pop an element and whenever the queue is empty, the consumer waits for the producer to push. That is, the producer waits for s_{pop}^{i+10} while holding an obligation for s_{push}^{i} and the consumer waits for s_{push}^{i} while holding an obligation for s_{pop}^{i}. So, we have to choose the signal levels such that $s_{\text{pop}}^{i+10}.\mathsf{lev} < s_{\text{push}}^{i}.\mathsf{lev}$ and $s_{\text{push}}^{i}.\mathsf{lev} < s_{\text{pop}}^{i}.\mathsf{lev}$ hold. We solve this by choosing $s_{\text{pop}}^{i}.\mathsf{lev} = 102 - i$ and $s_{\text{push}}^{i}.\mathsf{lev} = 101 - i$.

Verifying Termination. This setup suffices to verify the example program. Via the lock invariant, each thread has access to both active signals. Whenever the producer pushes a number i into the queue, it sets s_{push}^{i} which discharges the held obligation and decreases its counter. Afterwards, if $i > 1$, it uses the uninitialized signal chunk $\mathsf{uninitSig}(id_{\text{push}}^{i-1})$ to initialize $s_{\text{push}}^{i-1} = (id_{\text{push}}^{i-1}, 101 - (i-1))$ and replaces s_{push}^{i} in the lock invariant by s_{push}^{i-1} before it releases the lock. If $i = 1$, the counter reached 0 and the loop ends. In this case, the producer holds no obligation. The consumer behaves similarly. Since we proved that each thread discharged all its obligations, we proved that the program terminates. Figure 10a illustrates the most important proof steps. We present the program's verification in full detail in the extended version of this paper [28] and in the technical report [29]. Furthermore, we encoded [27] the proof in VeriFast [12].

The number of threads in this program is fixed. However, our approach also supports the verification of programs where the number of threads is not even statically bounded. In [28] we present and verify such a program. It involves N producer and N consumer threads that communicate via a shared buffer of size 1, for a random number $N > 0$ determined during runtime.

4 Specifying Busy-Waiting Concurrent Objects

Our approach can be used to verify busy-waiting concurrent objects with respect to abstract specifications. For example, we have verified [26] the CLH lock [7] against a specification that is very similar to our proof rules for built-in mutexes shown in Fig. 6. The main difference is that it is slightly more abstract: when a lock is initialized, it is associated with a *bounded infinite set* of levels rather than with a single particular level. (To make this possible, an appropriate universe of levels should be used, such as the set of lists of natural numbers, ordered lexicographically.) To acquire a lock, the levels of the obligations held by the thread must be above the elements of the set; the new obligation's level is an element of the set.

5 Tool Support

We have extended the VeriFast tool [10] for separation logic-based modular verification of C and Java programs so that it supports verifying termination of busy-waiting C or Java programs. When verifying termination, VeriFast consumes a *call permission* at each recursive call or loop iteration. In the technical report [29] we define a generalised version of our logic that instead of providing a special proof rule for busy-waiting loops, provides *wait permissions* and a *wait view shift*. A call permission of a *degree* δ can be turned into a wait permission of a degree $\delta' < \delta$ for a given signal s. A wait view shift for an unset signal s for which a wait permission of degree δ exists produces a call permission of degree δ, which can be used to fuel a busy-waiting loop. When busy-waiting for some signal s, we can generate new permissions to justify each iteration as long as s remains unset.

VeriFast allows threads to freely exchange permissions. This is useful to verify termination of non-blocking algorithms involving compare-and-swap loops [11]. However, we must be careful to prevent self-fueling busy-waiting loops. Hence, we restrict where a permission can be consumed based on the *thread phase* it was created in. The main thread's initial phase is ϵ. When a thread in phase p forks a new thread, its phase changes to $p.\mathsf{Forker}$ and the new thread starts in phase $p.\mathsf{Forkee}$. We allow a thread in phase p to consume a permission only if it was produced in an *ancestor thread phase* $p' \sqsubseteq p$.

The only change we had to make to VeriFast's symbolic execution engine was to enforce the thread phase rule. We encoded the other aspects of the logic simply as axioms in a *trusted header file*. We used this tool support to verify the bounded FIFO (Sect. 3) and the CLH lock (Sect. 4). The bounded FIFO proof [27] contains 160 lines of proof annotations for 37 lines of code (an annotation overhead of 435%) and takes 0.08 s to verify. The CLH lock proof [26] contains 343 lines of annotations for 49 lines of code (an overhead of 700%) and takes 0.1 s to verify.

6 Integrating Higher-Order Features

The logic we presented in this paper does not support higher-order features such as assertions that quantify over assertions, or storing assertions in the (logical) heap as the values of ghost cells. While we did not need such features to carry out our example proofs, they are generally useful to verify higher-order program modules against abstract specifications. The typical way to support such features in a program logic is by applying *step indexing* [1,17], where the domain of logical heaps is indexed by the number of execution steps left in the (partial) program trace under consideration. Assertions stored in a logical heap at index $n+1$ talk about logical heaps at index n; i.e., they are meaningful only *later*, after at least one more execution step has been performed.

It follows that such logics apply directly only to *partial* correctness properties. Fortunately, we can reduce a termination property to a safety property by writing our program in a programming language *instrumented* with runtime checks that guarantee termination. Specifically, we can write our program in a programming language that fulfils the following criteria: It tracks signals, obligations and permissions at runtime and has constructs for signal creation, waiting and setting a signal. The **fork** command takes as an extra operand the list of obligations to be transferred to the new thread (and the other constructs similarly take sufficient operands to eliminate any need for angelic choice). Threads get stuck when these constructs' preconditions are not satisfied, such as when a thread waits for a signal while holding the obligation for that signal. We can then use a step-indexing-based higher-order logic such as Iris [14] to verify that no thread in our program ever gets stuck. Once we established this, we know none of the instrumentation has any effect and can be safely *erased* from the program.

7 Related and Future Work

In recent work [30] we propose a separation logic to verify termination of programs where threads busy-wait to be abruptly terminated. We generalize this work to support busy waiting for arbitrary conditions.

In [11] we propose an approach based on *call permissions* to verify termination of single- and multithreaded programs that involve loops and recursion. However, that work does not consider busy-waiting loops. In the technical report, we present a generalised logic that uses call permissions and allows busy waiting to be implemented using arbitrary looping and/or recursion. Furthermore, the use of call permissions allowed us to encode our case studies in our VeriFast tool which also uses call permissions for termination verification.

Liang and Feng [20,21] propose LiLi, a separation logic to verify liveness of blocking constructs implemented via busy waiting. In contrast to our verification approach, theirs is based on the idea of contextual refinement. In their approach, client code involving calls of blocking methods of the concurrent object is verified by first applying the contextual refinement result to replace these calls by code involving primitive blocking operations and then verifying the resulting client code using some other approach. In contrast, specifications in our approach are regular Hoare-style triples and proofs are regular Hoare-style proofs.

In [9] we propose a Hoare logic to verify liveness properties of the I/O behaviour of programs that do not perform busy waiting. By combining that approach with the one we proposed in this paper, we expect to be able to verify I/O liveness of realistic concurrent programs involving both I/O and busy waiting, such as a server where one thread receives requests and enqueues them into a bounded FIFO, and another one dequeues them and responds. To support this claim, we encoded the combined logic in VeriFast and verified a simple server application where the receiver and responder thread communicate via a shared buffer [25].

8 Conclusion

We propose what is to the best of our knowledge the first separation logic for verifying termination of programs with busy waiting. We offer a soundness proof of the system of the paper in its extended version [28], and of a more general system in the technical report [29]. Further, we demonstrated its usability by verifying a realistic example. We encoded our logic and the realistic example in VeriFast [27] and used this encoding also to verify the CLH lock [26]. Moreover, we expect that our approach can be integrated into other existing concurrent separation logics such as Iris [14].

References

1. Appel, A.W., McAllester, D.: An indexed model of recursive types for foundational proof-carrying code. ACM Trans. Program. Lang. Syst. **23**(5), 657–683 (2001). https://doi.org/10.1145/504709.504712
2. Boström, P., Müller, P.: Modular verification of finite blocking in non-terminating programs. In: Boyland, J.T. (ed.) 29th European Conference on Object-Oriented Programming, ECOOP 2015, 5–10 July 2015, Prague, Czech Republic. LIPIcs, vol. 37, pp. 639–663. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2015). https://doi.org/10.4230/LIPIcs.ECOOP.2015.639
3. Boyapati, C., Lee, R., Rinard, M.: Ownership types for safe programming: Preventing data races and deadlocks. OOPSLA (2002). https://doi.org/10.1145/582419.582440
4. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: PLDI 2002 (2002). https://doi.org/10.1145/512529.512558
5. Hamin, J., Jacobs, B.: Deadlock-free monitors. In: Ahmed, A. (ed.) ESOP 2018. LNCS, vol. 10801, pp. 415–441. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89884-1_15
6. Hamin, J., Jacobs, B.: Transferring Obligations Through Synchronizations. In: 33rd European Conference on Object-Oriented Programming (ECOOP 2019). Leibniz International Proceedings in Informatics (LIPIcs), vol. 134, pp. 19:1–19:58 (2019). https://doi.org/10.4230/LIPIcs.ECOOP.2019.19
7. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming, 1st edn. Revised Reprint. Morgan Kaufmann Publishers Inc., San Francisco (2012)
8. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**, 576–580 (1968). https://doi.org/10.1145/363235.363259
9. Jacobs, B.: Modular verification of liveness properties of the I/O behavior of imperative programs. In: Margaria, T., Steffen, B. (eds.) ISoLA 2020. LNCS, vol. 12476, pp. 509–524. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-61362-4_29
10. Jacobs, B. (ed.): VeriFast 21.04. Zenodo (2021). https://doi.org/10.5281/zenodo.4705416
11. Jacobs, B., Bosnacki, D., Kuiper, R.: Modular termination verification of single-threaded and multithreaded programs. ACM Trans. Program. Lang. Syst. **40**, 12:1–12:59 (2018). https://doi.org/10.1145/3210258
12. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 41–55. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_4
13. Jung, R., Krebbers, R., Birkedal, L., Dreyer, D.: Higher-order ghost state. In: Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (2016). https://doi.org/10.1145/2951913.2951943
14. Jung, R., Krebbers, R., Jourdan, J.H., Bizjak, A., Birkedal, L., Dreyer, D.: Iris from the ground up: a modular foundation for higher-order concurrent separation logic. J. Funct. Program. **28**, e20 (2018). https://doi.org/10.1017/S0956796818000151
15. Kim, J., Sjöberg, V., Gu, R., Shao, Z.: Safety and liveness of MCS lock-layer by layer. In: Asian Symposium on Programming Languages and Systems (2017)

16. Kobayashi, N.: A new type system for deadlock-free processes. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 233–247. Springer, Heidelberg (2006). https://doi.org/10.1007/11817949_16
17. Lars Birkedal, Kristian Støvring, J.T.: The category-theoretic solution of recursive metric-space equations. Theoret. Comput. Sci. **411**(47), 4102–4122 (2010). https://doi.org/10.1016/j.tcs.2010.07.010
18. Leino, K.R.M., Müller, P.: A basis for verifying multi-threaded programs. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 378–393. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00590-9_27
19. Leino, K.R.M., Müller, P., Smans, J.: Deadlock-free channels and locks. In: Gordon, A.D. (ed.) Programming Languages and Systems. LNCS, vol. 6012, pp. 407–426. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11957-6_22
20. Liang, H., Feng, X.: A program logic for concurrent objects under fair scheduling. POPL (2016). https://doi.org/10.1145/2837614.2837635
21. Liang, H., Feng, X.: Progress of concurrent objects with partial methods. Proc. ACM Program. Lang. **2**, 20:1–20:31 (2017). https://doi.org/10.1145/3158108
22. Mellor-Crummey, J.M., Scott, M.L.: Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Trans. Comput. Syst. **9**, 21–65 (1991). https://doi.org/10.1145/103727.103729
23. Mühlemann, K.: Method for reducing memory conflicts caused by busy waiting in multiple processor synchronisation. IEE Proc. E - Comput. Digit. Techniques **127**(3), 85–87 (1980). https://doi.org/10.1049/ip-e.1980.0017
24. O'Hearn, P., Reynolds, J., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) CSL 2001. LNCS, vol. 2142, pp. 1–19. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44802-0_1
25. Reinhard, T., Jacobs, B.: VeriFast proof of I/O liveness for a simple server with a receiver and a responder thread communicating via a shared buffer (2020). https://github.com/verifast/verifast/blob/master/examples/busywaiting/ioliveness/echo_live_mt.c
26. Reinhard, T., Jacobs, B.: VeriFast proof of safety for CLH lock (2020). https://github.com/verifast/verifast/blob/master/examples/busywaiting/clhlock/clhlock.c
27. Reinhard, T., Jacobs, B.: VeriFast proof of termination for consumer-producer problem with bounded FIFO (2020). https://github.com/verifast/verifast/blob/master/examples/busywaiting/bounded_fifo.c
28. Reinhard, T., Jacobs, B.: Ghost signals: verifying termination of busy waiting (extended version) (2021). https://arxiv.org/abs/2010.11762
29. Reinhard, T., Jacobs, B.: Ghost signals: verifying termination of busy waiting (technical report). Zenodo (2021). https://doi.org/10.5281/zenodo.4775181
30. Reinhard, T., Timany, A., Jacobs, B.: A Separation Logic to Verify Termination of Busy-Waiting for Abrupt Program Exit, New York, NY, USA, pp. 26–32. Association for Computing Machinery (2020). https://doi.org/10.1145/3427761.3428345
31. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: Proceedings 17th Annual IEEE Symposium on Logic in Computer Science, pp. 55–74 (2002). https://doi.org/10.1109/LICS.2002.1029817
32. da Rocha Pinto, P., Dinsdale-Young, T., Gardner, P., Sutherland, J.: Modular termination verification for non-blocking concurrency. In: Thiemann, P. (ed.) ESOP 2016. LNCS, vol. 9632, pp. 176–201. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49498-1_8

33. Vafeiadis, V.: Concurrent separation logic and operational semantics. In: Electronic Notes in Theoretical Computer Science, 276, pp. 335–351 (2011). https://doi.org/10.1016/j.entcs.2011.09.029, twenty-seventh Conference on the Mathematical Foundations of Programming Semantics (MFPS XXVII)

Reflections on Termination of Linear Loops

Shaowei Zhu(✉) and Zachary Kincaid

Princeton University, Princeton, NJ 08544, USA
{shaoweiz,zkincaid}@cs.princeton.edu

Abstract. This paper shows how techniques for linear dynamical systems can be used to reason about the behavior of general loops. We present two main results. First, we show that every loop that can be expressed as a transition formula in linear integer arithmetic has a *best* model as a *deterministic affine transition system*. Second, we show that for any linear dynamical system f with integer eigenvalues and any integer arithmetic formula G, there is a linear integer arithmetic formula that holds exactly for the states of f for which G is eventually invariant. Combining the two, we develop a monotone conditional termination analysis for general loops.

Keywords: Termination · Conditional termination · Best abstraction · Reflective subcategory · Linear dynamical systems · Monotone analysis

1 Introduction

Linear and affine dynamical systems are a model of computation that is easy to analyze (relative to non-linear systems), making them useful across a broad array of applications. In the context of program analysis, affine dynamical systems correspond to loops of the form

$$\textbf{while } (G(\mathbf{x})) \textbf{ do } \mathbf{x} := A\mathbf{x} + \mathbf{b} \qquad (\dagger)$$

where G is a formula, A is a matrix, $\mathbf{x}$ is a vector of program variables, and $\mathbf{b}$ is a constant vector. The termination problem for such loops has been shown to be decidable for several variations of this model [4,9,12,24,29]. However, few loops in real programs take this form, and so this work has not yet made an impact on practical termination analysis tools. This paper bridges the gap between theory and practice, showing how techniques for linear and affine dynamical systems can be used to reason about general programs.

Example 1. We illustrate our methodology using the example program in Fig. 1 (left). First, observe that although the body of this loop is not of the form (†), the value of the sum $x + y$ decreases by z each iteration, and z remains the same. Thus, we can approximate the loop by the linear dynamical system in Fig. 1 (right), where the nature of the approximation is given by the linear map in the center of Fig. 1 (i.e., the a coordinate corresponds to $x + y$, and the b coordinate

```
z := 1
while (x ≥ 0 ∧ y ≥ 0) do
  w := 3w + x + 1
  if ((x - y) % 2 == 0):
    x := x - z
   else:
    y := y - z
```

$$\begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} w \\ x \\ y \\ z \end{bmatrix} \qquad \dashrightarrow \qquad \begin{bmatrix} a \\ b \end{bmatrix} := \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix}$$

Fig. 1. Over-approximation of a loop by a linear dynamical system.

to z). The linear map is a simulation, in the sense that it transforms the state space of the program into the state space of the linear dynamical system so that every step in the loop has a corresponding step in the linear dynamical system.

Next, we compute the image of the guard of the loop $(x \geq 0 \land y \geq 0)$ under the simulation, which yields $a \geq 0$ (corresponding to the constraint $x + y \geq 0$ over the original program variables). We can compute a closed form for this constraint holding on the kth iteration of the loop by exponentiating the dynamics matrix of the linear dynamical system, multiplying on the left by the row vector corresponding to the constraint, and on the right by the simulation:

$$\underbrace{\begin{bmatrix} 1 & 0 \end{bmatrix}}_{\text{Constraint}} \underbrace{\begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}^k}_{\text{Dynamics}} \underbrace{\begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\text{Simulation}} \begin{bmatrix} w \\ x \\ y \\ z \end{bmatrix} = (x + y) - kz.$$

We then analyze the asymptotic behavior of the closed form:

$$\text{As } k \to \infty, (x + y) - kz \to \begin{cases} -\infty & \text{if } z > 0 \\ x + y & \text{if } z = 0 \\ \infty & \text{if } z < 0 \end{cases}$$

We conclude that $z > 0 \lor (x + y) < 0$ is a sufficient condition for the loop to terminate. ⌟

The paper is organized as follows. To serve as the class of "linear models" of loops, we introduce *deterministic affine transition systems* (DATS), a computational model that generalizes affine dynamical systems. Sect. 3 shows that any loop expressed as a linear integer arithmetic formula has a *DATS-reflection*, which is a best representation of the behavior of the loop as a DATS. Moreover, this holds for a restricted class of DATS with rational eigenvalues. Section 4 shows that for a linear map f with integer eigenvalues and a linear integer arithmetic formula G, there is a linear integer arithmetic formula that holds exactly for those states x such that $G(f^k(x))$ holds for all but finitely many $k \in \mathbb{N}$. Section 5 brings the results together, showing that the analysis of a DATS with rational eigenvalues can be reduced to the analysis of a linear dynamical system with integer eigenvalues. The fact that DATS-reflections are *best* implies monotonicity of the analysis. Finally, in Sect. 6, we demonstrate experimentally that the analysis can be successfully applied to general programs, using the framework of algebraic termination analysis [34] to lift our loop analysis to a whole-program conditional termination analysis. Some proofs are omitted for space, but may be found in the extended version of this paper [33].

2 Preliminaries

This paper assumes familiarity with linear algebra – see for example [19]. We recall some basic definitions below.

In the following, a **linear space** refers to a finite-dimensional linear space over the field of rational numbers $\mathbb{Q}$. For V a linear space and $U \subseteq V$, $\mathit{span}(U)$ is the linear space generated by U; i.e., the smallest linear subspace of V that contains U. An **affine subspace** of a linear space V is the image of a linear subspace of V under a translation (i.e., a set of the form $\{v + v_0 : v \in U\}$ for some linear subspace $U \subseteq V$ and some $v_0 \in V$). For any scalar $a \in \mathbb{Q}$, and any linear space V, we use $\underline{a}$ to denote the linear map $\underline{a} : V \to V$ that maps $v \mapsto av$ (in particular, $\underline{1}$ is the identity). A **linear functional** on a linear space V is a linear map $V \to \mathbb{Q}$; the set of all linear functionals on V forms a linear space called the **dual space** of V, denoted $V^\star$. A linear map $f : V_1 \to V_2$ induces a dual linear map $f^\star : V_2^\star \to V_1^\star$ where $f^\star(g) \triangleq g \circ f$. For any linear space V, V is naturally isomorphic to $V^{\star\star}$, where the isomorphism maps $x \mapsto \lambda f : V^\star.f(x)$.

Let V be a linear space. A linear map $f : V \to V$ is associated with a **characteristic polynomial** $p_f(x)$, which is defined to be the determinant of $(xI - A_f)$, where A_f is a matrix representation of f with respect to some basis (the choice of which is irrelevant). Define the **spectrum** (set of eigenvalues) of f to be the set of (possibly complex) roots of its characteristic polynomial, $\mathit{spec}(f) \triangleq \{\lambda \in \mathbb{C} : p_f(\lambda) = 0\}$. We say that f has **rational spectrum** if $\mathit{spec}(f) \subseteq \mathbb{Q}$; equivalently (by the spectral theorem – see e.g. [19, Ch. 6, Theorem 7]):

- There is a basis $\{x_1, ..., x_n\}$ for V consisting of *generalized (right) eigenvectors*, satisfying $(f - \underline{\lambda_i})^{r_i}(x_i) = 0$ for some $\lambda_i \in \mathit{spec}(f)$ and some $r_i \geq 1$ (r_i is called the *rank* of x_i)
- There is a basis $\{g_1, ..., g_n\}$ for $V^\star$ consisting of *generalized left eigenvectors*, satisfying $g_i \circ (f - \underline{\lambda_i})^{r_i} = 0$ for some $\lambda_i \in \mathit{spec}(f)$ and some $r_i \geq 1$

It is possible to determine whether a linear map has rational spectrum (and compute the basis of eigenvectors for V and $V^\star$) in polynomial time by computing its characteristic polynomial [15], factoring it [22], and checking whether each factor is linear.

The syntax of linear integer arithmetic (LIA) is given as follows:

$$\begin{aligned} x &\in \mathsf{Variable} \\ n &\in \mathbb{Z} \\ t &\in \mathsf{Term} ::= x \mid n \mid n \cdot t \mid t_1 + t_2 \\ F &\in \mathsf{Formula} ::= t_1 \leq t_2 \mid (n \mid t) \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid \neg F \mid \exists x.F \mid \forall x.F \end{aligned}$$

Let $X \subseteq \mathsf{Variable}$ be a set of variables. A **valuation** over X is a map $v : X \to \mathbb{Z}$. If F is a formula whose free variables range over X and v is a valuation over X, then we say that v satisfies F (written $v \models F$) if the formula F is true when interpreted over the standard model of the integers, using v to interpret the free variables. We write $F \models G$ if every valuation that satisfies F also satisfies G.

2.1 Transition Systems

A **transition system** T is a pair $T = \langle S_T, R_T \rangle$ where S_T is a set of states and $R_T \subseteq S_T \times S_T$ is a transition relation. Within this paper, we shall assume that the state space of any transition system is a finite-dimensional linear space (over $\mathbb{Q}$). We write $x \to_T x'$ to denote that the pair $\langle x, x' \rangle$ belongs to R_T. We define the **domain** of a transition system T, $\text{dom}(T) \triangleq \{x \in S_T : \exists x'.x \to_T x'\}$, to be the set of states that have a T-successor. We define the **ω-domain** $\text{dom}^\omega(T)$ of T to be the set of states from which there exist infinite T-computations:

$$\text{dom}^\omega(T) \triangleq \{x_0 \in S_T : \exists x_1, x_2, ... \text{ such that } x_0 \to_T x_1 \to_T x_2 \to_T \cdots\} \ .$$

A **transition formula** $F(X, X')$ is an LIA formula whose free variables range over a designated finite set of variables X and a set of "primed copies" $X' = \{x' : x \in X\}$. For example, a transition formula that represents the body of the loop in Fig. 1 is

$$\begin{aligned} & x \geq 0 \wedge y \geq 0 \wedge w' = 3w + x + 1 \wedge z' = z \\ \wedge & \begin{pmatrix} ((2 \mid x - y) \wedge x' = x - z \wedge y' = y) \\ \vee\ (\neg(2 \mid x - y) \wedge y' = y - z \wedge x' = x) \end{pmatrix} \end{aligned} \qquad (1)$$

We use **TF** to denote the set of transition formulas. A transition formula $F(X, X')$ defines a transition system where the state space is the set of functions $X \to \mathbb{Q}$, and where $v \to_F v'$ if and only if both (1) v and v' map each $x \in X$ to an integer and (2) $[v, v'] \models F$, where $[v, v']$ denotes the valuation that maps each $x \in X$ to $v(x)$ and each $x' \in S'$ to $v'(x)$. Defining the state space of F to be $X \to \mathbb{Q}$ rather than $X \to \mathbb{Z}$ is a technical convenience ($X \to \mathbb{Q} \cong \mathbb{Q}^{|X|}$ is a linear space), but does not materially affect the results of this paper since only (integral) valuations are involved in transitions.

Let $T = \langle S_T, R_T \rangle$ be a transition system. We say that T is:

- **linear** if R_T is a linear subspace of $S_T \times S_T$,
- **affine** if R_T is an affine subspace of $S_T \times S_T$,
- **deterministic** if $x \to_T x'_1$ and $x \to_T x'_2$ implies $x'_1 = x'_2$

– **total** if for all $x \in S_T$ there exists some $x' \in S_T$ with $x \to_T x'$

For example, the transition system T with transition relation

$$R_T \triangleq \left\{ \left\langle \begin{bmatrix} x \\ y \end{bmatrix}, \begin{bmatrix} x' \\ y' \end{bmatrix} \right\rangle : \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 0 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \right\}$$

is deterministic and affine, but not linear or total. The transition system U with transition relation

$$R_U \triangleq \left\{ \left\langle \begin{bmatrix} x \\ y \end{bmatrix}, \begin{bmatrix} x' \\ y' \end{bmatrix} \right\rangle : \begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \right\}$$

is total, linear (and affine), but not deterministic. The classical notion of a **linear dynamical system**—a transition system where the state evolves according to a linear map—corresponds to a *total*, *deterministic*, *linear* transition system. Similarly, an **affine dynamical system** is a transition system that is total, deterministic, and affine.

For any map $s : X \to Y$, and any relation $R \subseteq X \times X$, define the image of R under s to be the relation $s[R] = \{\langle s(x), s(x')\rangle : \langle x, x'\rangle \in R\}$. For any relation $R \subseteq Y \times Y$, define the inverse image of R under s to be the relation $s^{-1}[R] = \{\langle x, x'\rangle : \langle s(x), s(x')\rangle \in R\}$. Let $T = \langle S_T, R_T\rangle$ and $U = \langle S_U, R_U\rangle$ be transition systems. We say that a linear map $s : S_T \to S_U$ is a **linear simulation from** T **to** U, and write $s : T \to U$, if for all $x \to_T x'$, we have $s(x) \to_U s(x')$. Observe that the following are equivalent: (1) s is a simulation, (2) $s[R_T] \subseteq R_U$, and (3) $R_T \subseteq s^{-1}[R_U]$.

An example of a simulation between a transition formula and a linear dynamical system is given in Fig. 1. In fact, there are many linear dynamical systems that over-approximate this loop; however, the simulation and linear dynamical system given in Fig. 1 is its *best abstraction.*

To formalize the meaning of *best abstractions*, it is convenient to use the language of category theory [17]. Any class of transition systems defines a category, where the objects are transitions systems of that class, and the arrows are linear simulations between them. We use boldface letters (**L**inear, **A**ffine, **D**eterministic, **T**otal) to denote categories of transition systems (e.g., **DATS** denotes the category of **D**eterministic **A**ffine **T**ransition **S**ystems).

If T is a transition system and **C** is a category of transition systems, a **C-abstraction** of T is a pair $\langle U, s\rangle$ consisting of a transition system U belonging to **C** and a linear simulation $s : T \to U$. A **C-reflection** of T is a **C**-abstraction that satisfies a universal property among **C**-abstractions of T: for any **C**-abstraction $\langle V, t\rangle$ of T there exists a unique simulation $\bar{t} : U \to V$ such that $\bar{t} \circ s = t$; i.e., the following diagram commutes:

$$\begin{tikzcd} & V \\ T \arrow[ur, "t"] \arrow[r, "s"'] & U \arrow[u, dashed, "\bar{t}"'] \end{tikzcd}$$

If **D** is a category of transition systems and **C** is a subcategory such that every transition system in **D** has a **C**-reflection, we say that **C** is a **reflective subcategory** of **D**.

Our ultimate goal is to bring techniques from linear dynamical systems to bear on transition formulas. Fig. 1 gives an example of a program and its linear dynamical system reflection. Unfortunately, such reflections do not exist for *all* transition formulas, which motivates our investigation of alternative models.

Proposition 1. *The transition formula $x' = x \wedge x = 0$ has no* **TDATS***-reflection.*

Proof. Let F be the 1-dimensional transition formula $x' = x \wedge x = 0$. For a contradiction, suppose that $\langle A, s\rangle$ is a **TDATS**-reflection of F. Since F contains the origin, then so must the transition relation of A, and so A is linear. Next, consider that for any $\lambda \in \mathbb{Q}$, we have the simulation $id : F \to A_\lambda$, where id is the identity function and $A_\lambda = \langle \mathbb{Q}, x \mapsto \lambda x\rangle$. Since $\langle A, s\rangle$ is a reflection of F, for any λ, there is some t_λ such that $t_\lambda : A \to A_\lambda$ and $id = t_\lambda \circ s$. Since t_λ is a simulation, we have $\lambda t_\lambda = A_\lambda \circ t_\lambda = t_\lambda \circ A$. Since $id = t_\lambda \circ s$, we must have t_λ non-zero, and so t_λ is a left eigenvector of A with eigenvalue λ. Since this holds for all λ, A must have infinitely many eigenvalues, a contradiction.

3 Linear Abstractions of Transition Formulas

Proposition 1 shows that not every transition formula has a total deterministic affine reflection. In the following we show that *totality* is the only barrier: every transition formula has a (computable) **DATS**-reflection. Moreover, we show that every transition formula has a *rational spectrum* **DATS** ($\mathbb{Q}$-**DATS**)-reflection, a restricted class of **DATS** that generalizes affine maps $x \mapsto A\mathbf{x} + \mathbf{b}$ where A has rational eigenvalues. The restriction on eigenvalues makes it easier to reason about the termination behavior of $\mathbb{Q}$-**DATS**.

In the remainder of this section, we show that every transition formula has a $\mathbb{Q}$-**DATS**-reflection by establishing a chain of reflective subcategories:

$$\begin{tikzcd}\mathbf{TF} \arrow[r, "\text{Lemma 1}"] & \mathbf{ATS} \arrow[r, "\text{Lemma 3}"] & \mathbf{DATS} \arrow[r, "\text{Corollary 1}"] & \mathbb{Q}\text{-}\mathbf{DATS}\end{tikzcd}$$

The fact that $\mathbb{Q}$-**DATS** is a reflective subcategory of **TF** then follows from the fact that a reflective subcategory of a reflective subcategory is reflective.

3.1 Affine Abstractions of Transition Formulas

Let $F(X, X')$ be a transition formula. The **affine hull** of F, denoted *aff*(F), is the smallest affine set *aff*$(F) \subseteq (X \cup X') \to \mathbb{Q} \cong (X \to \mathbb{Q}) \times (X \to \mathbb{Q})$ that contains all of the models of F. Reps et al. give an algorithm that can be used to compute *aff*(F), by using an SMT solver to sample a set of generators [26].

Lemma 1. *Let $F(X, X')$ be a transition formula. The affine hull of F (considered as a transition system) is the best affine abstraction of F (where the simulation from F to aff(F) is the identity).*

Example 2. Consider the example program in Fig. 1. Letting F denote the transition formula corresponding to the program, *aff*(F) can be represented as the solutions to the constraints

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} w' \\ x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} 3 & 1 & 0 & 0 \\ 0 & 1 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} w \\ x \\ y \\ z \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{2}$$

Notice that *aff*(F) is 4-dimensional and has a transition relation defined by 3 constraints, and thus is *not* deterministic. The next step is to find a suitable projection onto a lower-dimensional space so that the resulting transition system is deterministic.

3.2 Reflections via the Dual Space

This section presents a key technical tool that will be used in the next two subsections to prove the existence of reflections. For any transition system T, an abstraction $\langle U, s \rangle$ of T consisting of a transition system U and a simulation $s : S_T \to S_U$ induces a subspace of $S_T^\star$, which is the range of the dual map $s^\star$ (i.e., the set of all linear functionals on S_T of the form $g \circ s$ where $g \in S_U^\star$). The essential idea is we can apply this in reverse: any subspace Λ of $S_T^\star$ induces a transition system U and a simulation $s : T \to U$ that satisfies a universal property among all abstractions $\langle V, v \rangle$ of T where the range of $v^\star$ is contained in Λ. We will now formalize this idea.

Let T be a transition system, and let Λ be a subspace of $S_T^\star$. Define $\alpha_\Lambda(T)$ to be the pair $\alpha_\Lambda(T) \triangleq \langle U, s \rangle$ consisting of a transition system U and a linear simulation $s : T \to U$ where

- $s : S_T \to \Lambda^\star$ sends each $x \in S_T$ to $\lambda f : \Lambda . f(x)$
- $S_U \triangleq \Lambda^\star$, and $R_U \triangleq s[R_T] = \{\langle s(x), s(x') \rangle : \langle x, x' \rangle \in R_T\}$

Lemma 2 (Dual space simulation). *Let T be a transition system, let Λ be a subspace of $S_T^\star$, and let $\langle U, s \rangle = \alpha_\Lambda(T)$. Suppose that Z is a transition system and $z : T \to Z$ is a simulation such that the range of $z^\star$ is contained in Λ. Then there exists a unique simulation $\overline{z} : U \to Z$ such that $\overline{z} \circ s = z$.*

Proof. The high-level intuition is that since the range of $z^\star$ is contained in Λ, we may consider it to be a map $z^\star : S_Z^\star \to \Lambda$; dualizing again, we get a map $z^{\star\star} : \Lambda^\star \to S_Z^{\star\star}$, whose domain is S_U and codomain is (isomorphic to) S_Z.

More formally, let $j : S_Z \to S_Z^{\star\star}$ be the natural isomorphism between S_Z and $S_Z^{\star\star}$ defined by $j(y) \triangleq \lambda g : S_Z^\star . g(y)$. Define $\overline{z} : \Lambda^\star \to S_Z$ by

$$\overline{z}(h) \triangleq j^{-1}(\lambda g : S_Z^\star . h(g \circ z)) \ .$$

First we show that $\overline{z} \circ s = z$. Let $x \in S_Z$. Then we have

$$\begin{aligned}(\overline{z} \circ s)(x) &= \overline{z}(s(x)) \\ &= j^{-1}(\lambda g : S_Z^\star.(s(x))(g \circ z)) \\ &= j^{-1}(\lambda g : S_Z^\star.(\lambda f : \Lambda.f(x))(g \circ z)) \\ &= j^{-1}(\lambda g : S_Z^\star.g(z(x))) \\ &= z(x) \ .\end{aligned}$$

Next we show that $\overline{z}$ is a simulation. Suppose $y \to_U y'$. Since $R_U = s[R_T]$, there is some $x, x' \in S_T$ such that $x \to_T x'$, $s(x) = y$, and $s(x') = y'$. Since $z : T \to Z$ is a simulation, we have that $z(x) \to_Z z(x)$, and so $\overline{z}(s(x)) \to_Z \overline{z}(s(x'))$, and we may conclude that $\overline{z}(y) \to_Z \overline{z}(y')$.

Finally, observe that s is surjective, and therefore the solution to the equation $\overline{z} \circ s = z$ is unique.

We conclude this section by illustrating how to compute the function α for affine transition systems. Suppose that T is an affine transition system of dimension n. We can represent states in S_T by vectors in $\mathbb{Q}^n$, and the transition relation R_T by a finite set of transitions $B \subseteq \mathbb{Q}^n \times \mathbb{Q}^n$ that generates R_T (i.e., $R_T = \mathit{aff}(B)$). Suppose that Λ is an m-dimensional subspace of $S_T^\star$; elements of $S_T^\star$ can be represented by n-dimensional row vectors, and Λ can be represented by a basis $\mathbf{f}_1^\mathsf{T}, \dots, \mathbf{f}_m^\mathsf{T}$. We can compute a representation of $\langle U, s\rangle = \alpha_\Lambda(T)$ as follows. The elements of $S_U = \Lambda^\star$ can be represented by m-dimensional vectors (with respect to the basis $g_1, \dots, g_m$ such that g_i is the linear map that sends $\mathbf{f}_j^\mathsf{T}$ to 1 if $i = j$ and to 0 otherwise). The simulation s can be represented by the $m \times n$ matrix where the ith row is $\mathbf{f}_i^\mathsf{T}$. Finally, the transition relation R_U can be represented by a set of generators $\{\langle s(\mathbf{x}), s(\mathbf{x}')\rangle : \langle \mathbf{x}, \mathbf{x}'\rangle \in B\}$.

3.3 Determinization

In this section, we show that any transition system operating over a finite-dimensional vector space has a best deterministic abstraction, and give an algorithm for computing the best deterministic affine abstraction (or *determinization*) of an affine transition system.

Towards an application of Lemma 2, we seek to characterize the determinization of a transition system by a space of functionals on its state space. For any linear space V and space of functionals Λ on V, define an equivalence relation $\equiv_\Lambda$ on V by $x \equiv_\Lambda y$ iff $f(x) = f(y)$ for all $f \in \Lambda$. If T is a transition system and Λ, Λ' are spaces of functionals on S_T, we say that T is (Λ, Λ')**-deterministic** if for all x_1, x_2 x_1', x_2' such that $x_1 \equiv_\Lambda x_2$, $x_1 \to_T x_1'$, and $x_2 \to_T x_2'$, then we also have $x_1' \equiv_{\Lambda'} x_1'$. Observe that if D is a deterministic transition system and $d : T \to D$ is a simulation, then T must be (Λ_d, Λ_d)-deterministic, where Λ_d is the range of the dual map $d^\star$.

For any T and Λ, define $\mathsf{Det}(T, \Lambda) \triangleq \{f : T \text{ is } (\Lambda, \{f\})\text{-deterministic}\}$ to be the greatest set of functionals such that T is $(\Lambda, \mathsf{Det}(T, \Lambda))$-deterministic.

Observe that $\mathsf{Det}(T, -)$ is a monotone operator on the complete lattice of linear subspaces of $S_T^\star$ (i.e., if $\Lambda_1 \subseteq \Lambda_2$ then $\mathsf{Det}(T, \Lambda_1) \subseteq \mathsf{Det}(T, \Lambda_2)$, since Λ_1 induces a coarser equivalence relation than Λ_2). By the Knaster-Tarski fixpoint theorem [28], $\mathsf{Det}(T, -)$ has a greatest fixpoint, which we denote by $\mathsf{Det}(T)$. Then we have that T is $(\mathsf{Det}(T), \mathsf{Det}(T))$-deterministic, and $\mathsf{Det}(T)$ contains every space Λ such that T is (Λ, Λ)-deterministic.

Lemma 3 (Determinization). *For any transition system T, $\alpha_{\mathsf{Det}(T)}(T)$ is a deterministic reflection of T.*

Proof. Let $\langle D, d\rangle \triangleq \alpha_{\mathsf{Det}(T)}(T)$. First, we show that D is deterministic. Suppose that $y \to_D y_1'$ and $y \to_D y_2'$; we must show that $y_1' = y_2'$. Since R_D is defined to be $d[R_T]$, there must be x_1, x_2, x_1', and x_2' in S_T such that $x_1 \to_T x_1'$, $x_2 \to_T x_2'$, $d(x_1) = d(x_2) = y$, $d(x_1') = y_1'$, and $d(x_2') = y_2$. Since $d(x_1) = d(x_2)$, we have $(\lambda f : \mathsf{Det}(T).f(x_1)) = (\lambda f : \mathsf{Det}(T).f(x_2))$, and therefore $x_1 \equiv_{\mathsf{Det}(T)} x_2$. We thus have $x_1' \equiv_{\mathsf{Det}(T,\mathsf{Det}(T))} x_2'$, and since $\mathsf{Det}(T, \mathsf{Det}(T)) = \mathsf{Det}(T)$, we have $y_1' = d(x_1') = d(x_2') = y_2'$.

It remains to show that $\langle D, d\rangle$ is a deterministic *reflection* of T. Suppose that $\langle U, u\rangle$ is another deterministic abstraction of T. Define G to be the range of $u^\star$. Since U is deterministic, we must have $G \subseteq \mathsf{Det}(T, G)$, and since $\mathsf{Det}(T)$ is the greatest fixpoint of $\mathsf{Det}(T, -)$ we have $G \subseteq \mathsf{Det}(T)$. By Lemma 2, there is a unique linear simulation $\overline{u} : D \to U$ such that $\overline{u} \circ d = u$.

If a transition system T is affine, then its determinization can be computed in polynomial time. Fixing a basis for the state space S_T (of some dimension n), we can represent the transition relation of T in the form $R_T = \{\langle \mathbf{x}, \mathbf{x}'\rangle : A\mathbf{x}' = B\mathbf{x} + \mathbf{c}\}$ where $A, B \in \mathbb{Q}^{m\times n}$ and $\mathbf{c} \in \mathbb{Q}^m$ (for some m). We can represent functionals on S_T by n-dimensional vectors, where the vector $\mathbf{v} \in \mathbb{Q}^n$ corresponds to the functional that maps $\mathbf{u} \mapsto \mathbf{v}^\intercal\mathbf{u}$. A linear space of functionals Λ can be represented by a system of linear equations $\Lambda = \{\mathbf{x} : M\mathbf{x} = 0\}$. The ith row $\mathbf{a}_i^\intercal\mathbf{v} = \mathbf{b}_i^\intercal\mathbf{u} + c_i$, of the system of equations $A\mathbf{x}' = B\mathbf{x} + \mathbf{c}$ can be read as "T is $(\{\mathbf{b}_i^\intercal\}, \{\mathbf{a}_i^\intercal\})$-deterministic." Thus, the functionals $\mathbf{f}^\intercal$ such that T is $(\Lambda, \{\mathbf{f}^\intercal\})$-deterministic are those that can be written as a linear combination of the rows of A such that the corresponding linear combination of the rows of B belongs to Λ; i.e.,

$$\mathsf{Det}(\{\langle \mathbf{x}, \mathbf{x}'\rangle : A\mathbf{x}' = B\mathbf{x} + \mathbf{c}\}, \{\mathbf{f} : M\mathbf{f} = 0\}) = \{\mathbf{d} : \exists \mathbf{y}. MB^\intercal\mathbf{y} = 0 \land A^\intercal\mathbf{y} = \mathbf{d}\} \ .$$

A representation of $\mathsf{Det}(T, \Lambda)$ can be computed in polynomial time using Gaussian elimination. Since the lattice of linear subspaces of $S_T^\star$ has height n, the greatest fixpoint of $\mathsf{Det}(T, -)$ can be computed in polynomial time.

Example 3. Continuing the example from Fig. 1 and Example 2, we consider the determinization of the affine transition system in Eq. (2). The rows of the matrix on the left-hand side correspond to generators for $\mathsf{Det}(\mathit{aff}(F), \mathbb{Q}^{4\star})$:

$$\mathsf{Det}(\mathit{aff}(F), \mathbb{Q}^{4\star}) = \mathit{span}(\{\begin{bmatrix}1 & 0 & 0 & 0\end{bmatrix}, \begin{bmatrix}0 & 1 & 1 & 0\end{bmatrix}, \begin{bmatrix}0 & 0 & 0 & 1\end{bmatrix}\})$$

$$\mathsf{Det}(\mathit{aff}(F), \mathsf{Det}(\mathit{aff}(F), \mathbb{Q}^{4\star})) = \mathit{span}(\{\begin{bmatrix}0 & 1 & 1 & 0\end{bmatrix}, \begin{bmatrix}0 & 0 & 0 & 1\end{bmatrix}\})$$

which is the greatest fixpoint $\mathsf{Det}(\mathit{aff}(F))$. Intuitively: after one step of $\mathit{aff}(F)$, the values of w, $x + y$, and z are affine functions of the input; after two steps $x+y$ and z are affine functions of the input but w is not, since the value of w on the second step depends upon the value of x in the first, and x is not an affine function of the input.

This yields the deterministic reflection $\langle D, d\rangle$ (pictured in Fig. 1) where

$$R_D = \left\{ \left\langle \begin{bmatrix} a \\ b \end{bmatrix}, \begin{bmatrix} a' \\ b' \end{bmatrix} \right\rangle : \begin{bmatrix} a' \\ b' \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} \right\} \qquad \text{and} \qquad d = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \lrcorner$$

3.4 Rational-Spectrum Reflections of DATS

In this section, we define rational-spectrum **DATS** and show that every **DATS** has a rational-spectrum-reflection.

In the following, it is convenient to work with transition systems that are linear rather than affine. We will prove that every deterministic *linear* transition system has a best abstraction with rational spectrum. The result extends to the affine case through the use of *homogenization*: i.e., we embed a (non-empty) affine transition system into a linear transition system with one additional dimension, such that if we fix that dimension to be 1 then we recover the affine transition system. If the transition relation of a **DATS** is represented in the form $A\mathbf{x}' = B\mathbf{x} + \mathbf{c}$, then its homogenization is simply

$$\begin{bmatrix} A & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{x}' \\ y \end{bmatrix} = \begin{bmatrix} B & \mathbf{c} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ y \end{bmatrix} .$$

For a **DATS** T, we use $\mathsf{homog}(T)$ to denote the pair $\langle L, h\rangle$, consisting the **DLTS** L resulting from homogenization and the affine simulation $h : T \to L$ that maps each $\mathbf{x} \in S_T$ to $\begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix}$ (i.e., the affine simulation h formalizes the idea that if we fix the extra dimension y to be 1, we recover the original **DATS** T).

Let T be a deterministic linear transition system. Since our goal is to analyze the asymptotic behavior of T, and all long-running behaviors of T reside entirely within $\text{dom}^\omega(T)$, we are interested in the structure of $\text{dom}^\omega(T)$ and T's behavior on this set. First, we observe that $\text{dom}^\omega(T)$ is a linear subspace of S_T and is computable. For any k, let T^k denote the linear transition system whose transition relation is the k-fold composition of the transition relation of R. Consider the descending sequence of linear spaces

$$\text{dom}(T) \supseteq \text{dom}(T^2) \supseteq \text{dom}(T^3) \supseteq \dots$$

(i.e., the set of states from which there are T computations of length 1, length 2, length 3, ...). Since the space S_T is finite dimensional, this sequence must stabilize at some k. Since the states in $\text{dom}(T^k)$ have T-computations of any length and T is deterministic, we have that $\text{dom}(T^k)$ is precisely $\text{dom}^\omega(T)$.

Since T is total on $\text{dom}^\omega(T)$ and the successor of a state in $\text{dom}^\omega(T)$ must also belong to $\text{dom}^\omega(T)$, T defines a linear map $T|_\omega : \text{dom}^\omega(T) \to \text{dom}^\omega(T)$. In this way, we can essentially reduce asymptotic analysis of **DATS** to asymptotic analysis of linear dynamical systems. The asymptotic analysis of linear dynamical systems developed in Sects. 4 and 5 requires rational eigenvalues; thus we are interested in **DATS** T such that $T|_\omega$ has rational eigenvalues. With this in mind, we define $\mathit{spec}(T) = \mathit{spec}(T|_\omega)$, and say that T **has rational spectrum** if $\mathit{spec}(T) \subseteq \mathbb{Q}$. Define $\mathbb{Q}$-**DLTS** to be the subcategory of **DLTS** with rational spectrum, and $\mathbb{Q}$-**DATS** to be the subcategory of **DATS** whose homogenization lies in $\mathbb{Q}$-**DLTS**.

Example 4. Consider the **DLTS** T with

$$R_T \triangleq \left\{ \left\langle \begin{bmatrix} x \\ y \\ z \end{bmatrix}, \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} \right\rangle : \begin{bmatrix} 1\,0\,0 \\ 0\,1\,0 \\ 0\,0\,1 \\ 0\,0\,0 \end{bmatrix} \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} 2 & 0 & 1 \\ 0 & 2 & 2 \\ 0 & 0 & 3 \\ 1 & -1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \right\}$$

The bottom-most equation corresponds to a constraint that only vectors where the x and y coordinates are equal have successors, so we have:

$$\mathrm{dom}(T) = \left\{ \begin{bmatrix} x \; y \; z \end{bmatrix}^\intercal : x = y \right\}$$

Supposing that the x and y coordinates are equal in some pre-state, they are equal in the post-state exactly when $z = 0$, so we have

$$\mathrm{dom}(T^2) = \left\{ \begin{bmatrix} x \; y \; z \end{bmatrix}^\intercal : x = y \wedge z = 0 \right\}$$

It is easy to check that $\mathrm{dom}(T^3) = \mathrm{dom}(T^2)$, and therefore $\mathrm{dom}^\omega(T) = \mathrm{dom}(T^2)$. The vector $\begin{bmatrix} 1 \; 1 \; 0 \end{bmatrix}^\intercal$ is a basis for $\mathrm{dom}^\omega(T)$, and the matrix representation of $T|_\omega$ with respect to this basis is $\begin{bmatrix} 2 \end{bmatrix}$ (i.e., $\begin{bmatrix} 1 \; 1 \; 0 \end{bmatrix}^\intercal \to_T \begin{bmatrix} 2 \; 2 \; 0 \end{bmatrix}^\intercal$). Thus we can see $\mathit{spec}(T) = \{2\}$, and T is a $\mathbb{Q}$-**DLTS**. ⌟

Towards an application of Lemma 2, define the **generalized rational eigenspace** of a DLTS T to be

$$E_{\mathbb{Q}}(T) \triangleq \mathit{span}\left(\left\{ f \in S_T^\star : \exists \lambda \in \mathbb{Q}, \exists r \in \mathbb{N}^+ . f \circ (T|_\omega - \underline{\lambda})^r = 0 \right\} \right).$$

Lemma 4. *Let T be a DLTS, and define $\langle Q, q \rangle \triangleq \alpha_{E_{\mathbb{Q}}(T)}(T)$. Then for any* $\mathbb{Q}$-**DLTS** *U and any simulation $s : T \to U$, there is a unique simulation $\overline{s} : Q \to U$ such that $\overline{s} \circ q = s$.*

While $\alpha_{E_{\mathbb{Q}}(T)}(T)$ satisfies a universal property for $\mathbb{Q}$-**DLTS**, it does not necessary belong to $\mathbb{Q}$-**DLTS** itself because it need not be deterministic. However, by iterative interleaving of Lemma 4 and determinization as shown in Algorithm 1, we arrive at a $\mathbb{Q}$-**DLTS**-reflection. Example 5 demonstrates how we calculate a $\mathbb{Q}$-**DLTS**-reflection of a particular **DLTS**.

Example 5. Consider the **DLTS** T with transition relation

$$R_T \triangleq \left\{ \left\langle \begin{bmatrix} w \\ x \\ y \\ z \end{bmatrix}, \begin{bmatrix} w' \\ x' \\ y' \\ z' \end{bmatrix} \right\rangle : \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} w' \\ x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -1 & 0 \\ 1 & -1 & 0 & 0 \end{bmatrix} \begin{bmatrix} w \\ x \\ y \\ z \end{bmatrix} \right\}$$

We can calculate the ω-domain of T $\text{dom}^\omega(T) = \left\{ \begin{bmatrix} w & x & y & z \end{bmatrix}^\intercal : w = x \right\}$, which has a basis $B = \begin{bmatrix} 1 & 1 & 0 & 0 \end{bmatrix}^\intercal, \begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix}^\intercal, \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}^\intercal$. With respect to B, $T|_\omega$ corresponds to the matrix

$$T|_\omega = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{bmatrix}$$

and so we have $spec(T) = \{2, i, -i\}$. We may calculate $E_\mathbb{Q}(T)$ by finding (generalized) left eigenvectors with eigenvalue 2, the only rational number in $spec(T)$:

$$\begin{aligned} E_\mathbb{Q}(T) &= \left\{ \mathbf{v}^\intercal : \mathbf{v}^\intercal \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{B} \left(\underbrace{\begin{bmatrix} 2 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{bmatrix}}_{T|_\omega} - \underbrace{\begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{bmatrix}}_{2I} \right) = 0 \right\} \\ &= span(\begin{bmatrix} 1 & 1 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 1 & -1 & 0 & 0 \end{bmatrix}) \end{aligned}$$

Finally, we have $\langle Q, q \rangle = \alpha_{E_\mathbb{Q}(T)}(T)$, where

$$R_Q = \left\{ \left\langle \begin{bmatrix} a \\ b \end{bmatrix}, \begin{bmatrix} a' \\ b' \end{bmatrix} \right\rangle : \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} a' \\ b' \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} \right\} \qquad q = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \end{bmatrix}$$

Q is deterministic and has rational spectrum, so $\langle Q, q \rangle$ is a $\mathbb{Q}$-**DLTS**-reflection of T.

Theorem 1. *For any deterministic linear transition system, Algorithm 1 computes a* $\mathbb{Q}$-**DLTS**-*reflection.*

Finally, by homogenization and Theorem 1, we conclude with the desired result:

Corollary 1. $\mathbb{Q}$-**DATS** *is a reflective subcategory of* **DATS**.

4 Asymptotic Analysis of Linear Dynamical Systems

This section is concerned with analyzing the behavior of loops of the form

$$\textbf{while } (G(\mathbf{x})) \textbf{ do } \mathbf{x} := A\mathbf{x} \ ,$$

```
Input  : A DLTS $T$.
Output : $\mathbb{Q}$-DLTS-reflection of $T$
1 $U \leftarrow T$;
2 $s \leftarrow \lambda x.x$ ;                    /* Invariant: $s$ is a simulation from $T$ to $U$ */
3 while $spec(U|_\omega) \nsubseteq \mathbb{Q}$ do
4     $\langle Q, q\rangle \leftarrow \alpha_{E_{\mathbb{Q}}(U)}(U)$ ;                /* Lemma 4 */
5     $\langle U, d\rangle \leftarrow \alpha_{\mathsf{Det}(Q)}(Q)$ ;                 /* Lemma 3 */
6     $s \leftarrow d \circ q \circ s$;
7 return $\langle U, s\rangle$
```
Algorithm 1: Computation of a $\mathbb{Q}$-**DLTS**-reflection of a **DLTS**

where the $G(\mathbf{x})$ is an LIA formula and A is a matrix with integer spectrum. Our goal is to capture the asymptotic behavior of iterating the map A on an initial state $\mathbf{x}_0$ with respect to the formula G. Specifically, we show that

Theorem 2. *For any LIA formula G and any matrix A with integer spectrum, there is a periodic sequence of LIA formulas $H_0, H_1, H_2, \dots$ such that for any initial state $\mathbf{x}_0 \in \mathbb{Q}^n$, there exists K such that for any $k > K$, $G(A^k\mathbf{x}_0)$ holds if and only if $H_k(\mathbf{x}_0)$ does.*

Recall that an infinite sequence $H_0, H_1, H_2, \dots$ is *periodic* if it is of the form

$$(H_0, H_1, \dots, H_P)^\omega \triangleq H_0, H_1, \dots, H_P, H_0, H_1, \dots, H_P, \dots$$

We call the periodic sequence $(H_0, H_1, \dots, H_P)^\omega$ the *characteristic sequence* of the guard formula G with respect to dynamics matrix A, and denote it by $\chi(G, A)$. Note that $G(A^k\mathbf{x}_0)$ holds for all but finitely many k exactly when $\bigwedge_{i=0}^{P} H_i(\mathbf{x}_0)$ holds.

In the remainder of this section, we show how to compute characteristic sequences. Let G be an LIA formula and let A be a matrix with integer spectrum. To begin, we compute a quantifier-formula G' that is equivalent to G (using, for example, Cooper's algorithm [7]). We define $\chi(G', A)$ by recursion on the structure of G'. For the logical connectives $\wedge$, $\vee$, and $\neg$, characteristic sequences are defined pointwise:

$$\begin{aligned}
\chi(\neg H, A) &\triangleq (\neg(\chi(H, A)_0), \neg(\chi(H, A)_1), \dots)\\
\chi(H_1 \wedge H_2, A) &\triangleq (\chi(H_1, A)_0 \wedge \chi(H_2, A)_0, \chi(H_1, A)_1 \wedge \chi(H_2, A)_1, \dots)\\
\chi(H_1 \vee H_2, A) &\triangleq (\chi(H_1, A)_0 \vee \chi(H_2, A)_0, \chi(H_1, A)_1 \vee \chi(H_2, A)_1, \dots)
\end{aligned}$$

It remains to show how χ acts on atomic formulas, which take the form of inequalities $t_1 \leq t_2$ and divisibility constraints $n \mid t$. An important fact that we employ in both cases is that for any linear term $\mathbf{c}^\intercal\mathbf{x}$ over the variables $\mathbf{x}$, we can compute a closed form for $\mathbf{c}^\intercal A^k(\mathbf{x})$ by symbolically exponentiating A. Since (by assumption) A has integer eigenvalues, this closed form has the form $\frac{1}{Q}(p(\mathbf{x}, k))$ where $Q \in \mathbb{N}$ and p is an **integer exponential-polynomial term**, which takes the form

$$\lambda_1^k k^{d_1}\mathbf{a}_1^\intercal\mathbf{x} + \cdots + \lambda_m^k k^{d_m}\mathbf{a}_m^\intercal\mathbf{x} \quad (3)$$

where $\lambda_i \in \mathit{spec}(A)$, $d_i \in \mathbb{N}$, and $\mathbf{a}_i \in \mathbb{Z}^n$.[1]

Characteristic Sequences for Inequalities. Our method for computing characteristic sequences for inequalities is a variation of Tiwari's method for deciding termination of linear loops with real eigenvalues [29].

First, suppose that $\mathbf{p}(\mathbf{x}, k)$ is an integer exponential-polynomial of the form in Eq. (3) such that each λ_i is a *positive* integer. Further suppose that the summands are ordered by asymptotic growth, with the dominant term appearing earliest in the list; i.e., for $i < j$ we have either $\lambda_i > \lambda_j$, or $\lambda_i = \lambda_j$ and $d_i > d_j$. If we imagine that the variables $\mathbf{x}$ are fixed to some $\mathbf{x}_0 \in \mathbb{Z}^n$, then we see that $p(\mathbf{x}_0, k)$ is either identically zero or has finitely many zeros, and therefore its sign is eventually stable. Furthermore, the sign of $p(\mathbf{x}_0, k)$ as k tends to ∞ is simply the sign of its *dominant term* – that is, the sign of $\mathbf{a}_i^\intercal \mathbf{x}_0$ for the least i such that $\mathbf{a}_i^\intercal \mathbf{x}_0$ is non-zero. Thus, we may define a function DTA that maps any exponential-polynomial term $p(\mathbf{x}, k)$ (with positive integral λ_i) to an LIA formula such that for any $\mathbf{x}_0 \in \mathbb{Z}^n$, $\mathbf{x}_0 \models \mathsf{DTA}(p)$ holds if and only if $\mathbf{p}(\mathbf{x}_0, k)$ is eventually non-negative ($\mathbf{p}(\mathbf{x}_0, k) \geq 0$ for all but finitely many $k \in \mathbb{N}$). DTA is defined as follows:

$$\mathsf{DTA}(0) \triangleq \mathit{true}$$
$$\mathsf{DTA}(\lambda^k k^d \mathbf{a}^\intercal \mathbf{x} + p) \triangleq \mathbf{a}^\intercal \mathbf{x} \geq 1 \vee (\mathbf{a}^\intercal \mathbf{x} = 0 \wedge \mathsf{DTA}(p))$$

Finally, we define the characteristic sequence of an inequality atom as follows. An inequality $t_1 \leq t_2$ over the variables $\mathbf{x}$ can be written as $\mathbf{c}^\intercal \mathbf{x} + d \geq 0$ for $\mathbf{c} \in \mathbb{Z}^n$ and $d \in \mathbb{Z}$. Let $\frac{1}{Q_{even}} p_{even}(\mathbf{x}, k)$ and $\frac{1}{Q_{odd}} p_{odd}(\mathbf{x}, k)$ be the closed forms of $\mathbf{c}^\intercal A^{2k}(\mathbf{x})$ and $\mathbf{c}^\intercal A^{2k+1}(\mathbf{x})$, respectively; by splitting into "even" and "odd" cases, we ensure that the exponential-polynomial terms $p_{even}(\mathbf{x}, k)$ and $p_{odd}(\mathbf{x}, k)$ have only *positive* λ_i and thus are amenable to the dominant term analysis DTA described above. Then we define:

$$\chi\left(\mathbf{c}^\intercal \mathbf{x} + d \geq 0, A\right) \triangleq \left(\mathsf{DTA}(p_{even}(\mathbf{x}, k) + dQ_{even}), \mathsf{DTA}(p_{odd}(\mathbf{x}, k) + dQ_{odd})\right)^\omega$$

Example 6. Consider the matrix A and its exponential A^k below:

$$A\left(\begin{bmatrix} x \\ y \\ z \\ a \\ b \end{bmatrix}\right) = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -3 & 0 \\ 0 & 0 & 0 & 0 & 2 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ a \\ b \end{bmatrix}$$

$$A^k\left(\begin{bmatrix} x \\ y \\ z \\ a \\ b \end{bmatrix}\right) = \begin{bmatrix} 1 & k & \frac{k(k-1)}{2} & 0 & 0 \\ 0 & 1 & k & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & (-3)^k & 0 \\ 0 & 0 & 0 & 0 & 2^k \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ a \\ b \end{bmatrix} = \begin{bmatrix} \frac{1}{2}(zk^2 + (2y - z)k + 2x) \\ zk + y \\ z \\ (-3)^k a \\ 2^k b \end{bmatrix}$$

[1] Technically, we have $\frac{1}{Q}(\lambda_1^k k^{d_1} \mathbf{a}_1^\intercal + \cdots + \lambda_m^k k^{d_m} \mathbf{a}_m^\intercal) = \mathbf{c}^\intercal A^k \mathbf{x}$ for all k greater than rank of the highest-rank generalized eigenvector of 0, but since we are only interested in the asymptotic behavior of A we can disregard the first steps of the computation.

First we compute the characteristic sequence $\chi(x \geq 0, A)$. Applying the dominant term analysis of the closed form of x yields

$$\mathsf{DTA}\left(zk^2 + (2y - z)\,k + x\right) = \begin{pmatrix} z > 0 \\ \vee\,(z = 0 \wedge 2y - z > 0) \\ \vee\,(z = 0 \wedge 2y - z = 0 \wedge x \geq 0) \end{pmatrix} ,$$

Since the closed form involves only positive exponential terms, we need not split into an even and odd case, and we simply have:

$$\chi(x \geq 0, A) = (z > 0 \vee (z = 0 \wedge 2y - z > 0) \vee (z = 0 \wedge 2y - z = 0 \wedge x \geq 0))^\omega$$

Next we compute the characteristic sequence $\chi(a - b \geq 0, A)$, which does require a case split. Applying dominant term analysis of the closed form of $(a - b)$ yields

$$\begin{aligned} \mathsf{DTA}(a \cdot (-3)^{2k} - b \cdot 2^{2k}) &= a > 0 \vee (a = 0 \wedge -b \geq 0) \\ \mathsf{DTA}(a \cdot (-3)^{2k+1} - b \cdot 2^{2k+1}) &= -a > 0 \vee (-a = 0 \wedge -b \geq 0) \ . \end{aligned}$$

and thus we have

$$\chi(a - b \geq 0, A) = (a > 0 \vee (a = 0 \wedge -b \geq 0), -a > 0 \vee (-a = 0 \wedge -b \geq 0))^\omega \ .$$

⌟

Characteristic Sequences for Divisibility Atoms. Last we show how to define χ for divisibility atoms $n \mid t$. Write the term t as $\mathbf{c}^\intercal \mathbf{x} + d$ and let the closed form of $\mathbf{c}^\intercal A^k(\mathbf{x})$ be

$$\frac{1}{Q}(\lambda_1^k k^{d_1} \mathbf{a}_1^\intercal \mathbf{x} + \cdots + \lambda_m^k k^{d_m} \mathbf{a}_m^\intercal \mathbf{x}) \ .$$

The formula $n \mid \mathbf{c}^\intercal A^k(\mathbf{x}) + d$ is equivalent to $Qn \mid \lambda_1^k k^{d_1} \mathbf{a}_1^\intercal \mathbf{x} + \cdots + \lambda_m^k k^{d_m} \mathbf{a}_m^\intercal \mathbf{x} + Qd$. For any i, the sequence $\langle \lambda_i^k k^{d_i} \bmod Qn \rangle_{k=0}^\infty$ is ultimately periodic, since (1) $\langle k \bmod Qn \rangle_{k=0}^\infty = (0, 1, \ldots, Qn - 1)^\omega$, (2) $\langle \lambda_i^k \bmod Qn \rangle_{k=0}^\infty$ is ultimately periodic (with period and transient length bounded above by Qn)[2], and (3) ultimately periodic sequences are closed under pointwise product. It follows that for each i, there is a periodic sequence of integers $\langle z_{i,k} \rangle_{k=0}^\infty$ that agrees with $\langle \lambda_i^k k^{d_i} \bmod Qn \rangle_{k=0}^\infty$ on all but finitely many terms. Finally, we take

$$\chi(n \mid t, A) \triangleq \langle Qn \mid z_{1,k} \mathbf{a}_1^\intercal \mathbf{x} + \cdots + z_{m,k} \mathbf{a}_m^\intercal \mathbf{x} + Qd \rangle_{k=0}^\infty \ .$$

Example 7. Consider matrix A and the closed form of its exponents below

$$A\left(\begin{bmatrix} x \\ y \\ z \end{bmatrix}\right) = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 5 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \qquad A^k\left(\begin{bmatrix} x \\ y \\ z \end{bmatrix}\right) = \begin{bmatrix} 1 & k & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 5^k \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

[2] An infinite sequence $s_0, s_1, s_2, \ldots$ is *ultimately periodic*, if there exists N such that $s_N, s_{N+1}, s_{N+2}, \ldots$ is a periodic sequence. We call N the transient length of this sequence.

We show the characteristic sequences for some divisibility atoms w.r.t A:

$$\begin{aligned}\chi(3 \mid x, A) &= (3 \mid x, 3 \mid x+y, 3 \mid x+2y)^\omega \\ \chi(3 \mid x+2, A) &= (3 \mid x+2, 3 \mid x+y+2, 3 \mid x+2y+2)^\omega \\ \chi(3 \mid z, A) &= (3 \mid z, 3 \mid 2z)^\omega\end{aligned}$$ ⌟

5 A Conditional Termination Analysis for Programs

This section demonstrates how the results from Sects. 3 and 4 can be combined to yield a conditional termination analysis that applies to general programs.

Integer-Spectrum Restriction for $\mathbb{Q}$-DLTS. Section 3 gives a way to compute a $\mathbb{Q}$-**DATS**-reflection of any transition formula. Yet the analysis we developed in Sect. 4 only applies to linear dynamical systems with integer spectrum. We now show how to bridge the gap. Let V be a $\mathbb{Q}$-**DATS**. As discussed in Sect. 3.4, we may homogenize V to obtain a $\mathbb{Q}$-**DLTS** T. Define $\mathbb{Z}(T)$ to be the space spanned by the generalized (right) eigenvectors of $T|_\omega$ that correspond to integer eigenvalues:

$$\mathbb{Z}(T) \triangleq \mathit{span}(\{x \in \text{dom}^\omega(T) : \exists r \in \mathbb{N}^+, \lambda \in \mathbb{Z}.(T|_\omega - \underline{\lambda})^r(x) = 0\})$$

Since $\mathbb{Z}(T)$ is invariant under $T|_\omega$ and thus T, T defines a linear map $T|_\mathbb{Z} : \mathbb{Z}(T) \to \mathbb{Z}(T)$, and by construction $T|_\mathbb{Z}$ has integer spectrum. The following lemma justifies the restriction of our attention to the subspace $\mathbb{Z}(T)$.

Lemma 5. *Let F be a transition formula, let $\langle V, s\rangle$ be a $\mathbb{Q}$-**DATS**-reflection of F, and let $\langle T, h\rangle = \mathsf{homog}(V)$. For any state $v \in dom^\omega(F)$, we have $h(s(v)) \in \mathbb{Z}(T)$.*

Example 8. The following loop computes the number of trailing 0's in the binary representation of integer x and its corresponding transition formula:

```
c := 0
while (x % 2 == 0) do
  x = x / 2
  c = c + 1
```

$$F(x, c, x', c') = \begin{pmatrix} (2 \mid x) \\ \land\ (x - 1 \le 2x' \land 2x' \le x) \\ \land\ (c' = c + 1) \end{pmatrix}$$

The homogenization of the $\mathbb{Q}$-**DATS**-reflection of F is the $\mathbb{Q}$-**DLTS** T, where:

$$R_T \triangleq \left\{ \left\langle \begin{bmatrix} x \\ c \\ h \end{bmatrix}, \begin{bmatrix} x' \\ c' \\ h' \end{bmatrix} \right\rangle : \begin{bmatrix} x' \\ c' \\ h' \end{bmatrix} = \begin{bmatrix} \frac{1}{2} & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ c \\ h \end{bmatrix} \right\}$$

The ω-domain of T is the whole state space $\mathbb{Q}^3$. Since the eigenvector $\begin{bmatrix} 1 & 0 & 0 \end{bmatrix}^\intercal$ of the transition matrix corresponds to a non-integer eigenvalue $\frac{1}{2}$, the x-coordinate of states in $\mathbb{Z}(T)$ must be 0; i.e., $\mathbb{Z}(T) = \{(x, c, y) : x = 0\}$. We conclude that $x \neq 0$ is a sufficient condition for the loop to terminate.

```
Input  : A transition formula $F(\mathbf{x}, \mathbf{x}') \in \mathbf{TF}$ in linear integer arithmetic.
Output : A mortal precondition $mp(F)$ for $F$.
1  $A \leftarrow aff(F)$ ;                                   /* Affine hull [26]; Lemma 1 */
2  $\langle D, d\rangle \leftarrow \alpha_{\mathsf{Det}(A)}(A)$ ;                       /* Determinize; Lemma 3 */
3  $\langle V, q\rangle \leftarrow \mathbb{Q}$-DATS-reflection of $D$ ;                     /* Algorithm 1 */
4  $v \leftarrow q \circ d$ ;                         /* $\langle V, v\rangle$ is a $\mathbb{Q}$-DATS-reflection of $F$ */
5  $\langle T, h\rangle \leftarrow \mathsf{homog}(V)$ ;                            /* Homogenization of V */
6  $t \leftarrow h \circ v$ ;                              /* t is an affine simulation $F \to T$ */
7  $p \leftarrow$ (any) linear projection of $S_T$ onto $\mathbb{Z}(T)$;
8  $C \leftarrow$ matrix such that $C\mathbf{w} = 0 \iff \mathbf{w} \in \mathbb{Z}(T)$;
9  Let $G(\mathbf{w}) \leftarrow \exists \mathbf{x}, \mathbf{x}'.F(\mathbf{x}, \mathbf{x}') \land \mathbf{w} = p(t(\mathbf{x})) \land Ct(\mathbf{x}) = 0$;
10 $(H_0(\mathbf{w}), \dots, H_P(\mathbf{w}))^\omega \leftarrow \chi(G(\mathbf{w}), T|_{\mathbb{Z}})$ ;                  /* Section 4 */
11 return $\neg \left( \left( \bigwedge_i H_i(p(t(\mathbf{x}))) \right) \land Ct(\mathbf{x}) = 0 \right)$
```

Algorithm 2: Procedure for computing $mp(F)$.

The Mortal Precondition Operator. Algorithm 2 shows how to compute a mortal precondition for an LIA transition formula $F(\mathbf{x}, \mathbf{x}')$ (i.e., a sufficient condition for which F terminates). The algorithm operates as follows. First, we compute a $\mathbb{Q}$-**DATS**-reflection of F, and homogenize to get a $\mathbb{Q}$-**DLTS** T and an *affine* simulation $t : F \to T$. Let p denote an (arbitrary) projection from S_T onto $\mathbb{Z}(T)$ (so p is a simulation from T to $T|_{\mathbb{Z}}$). We then compute an LIA formula G which represents the states $\mathbf{w}$ of $T|_{\mathbb{Z}}$ such that there is some $v \in \text{dom}(F)$ such that $t(v) \in \mathbb{Z}(T)$ and $p(t(v)) = \mathbf{w}$. Letting $(H_0, ..., H_P)^\omega$ be the characteristic sequence $\chi(G, T|_{\mathbb{Z}})$, we have that for any $v \in \text{dom}^\omega(F)$, $t(v)$ must belong to $\mathbb{Z}(T)$ and $p(t(v))$ satisfies each H_i, so we define

$$mp(F) \triangleq \{v \in S_F : t(v) \notin \mathbb{Z}(T) \text{ or } v \not\models \bigwedge_i H_i(p(t(\mathbf{x}))).$$

Within the context of the algorithm, we suppose that states of F are represented by n-dimensional vectors, states of T are represented as m-dimensional vectors, and state of $T|_{\mathbb{Z}}$ are represented as q-dimensional vectors. The affine simulation t is represented in the form $\mathbf{x} \mapsto A\mathbf{x} + \mathbf{b}$, where $A \in \mathbb{Z}^{m \times n}$ and $\mathbf{b} \in \mathbb{Z}^m$, the projection p as a $\mathbb{Z}^{q \times m}$ matrix, and the linear map $T|_{\mathbb{Z}}$ as a $\mathbb{Q}^{q \times q}$ matrix. The fact that p and t have all integer (rather than rational) entries is without loss of generality, since any simulation can be scaled by the least common denominator of its entries.

Theorem 3 (Soundness). *For any transition formula F, for any state s such that $s \in mp(F)$, we have $s \notin dom^\omega(F)$.*

Proof. Let T, t, p, C, G, and $H_0, \dots, H_P$ be as in Algorithm 2. We prove the contrapositive: we assume $v \in \text{dom}^\omega(F)$ and prove $v \notin mp(F)$, or equivalently $v \models H_i(p(t(\mathbf{x})))$ for each i and $t(v) \in \mathbb{Z}(T)$. We have $t(v) \in \mathbb{Z}(T)$ by Lemma 5, so it remains only to show that $v \models H_i(p(t(\mathbf{x})))$ for each i.

Since $v \in \text{dom}^\omega(F)$, there exists an infinite trajectory of F starting from v: $v \to_F v_1 \to_F v_2 \to_F \dots$. For any j, let $\mathbf{w}_j = T|_{\mathbb{Z}}^j(p(t(v)))$. Since $p \circ t$

is an (affine) simulation, we have $\mathbf{w}_j = p(t(v_j))$ for all j. It follows that for any j, we have $[v_j, v_{j+1}] \models F(\mathbf{x}, \mathbf{x}') \wedge \mathbf{w}_j = p(t(\mathbf{x}_j)) \wedge Ct(\mathbf{x}_j) = 0$, and so $G(\mathbf{w}_j) = \exists \mathbf{x}, \mathbf{x}'. F(\mathbf{x}, \mathbf{x}') \wedge \mathbf{w}_j = p(t(\mathbf{x})) \wedge Ct(\mathbf{x}) = 0$ holds for all j. By Theorem 2, $H_i(p(t(\mathbf{x})))$ holds for all H_i.

The proof of soundness requires only that we can compute $\mathbb{Q}$-**DATS**-abstractions of transition formulas. The following is the culmination of our development of $\mathbb{Q}$-**DATS**-*reflections*:

Theorem 4 (Monotonicity). *For any transition formulas F_1 and F_2 such that $F_1 \models F_2$, we have $mp(F_2) \models mp(F_1)$.*

The desire for monotonicity is inspired by the principle that *changes to a program should have a predictable impact on its analysis* [34]. Monotonicity guarantees that more information into the analysis always leads to better results—for example, if a user annotates a procedure with pre-conditions or adds loop invariants into the program, our termination analysis can only produce weaker (that is, better) preconditions for termination. Moreover, in the context of this work, monotonicity also guarantees that if we cannot prove termination using the *mp* operator that we defined, then *any* linear abstraction of the loop has reachable non-terminating states.

6 Evaluation

Section 5 shows how to compute mortal preconditions for transition formulas. Using the framework of algebraic termination analysis [34], we can "lift" the analysis to compute mortal preconditions for whole programs. The essential idea is to compute summaries for loops and procedures in "bottom-up" fashion, apply the mortal precondition operator from Sect. 5 to each loop body summary, and then propagate the mortal preconditions for the loops back to the entry of the program (see [34] for more details). We can verify that a program terminates by using an SMT solver to check that its mortal precondition is valid.

We have implemented Algorithm 2 as a mortal precondition operator mp_{LR} ("mortal precondition via Linear Reflections") in ComPACT, a tool that implements the termination analysis framework presented in [34]. We compare the performance of our analysis against 2LS [5], Ultimate Automizer [10] and CPAchecker [23], the top three competitors in the termination category of Competition on Software Verification (SV-COMP) 2020.

Experiments are run on a virtual machine with Ubuntu 18.04, with a single-core Intel Core i7-9750H @ 2.60 GHz CPU and 8 GB of RAM. All tools were run with a time limit of 10 min.

Benchmarks. We tested on a suite of 263 programs divided into 4 categories. The `termination` and `recursive` suites contain small programs with challenging termination arguments, while the `polybench` suite contains larger real-world programs that have relatively simple termination arguments. The `termination`

Table 1. Termination verification benchmarks; time in seconds.

Benchmark	#tasks	mp_{LR}		2LS		UAutomizer		CPAChecker	
		#correct	time	#correct	time	#correct	time	#correct	time
Termination	171	98	**100.8**	115	1966.0	**161**	4772.2	126	12108.6
Recursive	42	4	**51.0**	–	–	**30**	1781.7	23	530.6
Polybench	30	**30**	**128.3**	0	7602.7	0	16241.6	0	4035.8
Linear	20	**20**	37.0	6	**17.6**	8	2841.3	3	3470.7
Total	263	152	**317.1**	121	9586.3	**199**	25636.8	152	20145.7

Table 2. Comparing mp_{LR} and ComPACT; time in seconds.

	#tasks	mp_{LR}		ComPACT-mp_{LR}		ComPACT+mp_{LR}	
		#correct	time	#correct	time	#correct	time
Termination	171	98	**100.8**	141	118.4	**146**	114.4
Recursive	42	4	**51.0**	31	95.4	**32**	94.6
Polybench	30	**30**	**128.3**	**30**	179.6	**30**	179.1
Linear	20	**20**	**37.0**	15	116.5	**20**	65.1
Total	263	152	**317.1**	217	509.9	**228**	453.3

category consists of the *non-recursive, terminating* benchmarks from SV-COMP 2020 in the `Termination-MainControlFlow` suite. The `recursive` category consists of the *recursive, terminating* benchmarks from the `recursive` directory and `Termination-MainControlFlow`. Note that 2LS does not handle recursive programs, so we exclude it from the `recursive` category. Finally, we created a new test suite `linear` consisting of programs with terminating linear abstractions. This suite is designed to exercise the capabilities of the mp_{LR}, and includes all examples from Ben-Amram and Genaim's article [1] on multi-phase ranking functions, loops with disjunctive and/or modular arithmetic guards, and loops that model integer division and remainder calculation.

How Does Our Analysis Compare with the State-of-the-Art? The comparison of ComPACT using the mp_{LR} operator against state-of-the-art termination analysis tools is shown in Table 1. ComPACT with mp_{LR} is competitive with (but not dominating) leading tools in terms of number of tasks solved across the suite, and uses substantially less time. The mp_{LR} analysis is least successful on the `termination` and `recursive` suites, which are designed to have difficult termination arguments. Most competitive tools use a portfolio of different termination techniques to approach such problems (e.g., Ultimate Automizer synthesizes linear, nested, multi-phase, lexicographic and piecewise ranking functions); we investigate the use of mp_{LR} in a portfolio solver in the following.

ComPACT with mp_{LR} solves all tasks in the `polybench` suite, which contains numerical programs that have simple termination arguments, but which are larger than the SV-COMP tasks. 2LS, Ultimate Automizer, and CPAChecker exhaust time or memory limits on all tasks. Nested loops are a problematic pattern that appears in these programs, e.g.,

```
for(int i = 0; i < 4096; i += step)
  for (int j = 0; j < 4096; j += step)
    // no modifications to i, j, or step
```

For such loops, mp_{LR} is guaranteed to synthesize a conditional termination argument that is *at least* as weak as *step* `> 0` (regardless of the contents of the inner loop) by monotonicity and the fact that the loop body formula entails $i < 4096 \wedge i' = i + \mathit{step} \wedge \mathit{step}' = \mathit{step}$. Ultimate Automizer, CPAChecker, and 2LS cannot make such theoretical guarantees.

The `linear` suite demonstrates that mp_{LR} is capable of proving termination of programs that lie outside the boundaries of the other tools.

Can Our Analysis Improve a Portfolio Solver? We compare mp_{LR} and ComPACT in Table 2. The columns correspond to running ComPACT with the following options: excluding the portfolio from [34] (mp_{LR}), including the portfolio but excluding mp_{LR} (ComPACT-mp_{LR}), and including the portfolio and mp_{LR} (ComPACT+mp_{LR}). ComPACT+mp_{LR} can solve 11 additional tasks over ComPACT-mp_{LR} while adding negligible runtime overhead. In fact, adding mp_{LR} to the portfolio *decreases* the amount of time it takes for ComPACT to complete all benchmark suites. Note that the combined tool is successful on the most termination tasks among all the tools we tested, both overall and for each individual suite except the `termination` category.

7 Related Work

Termination Analysis of Linear Loops. The universal termination problem for linear loops (or *total deterministic affine transition systems*, in the terminology of Sect. 4) was posed by Tiwari [29]. The case of linear loops over the reals was resolved by Tiwari [29], over the rationals by Braverman [4], and finally over the integers by Hosseini et al. [14]. In principle, we can combine any of these techniques with our algorithm for computing **DATS**-reflections of transition formulas to yield a sound (but incomplete) termination analysis. The significance of computing a **DATS**-reflection (rather than just "some" abstraction) is that is provides an algorithmic completeness result: if it is possible to prove termination of a loop by exhibiting a terminating linear dynamical system that simulates it, the algorithm will prove termination.

The method introduced in Sect. 4 to compute characteristic sequences of inequalities is based on the method that Tiwari used to prove decidability of the universal termination problem for linear loops with (positive) real spectra [29]. Tiwari's condition of having *real* spectra is strictly more general than the *integer* spectra used by our procedure; requiring that the spectrum be integer allows us express the **DTA** procedure in linear *integer* arithmetic rather than real arithmetic. Similar procedures appear also in [12,18]. We note in particular that our results in Sects. 4 and 5 subsume Frohn and Giesl's decision procedure for universal termination for upper-triangular linear loops [12]; since every rational upper-triangular linear loop has a rational spectrum (and is therefore a $\mathbb{Q}$-**DATS**), the mortal precondition computed for any rational upper-triangular linear loop is valid iff the loop is universally terminating.

Linear Abstractions. The formulation of "best abstractions" using reflective subcategories is based on the framework developed in [17]. A variation of this method was used in the context of invariant generation, based on computing (weak) reflections of linear rational arithmetic formulas in the category of rational vector addition systems [27]. This paper is the first to apply the idea to termination analysis.

A method for extracting polynomial recurrence (in)equations that are entailed by a transition formula appears in [16]. The algorithm can also be applied to compute a **TDATS**-abstraction of a transition formula. The procedure does not guarantee that the **TDATS**-abstraction is a reflection (*best* abstraction); Proposition 1 demonstrates that no such procedure exists. In this paper, we generalize the model to allow non-total transition systems, and show that best abstractions do exist. The techniques from Sect. 3 can be used for invariant generation, improving upon the methods of [16].

Kincaid et al. show that the category of linear dynamical systems with *periodic rational* spectrum is a reflective subcategory of the category of linear dynamical systems [18]. A complex number n is periodic rational if n^p is rational for some $p \in \mathbb{Z}^{>0}$. Combining this result with the technique from Sect. 3 yields the result that the category of **DATS** with periodic rational spectrum is a reflective subcategory of **TF**. The decision procedure from Sect. 4 extends easily to the periodic rational case, which results in a strictly more powerful decision procedure.

Termination Analysis. Termination analysis, and in particular conditional termination analysis, has been widely studied. Work on the subject can be divided into practical termination analyses that work on real programs (but offer few theoretical guarantees) [2,6,8,11,13,20,30–32], and work on simplified model (such as linear, octagonal, and polyhedral loops) with strong guarantees (but cannot be applied directly to real programs) [1,3,4,14,21,25,29]. This paper aims to help bridge the gap between the two, by showing how to apply analyses for linear loops to general programs, while preserving some of their desirable theoretical properties, in particular monotonicity.

Acknowledgments. This work was supported in part by the NSF under grant number 1942537 and by ONR under grant N00014-19-1-2318. Opinions, findings, conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the sponsoring agencies.

References

1. Ben-Amram, A.M., Genaim, S.: On multiphase-linear ranking functions. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 601–620. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_32
2. Borralleras, C., Brockschmidt, M., Larraz, D., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A.: Proving termination through conditional termination. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 99–117. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_6
3. Bradley, A.R., Manna, Z., Sipma, H.B.: Linear ranking with reachability. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 491–504. Springer, Heidelberg (2005). https://doi.org/10.1007/11513988_48
4. Braverman, M.: Termination of integer linear programs. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 372–385. Springer, Heidelberg (2006). https://doi.org/10.1007/11817963_34
5. Chen, H., David, C., Kroening, D., Schrammel, P., Wachter, B.: Bit-precise procedure-modular termination analysis. ACM Trans. Program. Lang. Syst. **40**(1), 1:1–1:38 (2018). https://doi.org/10.1145/3121136
6. Cook, B., Gulwani, S., Lev-Ami, T., Rybalchenko, A., Sagiv, M.: Proving conditional termination. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 328–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70545-1_32
7. Cooper, D.C.: Theorem proving in arithmetic without multiplication. Mach. Intell. **7**(91–99), 300 (1972)
8. Cousot, P., Cousot, R.: An abstract interpretation framework for termination. In: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, pp. 245–258. Association for Computing Machinery, New York (2012). https://doi.org/10.1145/2103656.2103687
9. Cyphert, J., Breck, J., Kincaid, Z., Reps, T.: Refinement of path expressions for static analysis. Proc. ACM Program. Lang. **3**(POPL) (2019). https://doi.org/10.1145/3290358
10. Dietsch, D., Heizmann, M., Nutz, A., Schätzle, C., Schüssele, F.: Ultimate taipan with symbolic interpretation and fluid abstractions. In: TACAS 2020. LNCS, vol. 12079, pp. 418–422. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45237-7_32
11. D'Silva, V., Urban, C.: Conflict-driven conditional termination. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9207, pp. 271–286. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21668-3_16
12. Frohn, F., Giesl, J.: Termination of triangular integer loops is decidable. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11562, pp. 426–444. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25543-5_24
13. Ganty, P., Genaim, S.: Proving termination starting from the end. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 397–412. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_27
14. Hosseini, M., Ouaknine, J., Worrell, J.: Termination of linear loops over the integers. In: Baier, C., Chatzigiannakis, I., Flocchini, P., Leonardi, S. (eds.) ICALP. LIPIcs, vol. 132, pp. 118:1–118:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019). https://doi.org/10.4230/LIPIcs.ICALP.2019.118
15. Keller-Gehrig, W.: Fast algorithms for the characteristic polynomial. Theor. Comput. Sci. **36**(2–3), 309–317 (1985)

16. Kincaid, Z., Cyphert, J., Breck, J., Reps, T.: Non-linear reasoning for invariant synthesis. PACMPL **2**(POPL), 54:1–54:33 (2018)
17. Kincaid, Z.: Numerical invariants via abstract machines. In: Podelski, A. (ed.) SAS 2018. LNCS, vol. 11002, pp. 24–42. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99725-4_3
18. Kincaid, Z., Breck, J., Cyphert, J., Reps, T.: Closed forms for numerical loops. Proc. ACM Program. Lang. **3**(POPL) (2019). https://doi.org/10.1145/3290368
19. Lax, P.D.: Linear Algebra and Its Applications, 2 edn. Wiley-Interscience (2007)
20. Le, T.C., Qin, S., Chin, W.N.: Termination and non-termination specification inference. In: PLDI, PLDI 2015, pp. 489–498. Association for Computing Machinery, New York (2015). https://doi.org/10.1145/2737924.2737993
21. Leike, J., Heizmann, M.: Ranking templates for linear loops. In: TACAS, pp. 172–186 (2014)
22. Lenstra, A.K., Lenstra, H.W., Lovász, L.: Factoring polynomials with rational coefficients. Math. Ann. **261**(4), 515–534 (1982)
23. Ott, S.: Implementing a termination analysis using configurable program analysis. Master's thesis, University of Passau (2016)
24. Ouaknine, J., Pinto, J.S., Worrell, J.: On termination of integer linear loops. In: SODA, pp. 957–969 (2015)
25. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: VMCAI, pp. 239–251 (2004)
26. Reps, T., Sagiv, M., Yorsh, G.: Symbolic implementation of the best transformer. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 252–266. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24622-0_21
27. Silverman, J., Kincaid, Z.: Loop summarization with rational vector addition systems. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11562, pp. 97–115. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25543-5_7
28. Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pac. J. Math. **5**(2), 285–309 (1955)
29. Tiwari, A.: Termination of linear programs. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 70–82. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27813-9_6
30. Urban, C.: The abstract domain of segmented ranking functions. In: Logozzo, F., Fähndrich, M. (eds.) SAS, pp. 43–62 (2013)
31. Urban, C., Miné, A.: An abstract domain to infer ordinal-valued ranking functions. In: Shao, Z. (ed.) ESOP 2014. LNCS, vol. 8410, pp. 412–431. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54833-8_22
32. Urban, C., Miné, A.: A decision tree abstract domain for proving conditional termination. In: Müller-Olm, M., Seidl, H. (eds.) SAS 2014. LNCS, vol. 8723, pp. 302–318. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10936-7_19
33. Zhu, S., Kincaid, Z.: Reflections on termination of linear loops (2021). https://arxiv.org/abs/2105.13941
34. Zhu, S., Kincaid, Z.: Termination analysis without the tears (2021)

Decision Tree Learning in CEGIS-Based Termination Analysis

Satoshi Kura[1,2](✉), Hiroshi Unno[3,4], and Ichiro Hasuo[1,2]

[1] National Institute of Informatics, Tokyo, Japan
kura@nii.ac.jp
[2] The Graduate University for Advanced Studies (SOKENDAI), Kanagawa, Japan
[3] University of Tsukuba, Ibaraki, Japan
[4] RIKEN AIP, Tokyo, Japan

Abstract. We present a novel decision tree-based synthesis algorithm of ranking functions for verifying program termination. Our algorithm is integrated into the workflow of CounterExample Guided Inductive Synthesis (CEGIS). CEGIS is an iterative learning model where, at each iteration, (1) a synthesizer synthesizes a candidate solution from the current examples, and (2) a validator accepts the candidate solution if it is correct, or rejects it providing counterexamples as part of the next examples. Our main novelty is in the design of a synthesizer: building on top of a usual decision tree learning algorithm, our algorithm detects *cycles* in a set of example transitions and uses them for refining decision trees. We have implemented the proposed method and obtained promising experimental results on existing benchmark sets of (non-)termination verification problems that require synthesis of piecewise-defined lexicographic affine ranking functions.

1 Introduction

Termination Verification by Ranking Functions and CEGIS. Termination verification is a fundamental but challenging problem in program analysis. Termination verification usually involves some well-foundedness arguments. Among them are those methods which synthesize *ranking functions* [16]: a ranking function assigns a natural number (or an ordinal, more generally) to each program state, in such a way that the assigned values strictly decrease along transition. Existence of such a ranking function witnesses termination, where well-foundedness of the set of natural numbers (or ordinals) is crucially used.

We study synthesis of ranking functions by CounterExample Guided Inductive Synthesis (CEGIS) [29]. CEGIS is an iterative learning model in which a synthesizer and a validator interact to find solutions for given constraints. At each iteration, (1) a synthesizer tries to find a candidate solution from the current examples, and (2) a validator accepts the candidate solution if it is correct, or rejects it providing counterexamples. These counterexamples are then used as part of the next examples (Fig. 1).

A. Silva and K. R. M. Leino (Eds.): CAV 2021, LNCS 12760, pp. 75–98, 2021.
https://doi.org/10.1007/978-3-030-81688-9_4

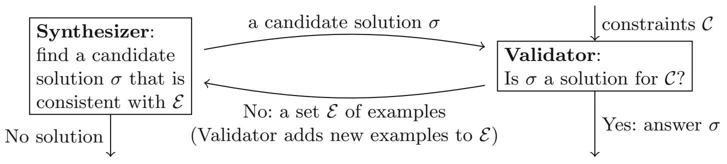

Synthesizer: find a candidate solution σ that is consistent with ℰ
a candidate solution σ
constraints 𝒞
Validator: Is σ a solution for 𝒞?
No: a set ℰ of examples (Validator adds new examples to ℰ)
No solution
Yes: answer σ

Fig. 1. The CEGIS architecture

CEGIS has been applied not only to program verification tasks (synthesis of inductive invariants [17,18,25,26], that of ranking functions [19], etc.) but also to constraint solving (for CHC [12,14,28,36], for pwCSP($\mathcal{T}$) [30,31], etc.). The success of CEGIS is attributed to the degree of freedom that synthesizers enjoy. In CEGIS, synthesizers receive a set of individual examples that synthesizers can use in various creative and speculative manners (such as machine learning). In contrast, in other methods such as [5–8,24,27], synthesizers receive logical constraints that are much more binding.

Segmented Synthesis in CEGIS-Based Termination Analysis. The choice of a *candidate space* for candidate solutions σ is important in CEGIS. A candidate space should be *expressive*: by limiting a candidate space, the CEGIS architecture may miss a genuine solution. At the same time, *complexity* should be low: a larger candidate space tends to be more expensive for synthesizers to handle.

This tradeoff is also in the choice of the type of examples: using an expressive example type, a small number of examples can prune a large portion of the candidate space; however, finding such expressive examples tends to be expensive.

In this paper, we use *piecewise affine functions* as our candidate space for ranking functions. Piecewise affine functions are functions of the form

$$f(\tilde{x}) \quad = \quad \begin{cases} \tilde{a}_1 \cdot \tilde{x} + b_1 & \tilde{x} \in L_1 \\ \quad\vdots \\ \tilde{a}_n \cdot \tilde{x} + b_n & \tilde{x} \in L_n \end{cases} \tag{1}$$

where $\{L_1, \ldots, L_n\}$ is a partition of the domain of $f(\tilde{x})$ such that each L_i is a polyhedron (i.e. a conjunction of linear inequalities). We say *segmented synthesis* to emphasize that our synthesis targets are piecewise affine functions with case distinction. Piecewise affine functions stand on a good balance between expressiveness and complexity: the tasks of synthesizers and validators can be reduced to linear programming (LP); at the same time, case distinction allows them to model a variety of situations, especially where there are discontinuities in the function values and/or derivatives.

We use *transition examples* as our example type (Table 1). Transition examples are pairs of program states that represent transitions; they are much cheaper to handle compared to *trace examples* (finite traces of executions until termination) used e.g. in [15,33]. The current work is the first to pursue segmented synthesis of ranking functions with transition examples; see Table 1.

Table 1. Ranking function synthesis by CEGIS

Candidate space\Example type	Trace examples	Transition examples
Affine ranking functions	[15,33]	[19]
Piecewise affine ranking functions	[15,33]	Our method

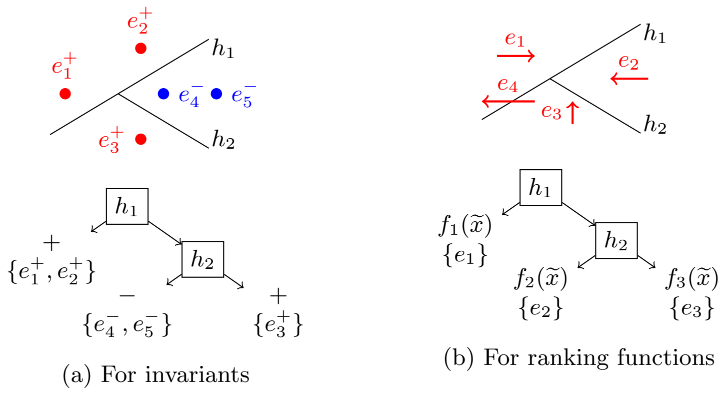


Fig. 2. Decision tree learning

Decision Tree Learning for CEGIS-Based Termination Analysis: a Challenge. In this paper, we represent piecewise affine functions (1) by the data structure of *decision trees*. The data structure suits the CEGIS architecture (Fig. 1): iterative refinement of candidate solutions can be naturally expressed by growing decision trees. The main challenge of this paper is the design of an effective synthesizer for decision trees—such a synthesizer *learns* decision trees from examples.

In fact, decision tree learning in the CEGIS architecture has already been actively pursued, for the synthesis of *invariants* as opposed to ranking functions [12,14,18,22,36]. It is therefore a natural idea to adapt the decision tree learning algorithms used there, from invariants to ranking functions. However, we find that a naive adaptation of those algorithms for invariants does not suffice: they are good at handling *state examples* that appear in CEGIS for invariants; but they are not good at handling transition examples.

More specifically, when decision tree learning is applied to invariant synthesis (Fig. 2a), examples are given in the form of program states labeled as positive or negative. Decision trees are then built by iteratively selecting the best halfspaces—where "best" is in terms of some quality measures—until each leaf contains examples with the same label. One common quality measure used here is an information-theoretic notion of *information gain*.

We extend this from invariant synthesis to ranking function synthesis where examples are given by transitions instead of states (Fig. 2b). In this case, a major challenge is to cope with examples that cross a border of the current segmentation—such as the transition e_4 crossing the border h_1 in Fig. 2b. Our decision tree learning algorithm should handle such crossing examples, taking into account the constraints imposed on the leaf labels affected by those examples (the affected leaf labels are $f_1(\widetilde{x})$ and $f_3(\widetilde{x})$ in the case of e_4).

Our Algorithm: Cycle-Based Decision Tree Learning for Transition Examples. We use what we call the *cycle detection theorem* (Theorem 17) as a theoretical tool to handle such crossing examples. The theorem claims the following: if there is no piecewise affine ranking function with the current segmentation of the domain (such as the one in Fig. 2b given by h_1 and h_2), then this must be caused by a certain type of cycle of constraints, which we call an *implicit cycle*.

In our decision tree learning algorithm, when we do not find a piecewise affine ranking function with the current segmentation, we find an implicit cycle and refine the segmentation to break the cycle. Once all the implicit cycles are gone, the cycle detection theorem guarantees the existence of a candidate piecewise affine ranking function with the segmentation.

We integrate this decision tree learning algorithm in the CEGIS architecture (Fig. 1) and use it as a synthesizer. Our implementation of this framework gives promising experimental results on existing benchmark sets.

Contribution. Our contribution is summarized as follows.

- We provide a decision tree-based synthesizer for ranking functions integrated into the CEGIS architecture. Our synthesizer uses transition examples to find candidate piecewise affine ranking functions. A major challenge here, namely handling constraints arising from crossing examples, is coped with by our theoretical observation of the cycle detection theorem.
- We implement our synthesizer for ranking functions implemented in MuVal and report the experience of using MuVal for termination and non-termination analysis. The experiment results show that MuVal's performance is comparable to state-of-the-art termination analyzers [7,10,13,21] from Termination Competition 2020, and that MuVal can prove (non-)termination of some benchmarks with which other analyzers struggle.

Organization. Section 2 shows the overview of our method via examples. Section 3 explains our target class of predicate constraint satisfaction problems and how to encode (non-)termination problem into such constraints. In Sect. 4, we review CEGIS architecture, and then explain simplification of examples into positive/negative examples. Section 5 proposes our main contribution, our decision tree-based ranking function synthesizer. Section 6 shows our implementation and experimental results. Related work is discussed in Sect. 7, and we conclude in Sect. 8.

2 Preview by Examples

We present a preview of our method using concrete examples. We start with an overview of the general CEGIS architecture, after which we proceed to our main contribution, namely a decision tree learning algorithm for transition examples.

2.1 Termination Verification by CEGIS

Our method follows the usual workflow of termination verification by CEGIS. It works as follows: given a program, we encode the termination problem into a constraint solving problem, and then use the CEGIS architecture to solve the constraint solving problem.

Encoding the Termination Problem. The first step of our method is to encode the termination problem as the set $\mathcal{C}$ of constraints.

Example 1. As a running example, consider the following C program.

```
while(x != 0) { if(x < 0) { x++; } else { x--; } }
```

The termination problem is encoded as the following constraints.

$$x < 0 \land x' = x + 1 \implies R(x, x') \tag{2}$$
$$\neg(x < 0) \land x' = x - 1 \implies R(x, x'). \tag{3}$$

Here, R is a predicate variable representing a well-founded relation, and term variables x, x' are universally quantified implicitly.

The set $\mathcal{C}$ of constraints claims that the transition relation for the given program is subsumed by a well-founded relation. So, verifying termination is now rephrased as the existence of a solution for $\mathcal{C}$. Note that we omitted constraints for invariants for simplicity in this example (see Sect. 3 for the full encoding).

Constraint solving by CEGIS. The next step is to solve $\mathcal{C}$ by CEGIS.

In the CEGIS architecture, a synthesizer and a validator iteratively exchange a set $\mathcal{E}$ of examples and a candidate solution $R(x, x')$ for $\mathcal{C}$. At the moment, we present a rough sketch of CEGIS, leaving the details of our implementation to Sect. 2.2.

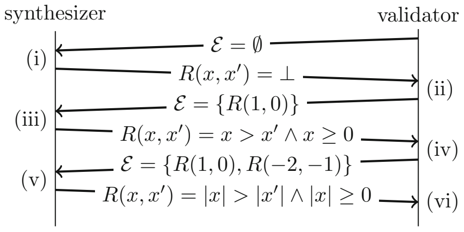

synthesizer
validator
$\mathcal{E} = \emptyset$
(i)
$R(x,x') = \bot$
(ii)
$\mathcal{E} = \{R(1,0)\}$
(iii)
$R(x,x') = x > x' \land x \geq 0$
(iv)
$\mathcal{E} = \{R(1,0), R(-2,-1)\}$
(v)
$R(x,x') = |x| > |x'| \land |x| \geq 0$
(vi)

Fig. 3. An example of CEGIS iterations

Example 2. Figure 3 shows how the CEGIS architecture solves the set $\mathcal{C}$ of constraints shown in (2) and (3). Figure 3 consists of three pairs of interactions (i)–(vi) between a synthesizer and a validator.

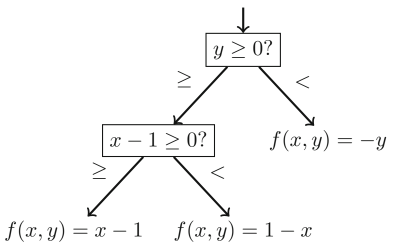
(i) The synthesizer takes $\mathcal{E} = \emptyset$ as a set of examples and returns a candidate solution $R(x, x') = \bot$ synthesized from $\mathcal{E}$. In general, candidate solutions are required to satisfy all constraints in $\mathcal{E}$, but the requirement is vacuously true in this case.

(ii) The validator receives the candidate solution and finds out that the candidate solution is not a genuine solution. The validator finds that the assignment $x = 1, x' = 0$ is a counterexample for (3), and thus adds $R(1, 0)$ to $\mathcal{E}$ to prevent the same candidate solution in the next iteration.
(iii) The synthesizer receives the updated set $\mathcal{E} = \{R(1, 0)\}$ of examples, finds a ranking function $f(x) = x$ for $\mathcal{E}$ (i.e. for the transition from $x = 1$ to $x' = 0$), and returns a candidate solution $R(x, x') = x > x' \wedge x \geq 0$.
(iv) The validator checks the candidate solution, finds a counterexample $x = -2, x' = -1$ for (2), and adds $R(-2, -1)$ to $\mathcal{E}$.
(v) The synthesizer finds a ranking function $f(x) = |x|$ for $\mathcal{E}$ and returns $R(x, x') = |x| > |x'| \wedge |x| \geq 0$ as a candidate solution. Note that the synthesizer have to synthesize a piecewise affine function here, but details are deferred to Sect. 2.2.
(vi) The validator accepts the candidate solution because it is a genuine solution for $\mathcal{C}$.

2.2 Handling Cycles in Decision Tree Learning

We explain the importance of handling cycles in our decision tree-based synthesizer of piecewise affine ranking functions.

In what follows, we deal with such decision trees as shown in Fig. 4: their internal nodes have affine inequalities (i.e. half-spaces); their leaves have affine functions; and overall, such a decision tree expresses a piecewise affine function (Fig. 4). When we remove leaf labels from such a decision tree, then we obtain a template of piecewise functions where condition guards are given but function bodies are not. We shall call the latter a *segmentation*.



$$f(x, y) = \begin{cases} x - 1 & y \geq 0 \wedge x - 1 \geq 0 \\ 1 - x & y \geq 0 \wedge x - 1 < 0 \\ -y & y < 0 \end{cases}$$

Fig. 4. An example of a decision tree that represents a piecewise affine ranking function $f(x, y)$

Input and Output of our Synthesizer. The input of our synthesizer is a set $\mathcal{E}$ of transition examples (e.g. $\mathcal{E} = \{R(1, 0), R(-2, -1)\}$) as explained in Sect. 2.1. The output of our synthesizer is a well-founded relation $R(\widetilde{x}, \widetilde{x}') := f(\widetilde{x}) > f(\widetilde{x}') \wedge f(\widetilde{x}) \geq 0$ where $\widetilde{x}$ is a sequence of variables and $f(\widetilde{x})$ is a piecewise affine function, which is represented by a decision tree (Fig. 4). Therefore our synthesizer aims at *learning* a suitable decision tree.

Refining Segmentations and Handling Cycles. Roughly speaking, our synthesizer learns decision trees in the following steps.

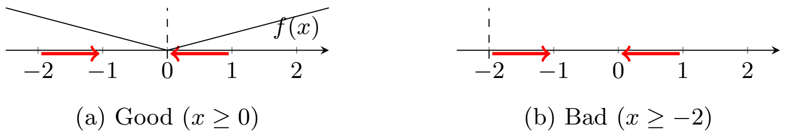


Fig. 5. Selecting halfspaces. Transition examples are shown by red arrows. Boundaries of halfspaces are shown by dashed lines.

1. Generate a set H of halfspaces from the given set $\mathcal{E}$ of examples. This H serves as the vocabulary for internal nodes. Set the initial segmentation to be the one-node tree (i.e. the trivial segmentation).
2. Try to synthesize a piecewise affine ranking function f for $\mathcal{E}$ with the current segmentation—that is, try to find suitable leaf labels. If found, then use this f in a candidate well-founded relation $R(\widetilde{x}, \widetilde{x}') = f(\widetilde{x}) > f(\widetilde{x}') \wedge f(\widetilde{x}) \geq 0$.
3. Otherwise, refine the current segmentation with some halfspace in H, and go to Step 2.

The key step of our synthesizer is Step 3. We show a few examples.

Example 3. Suppose we are given $\mathcal{E} = \{R(1, 0), R(-2, -1)\}$ as a set of examples. Our synthesizer proceeds as follows: (1) Our synthesizer generates the set $H := \{x \geq 1, x \geq 0, x \geq -2, x \geq -1\}$ from the examples in $\mathcal{E}$. (2) Our synthesizer tries to find a ranking function of the form $f(x) = ax + b$ (with the trivial segmentation), but there is no such ranking function. (3) Our synthesizer refines the current segmentation with $(x \geq 0) \in H$ because $x \geq 0$ "looks good". (4) Our synthesizer tries to find a ranking function of the form $f(x) = \textbf{if } x \geq 0 \textbf{ then } ax + b \textbf{ else } cx + d$, using the current segmentation. Our synthesizer obtains $f(x) = \textbf{if } x \geq 0 \textbf{ then } x \textbf{ else } -x$ and use this $f(x)$ for a candidate solution.

How can we decide which halfspace in H "looks good"? We use *quality measure* that is a value representing the quality of each halfspace and select the halfspace with the maximum quality measure.

Figure 5 shows the comparison of the quality of $x \geq 0$ and $x \geq -2$ in this example. Intuitively, $x \geq 0$ is better than $x \geq -2$ because we can obtain a simple ranking function **if** $x \geq 0$ **then** x **else** $-x$ with $x \geq 0$ (Fig. 5a) while we need further refinement of the segmentation with $x \geq -2$ (Fig. 5b). In Sect. 5, we introduce a quality measure for halfspaces following this intuition.

Our synthesizer iteratively refines segmentations following this quality measure, until examples contained in each leaf of the decision tree admit an affine ranking function. This approach is inspired by the use of information gain in the decision tree learning for invariant synthesis.

Example 3 showed a natural extension of a decision tree learning method for invariant synthesis. However, this is not enough for transition examples, for the reasons of *explicit* and *implicit cycles*. Here are their examples.

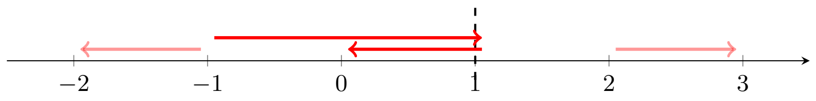


Fig. 6. Two examples $R(-1, 1)$ and $R(1, 0)$ make an implicit cycle between $x \geq 1$ and $\neg(x \geq 1)$.

Example 4. Suppose we are given $\mathcal{E} = \{R(1, 0), R(0, 1)\}$. In this case, there is no ranking function because $\mathcal{E}$ contains a cycle $1 \to 0 \to 1$ witnessing non-termination. We call such a cycle an *explicit cycle*.

Example 5. Let $\mathcal{E} = \{R(-1, 1), R(1, 0), R(-1, -2), R(2, 3)\}$ (Fig. 6). Our synthesizer proceeds as follows. (1) Our synthesizer generates the set $H := \{x \geq 1, x \geq 0, \dots\}$ of halfspaces. (2) Our synthesizer tries to find a ranking function of the form $f(x) = ax + b$ (with the trivial segmentation), but there is no such. (3) Our synthesizer refines the current segmentation with $(x \geq 1) \in H$ because $x \geq 1$ "looks good" (i.e. is the best with respect to a quality measure).

We have reached the point where the naive extension of decision tree learning explained in Example 3 no longer works: although all constraints contained in each leaf of the decision tree admit an affine ranking function, there is no piecewise affine ranking function for $\mathcal{E}$ of the form $f(x) = \textbf{if } x \geq 1 \textbf{ then } ax + b \textbf{ else } cx + d$.

More specifically, in this example, the leaf representing $x \geq 1$ contains $R(2, 3)$, and the other leaf representing $\neg(x \geq 1)$ contains $R(-1, -2)$. The example $R(2, 3)$ admits an affine ranking function $f_1(x) = -x + 2$, and $R(-1, -2)$ admits $f_2(x) = x + 1$, respectively. However, the combination $f(x) = \textbf{if } x \geq 1 \textbf{ then } f_1(x) \textbf{ else } f_2(x)$ is not a ranking function for $\mathcal{E}$. Moreover, there is no ranking function for $\mathcal{E}$ of the form $f(x) = \textbf{if } x \geq 1 \textbf{ then } ax + b \textbf{ else } cx + d$.

It is clear that this failure is caused by the *crossing examples* $R(-1, 1)$ and $R(1, 0)$. It is not that every crossing example is harmful. However, in this case, the set $\{R(-1, 1), R(1, 0)\}$ forms a cycle between the leaf for $x \geq 1$ and the leaf for $\neg(x \geq 1)$ (see Fig. 6). This "cycle" among leaves—in contrast to *explicit* cycles such as $\{R(1, 0), R(0, 1)\}$ in Example 4—is called an *implicit cycle*.

Once an implicit cycle is found, our synthesizer cuts it by refining the current segmentation. Our synthesizer continues the above steps (1–3) of decision tree learning as follows. (4) Our synthesizer selects $(x \geq 0) \in H$ and cuts the implicit cycle $\{R(-1, 1), R(1, 0)\}$ by refining segmentations. (5) Using the refined segmentation, our synthesizer obtains $f(x) = \textbf{if } x \geq 1 \textbf{ then } -x + 2 \textbf{ else if } x \geq 0 \textbf{ then } 0 \textbf{ else } x + 3$ as a ranking function for $\mathcal{E}$.

As explained in Example 4, and 5, handling (explicit and implicit) cycles is crucial in decision tree learning for transition examples. Moreover, our *cycle detection theorem* (Theorem 17) claims that if there is no explicit or implicit cycle, then one can find a ranking function for $\mathcal{E}$ without further refinement of segmentations.

3 (Non-)Termination Verification as Constraint Solving

We explain how to encode (non-)termination verification to constraint solving.

Following [31], we formalize our target class pwCSP of predicate constraint satisfaction problems parametrized by a first-order theory $\mathcal{T}$.

Definition 6. Given a formula ϕ, let $\mathit{ftv}(\phi)$ be the set of free term variables and $\mathit{fpv}(\phi)$ be the set of free predicate variables in ϕ.

Definition 7. A pwCSP is defined as a pair $(\mathcal{C}, \mathcal{R})$ where $\mathcal{C}$ is a finite set of clauses of the form

$$\phi \vee \left(\bigvee_{i=1}^{\ell} X_i(\widetilde{t_i})\right) \vee \left(\bigvee_{i=\ell+1}^{m} \neg X_i(\widetilde{t_i})\right) \tag{4}$$

and $\mathcal{R} \subseteq \mathit{fpv}(\mathcal{C})$ is a set of predicate variables that are required to denote *well-founded* relations. Here, $0 \leq \ell \leq m$. Meta-variables t and ϕ range over $\mathcal{T}$-terms and $\mathcal{T}$-formulas, respectively, such that $\mathit{ftv}(\phi) = \emptyset$. Meta-variables x and X range over term and predicate variables, respectively.

A pwCSP $(\mathcal{C}, \mathcal{R})$ is called CHCs (constrained Horn clauses, [9]) if $\mathcal{R} = \emptyset$ and $\ell \leq 1$ for all clauses $c \in \mathcal{C}$. The class of CHCs has been widely studied in the verification community [12,14,28,36].

Definition 8. A *predicate substitution* σ is a finite map from predicate variables X to closed predicates of the form $\lambda x_1, \ldots, x_{\mathrm{ar}(X)}.\phi$. We write $\mathrm{dom}(\sigma)$ for the domain of σ and $\sigma(\mathcal{C})$ for the application of σ to $\mathcal{C}$.

Definition 9. A predicate substitution σ is a *(genuine) solution* for $(\mathcal{C}, \mathcal{R})$ if (1) $\mathit{fpv}(\mathcal{C}) \subseteq \mathrm{dom}(\sigma)$; (2) $\models \bigwedge \sigma(\mathcal{C})$ holds; and (3) for all $X \in \mathcal{R}$, $\sigma(X)$ represents a well-founded relation, that is, $\mathrm{sort}(\sigma(X)) = (\widetilde{s}, \widetilde{s}) \to \bullet$ for some sequence $\widetilde{s}$ of sorts and there is no infinite sequence $\widetilde{v}_1, \widetilde{v}_2, \ldots$ of sequences $\widetilde{v}_i$ of values of the sorts $\widetilde{s}$ such that $\models \rho(X)(\widetilde{v}_i, \widetilde{v}_{i+1})$ for all $i \geq 1$.

Encoding Termination. Given a set of initial state $\iota(\widetilde{x})$ and a transition relation $\tau(\widetilde{x}, \widetilde{x}')$, the termination verification problem is expressed by the pwCSP $(\mathcal{C}, \mathcal{R})$ where $\mathcal{R} = \{R\}$, and $\mathcal{C}$ consists of the following clauses.

$$\iota(\widetilde{x}) \implies I(\widetilde{x}) \qquad \tau(\widetilde{x}, \widetilde{x}') \wedge I(\widetilde{x}) \implies I(\widetilde{x}') \qquad \tau(\widetilde{x}, \widetilde{x}') \wedge I(\widetilde{x}) \implies R(\widetilde{x}, \widetilde{x}')$$

We use $\phi \implies \psi$ as syntax sugar for $\neg\phi \vee \psi$, so this is a pwCSP. The well-founded relation R asserts that τ is terminating. We also consider an invariant I for τ to avoid synthesizing ranking functions on unreachable program states.

Encoding Non-termination. We can also encode a problem of non-termination verification to pwCSP via recurrent sets [20]. For simplicity, we explain the encoding for the case of only one program variable x. We consider a recurrent set R satisfying the following conditions.

$$\iota(x) \implies R(x) \tag{5}$$
$$R(x) \implies \exists x'.\tau(x,x') \wedge R(x') \tag{6}$$

To remove $\exists$ from (6), we use the following constraint that is equivalent to (6).

$$R(x) \implies E(x,0) \tag{7}$$
$$\begin{aligned} E(x,x') \implies &\big(\tau(x,x') \wedge R(x')\big) \\ &\vee \big(S(x',x'-1) \wedge E(x,x'-1)\big) \vee \big(S(x',x'+1) \wedge E(x,x'+1)\big) \end{aligned} \tag{8}$$

The intuition is as follows. Given x in the recurrent set R, the relation $E(x,x')$ searches for the value of $\exists x'$ in (6). The search starts from $x'=0$ in (7), and x' is nondeterministically incremented or decremented in (8). The well-founded relation S asserts that the search finishes within finite steps. As a result, we obtain a pwCSP for non-termination defined by $(\mathcal{C},\mathcal{R})$ where $\mathcal{R}=\{S\}$ and $\mathcal{C}$ is given by (5), (7), and (the disjunctive normal form of) (8).

Example 10. Consider the following C program.

```
while(x > 0) { x = -2 * x + 9; }
```

The non-termination problem is encoded as the pwCSP $(\mathcal{C},\mathcal{R})$ where $\mathcal{R}=\{S\}$, and $\mathcal{C}$ consists of

$$x>0 \implies R(x) \qquad\qquad R(x) \implies E(x,0)$$
$$\begin{aligned} E(x,x') \implies &\, x'=-2x+9 \wedge R(x') \\ &\vee (S(x',x'-1) \wedge E(x,x'-1)) \vee (S(x',x'+1) \wedge E(x,x'+1)). \end{aligned}$$

The program is non-terminating when $x=3$. This is witnessed by a solution σ for $(\mathcal{C},\mathcal{R})$, which is given by $\sigma(R)(x) \coloneqq x=3$, $\sigma(E)(x,x') \coloneqq x=3 \wedge 0 \le x' \wedge x' \le 3$, and $\sigma(S)(x',x'') \coloneqq x''=x'+1 \wedge x'' \le 3$.

4 CounterExample-Guided Inductive Synthesis (CEGIS)

We explain how CounterExample-Guided Inductive Synthesis [29] (CEGIS for short) works for a given pwCSP $(\mathcal{C},\mathcal{R})$ following [31]. Then, we add the extraction of positive/negative examples to the CEGIS architecture, which enables our decision tree-based synthesizer to use a simplified form of examples.

CEGIS proceeds through the iterative interaction between a synthesizer and a validator (Fig. 1), in which they exchange examples and candidate solutions.

Definition 11. A formula ϕ is an *example* of $\mathcal{C}$ if $\mathit{ftv}(\phi)=\emptyset$ and $\bigwedge\mathcal{C} \models \phi$ hold. Given a set $\mathcal{E}$ of examples of $\mathcal{C}$, a predicate substitution σ is a *candidate solution* for $(\mathcal{C},\mathcal{R})$ that is consistent with $\mathcal{E}$ if σ is a solution for $(\mathcal{E},\mathcal{R})$.

Synthesizer. The input for a synthesizer is a set $\mathcal{E}$ of examples of $\mathcal{C}$ collected from previous CEGIS iterations. The synthesizer tries to find a candidate solution σ consistent with $\mathcal{E}$ instead of a genuine solution for $(\mathcal{C}, \mathcal{R})$. If the candidate solution σ is found, then σ is passed to the validator. If $\mathcal{E}$ is unsatisfiable, then $\mathcal{E}$ witnesses unsatisfiability of $(\mathcal{C}, \mathcal{R})$. Details of our synthesizer is described in Sect. 5.

Validator. A validator checks whether the candidate solution σ from the synthesizer is a genuine solution of $(\mathcal{C}, \mathcal{R})$ by using SMT solvers. That is, satisfiability of $\models \neg \bigwedge \sigma(\mathcal{C})$ is checked. If $\models \neg \bigwedge \sigma(\mathcal{C})$ is not satisfiable, then σ is a genuine solution of the original pwCSP $(\mathcal{C}, \mathcal{R})$, so the validator accepts this. Otherwise, the validator adds new examples to the set $\mathcal{E}$ of examples. Finally, the synthesizer is invoked again with the updated set $\mathcal{E}$ of examples.

If $\models \neg \bigwedge \sigma(\mathcal{C})$ is satisfiable, new examples are constructed as follows. Using SMT solvers, the validator obtains an assignment θ to term variables such that $\models \neg\theta(\psi)$ holds for some $\psi \in \sigma(\mathcal{C})$. By (4), $\models \neg\theta(\psi)$ is a clause of the form $\models \neg\theta(\phi) \wedge \left(\bigwedge_{i=1}^{\ell} \neg\sigma(X_i)(\theta(\widetilde{t_i}))\right) \wedge \left(\bigwedge_{i=\ell+1}^{m} \sigma(X_i)(\theta(\widetilde{t_i}))\right)$. To prevent this counterexample from being found in the next CEGIS iteration again, the validator adds the following example to $\mathcal{E}$.

$$\bigvee_{i=1}^{\ell} X_i(\theta(\widetilde{t_i})) \vee \bigvee_{i=\ell+1}^{m} \neg X_i(\theta(\widetilde{t_i})) \tag{9}$$

The CEGIS architecture repeats this interaction between the synthesizer and the validator until a genuine solution for $(\mathcal{C}, \mathcal{R})$ is found or $\mathcal{E}$ witnesses unsatisfiability of $(\mathcal{C}, \mathcal{R})$.

Extraction of Positive/Negative Examples. Examples obtained in the above explanation are a bit complex to handle in our decision tree-based synthesizer: each example in $\mathcal{E}$ is a disjunction (9) of literals, which may contain multiple predicate variables.

To simplify the form of examples, we extract from $\mathcal{E}$ the sets $\mathcal{E}_X^+$ and $\mathcal{E}_X^-$ of *positive examples* (i.e., examples of the form $X(\widetilde{v})$) and *negative examples* (i.e., examples of the form $\neg X(\widetilde{v})$) for each $X \in \mathit{fpv}(\mathcal{E})$. This allows us to synthesize a predicate $\sigma(X)$ for each predicate variable $X \in \mathit{fpv}(\mathcal{E})$ separately. For simplicity, we write $\widetilde{v} \in \mathcal{E}_X^+$ and $\widetilde{v} \in \mathcal{E}_X^-$ instead of $X(\widetilde{v}) \in \mathcal{E}_X^+$ and $\neg X(\widetilde{v}) \in \mathcal{E}_X^-$.

The extraction is done as follows. We first substitute for each predicate variable application $X(\widetilde{v})$ in $\mathcal{E}$ a boolean variable $b_{X(\widetilde{v})}$ to obtain a SAT problem $\mathbf{SAT}(\mathcal{E})$. Then, we use SAT solvers to obtain an assignment η that is a solution for $\mathbf{SAT}(\mathcal{E})$. If a solution η exists, then we construct positive/negative examples from η; otherwise, $\mathcal{E}$ is unsatisfiable.

Definition 12. Let η be a solution for $\mathbf{SAT}(\mathcal{E})$. For each predicate variable $X \in \mathit{fpv}(\mathcal{E})$, we define the set $\mathcal{E}_X^+$ of *positive examples* and the set $\mathcal{E}_X^+$ of *negative examples* under the assignment η by $\mathcal{E}_X^+ := \{\widetilde{v} \mid \eta(b_{X(\widetilde{v})}) = \mathbf{true}\}$ and $\mathcal{E}_X^- := \{\widetilde{v} \mid \eta(b_{X(\widetilde{v})}) = \mathbf{false}\}$.

Note that some of predicate variable applications $X(\widetilde{v})$ may not be assigned true nor false because they do not affect the evaluation of $\mathbf{SAT}(\mathcal{E})$. Such predicate variable applications are discarded from $\{(\mathcal{E}_X^+, \mathcal{E}_X^-)\}_{X \in \mathit{fpv}(\mathcal{E})}$.

Our method uses the extraction of positive and negative examples when the validator passes examples to the synthesizer. If $X \in \mathit{fpv}(\mathcal{E}) \cap \mathcal{R}$, then we apply our ranking function synthesizer to $(\mathcal{E}_X^+, \mathcal{E}_X^-)$. If $X \in \mathit{fpv}(\mathcal{E}) \setminus \mathcal{R}$, then we apply an invariant synthesizer.

We say a candidate solution σ is consistent with $\{(\mathcal{E}_X^+, \mathcal{E}_X^-)\}_{X \in \mathit{fpv}(\mathcal{E})}$ if $\models \sigma(X)(\widetilde{v}^+)$ and $\models \neg\sigma(X)(\widetilde{v}^-)$ hold for each predicate variable $X \in \mathit{fpv}(\mathcal{E})$, $\widetilde{v}^+ \in \mathcal{E}_X^+$, and $\widetilde{v}^- \in \mathcal{E}_X^-$. If a candidate solution σ is consistent with $\{(\mathcal{E}_X^+, \mathcal{E}_X^-)\}_{X \in \mathit{fpv}(\mathcal{E})}$, then σ is also consistent with $\mathcal{E}$.

Note that unsatisfiability of $\{(\mathcal{E}_X^+, \mathcal{E}_X^-)\}_{X \in \mathit{fpv}(\mathcal{E})}$ does not immediately implies unsatisfiability of $\mathcal{E}$ nor $(\mathcal{C}, \mathcal{R})$ because $\{(\mathcal{E}_X^+, \mathcal{E}_X^-)\}_{X \in \mathit{fpv}(\mathcal{E})}$ depends on the choice of the assignment η. Therefore, the CEGIS architecture need to be modified: if synthesizers find unsatisfiability of $\{(\mathcal{E}_X^+, \mathcal{E}_X^-)\}_{X \in \mathit{fpv}(\mathcal{E})}$, then we add the negation of an unsatisfiability core to $\mathcal{E}$ to prevent using the same assignment η again.

Note that some restricted forms of (9) have also been considered in previous work and are called implication examples in [17] and implication/negation constraints in [12]. Our extraction of positive and negative examples is applicable to the general form of (9).

5 Ranking Function Synthesis

In this section, we describe one of the main contributions, that is, our decision tree-based synthesizer, which synthesizes a candidate well-founded relation $\sigma(R)$ from a finite set $\mathcal{E}_R^+$ of examples. We assume that only positive examples are given because well-founded relations occur only positively in pwCSP for termination analysis (see Sect. 3). The aim of our synthesizer is to find a piecewise affine lexicographic ranking function $\widetilde{f}(\widetilde{x})$ for the given set $\mathcal{E}_R^+$ of examples. Below, we fix a predicate variable $R \in \mathcal{R}$ and omit the subscript $\mathcal{E}_R^+ = \mathcal{E}^+$.

5.1 Basic Definitions

To represent piecewise affine lexicographic ranking functions, we use decision trees like the one in Fig. 4. Let $\widetilde{x} = (x_1, \ldots, x_n)$ be the program variables where each x_i ranges over $\mathbb{Z}$.

Definition 13. A *decision tree* D is defined by $D \coloneqq \widetilde{g}(\widetilde{x}) \mid \textbf{if } h(\widetilde{x}) \geq 0 \textbf{ then } D \textbf{ else } D$ where $\widetilde{g}(\widetilde{x}) = (g_k(\widetilde{x}), \ldots, g_0(\widetilde{x}))$ is a tuple of affine functions and $h(\widetilde{x})$ is an affine function. A *segmentation tree* S is defined as a decision tree with undefined leaves $\bot$: that is, $S \coloneqq \bot \mid \textbf{if } h(\widetilde{x}) \geq 0 \textbf{ then } S \textbf{ else } S$. For each decision tree D, we can canonically assign a segmentation tree by replacing the label of each leaf with $\bot$. This is denoted by $S(D)$. For each decision tree D, we denote the corresponding piecewise affine function by $\widetilde{f}_D(\widetilde{x}) : \mathbb{Z}^n \to \mathbb{Z}^{k+1}$.

Each leaf in a segmentation tree S corresponds to a polyhedron. We often identify the segmentation tree S with the set of leaves of S and a leaf with the polyhedron corresponding to the leaf. For example, we say something like "for each $L \in S$, $\widetilde{v} \in L$ is a point in the polyhedron L".

Suppose we are given a segmentation tree S and a set $\mathcal{E}^+$ of examples.

Definition 14. For each $L_1, L_2 \in S$, we denote the set of example transitions from L_1 to L_2 by $\mathcal{E}^+_{L_1,L_2} := \{(\widetilde{v}, \widetilde{v}') \in \mathcal{E}^+ \mid \widetilde{v} \in L_1, \widetilde{v}' \in L_2\}$. An example $(\widetilde{v}, \widetilde{v}') \in \mathcal{E}^+$ is *crossing* w.r.t. S if $(\widetilde{v}, \widetilde{v}') \in \mathcal{E}^+_{L_1,L_2}$ for some $L_1 \neq L_2$, and *non-crossing* if $(\widetilde{v}, \widetilde{v}') \in \mathcal{E}^+_{L,L}$ for some L.

Definition 15. We define the *dependency graph* $G(S, \mathcal{E}^+)$ for S and $\mathcal{E}^+$ by the graph (V, E) where vertices $V = S$ are leaves, and edges $E = \{(L_1, L_2) \mid L_1 \neq L_2, \exists(\widetilde{v}, \widetilde{v}') \in \mathcal{E}^+_{L_1,L_2}\}$ are crossing examples.

We denote the set of start points $\widetilde{v}$ and end points $\widetilde{v}'$ of examples $(\widetilde{v}, \widetilde{v}') \in \mathcal{E}^+$ by $\underline{\mathcal{E}^+} := \{\widetilde{v} \mid (\widetilde{v}, \widetilde{v}') \in \mathcal{E}^+\} \cup \{\widetilde{v}' \mid (\widetilde{v}, \widetilde{v}') \in \mathcal{E}^+\}$.

5.2 Segmentation and (Explicit and Implicit) Cycles: One-Dimensional Case

For simplicity, we first consider the case where $\widetilde{f}(\widetilde{x}) = f(\widetilde{x}) : \mathbb{Z}^n \to \mathbb{Z}$ is a one-dimensional ranking function. Our aim is to find a ranking function $f(\widetilde{x})$ for $\mathcal{E}^+$, which satisfies $\forall(\widetilde{v}, \widetilde{v}') \in \mathcal{E}^+.\ f(\widetilde{v}) > f(\widetilde{v}')$ and $\forall(\widetilde{v}, \widetilde{v}') \in \mathcal{E}^+.\ f(\widetilde{v}) \geq 0$. If our ranking function synthesizer finds such a ranking function $f(\widetilde{x})$, then a candidate well-founded relation R_f is constructed as $R_f(\widetilde{x}, \widetilde{x}') := f(\widetilde{x}) \geq 0 \wedge f(\widetilde{x}) > f(\widetilde{x}')$.

Our synthesizer builds a decision tree D to find a ranking function $f_D(\widetilde{x})$ for $\mathcal{E}^+$. The main question in doing so is "when and how should we refine partitions of decision trees?" To answer this question, we consider the case where there is no ranking function $f_D(\widetilde{x})$ for $\mathcal{E}^+$ with a fixed segmentation S, and classify reasons for this into three cases as follows.

Case 1: Explicit Cycles in Examples. We define an *explicit cycle* in $\mathcal{E}^+$ as a cycle in the graph $(\mathbb{Z}^n, \mathcal{E}^+)$. An explicit cycle witnesses that there is no ranking function for $\mathcal{E}^+$ (see e.g., Example 4).

Case 2: Non-crossing Examples are Unsatisfiable. The second case is when there is a leaf $L \in S$ such that no affine (not *piecewise* affine) ranking function for the set $\mathcal{E}^+_{L,L}$ of non-crossing examples exists. This prohibits the existence of piecewise affine function $f_D(\widetilde{x})$ for $\mathcal{E}^+$ with segmentation $S = S(D)$ because the restriction of $f_D(\widetilde{x})$ to $L \in S$ must be an affine ranking function for $\mathcal{E}^+_{L,L}$.

Case 3: Implicit Cycles in the Dependency Graph. We define an *implicit cycle* by a cycle in the dependency graph $G(S, \mathcal{E}^+)$. Case 3 is the case where an implicit cycle prohibits the existence of piecewise affine ranking functions for $\mathcal{E}^+$ with the segmentation S (e.g., Example 5). If Case 1 and Case 2 do not hold but no piecewise affine ranking function for $\mathcal{E}^+$ with the segmentation S exists, then there must be an implicit cycle by (the contraposition of) the following proposition.

Proposition 16. *Assume $\mathcal{E}^+$ is a set of examples that does not contain explicit cycles (i.e. Case 1 does not hold). Let S be a segmentation tree and assume that for each $L \in S$, there exists an affine ranking function $f_L(\widetilde{x})$ for $\mathcal{E}^+_{L,L}$ (i.e. Case 2 does not hold). If the dependency graph $G(S, \mathcal{E}^+)$ is acyclic, then there exists a decision tree D with the segmentation $S(D) = S$ such that $f_D(\widetilde{x})$ is a ranking function for $\mathcal{E}^+$.*

Proof. By induction on the height (i.e. the length of a longest path from a vertex) of vertices in $G(S, \mathcal{E}^+)$. We construct a decision tree D as follows. If the height of $L \in S$ is 0, then we assign $f'_L(\widetilde{x}) := f_L(\widetilde{x})$ to the leaf L where $f_L(\widetilde{x})$ is a ranking function for $\mathcal{E}^+_{L,L}$. If the height of $L \in S$ is $n > 0$, then we assign $f'_L(\widetilde{x}) := f_L(\widetilde{x}) + c$ to the leaf L where $c \in \mathbb{Z}$ is a constant that satisfies $\forall(\widetilde{v}, \widetilde{v}') \in \mathcal{E}^+_{L,L'}, f_L(\widetilde{v}) + c > f'_{L'}(\widetilde{v}')$ for each cell L' with the height less than n. □

Note that the converse of Proposition 16 does not hold: the existence of implicit cycles in $G(S, \mathcal{E}^+)$ does not necessarily imply that no piecewise affine ranking function exists with the segmentation S.

5.3 Segmentation and (Explicit and Implicit) Cycles: Multi-Dimensional Lexicographic Case

We consider a more general case where $\widetilde{f}(\widetilde{x}) = (f_k(\widetilde{x}), \dots, f_0(\widetilde{x}))$ is a multi-dimensional lexicographic ranking function and k is a fixed nonnegative integer.

Given a function $\widetilde{f}(\widetilde{x})$, we consider the well-founded relation $R_{\widetilde{f}}(\widetilde{x}, \widetilde{x}')$ defined inductively as follows.

$$R_{()}(\widetilde{x}, \widetilde{x}') := \bot \quad R_{(f_k,\dots,f_0)}(\widetilde{x}, \widetilde{x}') := f_k(\widetilde{x}) \geq 0 \land f_k(\widetilde{x}) > f_k(\widetilde{x}') \\ \lor f_k(\widetilde{x}) = f_k(\widetilde{x}') \land R_{(f_{k-1},\dots,f_0)}(\widetilde{x}, \widetilde{x}') \tag{10}$$

Our aim here is to find a lexicographic ranking function $\widetilde{f}(\widetilde{x})$ for $\mathcal{E}^+$, i.e. a function $\widetilde{f}(\widetilde{x})$ such that $R_{\widetilde{f}}(\widetilde{v}, \widetilde{v}')$ holds for each $(\widetilde{v}, \widetilde{v}') \in \mathcal{E}^+$. Our synthesizer does so by building a decision tree. The same argument as the one-dimensional case holds for lexicographic ranking functions.

Theorem 17 (cycle detection). *Assume $\mathcal{E}^+$ is a set of examples that does not contain explicit cycles. Let S be a segmentation tree and assume that for each $L \in S$, there exists an affine function $\widetilde{f}_L(\widetilde{x})$ that satisfies $\forall(\widetilde{v}, \widetilde{v}') \in \mathcal{E}^+_{L,L}, R_{\widetilde{f}_L}(\widetilde{v}, \widetilde{v}')$. If the dependency graph $G(S, \mathcal{E}^+)$ is acyclic, then there exists a decision tree D with the segmentation $S(D) = S$ such that $R_{\widetilde{f}_D}(\widetilde{v}, \widetilde{v}')$ holds for each $(\widetilde{v}, \widetilde{v}') \in \mathcal{E}^+$.*

Proof. The proof is almost the same as Proposition 16. Here, note that if $\widetilde{f}'(\widetilde{x}) = \widetilde{f}(\widetilde{x}) + \widetilde{c}$ where $\widetilde{c}$ is a tuple of nonnegative integer constants, then $R_{\widetilde{f}'}(\widetilde{x}, \widetilde{x}')$ subsumes $R_{\widetilde{f}}(\widetilde{x}, \widetilde{x}')$. □

Algorithm 1 Building decision trees.

```
Input: a set $\mathcal{E}^+$ of examples, an integer $k \geq 0$
Output: a well-founded relation $R$ such that $\forall(\widetilde{x}, \widetilde{x}') \in \mathcal{E}^+, R(\widetilde{x}, \widetilde{x}')$
 1: if $E$ has a cycle then
 2:     return unsatisfiable
 3: end if
 4: $D \coloneqq$ ResolveCase2($E$)
 5: while true do
 6:     $C \coloneqq$ GetConstraints($D, E$)
 7:     $O \coloneqq$ SumAbsParams($D$)
 8:     $\rho \coloneqq$ Minimize($O, C$)
 9:     if $\rho$ is defined then
10:         $\widetilde{f}(\widetilde{x}) \coloneqq \widetilde{f}_{\rho(D)}(\widetilde{x})$
11:         return $R_{\widetilde{f}}$
12:     else
13:         get an unsat core in $C$
14:         find an implicit cycle $(\widetilde{v}_1, \widetilde{v}_1'), \ldots, (\widetilde{v}_l, \widetilde{v}_l')$ in the unsat core
15:         find a cell $C$ and two distinct points $\widetilde{v}_i', \widetilde{v}_{i+1} \in C$ in the implicit cycle
16:         add a halfspace to separate $\widetilde{v}_i'$ and $\widetilde{v}_{i+1}$ and update $D$
17:     end if
18: end while
```

5.4 Our Decision Tree Learning Algorithm

We design a concrete algorithm based on Theorem 17. It is shown in Algorithm 1 and consists of three phases. We shall describe the three phases one by one.

Phase 1. Phase 1 (Line 1–3) detects explicit cycles in $\mathcal{E}^+$ to exclude Case 1. Here, we use a cycle detection algorithm for directed graphs.

Phase 2. Phase 2 (Line 4) detects and resolves Case 2 by using ResolveCase2 (Algorithm 2), which is a function that grows a decision tree recursively. ResolveCase2 takes non-crossing examples in a leaf, divides the leaf, and returns a *template tree* that is fine enough to avoid Case 2. Here, template trees are decision trees whose leaves are labeled by affine templates.

Algorithm 2 shows the detail of ResolveCase2. ResolveCase2 builds a template tree recursively starting from the trivial segmentation $S = \bot$ and all given examples. In each polyhedron, ResolveCase2 checks whether the set C of constraints imposed by non-crossing examples can be satisfied by an affine lexicographic ranking function on the polyhedron (Line 2–3). If the set C of constraints is not satisfiable, then ResolveCase2 chooses a halfspace $h(\widetilde{x}) \geq 0$ (Line 6) and divides the current polyhedron by the halfspace.

There is a certain amount of freedom in the choice of halfspaces. To guarantee termination of the whole algorithm, we require that the chosen halfspace h separates at least one point in $\underline{\mathcal{E}'^+} \coloneqq \{\widetilde{v} \mid (\widetilde{v}, \widetilde{v}') \in \mathcal{E}'^+\} \cup \{\widetilde{v}' \mid (\widetilde{v}, \widetilde{v}') \in \mathcal{E}'^+\}$ from the other points in $\underline{\mathcal{E}'^+}$. That is:

Algorithm 2 Resolving Case 2.

```
1: function ResolveCase2($\mathcal{E}'^{+}$)
2:     $\widetilde{f} := $ MakeAffineTemplate($k$)
3:     $C := $ GetConstraints($\widetilde{f}, \mathcal{E}'^{+}$)
4:     $\rho := $ GetModel($C$)
5:     if $\rho$ is undefined then
6:         $h := $ ChooseQualifier($\mathcal{E}'^{+}$)
7:         $D_{\geq 0} := $ ResolveCase2($\{(\widetilde{v}, \widetilde{v}') \in \mathcal{E}'^{+} \mid h(\widetilde{v}) \geq 0 \wedge h(\widetilde{v}') \geq 0\}$)
8:         $D_{<0} := $ ResolveCase2($\{(\widetilde{v}, \widetilde{v}') \in \mathcal{E}'^{+} \mid h(\widetilde{v}) < 0 \wedge h(\widetilde{v}') < 0\}$)
9:         return (if $h(\widetilde{x}) \geq 0$ then $D_{\geq 0}$ else $D_{<0}$)
10:    else
11:        return $\widetilde{f}$
12:    end if
13: end function
14: function GetConstraints($D, \mathcal{E}^{+}$)
15:    return $\{R_{\widetilde{f}_D}(\widetilde{v}, \widetilde{v}') \mid (\widetilde{v}, \widetilde{v}') \in \mathcal{E}^{+}\}$ where $\widetilde{f}_D$ is the tuple of piecewise affine
    functions corresponding to $D$
16: end function
```

Algorithm 3 A criterion for eager qualifier selection.

```
1: function QualityMeasure($h, \mathcal{E}'^{+}$)
2:     $E_{++} := \{(\widetilde{v}, \widetilde{v}') \in \mathcal{E}'^{+} \mid h(\widetilde{v}) \geq 0 \wedge h(\widetilde{v}) \geq 0\}$
3:     $E_{+-} := \{(\widetilde{v}, \widetilde{v}') \in \mathcal{E}'^{+} \mid h(\widetilde{v}) \geq 0 \wedge h(\widetilde{v}) < 0\}$
4:     $E_{-+} := \{(\widetilde{v}, \widetilde{v}') \in \mathcal{E}'^{+} \mid h(\widetilde{v}) < 0 \wedge h(\widetilde{v}) \geq 0\}$
5:     $E_{--} := \{(\widetilde{v}, \widetilde{v}') \in \mathcal{E}'^{+} \mid h(\widetilde{v}) < 0 \wedge h(\widetilde{v}) < 0\}$
6:     $\widetilde{f} := $ MakeAffineTemplate($k$)
7:     $C_{+} := $ GetConstraints($\widetilde{f}, E_{++}$)        $C_{-} := $ GetConstraints($\widetilde{f}, E_{--}$)
8:     $N_{+} := $ MaxSmt($C_{+}$)                       $N_{-} := $ MaxSmt($C_{-}$)
9:     return $N_{+} + N_{-} + (|E_{+-}| + |E_{-+}|)(1 - \text{entropy}(|E_{+-}|, |E_{-+}|))$
10: end function
```

Assumption 18. If halfspace $h(\widetilde{x}) \geq 0$ is chosen in Line 6 of Algorithm 2, then there exist $\widetilde{v}, \widetilde{u} \in \underline{\mathcal{E}'^{+}}$ such that $h(\widetilde{v}) \geq 0$ and $h(\widetilde{u}) < 0$.

We explain two strategies (eager and lazy) to choose halfspaces that can be used to implement ChooseQualifier. Both of them are guaranteed to terminate, and moreover, intended to yield simple decision trees.

Eager Strategy. In the eager strategy, we eagerly generate a finite set H of halfspaces from the set $\mathcal{E}^{+}$ of all examples beforehand and choose the best one from H with respect to a certain quality measure. To satisfy Assumption 18, H are generated so that any two points $\widetilde{u}, \widetilde{v} \in \underline{\mathcal{E}^{+}}$ can be separated by some halfspace $(h(\widetilde{x}) \geq 0) \in H$.

For example, we can use intervals $H = \{\pm(x_i - a_i) \geq 0 \mid i = 1, \dots, n \wedge (a_1, \dots, a_n) \in \underline{\mathcal{E}^{+}}\}$ and octagons $H = \{\pm(x_i - a_i) \pm (x_j - a_j) \geq 0 \mid i \neq j \wedge (a_1, \dots, a_n) \in \underline{\mathcal{E}^{+}}\}$ where $\widetilde{x} = (x_1, \dots, x_n)$. For any input $\mathcal{E}'^{+} \subseteq \mathcal{E}^{+}$ of

ResolveCase2, intervals and octagons satisfy $\emptyset \neq H' := \{h(\widetilde{x}) \geq 0 \mid \exists \widetilde{v}, \widetilde{u} \in \mathcal{E}'^{+}. h(\widetilde{v}) \geq 0 \wedge h(\widetilde{u}) < 0\}$, so Assumption 18 is satisfied by choosing the best halfspace with respect to the quality measure from H'.

For each halfspace $(h(\widetilde{x}) \geq 0) \in H'$, we calculate QualityMeasure in Algorithm 3, and choose one that maximizes QualityMeasure$(h, \mathcal{E}'^{+})$. QualityMeasure$(h, \mathcal{E}'^{+})$ calculates the sum of the maximum number of satisfiable constraints in each leaf divided by $h(\widetilde{x}) \geq 0$ plus an additional term $(|E_{+-}| + |E_{-+}|)(1 - \mathrm{entropy}(|E_{+-}|, |E_{-+}|))$ where $\mathrm{entropy}(x, y) = -\frac{x}{x+y} \log_2 \frac{x}{x+y} - \frac{y}{x+y} \log_2 \frac{y}{x+y}$. Therefore, the term $(|E_{+-}| + |E_{-+}|)(1 - \mathrm{entropy}(|E_{+-}|, |E_{-+}|))$ is close to $|E_{+-}| + |E_{-+}|$ if almost all examples in $E_{+-} \cup E_{-+}$ cross h in the same direction and close to 0 if $|E_{+-}|$ is almost equal to $|E_{-+}|$.

Lazy Strategy. In the lazy strategy, we lazily generate halfspaces. We divide the current polyhedron so that non-crossing examples in the cell point to almost the same direction.

First, we label states that occur in $\mathcal{E}^{+}_{C,C}$ as follows. We find a direction that most examples in C point to by solving the MAX-SMT $\boldsymbol{a} := \max_{\boldsymbol{a}} \left|\{(\widetilde{v}, \widetilde{v}') \in \mathcal{E}^{+}_{C,C} \mid \boldsymbol{a} \cdot (\widetilde{v} - \widetilde{v}') > 0\}\right|$. For each $(\widetilde{v}, \widetilde{v}') \in \mathcal{E}^{+}_{C,C}$, we label two points $\widetilde{v}, \widetilde{v}'$ with $+1$ if $\boldsymbol{a} \cdot (\widetilde{v} - \widetilde{v}') > 0$ and with -1 otherwise.

Then we apply weighted C-SVM to generate a hyperplane that separates most of the positive and negative points. To guarantee termination of Algorithm 1, we avoid "useless" hyperplanes that classify all the points by the same label. If we obtain such a useless hyperplane, then we undersample a majority class and apply C-SVM again. By undersampling suitably, we eventually get linearly separable data with at least one positive point and one negative point.

Note that since coefficients of hyperplanes extracted from C-SVM are floating point numbers, we have to approximate them by hyperplanes with rational coefficients. This is done by truncating continued fraction expansions of coefficients by a suitable length.

Phase 3. In Line 5–18 of Algorithm 1, we further refine the segmentation $S(D)$ to resolve Case 3. Once Case 2 is resolved by ResolveCase2, Case 2 never holds even after refining $S(D)$ further. This enables to separate Phases 2 and 3.

Given a template tree D, we consider the set C of constraints on parameters in D that claims $\widetilde{f_D}(\widetilde{x})$ is a ranking function for $\mathcal{E}^{+}$ (Line 6).

If C is satisfiable, we use an SMT solver to obtain a solution of C (i.e. an assignment ρ of integers to parameters) while minimizing the sum of absolute values of unknown parameters in D at the same time (Line 8). This minimization is intended to give a simple candidate ranking function. The solution ρ is used to instantiate the template tree D (Line 11).

If C cannot be satisfied, there must be an implicit cycle in the dependency graph $G(S(D), \mathcal{E}^{+})$ by Theorem 17. The implicit cycle can be found in an unsatisfiable core of C. We refine the segmentation of D to cut the implicit cycle in Line 16. To guarantee termination, we choose a halfspace satisfying the following assumption, which is similar to Assumption 18.

Assumption 19. If halfspace $h(\widetilde{x}) \geq 0$ is chosen in Line 16 of Algorithm 1, then there exist $\widetilde{v}, \widetilde{u} \in \underline{\mathcal{E}^+}$ such that $h(\widetilde{v}) \geq 0$ and $h(\widetilde{u}) < 0$.

We have two strategy (eager and lazy) to refine the segmentation of D.

In eager strategy, we choose a halfspace $(h(\widetilde{x}) \geq 0) \in H$ that separates two distinct points $\widetilde{v}'_i$ and $\widetilde{v}_{i+1}$ in the implicit cycle. In doing so, we want to reduce the number of implicit cycles in $G(S(D), \mathcal{E}^+)$, but adding a new halfspace may introduce new implicit cycles if there exists $(\widetilde{v}, \widetilde{v}') \in \mathcal{E}^+_{C,C}$ that crosses the new border from the side of $\widetilde{v}'_i$ to the side of $\widetilde{v}_{i+1}$. Therefore, we choose a hyperplane that minimizes the number of new crossing examples.

In lazy strategy, we use an SMT solver to find a hyperplane $h(\widetilde{x}) \in H$ that separates $\widetilde{v}'_i$ and $\widetilde{v}_{i+1}$ and minimizes the number of new crossing examples.

Termination. Assumption 18 and Assumption 19 guarantees that every leaf in $S(D)$ contains at least one point in the finite set $\underline{\mathcal{E}^+}$. Because the number of leaves in $S(D)$ strictly increases after each iteration of Phase 2 and Phase 3, we eventually get a segmentation $S(D)$ where each $L \in S(D)$ contains only one point in $\underline{\mathcal{E}^+}$ in the worst case. Since we have excluded Case 1 at the beginning, Theorem 17 guarantees the existence of ranking function with the segmentation $S(D)$. Therefore, the algorithm terminates within $|\underline{\mathcal{E}^+}|$ times of refinement.

Theorem 20. *If Assumption 18 and Assumption 19 hold, then Algorithm 1 terminates. If Algorithm 1 returns a piecewise affine lexicographic function $\widetilde{f}(\widetilde{x})$, then the function satisfies $R_{\widetilde{f}}(\widetilde{x}, \widetilde{x}')$ for each $(\widetilde{x}, \widetilde{x}') \in \mathcal{E}^+$ where $\mathcal{E}^+$ is the input of the algorithm.* ☐

5.5 Improvement by Degenerating Negative Values

There is another way to define well-founded relation from the tuple $\widetilde{f}(\widetilde{x}) = (f_k(\widetilde{x}), \ldots, f_0(\widetilde{x}))$ of functions, that is, the well-founded relation $R'_{\widetilde{f}}(\widetilde{x}, \widetilde{x}')$ defined inductively by $R'_{()}(\widetilde{x}, \widetilde{x}') :=\ \perp$ and $R'_{(f_k,\ldots,f_0)}(\widetilde{x}, \widetilde{x}') := f_k(\widetilde{x}) \geq 0 \wedge f_k(\widetilde{x}) > f_k(\widetilde{x}') \vee \big(f_k(\widetilde{x}') < 0 \vee f_k(\widetilde{x}) = f_k(\widetilde{x}')\big) \wedge R'_{(f_{k-1},\ldots,f_0)}(\widetilde{x}, \widetilde{x}')$.

In this definition, we loosen the equality $f_i(\widetilde{x}) = f_i(\widetilde{x}')$ (where $i = 1, \ldots, k$) of the usual lexicographic ordering (10) to $f_i(\widetilde{x}') < 0 \vee f_i(\widetilde{x}) = f_i(\widetilde{x}')$. This means that once $f_i(\widetilde{x})$ becomes negative, $f_i(\widetilde{x})$ must stay negative but the value do not have to be the same, which is useful for the synthesizer to avoid complex candidate lexicographic ranking functions and thus improves the performance.

However, if we use this well-founded relation $R'_{\widetilde{f}}(\widetilde{x}, \widetilde{x}')$ instead of $R_{\widetilde{f}}(\widetilde{x}, \widetilde{x}')$ in (10), then Theorem 17 fails because $R'_{\widetilde{f}}(\widetilde{x}, \widetilde{x}')$ is not necessarily subsumed by $R'_{\widetilde{f}+\widetilde{c}}$ where $\widetilde{c} = (c_k, \ldots, c_0)$ is a nonnegative constant (see the proof of Proposition 16 and Theorem 17). As a result, there is a chance that no implicit cycle can be found in line 14 of Algorithm 1. Therefore, when we use $R'_{\widetilde{f}}(\widetilde{x}, \widetilde{x}')$, we modify Algorithm 1 so that if no implicit cycle can be found in line 14, then we fall back on the former definition of $R_{\widetilde{f}}(\widetilde{x}, \widetilde{x}')$ and restart Algorithm 1.

6 Implementation and Evaluation

Implementation. We implemented a constraint solver MuVal that supports invariant synthesis and ranking function synthesis. For invariant synthesis, we apply an ordinary decision tree learning (see [12,14,18,22,36] for existing techniques). For ranking function synthesis, we implemented the algorithm in Sect. 5 with both eager and lazy strategies for halfspace selection. Our synthesizer uses well-founded relation explained in Sect. 5.5. Given a benchmark, we run our solver for both termination and non-termination verification in parallel, and when one of the two returns an answer, we stop the other and use the answer. MuVal is written in OCaml and uses Z3 as an SMT solver backend. We used clang and llvm2kittel [1] to convert C benchmarks to T2 [3] format files, which are then translated to pwCSP by MuVal.

Experiments. We evaluated our implementation MuVal on C benchmarks from Termination Competition 2020 (C Integer) [4]. We compared our tool with AProVE [10,13], iRankFinder [7], and Ultimate Automizer [21]. Experiments are conducted on StarExec [2] (CentOS 7.7 (1908) on Intel(R) Xeon(R) CPU E5-2609 0 @ 2.40GHz (2393 MHZ) with 263932744 kB main memory). The time limit was 300 s.

Results. Results are shown in Table 2. Yes/No/TO/U means the number of benchmarks that these tools could verify termination/could verify non-termination/could not answer within 300 s and timed out (TimeOut)/gave up before 300 s (Unknown), respectively. We also show scatter plots of runtime in Fig. 7.

Table 2. Numbers of solved benchmarks

	Yes	No	TO	U
MuVal (eager)	204	89	42	0
MuVal (lazy)	200	84	51	0
AProVE	216	100	16	3
iRankFinder	208	92[a]	0	34
Ultimate Automizer	180	83	2	70

[a]We removed one benchmarks from the result of iRankFinder because the answer was wrong.

MuVal was able to solve more benchmarks than Ultimate Automizer. Compared to iRankFinder, MuVal solved slightly fewer benchmarks, but was faster in a large number of benchmarks: 265 benchmarks were solved faster by MuVal, 68 by iRankFinder, and 2 were not solved by both tools within 300 s (here, we regard U (unknown) as 300 s). Compared to AProVE, MuVal solved fewer benchmarks. However, there are several benchmarks that MuVal could solve but AProVE could not. Among them is “TelAviv-Amir-Minimum_true-termination.c”, which does require piecewise affine ranking functions. MuVal found a ranking function $f(x, y) = \textbf{if } x - y \geq 0 \textbf{ then } y \textbf{ else } x$, while AProVE timed out.

We also observed that using CEGIS with transition examples itself showed its strengths even for benchmarks that do not require piecewise affine ranking functions. Notably, there are three benchmarks that MuVal could solve but the other tools could not; they are examples that do not require segmentations. Further analysis of these benchmarks indicates the following strengths of our framework: (1) the ability to handle nonlinear constraints (to some extent) thanks to

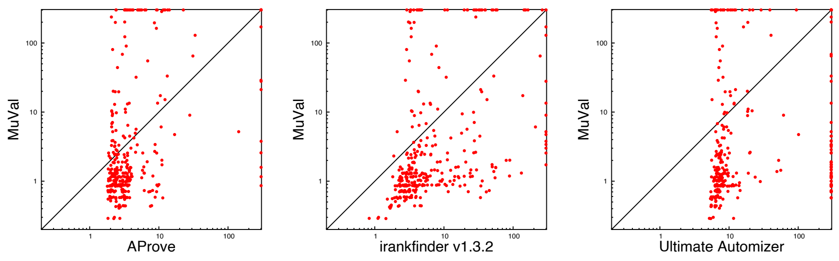


Fig. 7. Scatter plots of runtime. Ultimate Automizer and AProVE sometimes gave up before the time limit, and such cases are regarded as 300s.

the example-based synthesis and the recent development of SMT solvers; and (2) the ability to find a long lasso-shaped non-terminating trace assembled from multiple transition examples. See [23, Appendix A] for details.

7 Related Work

There are a bunch of works that synthesize ranking functions via constraint solving. Among them is a counterexample-guided method like CEGIS [29]. CEGIS is sound but not guaranteed to be complete in general: even if a given constraint has a solution, CEGIS may fail to find the solution. A complete method for ranking function synthesis is proposed in [19]. They collect only extremal counterexamples instead of arbitrary transition examples to avoid infinitely many examples. A limitation of their method is that the search space is limited to (lexicographic) affine ranking functions.

Another counterexample-guided method is proposed in [33] and implemented in SeaHorn. This method can synthesize piecewise affine functions, but their approach is quite different from ours. Given a program, they construct a *safety* property that the number of loop iterations does not exceed the value of a candidate ranking function. The safety property is checked by a verifier. If it is violated, then a trace is obtained as a counterexample and the candidate ranking function is updated by the counterexample. The main difference from our method is that their method uses trace examples while our method uses transition examples (which is less expensive to handle). FreqTerm [15] also uses the connection to safety property, but they exploit syntax-guided synthesis for synthesizing ranking functions.

Aside from counterexample-guided methods, constraint solving is widely studied for affine ranking functions [27], lexicographic affine ranking functions [5,7,24], and multiphase affine ranking functions [6,8]. Their implementation includes RankFinder and iRankFinder. Farkas' lemma or Motzkin's transposition theorem are often used as a tool to transform $\exists\forall$-constraints to $\exists$-constraints. However, when we apply this technique to piecewise affine ranking functions, we get nonlinear constraints [24].

Abstract interpretation is also applied to segmented synthesis of ranking functions and implemented in FuncTion [32,34,35]. In this series of work, decision tree representation of ranking functions is used in [35] for better handling of disjunctions. Compared to their work, we believe that our method is more easily extensible to other theories than linear integer arithmetic as long as the theories are supported by SMT solvers (although such extensions are out of the scope of this paper).

Other state-of-the-art termination verifiers include the following. Ultimate Automizer [21] is an automata-based method. It repeatedly finds a trace and computes a termination argument that contains the trace until termination arguments cover the set of all traces. Büchi automata are used to handle such traces. AProVE [10,13] is based on term rewriting systems.

8 Conclusions and Future Work

In this paper, we proposed a novel decision tree-based synthesizer for ranking functions, which is integrated into the CEGIS architecture. The key observation here was that we need to cope with explicit and implicit cycles contained in given examples. We designed a decision tree learning algorithm using the theoretical observation of the cycle detection theorem. We implemented the framework and observed that its performance is comparable to state-of-the-art termination analyzers. In particular, it solved three benchmarks that no other tool solved, a result that demonstrates the potential of the current combination of CEGIS, segmented synthesis, and transition examples.

We plan to extend our ranking function synthesizer to a synthesizer of piecewise affine ranking supermartingales. Ranking supermartingales [11] are probabilistic version of ranking functions and used for verification of almost-sure termination of probabilistic programs.

We also plan to implement a mechanism to automatically select a suitable set of halfspaces with which decision trees are built. In our ranking function synthesizer, intervals/octagons/octahedron/polyhedra can be used as the set of halfspaces. However, selecting an overly expressive set of halfspaces may cause the problem of overfitting [25] and result in poor performance. Therefore, applying heuristics that adjusts the expressiveness of halfspaces based on the current examples may improve the performance of our tool.

Acknowledgement. We thank Andrea Peruffo and the anonymous referees for many suggestions. This work was supported by JST ERATO HASUO Metamathematics for Systems Design Project (No. JPMJER1603) and JSPS KAKENHI Grant Numbers 20H04162, 20H05703, 19H04084, and 17H01720.

References

1. llvm2KITTeL. https://github.com/hkhlaaf/llvm2kittel
2. StarExec. https://www.starexec.org
3. T2 temporal logic prover. https://github.com/mmjb/T2
4. Termination Competition 2020: C Integer. https://termcomp.github.io/Y2020/job_41519.html
5. Alias, C., Darte, A., Feautrier, P., Gonnord, L.: Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 117–133. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15769-1_8
6. Ben-Amram, A.M., Doménech, J.J., Genaim, S.: Multiphase-linear ranking functions and their relation to recurrent sets. In: Chang, B.-Y.E. (ed.) SAS 2019. LNCS, vol. 11822, pp. 459–480. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32304-2_22
7. Ben-Amram, A.M., Genaim, S.: Ranking functions for linear-constraint loops. J. ACM **61**(4), 1–55 (2014)
8. Ben-Amram, A.M., Genaim, S.: On multiphase-linear ranking functions. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 601–620. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_32
9. Bjørner, N., Gurfinkel, A., McMillan, K., Rybalchenko, A.: Horn clause solvers for program verification. In: Beklemishev, L.D., Blass, A., Dershowitz, N., Finkbeiner, B., Schulte, W. (eds.) Fields of Logic and Computation II. LNCS, vol. 9300, pp. 24–51. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23534-9_2
10. Brockschmidt, M., Ströder, T., Otto, C., Giesl, J.: Automated detection of non-termination and NullPointerExceptions for Java bytecode. In: Beckert, B., Damiani, F., Gurov, D. (eds.) FoVeOOS 2011. LNCS, vol. 7421, pp. 123–141. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31762-0_9
11. Chakarov, A., Sankaranarayanan, S.: Probabilistic program analysis with martingales. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 511–526. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_34
12. Champion, A., Chiba, T., Kobayashi, N., Sato, R.: ICE-based refinement type discovery for higher-order functional programs. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10805, pp. 365–384. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89960-2_20
13. Emmes, F., Enger, T., Giesl, J.: Proving non-looping non-termination automatically. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS (LNAI), vol. 7364, pp. 225–240. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31365-3_19
14. Ezudheen, P., Neider, D., D'Souza, D., Garg, P., Madhusudan, P.: Horn-ICE learning for synthesizing invariants and contracts. Proc. ACM Program. Lang. **2**(OOPSLA), 131:1–131:25 (2018)
15. Fedyukovich, G., Zhang, Y., Gupta, A.: Syntax-guided termination analysis. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 124–143. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_7
16. Floyd, R.W.: Assigning meanings to programs. In: Proceedings of Symposium on Applied Mathematics, vol. 19, pp. 19–32 (1967)
17. Garg, P., Löding, C., Madhusudan, P., Neider, D.: ICE: a robust framework for learning invariants. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 69–87. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_5

18. Garg, P., Neider, D., Madhusudan, P., Roth, D.: Learning invariants using decision trees and implication counterexamples. In: POPL 2016, pp. 499–512. ACM (2016)
19. Gonnord, L., Monniaux, D., Radanne, G.: Synthesis of ranking functions using extremal counterexamples. In: PLDI 2015, pp. 608–618. ACM (2015)
20. Gupta, A., Henzinger, T.A., Majumdar, R., Rybalchenko, A., Xu, R.G.: Proving non-termination. In: POPL 2008, pp. 147–158. ACM (2008)
21. Heizmann, M., Hoenicke, J., Podelski, A.: Termination analysis by learning terminating programs. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 797–813. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_53
22. Krishna, S., Puhrsch, C., Wies, T.: Learning invariants using decision trees. CoRR abs/1501.04725 (2015). http://arxiv.org/abs/1501.04725
23. Kura, S., Unno, H., Hasuo, I.: Decision tree learning in CEGIS-based termination analysis. CoRR abs/2104.11463 (2021). https://arxiv.org/abs/2104.11463
24. Leike, J., Heizmann, M.: Ranking templates for linear loops. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 172–186. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_12
25. Padhi, S., Millstein, T., Nori, A., Sharma, R.: Overfitting in synthesis: theory and practice. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 315–334. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_17
26. Padhi, S., Sharma, R., Millstein, T.D.: Data-driven precondition inference with learned features. In: PLDI 2016, pp. 42–56 (2016)
27. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 239–251. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24622-0_20
28. Satake, Y., Unno, H., Yanagi, H.: Probabilistic inference for predicate constraint satisfaction. In: Proceedings of AAAI 2020 (2020)
29. Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., Saraswat, V.: Combinatorial sketching for finite programs. In: ASPLOS XII, pp. 404–415. ACM (2006)
30. Unno, H., Satake, Y., Terauchi, T., Koskinen, E.: Program verification via predicate constraint satisfiability modulo theories. CoRR abs/2007.03656 (2020). https://arxiv.org/abs/2007.03656
31. Unno, H., Terauchi, T., Koskinen, E.: Constraint-based relational verification. In: CAV 2021. Springer, Cham (2021)
32. Urban, C.: The abstract domain of segmented ranking functions. In: Logozzo, F., Fähndrich, M. (eds.) SAS 2013. LNCS, vol. 7935, pp. 43–62. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38856-9_5
33. Urban, C., Gurfinkel, A., Kahsai, T.: Synthesizing ranking functions from bits and pieces. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 54–70. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_4
34. Urban, C., Miné, A.: An abstract domain to infer ordinal-valued ranking functions. In: Shao, Z. (ed.) ESOP 2014. LNCS, vol. 8410, pp. 412–431. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54833-8_22
35. Urban, C., Miné, A.: A decision tree abstract domain for proving conditional termination. In: Müller-Olm, M., Seidl, H. (eds.) SAS 2014. LNCS, vol. 8723, pp. 302–318. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10936-7_19
36. Zhu, H., Magill, S., Jagannathan, S.: A data-driven CHC solver. In: PLDI 2018, pp. 707–721. ACM (2018)

ATLAS: Automated Amortised Complexity Analysis of Self-adjusting Data Structures

Lorenz Leutgeb[2(✉)], Georg Moser[1], and Florian Zuleger[2]

[1] Department of Computer Science, Universität Innsbruck, Innsbruck, Austria
[2] Institute of Logic and Computation 192/4, Technische Universität Wien, Vienna, Austria
lorenz@leutgeb.xyz

Abstract. Being able to argue about the performance of self-adjusting data structures such as splay trees has been a main objective, when Sleator and Tarjan introduced the notion of *amortised* complexity.

Analysing these data structures requires sophisticated potential functions, which typically contain logarithmic expressions. Possibly for these reasons, and despite the recent progress in automated resource analysis, they have so far eluded automation. In this paper, we report on the first fully-automated amortised complexity analysis of self-adjusting data structures. Following earlier work, our analysis is based on potential function templates with unknown coefficients.

We make the following contributions: 1) We encode the search for concrete potential function coefficients as an optimisation problem over a suitable constraint system. Our target function steers the search towards coefficients that minimise the inferred amortised complexity. 2) Automation is achieved by using a linear constraint system in conjunction with suitable lemmata schemes that encapsulate the required non-linear facts about the logarithm. We discuss our choices that achieve a scalable analysis. 3) We present our tool ATLAS and report on experimental results for *splay trees*, *splay heaps* and *pairing heaps*. We completely automatically infer complexity estimates that match previous results (obtained by sophisticated pen-and-paper proofs), and in some cases even infer better complexity estimates than previously published.

Keywords: Amortised cost analysis · Functional programming · Self-adjusting data structures · Automation · Constraint solving

1 Introduction

Amortised analysis, as introduced by Sleator and Tarjan [47,49], is a method for the worst-case cost analysis of data structures. The innovation of amortised analysis lies in considering the cost of a single data structure operation as part of a sequence of data structure operations. The methodology of amortised analysis allows one to assign a low (e.g., constant or logarithmic) amortised cost to a

A. Silva and K. R. M. Leino (Eds.): CAV 2021, LNCS 12760, pp. 99–122, 2021.
https://doi.org/10.1007/978-3-030-81688-9_5

data structure operation even though the worst-case cost of a single operation might be high (e.g., linear, polynomial or worse). The setup of amortised analysis guarantees that for a sequence of data structure operations the worst-case cost is indeed the number of data structure operations times the amortised cost. In this way amortised cost analysis provides a methodology for worst-case cost analysis. Notably, the cost analysis of self-adjusting data structures, such as splay trees, has been a main objective already in the initial proposal of amortised analysis [47,49]. Analysing these data structures requires sophisticated potential functions, which typically contain logarithmic expressions. Possibly for these reasons, and despite the recent progress in automated complexity analysis, they have so far eluded automation.

In this paper, we present the first fully-automated amortised cost analysis of self-adjusting data structures, that is, of *splay trees*, *splay heaps* and *pairing heaps*, which so far have only (semi-) manually been analysed in the literature. We implement and extend a recently proposed type-and-effect system for amortised resource analysis [26,27]. This system belongs to a line of work (see [20,22–25,28] and the references therein), where types are template potential functions with unknown coefficients and the type-and-effect system extracts constraints over these coefficients in a syntax directed way from the program under analysis. Our work improves over [26,27] in three regards: 1) The approach of [26,27] only supports *type checking*, i.e. verifying that a manually provided type is correct. In this paper, we add an optimisation layer to the set-up of [26,27] in order to support *type inference*, i.e. our approach does not rely on manual annotations. Our target function steers the search towards coefficients that minimise the inferred amortised complexity. 2) The only case study of [26,27] is partial, focusing on the zig-zig case of the splay tree function `splay`, while we report on the full analysis of the operations of several data structures. 3) [26,27] does not report on a fully-automated analysis. Besides the requirement that the user needs to provide the resource annotation, the user also has to apply the structural rules of the type system manually. Our tool ATLAS is able to analyse our benchmarks fully automatically. Achieving full automation required substantial implementation effort as the structural rules need to be applied carefully—as we learned during our experiments—in order to avoid a size explosion of the generated constraint system. We evaluate and discuss our design choices that lead to a scalable implementation.

With our implementation and the obtained experimental results we make two contributions to the complexity analysis of data structures:

1.) We automatically infer complexity estimates that match previous results (obtained by sophisticated pen-and-paper proofs), and in some cases even infer better complexity estimates than previously published. In Table 1, we state the complexity bounds computed by ATLAS next to results from the literature. We match or improve the results from [37,41,42]. To the best of our knowledge, the bounds for splay trees and splay heaps represent the state-of-the-art. In particular, we improve the bound for the `delete` function of splay trees and all bounds for the splay heap functions. For pairing heaps, Iacono [29,30] has proven (using a more involved potential function) that `insert` and `merge` have constant amortised complexity,

Table 1. Amortised complexity bounds for splay trees (module name SplayTree, abbrev. ST), splay heaps (SplayHeap, SH) and pairing heaps (PairingHeap, PH).

Function name	ATLAS (automated)	[42] (manual)[a]	[37] (semi-automated)
`ST.splay`	$3/2 \log_2(\|t\|)$	$3/2 \log_2(\|t\|) + 1$	$3/2 \log_2(\|t\|) + 1$
`ST.splay_max`	$3/2 \log_2(\|t\|)$	-	$3/2 \log_2(\|t\|) + 1$
`ST.insert`	$2 \log_2(\|t\|) + 3/2$	$2 \log_2(\|t\| + 1) + O(1)$	$2 \log_2(\|t\|) + 3/2$
`ST.delete`	$5/2 \log_2(\|t\|) + 3$	$3 \log_2(\|t\| + 1) + O(1)$	$3 \log_2(\|t\|) + 2$
`SH.partition`	$3/4 \log_2(\|t\|) + \log_2(\|t\| + 1)$	-	$2 \log_2(\|t\| + 1) + 1$
`SH.insert`	$3/4 \log_2(\|t\|) + \log_2(\|t\| + 1) + 3/2$	-	$3 \log_2(\|t\| + 2) + 1$
`SH.del_min`	$\log_2(\|t\|)$	-	$2 \log_2(\|t\| + 1) + 1$
`PH.merge_pairs`	$3/2 \log_2(\|h\|)$	-	$3 \log_2(\|h\|) + 4$
`PH.insert`	$1/2 \log_2(\|h\|)$	-	$\log_2(\|h\| + 1) + 1$
`PH.merge`	$1/2 \log_2(\|h_1\| + \|h_2\|) + 1$	$1/2 \log_2(\|h_1\| + \|h_2\|)$	$\log_2(\|h_1\| + \|h_2\| + 1) + 2$
`PH.del_min`	$\log_2(\|h\|)$	$\log_2(\|h\|)$	$3 \log_2(\|h\| + 1) + 4$

[a][42] uses a different cost metric, i.e. the numbers of arithmetic comparisons, whereas we and [37] count the number of (recursive) function applications. We adapted the results of [42] to our cost metric to make the results easier to compare, i.e. the coefficients of the logarithmic terms are by a factor 2 smaller compared to [42].

while the other data structure operations continue to have an amortised complexity of $k \log_2(|t|)$; while we leave an automated analysis based on Iacono's potential function for future work, we note that his coefficients k in the logarithmic terms are large, and that therefore the small coefficients in Table 1 are still of interest. We will detail below that we used a simpler potential function than [37,41,42] to obtain our results. Hence, also the new proofs of the confirmed complexity bounds can be considered a contribution.

2.) We establish a new approach for the complexity analysis of data structures. Establishing the prior results in Table 1 required considerable effort. Schoenmakers studied in his PhD thesis [42] the best amortised complexity bounds that can be obtained using a parameterised potential function $\phi(t)$, where t is a binary tree, defined by $\phi(\texttt{leaf}) := 0$ and $\phi(\texttt{(}l\texttt{,}\ d\texttt{,}\ r\texttt{)}) := \phi(l) + \beta \log_\alpha(|l|+|r|) + \phi(r)$, for real-valued parameters $\alpha, \beta > 0$. Carrying out a sophisticated optimisation with pen and paper, he concluded that the best bounds are obtained by setting $\alpha = \sqrt[3]{4}$ and $\beta = \frac{1}{3}$ for splay trees, and by setting $\alpha = \sqrt{2}$ and $\beta = \frac{1}{2}$ for pairing heaps (splay heaps were proposed only some years later by Okasaki in [38]). Brinkop and Nipkow verify his complexity results for splay trees in the theorem prover Isabelle [37]. They note that manipulating the expressions corresponding to $\beta \log_\alpha(|t|)$ could only partly be automated[1].

[1] Nipkow et al. [37] state "The proofs in this subsection require highly nonlinear arithmetic. Only some of the polynomial inequalities can be automated with Harrison's sum-of-squares method [16]".

For splay heaps, there is to the best of our knowledge no previous attempt to optimise the obtained complexity bounds, which might explain why our optimising analysis was able to improve all bounds. For pairing heaps, Brinkop and Nipkow did not use the optimal parameters reported by Schoenmakers—probably in order to avoid reasoning about polynomial inequalities—, which explains the worse complexity bounds. In contrast to the discussed approaches, we were able to verify and improve the previous results fully automatically. Our approach uses a variation of Schoenmakers' potential function, where we roughly fix $\alpha = 2$ and leave β as a parameter for the optimisation phase (see Sect. 2 for more details). Despite this choice, our approach was able to derive bounds that match and improve the previous results, which came as a surprise to us. Looking back at our experiments and interpreting the obtained results, we recognise that we might have been in luck with the particular choice of the potential function (because we can obtain the previous results despite fixing $\alpha = 2$). However, we would not have expected that an automated analysis is able to match and improve all previously reported coefficients, which shows the power of the optimisation phase. *Thus, we believe that our results suggest a new approach for the complexity analysis of data structures.* So far, self-adjusting data structures had to be analysed manually. This is possibly due to the use of sophisticated potential functions, which may contain logarithmic expressions. Both features are challenging for automated reasoning. Our results suggest that the following alternative (see Sects. 2 and 4.2 for more details): (i) Fix a parameterised potential function; (ii) derive a (linear) constraint system over the function parameters from the AST of the program; (iii) capture the required non-linear reasoning in lemmata, and use Farkas' lemma to integrate the application of these lemmata into the constraint system (in our case two lemmata, one about an arithmetic property and one about the monotonicity of the logarithm, were sufficient for all of our benchmarks); and finally (iv) find values for the parameters by an (optimising) constraint solver. We believe that our approach will carry over to other data structures: one needs to adapt the potential functions and add suitable lemmata, but the overall setup will be the same. We compare the proposed methodology to program synthesis by sketching [48], where the synthesis engineer communicates her main insights to the synthesis engine (in our case the potential functions plus suitable lemmata), and a constraint solver then fills in the details. As conclusion from our benchmarking, we observe that an automated analysis of sophisticated data structures are possible without the need to (i) resort to user guidance; (ii) forfeit optimal results; or (iii) be bogged down in computation times. These results also show how dependencies on properties of functional correctness of the code can be circumvented.

Related Work. To the best of our knowledge the here presented automated amortised analysis of self-adjusting data-structures is novel and unparalleled in the literature. However, there is a vast amount of literature on (automated) resource analysis. Without hope for a completeness, we briefly mention [1–7,9–11,14,15,17,18,20,22–25,39,44–46,52] for an overview of the field. Logarithmic and sublinear bounds are typically not in the focus of the cited approaches, but can be inferred by some tools. In the recurrence relations based approach to cost analysis [1] refinements of linear ranking functions are combined with criteria for divide-and-conquer patterns; this allows the tool PUBS to recognise logarithmic bounds for some problems, but examples such as *mergesort* or *splaying* are beyond the scope of this approach. Logarithmic and exponential terms are integrated into the synthesis of ranking functions in [8], making use of an insightful adaption of Farkas' and Handelman's lemmas. The approach is able to handle examples such as *mergesort*, but again not suitable to handle self-balancing data structures. A type based approach to cost analysis for an ML-like language is presented in [50], which uses the Master Theorem to handle divide-and-conquer-like recurrences. Recently, support for the Master Theorem was also integrated for the analysis of rewriting systems [51], extending [4] on the modular resource analysis of rewriting to so-called logically constrained rewriting systems [12]. The resulting approach also supports the fully automated analysis of *mergesort*.

Structure. In Sects. 2 and 3 we review the type system of [26,27]. We sketch the challenges to automation in Sect. 4 and present our contributions in Sects. 5 and 6. Finally, we conclude in Sect. 7.

2 Step by Step to an Automated Analysis of Splaying

In this and the next section we sketch the theory developed by Hofmann et al. in [27], in order to be able to present the contributions of this article in Sect. 4 and 5. For brevity, we restrict our exposition to those parts essential in the analysis of a particular program code. As motivating example consider *splay trees*, introduced by Sleator and Tarjan [47,49]. *Splaying* is the most important operation on splay trees, which performs rotation. Consider Fig. 1, a depiction of the zig-zig case of `splay`, which implements *splaying*.

The analysis of [27] (see also [26]) is formulated in terms of the physicist's method of amortised analysis in the style of Sleator and Tarjan [47,49]. The central idea of this approach is to assign a *potential* to the data structures of interest such that the difference in potential before and after executing a function is sufficient to pay for the actual cost of the function, i.e. one chooses potential functions ϕ, ψ such that $\phi(v) \geqslant c_f(v) + \psi(f(v))$ holds for all inputs v to a function f, where $c_f(v)$ denotes the *worst-case cost* of executing function f on v. This generalises the original formulation, which can be seen by setting $\phi(v) := a_f(v) + \psi(v)$, where $a_f(v)$ denotes the *amortised cost* of f.

In order to be able to analyse self-adjusting data structures such as splay trees, one needs potential functions that can express *logarithmic* amortised cost. Hofmann et al. [26,27] propose to make use of a variant of Schoenmakers' potential, $\mathsf{rk}(t)$ for a tree t, cf. [37,41,42], defined inductively by

$$\mathsf{rk}(\texttt{leaf}) := 1 \qquad \mathsf{rk}(\texttt{(}l\texttt{,}\ d\texttt{,}\ r\texttt{)}) := \mathsf{rk}(l) + \log_2(|l|) + \log_2(|r|) + \mathsf{rk}(r)\ ,$$

where l, r are the left resp. right child of the tree $\texttt{(}l\texttt{,}\ d\texttt{,}\ r\texttt{)}$, $|t|$ denotes the size of a tree (defined as the number of leaves of the tree), and d is some

```
splay a t = match t with
 | (cl, c, cr) -> match cl with
   | (bl, b, br) -> let s = splay a bl in match s with
     | (al, a', ar) -> (al, a', (ar, b, (br, c, cr)))
```

Fig. 1. Zig-zig case of the `splay` function.

data element that is ignored by the potential function. Besides Schoenmakers' potential, further basic potential functions need to be added to the analysis: For a sequence of m trees $t_1, \ldots, t_m$ and coefficients $a_i, b \in \mathbb{N}$, the potential function

$$p_{(a_1,\ldots,a_m,b)}(t_1,\ldots,t_m) := \log_2(a_1 \cdot |t_1| + \cdots + a_m \cdot |t_m| + b)$$

denotes the logarithm of a linear combination of the sizes of the tree.

Following [37], we set the cost $c_{\texttt{splay}}(t)$ of splaying a tree t to be the number of recursive calls to `splay`. Splaying and all operations that depend on splaying can be done in $O(\log_2 n)$ amortised cost. Employing the above introduced potential functions, the analysis of [27] is able verify the following cost annotation for splaying (the annotation needs to be provided by the user):

$$\mathsf{rk}(t) + 3 \cdot p_{(1,0)}(t) + 1 \geqslant c_{\texttt{splay}}(t) + \mathsf{rk}(\texttt{splay a t}) \ . \tag{1}$$

From this result, one directly reads off $3 \cdot p_{(1,0)}(t) + 1 = 3 \cdot \log_2(|t|) + 1$ as bound on the amortised cost of splaying.[2]

Based on earlier work [6,20,22–25,27,28] employs a *type-and-effect system* that uses *template potential functions*, i.e. functions of a fixed shape with indeterminate coefficients. The key challenge is to identify templates that are suitable for logarithmic analysis and that are closed under the basic operations of the considered programming language. For example, one introduces the coefficients $q_*, q_{(1,0)}, q_{(0,2)}, q'_*, q'_{(1,0)}, q'_{(0,2)}$ and introduces the potential function templates

$$\begin{aligned}\Phi(t\colon \mathsf{T}|Q) &:= q_* \cdot \mathsf{rk}(t) + q_{(1,0)} \cdot p_{(1,0)}(t) + q_{(0,2)} \cdot p_{(0,2)}(t)\\ \Phi(\texttt{splay a t}\colon \mathsf{T}|Q') &:= q'_* \cdot \mathsf{rk}(\texttt{splay a t}) + \\ &\quad + q'_{(1,0)} \cdot p_{(1,0)}(\texttt{splay a t}) + q'_{(0,2)} \cdot p_{(0,2)}(\texttt{splay a t}) \ ,\end{aligned}$$

for the input and output of the `splay` function. The type system then derives constraints on the template function coefficients, as indicated in the sequel. We take up further discussion of the constraint system, in particular how to maintain a scalable analysis, in Sect. 4.

We explain the use of the type system on the motivating example. For brevity, type judgements and the type rules are presented in a simplified form. In particular, we restrict our attention to tree types, denoted as T. This omission is inessential to the actual complexity analysis. For the full set of rules see [27].

[2] For ease of presentation, we elide the underlying semantics for now and simply write "`splay a t`" for the resulting tree t', obtained after evaluating `splay a t`.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\texttt{splay}\colon \mathsf{T}|Q \to \mathsf{T}|Q'}{bl\colon \mathsf{T}|Q \vdash \texttt{splay a bl}\colon \mathsf{T}|Q'-1}\;(\mathsf{app}) \qquad \Delta|R \vdash^{\mathrm{cf}} \texttt{splay a bl}\colon \mathsf{T}|R' \quad cr\colon \mathsf{T}, br\colon \mathsf{T}, s\colon \mathsf{T}|Q_4 \vdash \texttt{match}\ x\ \texttt{with | (}al\texttt{,}a'\texttt{,}ar\texttt{) ->}\ t'\colon \mathsf{T}|Q'}{cr\colon \mathsf{T}, bl\colon \mathsf{T}, br\colon \mathsf{T}|Q_3 \vdash e_1'\colon \mathsf{T}|Q'}\;(\mathsf{let}:\mathsf{T})}{cr\colon \mathsf{T}, bl\colon \mathsf{T}, br\colon \mathsf{T}|Q_2 \vdash e_1'\colon \mathsf{T}|Q'}\;(\mathsf{w})}{cl\colon \mathsf{T}, cr\colon \mathsf{T}|Q_1 \vdash \texttt{match}\ cl\ \texttt{with | (}bl\texttt{,}b\texttt{,}br\texttt{) ->}\ e_1'\colon \mathsf{T}|Q'}\;(\mathsf{match})}{t\colon \mathsf{T}|Q \vdash \texttt{match}\ t\ \texttt{with| (}cl\texttt{,}c\texttt{,}cr\texttt{) ->}\ e_1\colon \mathsf{T}|Q'}\;(\mathsf{match})$$

Fig. 2. Partial typing derivation for the motivating example `splay`.

Let e denote the body of the function definition of `splay a t` , depicted in Fig. 1. Our automated analysis infers an *annotated type* of splaying, by verifying that the type judgement

$$t\colon \mathsf{T}|Q \vdash e\colon \mathsf{T}|Q' \ , \tag{2}$$

is derivable. As above, types are decorated with *annotations* $Q := [q_*, q_{(1,0)}, q_{(0,2)}]$ and $Q' := [q'_*, q'_{(1,0)}, q'_{(0,2)}]$—employed to express the potential carried by the arguments to `splay` and its results.

The soundness theorem of the type system (Theorem 1) expresses that if the above type judgement is derivable, then the total cost $c_{\texttt{splay}}(t)$ of splaying is bound by the difference between $\Phi(t\colon \mathsf{T}|Q)$ and $\Phi(\texttt{splay a t}\colon \mathsf{T}|Q')$, i.e. $\Phi(t\colon \mathsf{T}|Q) \geqslant c_{\texttt{splay}}(t) + \Phi(\texttt{splay a t}\colon \mathsf{T}|Q')$. In particular, Eq. 1 can be derived in this way.

We now provide an intuition on the type-and-effect system, stepping through the code of Fig. 1. The corresponding type derivation tree is depicted in Fig. 2. We note that the tree contains further annotations Q_1, Q_2, Q_3, Q_4 (besides the annotations Q and Q') which again represent the unknown coefficients of potential function templates. The goal of the type-and-effect system is to provide constraints for each programming construct that connect the annotations in subsequent derivation steps, e.g. Q_2 and Q_3. The type-and-effect system operates *syntax-directed* and formulates one rule per programming languages construct. We now discuss some of these rules for the partial derivation for `splay`.

The outermost command of e is a `match` statement, for which the following rule is applied:

$$\dfrac{cl\colon \mathsf{T}, cr\colon \mathsf{T}|Q_1 \vdash e_1\colon \mathsf{T}|Q'}{t\colon \mathsf{T}|Q \vdash \texttt{match}\ t\ \texttt{with | (}cl\texttt{,}c\texttt{,}cr\texttt{) ->}\ e_1\colon \mathsf{T}|Q'}\;(\mathsf{match}) \ .$$

Here e_1 denotes the subexpression of e, which constitutes the nested pattern match. Primarily, this is a standard type rule for pattern matching. The novelty are the constraints on the annotations Q, Q' and Q_1. More precisely, (match) induces the constraints

$$q^1_1 = q^1_2 = q_* \qquad q^1_{(1,1,0)} = q_{(1,0)} \qquad q^1_{(1,0,0)} = q^1_{(0,1,0)} = q_* \qquad q^1_{(0,0,2)} = q_{(0,2)} \ ,$$

which can be directly read-off the definition of $\mathsf{rk}(t) = \mathsf{rk}(cl) + \log_2(|cl|) + \log_2(|cr|) + \mathsf{rk}(cr)$. Similarly, the nested `match` command, starting expression e'_1, is subject to the same rule; the resulting constraints amount to

$$\begin{array}{lll} q^2_1 = q^2_2 = q^2_3 & q^2_{(0,0,0,2)} = q^1_{(0,0,2)} & q^2_{(1,1,1,0)} = q^1_{(1,1,0)} \\ q^2_{(0,1,1,0)} = q^1_{(1,0,0)} & q^2_{(1,0,0,0)} = q^1_{(0,1,0)} & q^2_{(0,1,0,0)} = q^2_{(0,0,1,0)} = q^1_1 \ . \end{array}$$

Besides the rules for programming language constructs, the type-and-effect system contains *structural rules*, which operate on the type annotations themselves. The *weakening* rule allows a suitable adaptation of the coefficients of the potential function $\Phi(\Gamma|Q_2)$ to obtain a new potential function $\Phi(\Gamma|Q_3)$, where we use the shorthand $\Gamma := cr\colon \mathsf{T}, bl\colon \mathsf{T}, br\colon \mathsf{T}$:

$$\frac{\Gamma|Q_3 \vdash e'_1\colon \mathsf{T}|Q' \quad \Phi(\Gamma|Q_2) \geqslant \Phi(\Gamma|Q_3)}{\Gamma|Q_2 \vdash e'_1\colon \mathsf{T}|Q'} \ (\mathsf{w})$$

The difficulty in applying the *weakening* rule, consists in discharging the constraint:

$$\Phi(\Gamma|Q_2) \geqslant \Phi(\Gamma|Q_3) \tag{3}$$

Note, that the comparison is to be performed *symbolically*, that is, abstracted from the concrete value of the variables. We emphasise that this step can neither be avoided, nor easily moved to the axioms of the derivation, as in related approaches in the literature [19, 21–23, 28, 31, 35]. We use Farkas' Lemma in conjunction with two facts about the logarithm to linearise this symbolic comparison, namely the monotonicity of the logarithm and the fact that $2 + \log_2(x) + \log_2(y) \leqslant 2\log_2(x + y)$ for all $x, y \geqslant 1$. For example, for the facts $\log_2(|bl|) \leq \log_2(|bl| + |br|)$ and $2 + \log_2(|bl|) + \log_2(|cr| + |br|) \leq 2\log_2(|cr| + |bl| + |br|)$, we use Farkas' Lemma to generate the constraints

$$\begin{array}{ll} q^2_{(0,0,0,2)} + 2f \geqslant q^3_{(0,0,0,2)} & \\ q^2_{(1,0,1,0)} + \ \ f \geqslant q^3_{(1,0,1,0)} & q^2_{(0,1,0,0)} + f + g \geqslant q^3_{(0,1,0,0)} \\ q^2_{(1,1,1,0)} - 2f \geqslant q^3_{(1,1,1,0)} & q^2_{(0,1,1,0)} \qquad - g \geqslant q^3_{(0,1,1,0)} \end{array}$$

for some coefficients $f, g \geqslant 0$ introduced by Farkas' Lemma. We note that Farkas' Lemma can be interpreted as systematically exploring all positive-linear combinations of the considered mathematical facts. This can be seen on the above example: one can combine g times the first fact with f times the second fact.

Next, we apply the rule for the `let` expression. This rule is the most involved typing rule in the system proposed by Hofmann et al. [27].

$$\frac{\Delta|Q \vdash e_2 : \mathsf{T}|Q' - 1 \quad \Delta|R \vdash^{\mathrm{cf}} e_2 : \mathsf{T}|R' \quad \Theta|Q_4 \vdash e_3 : \mathsf{T}|Q'}{cr : \mathsf{T}, bl : \mathsf{T}, br : \mathsf{T}|Q_3 \vdash \texttt{let}\ s\ \texttt{=}\ e_2\ \texttt{in}\ e_3 : \mathsf{T}|Q'}\ (\mathsf{let} : \mathsf{T})$$

Ignoring the annotations and in particular the second premise for a moment, the type rule specifies a standard typing for a `let` expression. We note that, as required by the rule, all variables in the type context Γ occur at most once in the let-expression. Γ can then be split into contexts $\Delta := bl : \mathsf{T}$ and $\Theta := cr : \mathsf{T}, br : \mathsf{T}$. Here, e_2 := `splay a bl` and e_3 denotes the last `match` statement in e. The let-rule facilitates a splitting of the potential Q_3 for the evaluation of e_2 and e_3 according to the type contexts Δ and Θ. Abusing notation, the distribution of potentials facilitated by the let-rule can be stated very roughly as two "equalities", that is, (i) "$Q_3 = Q + R + P$" and (ii) "$Q_4 = (Q' - 1) + R' + P$". (i) states that the potential Q_3 pays for evaluating the `splay` expression e_2 (with and without costs, requiring the potential Q and R) and leaves the remainder potential P. (ii) states that the potential Q_4 is constituted of the remainder potential P and of the potentials left after evaluating e_2 (with and without costs, i.e. potentials $Q' - 1$ and R'). E.g. Q_4 is given by the following constraints

$$\begin{array}{llll} q^4_1 = q^3_1 & q^4_3 = q'_* & q^4_{(1,0,0,0)} = q^3_{(1,0,0,0)} & q^4_{(1,1,1,0)} = r'_{(1,0)} \\ q^4_2 = q^3_3 & q^4_{(0,1,0,0)} = q^3_{(0,0,1,0)} & q^4_{(1,1,0,0)} = q^3_{(1,0,1,0)} & \end{array}$$

where the coefficients q^3 stem from the remainder potential of Q_3, the coefficient q'_* from $Q' - 1$ and $r'_{(1,0)}$ from R'.

The most original part of this type rule is the second premise $\Delta|R \vdash^{\mathrm{cf}}$ `splay a bl`$: \mathsf{T}|R'$. Here, $\vdash^{\mathrm{cf}}$ denotes the same kind of typing judgement as used in the overall typing derivation, but where all costs are set to zero (hence, the superscript *cost-free*). Let us assume $R = [r_{(1,0)}]$, $R' = [r'_{(1,0)}]$, and that ATLAS was able to establish that

$$\Phi(bl : \mathsf{T}|R) = \log_2(|bl|) \geqslant \log_2(|s|) = \Phi(s : \mathsf{T}|R')\ , \tag{4}$$

establishing the coefficients $r_{(1,0)} = 1$ and $r'_{(1,0)} = 1$. (We note that cost-free typing derivations as in Eq. (4) constitute a *size analysis* that relates the sizes of input and output). Then, ATLAS infers from (4), taking advantage of the monotonicity of log, that

$$\log_2(|cr| + |bl| + |br|) \geqslant \log_2(|cr| + |br| + |s|)\ .$$

This inequality expresses that if the summand $\log_2(|cr| + |bl| + |br|)$ is included in the potential $\Phi(\Gamma|Q_3)$, then the summand $\log_2(|cr| + |br| + |s|)$ may be included in the potential $\Phi(cr : \mathsf{T}, br : \mathsf{T}, s : \mathsf{T}|Q_4)$. (The two logarithmic terms correspond to the coefficients $q^3_{(1,1,1,0)}$ and $q^4_{(1,1,1,0)}$ marked in red above.) Thus, the cost-free derivation allows the potential R to pass from Q_3, via R', to Q_4. This is crucial for being able to pay for the evaluation of e_3.

The let-rule has the three premises $\Delta|Q \vdash e_2 : \mathsf{T}|Q' - 1$, $\Delta|R \vdash^{\text{cf}} e_2 : \mathsf{T}|R'$ and $\Theta|Q_4 \vdash e_3 : \mathsf{T}|Q'$. We focus here on the first premise and do not state the derivations for the other two premises (such derivations can be found in [27]). The judgement $\Delta|Q \vdash$ `splay a t`$: \mathsf{T}|Q' - 1$ can be derived by the rule for function application, which states a cost of 1 with regard to the type signature of `splay`, represented by decrementing the potential induced by the annotation Q'.

$$\frac{\texttt{splay}\colon \mathsf{T}|Q \to \mathsf{T}|Q'}{t\colon \mathsf{T}|Q \vdash \texttt{splay a t}\colon \mathsf{T}|Q' - 1}\ (\mathsf{app})$$

The rule for function application is an axiom, and closes this branch of the typing derivation. This concludes the presentation of the partial type inference given in Fig. 2. Similarly to the above example of `splay`, estimates for the amortised costs of insertion and deletion on splay trees can be automatically inferred by our tool ATLAS. Further, our analysis handles similar self-adjusting data structures like *pairing heaps* and *splay heaps* (see Sect. 6.1).

3 Technical Foundation

In this short section, we provide a more detailed account of the formal system underlying our tool ATLAS. We state the soundness of the system in Theorem 1.

A *typing context* is a mapping from variables $\mathcal{V}$ to types; denoted by uppercase Greek letters. A program P is a set of typed function definitions of the form $f(x_1, \dots, x_n) = e$, where the x_i are variables and e an expression. A *substitution* (or an *environment*) σ is a mapping from variables to values that respects types. Substitutions are denoted as sets of assignments: $\sigma = \{x_1 \mapsto t_1, \dots, x_n \mapsto t_n\}$. We employ a simple cost-sensitive big-step semantics based on eager evaluation, dressed up with cost assertions. The judgement $\sigma \overset{\ell}{\vdash} e \Rightarrow v$ means that under environment σ, expression e is evaluated to value v in exactly ℓ steps. Here only rule applications emit (unit) costs. For brevity, the formal definition of the semantics is omitted but can be found in [27].

In Sect. 2, we introduced a variant of Schoenmakers' potential function, denoted as $\mathsf{rk}(t)$, and the additional potential functions $p_{(a_1,\dots,a_m,b)}(t_1, \dots, t_m) := \log_2(a_1 \cdot |t_1| + \dots + a_m \cdot |t_m| + b)$, denoting the $\log_2$ of a linear combination of tree sizes. $\log_2$ denotes the logarithm to the base 2; throughout the paper we stipulate $\log_2(0) := 0$ in order to avoid case distinctions. Note that the constant function 1 is representable: $1 = \lambda t. \log_2(0 \cdot |t| + 2) = p_{(0,2)}$. We are now ready to state the resource annotation of a sequence of trees:

Definition 1. *A* resource annotation *or simple* annotation *of length m is a sequence $Q = [q_1, \dots, q_m] \cup [(q_{(a_m,\dots,a_n,b)})_{a_i, b \in \mathbb{N}}]$, vanishing almost everywhere.*

Let $t_1, \ldots, t_m$ be a sequence of trees. Then, the potential of $t_1, \ldots, t_m$ wrt. Q is given by

$$\Phi(t_1, \ldots, t_m|Q) := \sum_{i=1}^{m} q_i \cdot \mathsf{rk}(t_i) + \sum_{a_1,\ldots,a_m,b\in\mathbb{N}} q_{(a_1,\ldots,a_m,b)} \cdot p_{(a_1,\ldots,a_m,b)}(t_1, \ldots, t_m) .$$

In case of an annotation of length 1, we sometimes write q_* instead of q_1, as we already did above.

Example 1. Let t be a tree, then its potential could be defined as follows: $\mathsf{rk}(t) + 3 \cdot \log_2(|t|) + 1$. Wrt. the above definition this potential becomes representable by setting $q_* := 1, q_{(1,0)} := 3, q_{(0,2)} := 1$. Thus, $\Phi(t|Q) = \mathsf{rk}(t) + 3 \cdot \log_2(|t|) + 1$. □

Let σ be a substitution, let Γ denote a typing context and let $x_1 : \mathsf{T}, \ldots, x_m : \mathsf{T}$ denote all tree types in Γ. A *resource annotation for* Γ or simply *annotation* is an annotation for the sequence of trees $x_1\sigma, \ldots, x_m\sigma$. We define the *potential* of the annotated context $\Gamma|Q$ wrt. a substitution σ as $\Phi(\sigma; \Gamma|Q) := \Phi(x_1\sigma, \ldots, x_m\sigma|Q)$.

Definition 2. *An* annotated signature $\mathcal{F}$ *maps functions* f *to sets of pairs of the annotation type for the arguments and the annotation type of the result:*

$$\mathcal{F}(f) := \{\alpha_1 \times \cdots \times \alpha_n|Q \to \beta|Q' \colon Q, Q' \text{ are annotations, } Q \text{ is of length } m\}.$$

We suppose f takes n arguments of which m are trees; $m \leqslant n$ by definition.

Instead of $\alpha_1 \times \cdots \times \alpha_n|Q \to \beta|Q' \in \mathcal{F}(f)$, we sometimes succinctly write $f : \alpha_1 \times \cdots \times \alpha_n|Q \to \beta|Q'$. The *cost-free* signature, denoted as $\mathcal{F}^{\mathrm{cf}}$, is similarly defined.

Example 2. Consider the function `splay` from above. Its signature is formally represented as $\mathsf{B} \times \mathsf{T}|Q \to \mathsf{T}|Q'$, where $Q := [q_*] \cup [(q_{(a,b)})_{a,b\in\mathbb{N}}]$ and $Q' := [q'_*] \cup [(q'_{(a,b)})_{a,b\in\mathbb{N}}]$. We leave it to the reader to specify the coefficients in Q, Q' so that the rule (app) as depicted in Sect. 2 can indeed by employed to type the recursive call of `splay`.

Let $Q = [q_*] \cup [(q_{(a,b)})_{a,b\in\mathbb{N}}]$ be an annotation such that $q_{(a,b)} > 0$. Then $Q' := Q - 1$ is defined as follows: $Q' = [q_*] \cup [(q'_{(a,b)})_{a,b\in\mathbb{N}}]$, where $q'_{(0,2)} := q_{(0,2)} - 1$ and for all $(a, b) \neq (0, 2)$ $q'_{(a,b)} := q_{(a,b)}$. By definition the annotation coefficient $q_{(0,2)}$ is the coefficient of the basic potential function $p_{(0,2)}(t) = \log_2(0|t|+2) = 1$, so the annotation $Q - 1$, decrements cost 1 from the potential induced by Q.

Type-and-Effect System. The typing system makes use of a *cost-free* semantics, which does not attribute any costs to the calculation. I.e. the rule (app) (Sect. 2) is changed so that no cost is emitted. The cost-free application rule is denoted as ($\mathsf{app} : \mathsf{cf}$). The cost-free typing judgement is written as $\Gamma|Q \vdash^{\mathrm{cf}} e : \alpha|Q'$. The judgement $\Gamma|Q \vdash e : \alpha|Q'$ is governed by a plethora of typing rules. We have illustrated several typing rules in Sect. 2 (the complete set of typing rules can be found in [27]).

A program P is called *well-typed* if for any rule $f(x_1,\ldots,x_k) = e \in \mathsf{P}$ and any annotated signature $f\colon \alpha_1 \times \cdots \times \alpha_k|Q \to \beta|Q'$, we have $x_1\colon\alpha_1,\ldots,x_k\colon\alpha_k|Q \vdash e\colon\beta|Q'$. A program P is called *cost-free* well-typed, if the cost-free typing relation is employed.

Hofmann et al. establish the following soundness result:[3]

Theorem 1 (Soundness Theorem). *Let* P *be well-typed and let* σ *be an environment. Suppose* $\Gamma|Q \vdash e\colon\alpha|Q'$ *and* $\sigma \overset{\ell}{\vdash} e \Rightarrow v$*. Then* $\Phi(\sigma;\Gamma|Q) - \Phi(v|Q') \geqslant \ell$*. Further, if* $\Gamma|Q \vdash^{cf} e\colon\alpha|Q'$*, then* $\Phi(\sigma;\Gamma|Q) \geqslant \Phi(v|Q')$*.*

4 The Road to Automation, Continued

The above sketched type-and-effect system, originally proposed in [27], is only a first step towards full automation. Several challenges need to be overcome, which we detail in this section.

4.1 Type Checking

Comparison between logarithmic expressions, constitutes a first major challenge, as such a comparison cannot be directly encoded as a *linear* constraint problem. To achieve such *linearisation*, [27] makes use of the following: (i) a subtly and surprisingly effective variant of Schoenmakers potential (see Sect. 2); (ii) mathematical facts about the logarithm function—like Lemma 1 below—referred to as *expert knowledge*; and finally (iii) Farkas' Lemma for turning the universally-quantified premise of the weakening rule into an existentially-quantified statement that can be added to the constraint system—see Lemma 2.

A simple mathematical fact that is employed by Hofmann et al.— following earlier pen-and-paper proofs in the literature [37,38,41]—states as follows:

Lemma 1. *Let* $x, y \geqslant 1$*. Then* $2 + \log_2(x) + \log_2(y) \leqslant 2\log_2(x+y)$*.*

We remark that our automated analysis shows that this lemma is not only crucial in the analysis of splaying, but also for the other data structures we have investigated. Further, Hofmann et al. state and prove the following variant of Farkas' Lemma, which lies at the heart of an effective transformation of comparison demands like (3) into a linear constraint problem. Note that $\vec{u}$ and $\vec{f}$ denote column vectors of suitable length.

Lemma 2 (Farkas' Lemma). *Suppose* $A\vec{x} \leqslant \vec{b}, \vec{x} \geqslant 0$ *is solvable. Then the following assertions are equivalent. (i)* $\forall \vec{x} \geqslant 0.\ A\vec{x} \leqslant \vec{b} \Rightarrow \vec{u}^T\vec{x} \leqslant \lambda$ *and (ii)* $\exists \vec{f} \geqslant 0.\ \vec{u}^T \leqslant \vec{f}^T A \wedge \vec{f}^T\vec{b} \leqslant \lambda$*.*

[3] Note that soundness assumes a terminating execution $\sigma \overset{\ell}{\vdash} e \Rightarrow v$ of P. We point out that our analysis does not guarantee the termination of P for all environments σ.

The lemma allows the assumption of *expert knowledge* through the assumption $A\vec{x} \leqslant \vec{b}$ for all $\vec{x} \geqslant 0$. E.g., thus formalised expert knowledge is a clear point of departure for additional information. E.g. Hofmann et al. [27] propose the following potential extensions: (i) additional mathematical facts on the log function; (ii) a dedicated size analysis; (iii) incorporation of basic static analysis techniques. The incorporation of Farkas' Lemma with suitable expert knowledge is already essential for *type checking*, whenever the symbolic weakening rule (3) needs to be discharged.

ATLAS incorporates two facts into the expert knowledge: Lemma 2 and the monotonicity of the logarithm (see Sect. 5). We found these two facts to be sufficient for handling our benchmarks, i.e. expert knowledge of form (ii) and (iii) was not needed. (We note though that we have experimented with adding a dedicated size analysis (ii), which interestingly increased the solver performance, despite generating a large constraint system).

We indicate how ATLAS may be used to solve the constraints generated for the example in Sect. 2. We recall the crucial application of the *weakening* step between annotations Q_2 and Q_3. This weakening step can be automatically discharged using the monotonicity of logs and Lemma 1. (More precisely, ATLAS employs the mode `w{mono l2xy}` see, Sect. 5.) For example, ATLAS is able to verify the validity of the following concrete constants:

$$\begin{array}{lll@{\qquad}lll}
Q_2\colon q^2_1 = q^2_2 = q^2_3 = 1 & & & Q_3\colon q^3_1 = q^3_2 = q^3_3 = 1 & & \\
\quad q^2_{(0,0,0,2)} = 1 & q^2_{(0,1,1,0)} = 1 & & \quad q^3_{(0,0,0,2)} = 2 & q^3_{(1,0,0,0)} = 1 \\
\quad q^2_{(0,0,1,0)} = 1 & q^2_{(1,0,0,0)} = 1 & & \quad q^3_{(0,0,1,0)} = 1 & q^3_{(1,0,1,0)} = 1 \\
\quad q^2_{(0,1,0,0)} = 1 & q^2_{(1,1,1,0)} = 3 & & \quad q^3_{(0,1,0,0)} = 3 & q^3_{(1,1,1,0)} = 1
\end{array}$$

4.2 Type Inference

We extend the type-and-effect system of [27] from type checking to type inference. Further, we automate the application of structural rules like *sharing* or *weakening*, which have so far required user guidance.

The two central contributions of this paper, as delineated in the introduction, are based on significant improvement over the state-of-the-art as described above. Concretely, they came about by a novel (i) *optimisation layer*; (ii) a careful control of the *structural rules*; (iii) the generalisation of user-defined *proof tactics* into an overall strategy of type inference; and (iv) provision of an automated amortised analysis in the sense of Sleator and Tarjan. In the sequel of the section, we will discuss these stepping stones towards full automation in more details.

Optimisation Layer. We add an optimisation layer to the set-up, in order to support *type inference*. This allows for the inference of (optimal) type annotations based on user-defined type annotations. For example, assume the user-provided type annotation $\mathsf{rk}(t) + 3\log_2(|t|) + 1 \to \mathsf{rk}(\mathtt{splay}(t))$ can in principle be checked automatically. Then—instead of checking this annotation—ATLAS automatically *optimises* the signature, by minimising the deduced coefficients.

```
(match (* t *) leaf
  (match (* cl *) ?
    (w{l2xy} (let:tree:cf (* s *)
      app (* splay_eq a bl *)
      (match leaf
        (let:tree:cf node (let:tree:cf node (w{mono} node))))))))
```

Fig. 3. Tactic that matches the zig-zig case of `splay` as shown in Fig. 1.

(In Sect. 5 we discuss how this optimisation step is performed.) That is, ATLAS reports the following annotation

$$\texttt{splay}\colon {}^{1}\!/_{2}\,\mathsf{rk}(t) + {}^{3}\!/_{2}\log_2(|t|) \to {}^{1}\!/_{2}\,\mathsf{rk}(\texttt{splay}(t)) ,$$

which yields the *optimal* amortised cost of splaying of ${}^{3}\!/_{2}\log_2(|t|)$. Optimality here means that no better bound has been obtained by earlier pen-and-paper verification methods (compare the discussion in Sect. 1).

Structural Rules. We observed that an unchecked application of the structural rules, that is of the *sharing* and the *weakening* rule, quickly leads to an explosion of the size of the constraint system and thus to de-facto unsolvable problems. To wit, an earlier version of our implementation ran continuously for *24/7* without being able to infer a type for the complete definition of the function `splay`.[4]

The type-and-effect system proposed by Hofmann et al. is in principle *linear*, that is, variables occur at most once in the function body. For example, this is employed in the definition of the let-rule, cf. Sect. 2. However, a *sharing* rule is admissible, that allows to treat multiple occurrences of variables. Occurrences of non-linear variables are suitably renamed apart and the carried potential is shared among the variants. (See [27] for the details.) The number of variables strongly influences the size of the constraint problem. Hence, eager application of the sharing rule proved infeasible. Instead, we restricted its application to individual program traces. For the considered benchmark examples, this removed the need for sharing altogether.

With respect to *weakening*, a careful application of the weakening rule proved necessary for performance reasons: First, we apply weakening only selectively. Second, when applying weakening, we employ different levels of *granularity*. We may only perform a simple coefficient comparison, or we may apply monotonicity or Lemma 1 or both in conjunction with Farkas' Lemma. We give the details in Sect. 5.

Proof Tactics. Hofmann et al. [27] already propose user-defined proof plans, so-called *tactics*, to improve the effectivity of type checking. In combination with our optimisation framework, tactics allow to significantly improve type annotations. To wit, ATLAS can be invoked with user-defined resource annotations for the function `splay`, representing its "standard" amortised complexity (e.g. copied from Okasaki's book [38]) and an easily definable tactic, cf. Fig. 3.

[4] The code ran single-threaded on AMD® Ryzen 7 3800 @ 3.90 GHz.

Then, ATLAS automatically derives the optimal bound reported above. Still, for full-automation tactics are clearly not sufficient. In order to obtain *type inference* in general, we developed a generalisation of all the tactics that proved useful on our benchmark and incorporated this proof search strategy into the type inference algorithm. Using this, the aforementioned (unsuccessful) week-long quest for a type inference of splaying can now be successfully answered (in an optimal form) in mere minutes.

We'd like to argue that ATLAS proof search strategy for full automation is free of bias towards the provided complexity analysis. As detailed in Sect. 5, the heuristics incorporates common design principles of the data structures analysed. Thus, we exploit recurring patterns in the input (destructuring of input trees, handling base/recursive cases, rotations) not in the solution. The situation is similar to the choice of the potential functions, which we expect to generalise to other data structures. Similarly, we expect generalisability of the current proof search strategy.

Automated Amortised Analysis. In Sect. 2, we provided a high-level introduction into the potential method and remarked that Sleator and Tarjan's original formulation is re-obtained, if the corresponding potential functions are defined such that $\phi(v) := a_f(v) + \psi(x)$, see page 5. We now discuss how we can extract amortised complexities in the sense of Sleator and Tarjan from our approach. Suppose, we are interested in an amortised analysis of splay heaps. Then, it suffices to equate the right-hand sides of the annotated signatures of the splay heap functions. That is, we set `del_min`: $\mathsf{T}|Q_1 \to \mathsf{T}|Q'$, `insert`: $\mathsf{B} \times \mathsf{T}|Q_2 \to \mathsf{T}|Q'$ and `partition`: $\mathsf{B} \times \mathsf{T}|Q_3 \to \mathsf{T}|Q'$ for some unknown resource annotations Q_1, Q_2, Q_3, Q'. Note that we use the same annotation Q' for all signatures. We can then obtain a potential function from the annotation Q' in the sense of Sleator and Tarjan and deduce $Q_i - Q'$ as an upper bound on the amortised complexity of the respective function. In Sect. 5, we discuss how to automatically optimise $Q_i - Q'$ in order to minimise the amortised complexity bound. This automated minimisation is the second major contribution of our work. Our results suggest a new approach for the complexity analysis of data structures. On the one hand, we obtain novel insights into the automated worst-case runtime complexity analysis of involved programs. On the other hand, we provide a proof-of-concept of a computer-aided analysis of amortised complexities of data-structures that so far have only been analysed manually.

5 Implementation

In this section, we present our tool ATLAS, which implements type inference for the type system presented in Sects. 2 and 3. ATLAS operates in three phases:

1.) *Preprocessing*, ATLAS parses and normalises the input program;
2.) *Generation of the Constraint System*, ATLAS extracts constraints from the normalised program according to the typing rules (as sketched in Sect. 2);
3.) *Solving*, the derived constraint system is handed to an optimising constraint solver and the solver output is converted into a type annotation.

```
LNF[if a<a'
  then (l,a,(leaf,a',r))
  else ((l,a',leaf),a,r)]
```

```
let x1 = a<a' in if x1
  then LNF[(l,a,(leaf,a',r))]
  else LNF[((l,a',leaf),a,r)]
```

```
let x1 = a < a' in if x1
  then let x2 = leaf in let x3 = (x2,a',r) in (l,a,x3)
  else let x4 = leaf in let x5 = (l,a',x4) in (x5,a,r)
```

Fig. 4. Preprocessing: let normal forms.

In terms of overall resource requirements, the bottleneck of the system is phase three. Preprocessing is both simple and fast. While the code implementing constraint generation might be complex, its execution is fast. All of the underlying complexity is shifted into the third phase. On modern machines with multiple gibibytes of main memory, ATLAS is constrained by the CPU, and not by the available memory. In the remainder of this section, we first detail these phases of ATLAS. We then go into more details of the second phase. Finally, we elaborate the optimisation function which is the key enabler of type inference.

5.1 The Three Phases of ATLAS

1.) Preprocessing. The parser used in the first phase is generated with ANTLR[5] and transformation of the syntax is implemented in Java. The preprocessing performs two tasks: (i) Transformation of the input program into *let-normal-form*, which is the form of program input required by our type system. (ii) The *unsharing* conversion creates explicit copies for variables that are used multiple times. Making multiple uses of a variables explicit is required by the let-rule of the type system.

In order to satisfy the requirement of the let-rule, it is actually sufficient to track variable usage on the level of program paths. It turns out that in our benchmarks variables are only used multiple times in different branches of an if-statement, for which no unsharing conversion is needed. Hence, we do not discuss the unsharing conversion further in this paper and refer the interested reader to [27] for more details.

Let-Normal-Form Conversion. The let-normal-form conversion is performed recursively and rewrites composed expressions into simple expressions, where each operator is only applied to a variable or a constant. This conversion is achieved by introducing additional let-constructs. We exemplify let-normal-form conversion on a code snippet in Fig. 4.

2.) Generation of the Constraint System. After preprocessing, we apply the typing rules. Importantly, the application of all typing rules, except for the weakening rule, which we discuss in further detail below, is *syntax-directed*: This means

[5] See antlr.org.

that each node of the AST of the input program dictates which typing rule is to be applied. The weakening rule could in principle be applied at each AST node, giving the constraint solver more freedom to find a solution. This degree of freedom needs to be controlled by the tool designer. In addition, recall that the suggested implementation of the weakening rule (see Sect. 4.1) is to be parameterised by the expert knowledge, fed into the weakening rule. In our experiments we noticed that the weakening rule has to be applied sparingly in order to avoid an explosion of the resulting constraint system.

We summarise the degrees of freedom available to the tool designer, which can be specified as parameters to ATLAS on source level. 1.) The selected template potential functions, i.e. the family of indices $\vec{a}, b$ for which coefficients $q_{(\vec{a},b)}$ are generated (we assume not explicitly generated are set to zero). 2.) The number of annotated signatures (with costs and without costs) for each function. 3.) The policy for applying the (parameterised) weakening rule.

We detail our choices for instantiating the above degrees of freedom in Sect. 5.2.

3.) Solving. For solving the generated constraint system, we rely on the Z3 SMT solver. We employ Z3's Java bindings, load Z3 as a shared library, and exchange constraints for solutions. ATLAS forwards user-supplied configuration to Z3, which allows for flexible tuning of solver parameters. We also record Z3's statistics, most importantly memory usage. During the implementation of ATLAS, Z3's feature to extract unsatisfiable cores has proven valuable. It supplied us with many counterexamples, often directly pinpointing bugs in our implementation. The tool exports constraint systems in SMT-LIB format to the file system. This way, solutions could be cross-checked by re-computing them with other SMT solvers that support minimisation, such as OptiMathSAT [43].

5.2 Details on the Generation of the Constraint System

We now discuss our choices for the aforementioned degrees of freedom.

Potential Function Templates. Following [27], we create for each node in the AST of the considered input program, where n variables of tree-type are currently in context, the coefficients $q_1, \ldots, q_n$ for the rank functions and the coefficients $q_{(\vec{a},b)}$ for the logarithmic terms, where $\vec{a} \in \{0,1\}^n$ and $b \in \{0,2\}$. This choice turned out to be sufficient in our experiments.

Number of Function Signatures. We fix the number of annotations for each function $f : \alpha_1 \times \cdots \times \alpha_n | Q \to \beta | Q'$ to one regular and one cost-free signature. This was sufficient for our experiments.

Weakening. We need to discharge symbolic comparisons of form $\Phi(\Gamma|P) \leqslant \Phi(\Gamma|Q)$. As indicated in Sect. 4, we

$$\begin{tikzcd}
 & x_{(1,1,2)} \arrow[dl] \arrow[d] \arrow[dr] & \\
x_{(0,1,2)} \arrow[d] \arrow[dr] & x_{(1,0,2)} \arrow[dl] \arrow[dr] & x_{(1,1,0)} \arrow[dl] \arrow[d] \\
x_{(0,0,2)} \arrow[dr] & x_{(0,1,0)} \arrow[d] & x_{(1,0,0)} \arrow[dl] \\
 & x_{(0,0,0)} &
\end{tikzcd}$$

Fig. 5. Monotonicity Lattice for $|Q| = 2$.

employ Farkas' Lemma to derive constraints for the weakening rule. For context $\Gamma = t_1, \ldots, t_n$, we introduce variables $x_{(\vec{a},b)}$ where $\vec{a} \in \{0,1\}^n, b \in \{0,2\}$, which represent the potential functions $p_{(\vec{a},b)} = \log_2(a_1|t_1| + \ldots + a_n|t_n| + b)$. Next, we explain how the monotonicity of $\log_2$ and Lemma 1 can be used to derive inequalities on the variables $x_{(\vec{a},b)}$, which can then be used to instantiate matrix A in Farkas' Lemma as stated in Sect. 4.

Monotonicity. We observe that $p_{(\vec{a},b)} = \log_2(a_1|t_1| + \ldots + a_n|t_n| + b) \leqslant \log_2(a'_1|t_1| + \ldots + a'_n|t_n| + b') = p_{(\vec{a}',b')}$, if $a_1 \leqslant a'_1, \ldots, a_n \leqslant a'_n$ and $b \leqslant b'$. This allows us to obtain the lattice shown in Fig. 5. A path from $x_{(\vec{a}',b')}$ to $x_{(\vec{a},b)}$ signifies $x_{(\vec{a},b)} \leqslant x_{(\vec{a}',b')}$ resp. $x_{(\vec{a},b)} - x_{(\vec{a}',b')} \leqslant 0$, represented by a row with coefficients 1 and -1 in the corresponding columns of matrix A.

Mathematical Facts, Like Lemma 1. For an annotated context of length 2, Lemma 1 can be stated by the inequality $2x_{(0,0,2)} + x_{(0,1,0)} + x_{(1,0,0)} - 2x_{(1,1,0)} \leqslant 0$; we add a corresponding row with coefficients $2, 1, 1, -2$ to the matrix A. Likewise, for contexts of length > 2, we add, for each subset of 2 variables, a row with coefficients $2, 1, 1, -2$, setting the coefficients of all other variables to 0.

Sparse Expert Knowledge Matrix. We observe for both kinds of constraints that matrix A is sparse. We exploit this in our implementation and only store non-zero coefficients.

Parametrisation of Weakening. Each applications of the weakening rule is parameterised by the matrix A. In our tool, we instantiate A with either the constraints for (i) monotonicity, shortly referenced as `w{mono}`; (ii) Lemma 1 (`w{l2xy}`); (iii) both (`w{mono l2xy}`); or (iv) none of the constraints (`w`).

In the last case, Farkas' Lemma is not needed because weakening defaults to point-wise comparison of the coefficients $p_{(\vec{a},b)}$, which can be implemented more directly. Each time we apply weakening, we need to choose how to instantiate matrix A. Our experiments demonstrate that we need to apply monotonicity and Lemma 1 sparingly in order to avoid blowing up the constraint system.

Tactics and Automation. ATLAS supports manually applying the weakening rule—for this the user has to provide a tactic—and a fully-automated mode.

Naive Automation. Our first attempt to automation applied the weakening rule everywhere instantiated with the full amount of available expert knowledge. This approach did not scale.

Manual Mode via Tactics. A tactic is given as a text file that contains a tree of rule names corresponding to the AST nodes of the input program, into which the user can insert applications of the weakening rule, parameterised by the expert knowledge which should be applied. A simple tactic is depicted in Fig. 3. Tactics are distributed with ATLAS, see [32]. The user can name sub-trees for reference in the result of the analysis and include ML-style comments in the tactics text. We provide two special commands that allow the user to directly deal with a whole branch of the input program: The question mark (`?`) allows partial proofs; no constraints will be created for the part of the program thus marked. The underscore (`_`) switches to the naive automation of ATLAS and will apply the weakening rule with full expert knowledge everywhere. Both, `?` and `_`, were invaluable when developing and debugging the automated mode. We note that the manual mode still achieves solving times that are by a magnitude faster than the automated mode, which may be of interest to a user willing to hand-optimise solving times.

Automated Mode. For automation, we extracted common patterns from the tactics we developed manually: Weakening with mode `w{mono}` is applied before (var) and (leaf), `w{mono l2xy}` is applied only before (app). (We recall that the full set of rules employed by our analysis can be found in [27].) Further, for AST subtrees that construct trees, i.e. which only consist of (node), (var) and (leaf) rule applications, we apply `w{mono}` for each inner node, and `w{l2xy}` for each outermost node. For all other cases, no weakening is applied. This approach is sufficient to cover all benchmarks, with further improvements possible.

5.3 Optimisation

Given an annotated function $f\colon \alpha_1 \times \cdots \times \alpha_n | Q \to \beta | Q'$, we want to find values for the coefficients of the resource annotations Q and Q' that minimise $\Phi(\Gamma|Q) - \Phi(\Gamma|Q')$, since this difference is an upper bound on the amortised cost of f, cf. Sect. 4.2. However, as with weakening, we cannot directly express such a minimisation, and again resort to linearisation: We choose an optimisation function that directly maps from Q and Q' to $\mathbb{Q}$. Our optimisation function combines four measures, three of which involve a difference between coefficients of Q and Q', and a fourth one that only involves coefficients from Q in order to minimise the absolute values of the discovered coefficients. We first present these measures for the special case of $|Q| = 1$.

The first measure $d_1(Q, Q') := q_* - q'_*$ reflects our goal of preserving the coefficient for rk; note that for $d_1(Q, Q') \neq 0$, the resulting complexity bound would be super-logarithmic. The second measure $d_2(Q, Q') := \sum_{(a,b)} (q_{(a,b)} - q'_{(a,b)}) \cdot w(a, b)$ reflects the goal of achieving logarithmic bounds that are as small as possible. Weights are defined to penalise more complex terms, and to exclude constants. (Recall that 1 is representable as $\log_2(0 + 2)$.) We set

$$w(a, b) := \begin{cases} 0, & \text{for } (a, b) = (0, 2), \\ (a + (b + 1)^2)^2, & \text{otherwise.} \end{cases}$$

The third measure $d_3(Q, Q') := q_{(0,2)} - q'_{(0,2)}$ reflects the goal of minimising constant cost. Lastly, we set $d_4(Q, Q') := \sum_{(a,b)} q_{(a,b)}$ in order to obtain small absolute numbers. The last measure does not influence bounds on the amortised cost, but leads to more beautiful solutions. These measures are then composed to the linear objective function $\min \sum_{i=1}^{4} d_i(Q, Q') \cdot w_i$. In our implementation, we set $w_i = [16127, 997, 97, 2]$; these weights are chosen (almost) arbitrary, we only noticed that w_1 must be sufficiently large to guarantee its priority. (We note that these weights were sufficient for our experiments; we refer to the literature for more principled ways of choosing the weights of an aggregated cost function [34].)

Multiple Arguments. For $|Q| > 1$, we set $d_1 := \sum_{i=1}^{|Q|} q_i - q'_*$ and $d_2(Q, Q') := \sum_{(a,a,\ldots,b)}(q_{(a,a,\ldots,b)} - q'_{(a,b)}) \cdot w(a,b)$. The required changes for d_3 and d_4 are straight-forward. In our benchmarks, there is only one function (`merge` of pairing heaps) that requires this minimisation function.

6 Evaluation

We first describe the benchmark functions employed to evaluate ATLAS and then detail this experimental evaluation, already depicted in Table 1.

6.1 Automated Analysis of Splaying et al.

Splay Trees. Introduced by Sleator and Tarjan [47,49], *splay trees* are self-adjusting binary search trees with strictly increasing in-order traversal, but without an explicit balancing condition. Based on splaying, searching is performed by splaying with the sought element and comparing to the root of the result. Similarly, insertion and deletion are based on splaying. Above we used the zig-zig case of splaying, depicted in Fig. 1 as motivating code example. While the pen-and-paper analysis of this case is the most involved, type inference for this case alone did not directly yield the desired automation of the complete definition. Rather, full automation required substantial implementation effort, as detailed in Sect. 5. As already emphasised, it came as a surprise to us that our tool ATLAS is able match up and partly improve upon the sophisticated optimisations performed by Schoenmakers [41,42]. This seems to be evidence of the versatility of the employed potential functions. Further, we leverage the sophistication of our optimisation layer in conjunction with the current power of state-of-the-art constraint solvers, like Z3 [36].

Splay Heaps. To overcome deficiencies of splay trees when implemented functionally, Okasaki introduced *splay heaps*. Splay heaps are defined similarly to splay trees and their (manual) amortised cost analysis follows similar patterns as the one for splay trees. Due to the similarity in the definitions between splay heaps and splay trees, extension of our experimental results in this direction did not pose any problems. Notably, however, ATLAS improves the known complexity bounds on the amortised complexity for the functions studied. We also remark that typical assumptions made in pen-and-paper proofs are automatically discharged by our approach: Schoenmakers [41,42] as well as Nipkow and Brinkop [37] make use of the (obvious) fact that the size of the resulting tree t' or heap h' equals the size of the input. As discussed, this information is captured by a cost-free derivation, cf. Sect. 2.

Pairing Heaps. These are another implementation of heaps, which are represented as binary trees, subject to the invariant that they are either `leaf`, or the right child is `leaf`, respectively. The left child is conceivable as list of pairing heaps. Schoenmakers and Nipkow et al. provide a (semi-)manual

Table 2. Experimental results

Function	Proof / (w)	automated (naive)		automated (improved)		manual	
`ST.splay` (zig-zig)	Selective	n/a		7718	18S	2552	<1S
	All	11792	45S	9984	19S	2864	<1S
`ST.splay`	Selective	n/a		42095	8M1S	19111	12S
	All	68103	t/o 24H	54377	14M19S	23323	1M27S
`SH.partition`	Selective	n/a		33729	7M9S	15213	6S
	All	51995	t/o 24H	43549	15M2S	16829	10S
`PH.merge_pairs`	Selective	n/a		25860	1M3S	6414	<1S
	All	43515	t/o 24H	34918	13M41S	6558	<1S

(a) Comparison of the number of constraints generated and time taken for the type inference of the core operation of each benchmark plus the zig-zig case of `splay`.

Module	automated			manual		
	Assertions	Time	Memory	Assertions	Time	Memory
`ST`	54794	24M17S	3204	24677	43S	280
`SH`	37911	7M35S	1482	17877	12S	237
`PH`	29493	3M42S	760	7987	1S	29

(b) Number of assertions, solving time and maximum memory usage (in mebibytes) for the combined analysis of functions per-module.

analysis of pairing heaps, that ATLAS can verify or even improve fully-automatically. We note that we analyse a single function `merge_pairs`, whereas [37] breaks down the analysis and studies two functions `pass_1` and `pass_2` with `merge_pairs` $=$ `pass_2` $\circ$ `pass_1`. All definitions can be found at [33].

6.2 Experimental Results

Our main results have already been stated in Table 1 of Sect. 1. Table 2a compares the differences between the "naive automation" and our actual automation ("automated mode"), see Sect. 5. Within the latter, we distinguish between a "selective" and a "full" mode. The "selective" mode is as described on page 18. The "full" mode employs weakening for the same rule applications as the "selective" mode, but always with option `w{mono l2xy}`. The same applies to the "full" manual mode. The naive automation does not support selection of expert knowledge. Thus the "selective" option is not available, denoted as "n/a". Timeouts are denoted by "t/o". As depicted in the table, the naive automation does not terminate within 24 h for the core operations of the three considered data structures, whereas the improved automated mode produces optimised results within minutes. In Table 2b, we compare the (improved) automated mode with the manual mode, and report on the sizes of the resulting constraint system and on the resources required to produce the same results. Observe that even though our automated mode achieves reasonable solving times, there is still a significant gap between the manually crafted tactics and the automated mode, which invites future work.

7 Conclusion

In this paper we have for the first time been able to automatically conduct an amortised analysis for self-adjusting data structures. Our analysis is based on the "sum of logarithms" potential function and we have been able to automate reasoning about these potential functions by using Farkas' Lemma for the linear part of the calculations and adding necessary facts about the logarithm. Immediate future work is concerned with replacing the "sum of logarithms" potential function in order to analyse skew heaps and Fibonacci heaps [42]. In particular, the potential function for skew heaps, which counts "right heavy" nodes, is interesting, because it is also used as a building block by Iacono in his improved analysis of pairing heaps [29,30]. Further, we envision to extend our analysis to related probabilistic settings such as priority queues [13] and skip lists [40].

References

1. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Closed-form upper bounds in static cost analysis. JAR **46**(2), 161–203 (2011)
2. Alonso-Blas, D.E., Genaim, S.: On the limits of the classical approach to cost analysis. In: SAS, pp. 405–421 (2012)
3. Avanzini, M., Lago, U.D., Moser, G.: Analysing the complexity of functional programs: higher-order meets first-order. In: ICFP, pp. 152–164. ACM (2015)
4. Avanzini, M., Moser, G.: A combination framework for complexity. IC **248**, 22–55 (2016)
5. Avanzini, M., Moser, G., Schaper, M.: TcT: Tyrolean complexity tool. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 407–423. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_24
6. Bauer, S., Jost, S., Hofmann, M.: Decidable inequalities over infinite trees. In: LPAR, vol. 57, pp. 111–130 (2018)
7. Brázdil, T., Chatterjee, K., Kucera, A., Novotný, P., Velan, D., Zuleger, F.: Efficient algorithms for asymptotic bounds on termination time in VASS. In: LICS, pp. 185–194 (2018)
8. Chatterjee, K., Fu, H., Goharshady, A.K.: Non-polynomial worst-case analysis of recursive programs. In: CAV, pp. 41–63 (2017)
9. Colcombet, T., Daviaud, L., Zuleger, F.: Size-change abstraction and max-plus automata. In: MFCS, pp. 208–219 (2014)
10. Fiedor, T., Holík, L., Rogalewicz, A., Sinn, M., Vojnar, T., Zuleger, F.: From shapes to amortized complexity. In: VMCAI, pp. 205–225 (2018)
11. Flores-Montoya, A.: Cost analysis of programs based on the refinement of cost relations. Ph.D. thesis, Darmstadt University of Technology, Germany (2017)
12. Fuhs, C., Kop, C., Nishida, N.: Verifying procedural programs via constrained rewriting induction. TOCL **18**(2), 14:1–14:50 (2017)
13. Gambin, A., Malinowski, A.: Randomized meldable priority queues. In: SOFSEM, pp. 344–349 (1998)

14. Giesl, J., et al.: Analyzing program termination and complexity automatically with AProVE. JAR **1**, 3–31 (2017)
15. Gulwani, S., Zuleger, F.: The reachability-bound problem. In: PLDI, pp. 292–304 (2010)
16. Harrison, J.: Verifying nonlinear real formulas via sums of squares. In: TPHOLs, pp. 102–118 (2007)
17. Hermenegildo, M., et al.: An overview of ciao and its design philosophy. TPLP **12**(1–2), 219–252 (2012)
18. Hirokawa, N., Moser, G.: Automated complexity analysis based on the dependency pair method. In: IJCAR, pp. 364–380 (2008)
19. Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate amortized resource analysis. In: Proceedings of 38th POPL, pp. 357–370. ACM (2011)
20. Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate amortized resource analysis. TOPLAS **34**(3), 14 (2012)
21. Hoffmann, J., Aehlig, K., Hofmann, M.: Resource aware ML. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 781–786. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31424-7_64
22. Hoffmann, J., Das, A., Weng, S.C.: Towards automatic resource bound analysis for OCaml. In: POPL, pp. 359–373 (2017)
23. Hofmann, M., Jost, S.: Static prediction of heap space usage for first-order functional programs. In: POPL, pp. 185–197 (2003)
24. Hofmann, M., Moser, G.: Amortised resource analysis and typed polynomial interpretations. In: Proceedings of Joint 25th RTA and 12th TLCA, pp. 272–286 (2014)
25. Hofmann, M., Moser, G.: Multivariate amortised resource analysis for term rewrite systems. In: TLCA, pp. 241–256 (2015)
26. Hofmann, M., Moser, G.: Analysis of logarithmic amortised complexity (2018)
27. Hofmann, M., Leutgeb, L., Moser, G., Obwaller, D., Zuleger, F.: Type-based analysis of logarithmic amortised complexity. MSCS (2021, to appear). https://arxiv.org/abs/2101.12029
28. Hofmann, M., Rodriguez, D.: Automatic type inference for amortised heap-space analysis. In: ESOP, pp. 593–613 (2013)
29. Iacono, J.: Improved upper bounds for pairing heaps. In: SWAT, pp. 32–45 (2000)
30. Iacono, J., Yagnatinsky, M.V.: A linear potential function for pairing heaps. In: COCOA, pp. 489–504 (2016)
31. Jost, S., Vasconcelos, P., Florido, M., Hammond, K.: Type-based cost analysis for lazy functional languages. JAR **59**(1), 87–120 (2017)
32. Leutgeb, L.: ATLAS: Automated Amortised Complexity Analysis of Self-Adjusting Data Structures (2021). https://doi.org/10.5281/zenodo.4724917
33. Leutgeb, L.: ATLAS: Examples (2021). https://doi.org/10.5281/zenodo.4880499
34. Marques-Silva, J., Argelich, J., Graça, A., Lynce, I.: Boolean lexicographic optimization: algorithms & applications. Ann. Math. Artif. Intell. **62**(3–4), 317–343 (2011)
35. Moser, G., Schneckenreither, M.: Automated amortised resource analysis for term rewrite systems. Sci. Comput. Program. **185**, 102306 (2020)
36. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: TACAS, pp. 337–340 (2008)
37. Nipkow, T., Brinkop, H.: Amortized complexity verified. JAR **62**(3), 367–391 (2019)
38. Okasaki, C.: Purely Functional Data Structures. Cambridge University Press, Cambridge (1999)

39. Pani, T., Weissenbacher, G., Zuleger, F.: Rely-guarantee reasoning for automated bound analysis of lock-free algorithms. In: FMCAD, pp. 1–9 (2018)
40. Pugh, W.: Skip lists: a probabilistic alternative to balanced trees. Commun. ACM **33**(6), 668–676 (1990)
41. Schoenmakers, B.: A systematic analysis of splaying. IPL **45**(1), 41–50 (1993)
42. Schoenmakers, B.: Data structures and amortized complexity in a functional setting. Ph.D. thesis, Eindhoven University of Technology (1992)
43. Sebastiani, R., Trentin, P.: Optimathsat: a tool for optimization modulo theories. In: CAV, pp. 447–454 (2015)
44. Sinn, M., Zuleger, F., Veith, H.: A simple and scalable static analysis for bound analysis and amortized complexity analysis. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 745–761. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_50
45. Sinn, M., Zuleger, F., Veith, H.: Difference constraints: an adequate abstraction for complexity analysis of imperative programs. In: Kaivola, R., Wahl, T. (eds.) FMCAD, pp. 144–151. IEEE (2015)
46. Sinn, M., Zuleger, F., Veith, H.: Complexity and resource bound analysis of imperative programs using difference constraints. JAR **59**(1), 3–45 (2017)
47. Sleator, D., Tarjan, R.: Self-adjusting binary search trees. JACM **32**(3), 652–686 (1985)
48. Solar-Lezama, A.: The sketching approach to program synthesis. In: APLAS, pp. 4–13 (2009)
49. Tarjan, R.: Amortized computational complexity. SIAM J. Alg. Disc. Meth **6**(2), 306–318 (1985)
50. Wang, P., Wang, D., Chlipala, A.: TiML: a functional language for practical complexity analysis with invariants. Proc. ACM Program. Lang. **1**(OOPSLA), 1–26 (2017)
51. Winkler, S., Moser, G.: Runtime complexity analysis of logically constrained rewriting. In: Proceedings of LOPSTR 2020 (2020)
52. Zuleger, F.: The polynomial complexity of vector addition systems with states. In: FOSSACS, pp. 622–641 (2020)

Decision Procedures and Solvers

Theory Exploration Powered by Deductive Synthesis

Eytan Singher(✉) and Shachar Itzhaky

Technion, Haifa, Israel
{eytan.s,shachari}@cs.technion.ac.il

Abstract. This paper presents a symbolic method for automatic theorem generation based on deductive inference. Many software verification and reasoning tasks require proving complex logical properties; coping with this complexity is generally done by declaring and proving relevant sub-properties. This gives rise to the challenge of discovering useful sub-properties that can assist the automated proof process. This is known as the *theory exploration* problem, and so far, predominant solutions that emerged rely on evaluation using concrete values. This limits the applicability of these theory exploration techniques to complex programs and properties.

In this work, we introduce a new symbolic technique for theory exploration, capable of (offline) generation of a library of lemmas from a base set of inductive data types and recursive definitions. Our approach introduces a new method for using abstraction to overcome the above limitations, combining it with deductive synthesis to reason about abstract values. Our implementation has shown to find more lemmas than prior art, avoiding redundant lemmas (in terms of provability), while being faster in most cases. This new abstraction-based theory exploration method is a step toward applying theory exploration to software verification and synthesis.

Keywords: Theory exploration · Synthesis · Automatic theorem proving

1 Introduction

Most forms of software verification and synthesis rely on some form of logical reasoning to complete their task. Whether it is checking pre- and post-conditions, deriving specifications for sub-problems [1,19], or equivalence reduction [39], these methods rely on assumptions from both the input and relevant background knowledge. Domain-specific knowledge can reinforce these methods, whether via the design of a domain-specific language [29,36,45], specialized decision procedures [28], or decomposing specifications [35]. While hand-crafted techniques can treat whole classes of programs, every library or module contributes a collection of new primitives, requiring tweaking or extending these methods. Automatic formation of background knowledge can enable effortless treatment of such libraries and programs.

In the context of verification tools, such as Dafny [27] and Leon [7], as well as interactive proof assistants, such as Coq [12] and Isabelle/HOL [33], background knowledge is typically given as a set of *lemmas*. Usually, these libraries of lemmas (*i.e.* the background knowledge) are created by human engineers and researchers who are tasked with formulating them and proving their correctness. When a proof or verification task requires auxiliary lemmas missing from the existing background knowledge, the user is required to add and prove it, sometimes repeating this process until the proof is trivial or can be found automatically. For example, both Dafny and Leon fail to prove that addition is associative and commutative from first principles—based on an algebraic construction of the natural numbers. However, when given knowledge of these properties (*i.e.* encoded as lemmas: $(x + y) + z = x + (y + z)$ and $x + y = y + x)$[1], they readily prove composite facts such as $(x + 5) + y = 5 + (x + y)$.

A possible solution is to eagerly generate valid lemmas, and to do so automatically, offline, as a precursor to any work that would be built on top of the library. This paradigm is known as *theory exploration* [8,9], and differs from the common conjecture generation approach (in theorem provers and SMT solvers [37]) that is guided by a proof goal. As opposed to using proof goal as the basis for discovering sub-goals, when eagerly generating lemmas there is a vast space of possible lemmas to consider. Currently, two main approaches exist for filtering candidate conjectures, counterexample-based and observational equivalence-based [18,22,23,43]. These filtering techniques are all based on testing and therefore require automatic creation of concrete examples.

Testing with concrete values allows for fast evaluation and filtering of terms when the data types involved are simple. However, when scaling to larger data types and function types it becomes a bottleneck of the theory exploration process. Previous research effort has revealed that testing-based discovery is sensitive to the number and size of type definitions occurring in the code base. For example, QuickSpec, which is based on QuickCheck (as are all the existing testing-based theory exploration methods), employs a heuristic to restrict the set of types allowed in terms in order to make the checker's job easier. Compound data types such as lists can be nested up to two levels (lists of lists, but not lists of lists of lists). This presents an obstacle towards scaling the approach to real software libraries, since "*QuickCheck's size control interacts badly with deeply nested types* [...] *will generate extremely large test data.*" [38]

Following are two example scenarios that attempt to represent cases from software systems where structured data types and complicated APIs exist: (i) A series of tree data-types T_i where each T_i is a tree of height i with i children of type T_{i-1}, and the base case is an empty tree. Creating concrete examples for T_i will be resource expensive, as each tree has $O(i!)$ nodes, and each node requires a

[1] In fact, these properties are hard-wired into decision procedures for linear integer arithmetic in SMT solvers.

value. (ii) An ADT (Algebraic Data Type) A with multiple fields where each can contain a large amount of text or other ADTs, and a function over A that only accesses one of the fields. Even if evaluating the function is fast, fully creating A is expensive and will impact the theory exploration run-time.

This paper presents a new symbolic theory exploration approach that takes advantage of the characteristics of induction-based proofs. To overcome the blowup in the space of possible values, we make use of *symbolic values*, which contain interpreted symbols, uninterpreted symbols, or a mixture of the two. Conceptually, each symbolic value is an abstraction representing (infinitely) many possible values. This means that preexisting knowledge on the symbolic value can be applied without fully creating interpreted values. Still, when necessary, uninterpreted values can be expanded, creating larger symbolic values, thus refining the abstraction, and facilitating the necessary computation. We focus on the formation of *equational* theories, that is, lemmas that curtail the equivalence of two terms, with universal quantification over all free variables.

We show that our symbolic method for theory exploration is more applicable and faster in many different scenarios than state-of-the-art. As an example, given standard definitions for the list functions: ++ drop take filter our method proves facts that were not found by current state-of-the-art such as:

$$\text{(take } i\ xs) \mathbin{+\!\!+} (\text{drop } i\ xs) = xs$$
$$\text{filter } p\ (xs \mathbin{+\!\!+} ys) = (\text{filter } p\ xs) \mathbin{+\!\!+} (\text{filter } p\ ys)$$

Main Contributions. This paper provides the following contributions:

– A system for *theory synthesis* using symbolic values to take advantage of value abstraction. Our implementation, TheSy, can discover more lemmas than were found by testing-based tools, while being faster in most cases.
– A technique to compare universally quantified terms using term rewriting techniques and a given set of lemmas, called *symbolic observational equivalence* (SOE). SOE overapproximates term equalities deducible by the given lemmas (*i.e.*, will find more equalities), thus can be used for equality reduction in context of uninterpreted values, enabling fully symbolic reasoning over a large set of terms.
– An evaluation of our theory exploration system on a set of benchmarks for induction proofs taken from CVC4 [37] and TIP 2015 [11], specifically the IsaPlanner benchmarks [21]. We compare our implementation with a current leading theory exploration system, Hipster [18], using a novel metric. This metric is insensitive to the amount of found lemmas, but rather measures their usefulness in the context of theorem proving.

2 Overview

Our theory exploration method, named TheSy (Theory Synthesizer, pronounced *Tessy*), is based on syntax-guided enumerative synthesis. Similarly to previous approaches [10,20,38], TheSy generates a comprehensive set of terms from

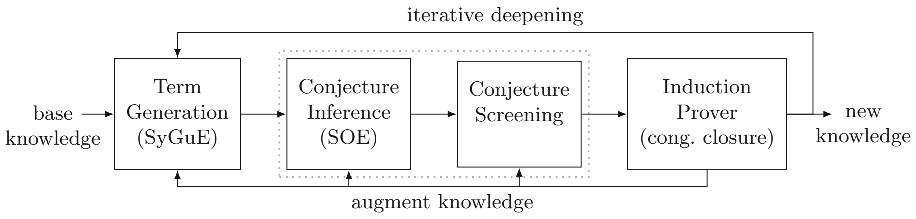


Fig. 1. TheSy system overview: breakdown into phases, with feedback loop.

the given vocabulary and looks for pairs that seem equivalent. Notably, TheSy employs deductive reasoning based on term rewriting systems to propose these pairs by extrapolating from a set of known equalities, employing a relatively lightweight (but unsound) reasoning procedure. The proposed pairs are passed as equality conjectures to a theorem prover capable of reasoning by induction.

The process (as shown in Fig. 1) is separated into four stages. These stages work in an iterative deepening fashion and are dependent on the results of each other. A short description is given to help the reader understand their context later on.

1. **Term Generation.** Build symbolic terms of increasing depth, based on the given vocabulary. Use known equalities for pruning via equivalence reduction.
2. **Conjecture Inference.** Evaluate terms on symbolic inputs, and apply deductive inference to extract new equalities, thus forming conjectures.
3. **Conjecture Screening.** Some of the conjectures, even valid ones, are special cases of known equalities or are trivially implied by them. We deem these conjectures redundant. TheSy culls such conjectures before continuing to prove the rest.
4. **Induction Prover.** The prover attempts to prove conjectures that passed screening using a normal induction scheme derived from algebraic data structure definitions in the given vocabulary. Conjectures that were successfully proven are then declared *lemmas* and added to the known equalities.

The phases are run iteratively in a loop, where each iteration deepens the generated terms and, hence, the discovered lemmas. These lemmas are fed back to earlier phases; this form of feedback contributes to discovering more lemmas thanks to several factors:

(i) Conjecture inference is dependent upon known equalities. Additional equalities enable finding new conjectures.
(ii) Accurate screening by merging equivalence classes based on known equalities.
(iii) The prover is based on known equalities with a congruence closure procedure. The more lemmas are known to the system, the more lemmas become provable by this method.
(iv) Term generation benefits from the equivalence reduction, avoiding duplicate work for equivalent terms.

$$\mathcal{V} = \{\, \begin{array}{ll} [\,] & \mathsf{list}\ T, \\ :: & T \to \mathsf{list}\ T \to \mathsf{list}\ T, \\ {+\!+} & \mathsf{list}\ T \to \mathsf{list}\ T \to \mathsf{list}\ T, \\ \mathrm{filter} & (T \to \mathsf{bool}) \to \mathsf{list}\ T \to \mathsf{list}\ T \ \} \end{array} \qquad \mathcal{C} = \{\, [\,], ::\, \}$$

$$\mathcal{E} = \{\, \begin{array}{ll} [\,] {+\!+}\, l = l, & (x :: xs) {+\!+}\, l = x :: (xs {+\!+}\, l), \\ \mathrm{filter}\ p\ [\,] = [\,], & \mathrm{filter}\ p\ (x :: xs) = \mathsf{if}\ p\,x\ \mathsf{then}\ x :: \mathrm{filter}\ p\ xs\ \mathsf{else}\ \mathrm{filter}\ p\ xs \ \} \end{array}$$

Fig. 2. An example input to TheSy.

Running Example. To illustrate TheSy's theory exploration procedure, we introduce a simple running example based on a list ADT. The input given to TheSy is shown in Fig. 2; it consists of a vocabulary $\mathcal{V}$ (of which $\mathcal{C}$ is a subset of ADT constructors) and a set of known equalities $\mathcal{E}$. The vocabulary $\mathcal{V}$ contains the canonical list constructors $[\,]$ and $::$, and two basic list operations $+\!+$ (concatenate) and filter. The equalities $\mathcal{E}$ consist of the definitions of the latter two.

At a very high level, the following process is about to take place: TheSy generates symbolic terms representing length-bound lists, *e.g.*, $[\,]$, $[v_1]$, $[v_2, v_1]$. Then, it will evaluate all combinations of function applications, up to a small depth, using these symbolic terms as arguments. If these evaluations yield common values for all possible assignments, the two application terms yielding them are conjectured to be equal. Since the evaluated expressions contain symbolic values, their result is a symbolic value. Comparing such symbolic values is done via congruence closure-based reasoning; we call this process *symbolic observational equivalence*, by way of analogy to observational equivalence [2] that is carried out using concrete values.

Out of the conjectures computed using symbolic observational equivalence, TheSy selects minimal ones according to a combined metric of compactness and generality. These are passed to a prover that employs both congruence closure and induction to verify the correction of the lemmas for *all* possible list values.

Some lemmas that TheSy can discover this way are:

$$\mathrm{filter}\ p\ (\mathrm{filter}\ p\ l) = \mathrm{filter}\ p\ l \qquad\qquad l_1 {+\!+} (l_2 {+\!+}\, l_3) = (l_1 {+\!+}\, l_2) {+\!+}\, l_3$$
$$\mathrm{filter}\ p\ l_1 {+\!+}\ \mathrm{filter}\ p\ l_2 = \mathrm{filter}\ p\ (l_1 {+\!+}\, l_2)$$

As briefly mentioned, our system design relies on congruence closure-based reasoning over universally quantified first-order formulas with uninterpreted functions. Congruence closure is weak but fast and constitutes one of the core procedures in SMT solvers [31,32]. On top of that, universally-quantified assumptions [4] are handled by formulating them as rewrite rules and applying some depth-bounded term rewriting as described in Subsect. 3.1. Additionally, TheSy implements a simple case splitting mechanism that enables reasoning on conditional expressions. Notably, this procedure *cannot* reason about recursive definitions since such reasoning routinely requires the use of induction. To that end, TheSy is geared towards discovering lemmas that can be proven by induction; a lemma is considered useful if it cannot be proven from existing lemmas by congruence closure alone, that is, without induction. Discovering such lemmas and adding them to the background knowledge evidently increases the reasoning power of the prover, since at least the fact of their own validity becomes provable, which it was not before.

3 Preliminaries

This work relies heavily on term rewriting techniques, which is employed across multiple phases of the exploration. Term rewriting is implemented efficiently using equality graphs (e-graphs). In this section, we present some minimal background of both, which will be relevant for the exploration procedure described later.

3.1 Term Rewriting Systems

Consider a formal language $\mathcal{L}$ of terms over some vocabulary of symbols. We use the notation $\mathcal{R} = t_1 \dot{\rightarrow} t_2$ to denote a rewrite rule from t_1 to t_2. For a (universally quantified) semantic equality law $t_1 = t_2$, we would normally create *both* $t_1 \dot{\rightarrow} t_2$ and $t_2 \dot{\rightarrow} t_1$. We refrain from assigning a direction to equalities since we do not wish to restrict the procedure to strongly normalizing systems, as is traditionally done in frameworks based on the Knuth-Bendix algorithm [24]. Instead, we define equivalence when a sequence of rewrites can identify the terms in either direction. A small caveat involves situations where $\mathrm{FV}(t_1) \neq \mathrm{FV}(t_2)$, that is, one side of the equality contains variables that do not occur on the other. We choose to admit only rules $t_i \dot{\rightarrow} t_j$ where $\mathrm{FV}(t_i) \supseteq \mathrm{FV}(t_j)$, because when $\mathrm{FV}(t_i) \subset \mathrm{FV}(t_j)$, applying the rewrite would have to create new symbols for the unassigned variables in t_j, which results in a large growth in the number of symbols and typically makes rewrites much slower as a result.

This slight asymmetry is what motivates the following definitions.

Definition 1. *Given a rewrite rule $\mathcal{R} = t_1 \dot{\rightarrow} t_2$, we define a corresponding relation $\xrightarrow{\mathcal{R}}$ such that $s_1 \xrightarrow{\mathcal{R}} s_2 \iff s_1 = C[t_1\sigma] \wedge s_2 = C[t_2\sigma]$ for some context C and substitution σ for the free variables of t_1, t_2. (A* context *is a term with a single hole, and $C[t]$ denotes the term obtained by filling the hole with t.)*

Definition 2. *Given a relation $\xrightarrow{\mathcal{R}}$ we define its symmetric closure:*

$$t_1 \xleftrightarrow{\mathcal{R}} t_2 \iff t_1 \xrightarrow{\mathcal{R}} t_2 \ \vee \ t_2 \xrightarrow{\mathcal{R}} t_1$$

Definition 3. *Given a set of rewrite rules $G_{\mathcal{R}} = \{\mathcal{R}_i\}$, we define a relation as union of the relations of the rewrites: $\xleftrightarrow{\{\mathcal{R}_i\}} \ \widehat{=} \ \bigcup_i \xleftrightarrow{\mathcal{R}_i}$.*
In the sequel, we will mostly use its reflexive transitive closure, $\xleftrightarrow{\{\mathcal{R}_i\}}^{*}$.

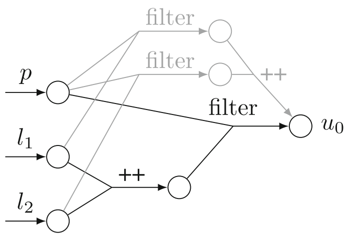


Fig. 3. An e-graph representing the expression filter $p\ (l_1 +\!\!+\, l_2)$ (dark) and the equivalent expression filter $p\ l_1 +\!\!+$ filter $p\ l_2$ (light).

The relation $\xleftrightarrow{\{\mathcal{R}_i\}}^*$ is reflexive, transitive, and symmetric, so it is an equivalence relation over $\mathcal{L}$. Under the assumption that all rewrite rules in $\{\mathcal{R}_i\}$ are semantics preserving, for any equivalence class $[t] \in \mathcal{L}/\xleftrightarrow{\{\mathcal{R}_i\}}^*$, all terms belonging to $[t]$ are definitionally equal. However, since $\mathcal{L}$ may be infinite, it is essentially impossible to compute $\xleftrightarrow{\{\mathcal{R}_i\}}^*$. Any algorithm can only explore a finite subset $\mathcal{T} \subseteq \mathcal{L}$, and in turn, construct a subset of $\xleftrightarrow{\{\mathcal{R}_i\}}^*$.

3.2 Compact Representation Using Equality Graphs

In order to be able to cover a large set of terms $\mathcal{T}$, we need a compact data structure that can efficiently represent many terms. Normally, terms are represented by their ASTs (Abstract Syntax Trees), but as there would be many instances of common subterms among the terms of $\mathcal{T}$, this would be highly inefficient. Instead, we adopt the concept of equality graphs (e-graphs) from automated theorem proving [15], which also saw uses in compiler optimizations and program synthesis [30,34,41], in which context they are known as Program Expression Graphs (PEGs). An e-graph is essentially a hypergraph where each vertex represents a set of equivalent terms (programs), and labeled, directed hyperedges represent function applications. Hyperedges therefore have exactly one target and zero or more sources, which form an ordered multiset (a vector, basically). Just to illustrate, the expression filter $p\ (l_1 +\!\!+\, l_2)$ will be represented by the nodes and edges shown in dark in Fig. 3. The nullary edges represent the constant symbols (p, l_1, l_2), and the node u_0 represents the entire term. The expression filter $p\ l_1 +\!\!+$ filter $p\ l_2$, which is equivalent, is represented by the light nodes and edges, and the equivalence is captured by sharing of the node u_0.

When used in combination with a rewrite system $\{\mathcal{R}_i\}$, each rewrite rule is represented as a premise pattern P and a conclusion pattern C. Applying a rewrite rule is then reduced to searching the e-graph for the search pattern and obtaining a substitution σ for the free variables of P. The result term is then obtained by substituting the free variables of C using σ. This term is added to the same equivalence class as the matched term (*i.e.* $P\sigma$), meaning they will both have the same root node. Consequently, a single node can represent a set of terms exponentially large in the number of edges, all of which will always be equivalent modulo $\overset{\{\mathcal{R}_i\}}{\longleftrightarrow}^{*}$.

In addition, since hyperedges always represent functions, a situation may arise in which two vertices represent the same term: This happens if two edges $\bar{u} \xrightarrow{f} v_1$ and $\bar{u} \xrightarrow{f} v_2$ are introduced by $\{\mathcal{R}_i\}$ for $v_1 \neq v_2$. In a purely functional setting, this means that v_1 and v_2 are equal. Therefore, when such duplication is found, it is beneficial to *merge* v_1 and v_2, eliminating the duplicate hyperedge. The e-graph data structure therefore supports a vertex merge operation and a congruence closure-based transformation [44] that finds vertices eligible for merge to keep the overall graph size small. This procedure can be quite expensive, so it is only run periodically.

4 Theory Synthesis

In this section, we go into a more detailed description of the phases of theory synthesis and explain how they are combined within an iterative deepening loop. To simplify the presentation, we describe all the phases first, then explain how the output from the last phase is fed back to the next iteration to complete a feedback loop. We continue with the input from the running example in Sect. 2 (Fig. 2) and dive deeper by showing intermediate states encountered during the execution of TheSy on this input. Throughout the execution, TheSy maintains a state, consisting of the following elements:

- $\mathcal{V}$, a sorted vocabulary
- $\mathcal{C} \subseteq \mathcal{V}$, a subset of constructors for some or all of the types
- $\mathcal{E}$, a set of equations initially consisting only of the definitions of the (non-constructor) functions in $\mathcal{V}$
- $\mathcal{T}$, a set of terms, initially containing just atomic terms corresponding to symbols from $\mathcal{V}$.

4.1 Term Generation

The first step is to generate a set of terms over the vocabulary $\mathcal{V}$. For the purpose of generating universally-quantified conjectures, we introduce a set of uninterpreted symbols, which we will call *placeholders*. Let $\mathcal{T}_{\mathcal{V}}$ be the set of types occurring as the type of some argument of a function symbol in $\mathcal{V}$. For each type τ occurring in $\mathcal{V}$ we generate placeholders $\overset{\tau}{\circ}_i$, two for each type (we will explain later why two are enough). These placeholders, together with all the symbols in $\mathcal{V}$, constitute the terms at depth 0.

At every iteration of deepening, TheSy uses the set of terms generated so far, and the (non-nullary) symbols of $\mathcal{V}$, to form new terms by placing existing ones in argument positions. For example, with the definitions from Fig. 2, we will have terms such as these at depths 1 and 2:

$$\begin{array}{lcc}
1 & \text{filter}\ \overset{T\to\mathsf{bool}}{\circ_1}\ \overset{\mathsf{list}\ T}{\circ_1} & \overset{\mathsf{list}\ T}{\circ_1} \mathbin{++} \overset{\mathsf{list}\ T}{\circ_2} \\
\hline
2 & [\,] \mathbin{++} \text{filter}\ \overset{T\to\mathsf{bool}}{\circ_1}\ \overset{\mathsf{list}\ T}{\circ_1} & \overset{\mathsf{list}\ T}{\circ_1} \mathbin{++} (\text{filter}\ \overset{T\to\mathsf{bool}}{\circ_1}\ \overset{\mathsf{list}\ T}{\circ_2}) \\
 & \text{filter}\ \overset{T\to\mathsf{bool}}{\circ_1}\ (\overset{\mathsf{list}\ T}{\circ_1} \mathbin{++} \overset{\mathsf{list}\ T}{\circ_2}) & (\text{filter}\ \overset{T\to\mathsf{bool}}{\circ_1}\ \overset{\mathsf{list}\ T}{\circ_1}) \mathbin{++} (\text{filter}\ \overset{T\to\mathsf{bool}}{\circ_1}\ \overset{\mathsf{list}\ T}{\circ_2})
\end{array} \tag{1}$$

It is easy to see that $\mathsf{filter}\ \overset{T\to\mathsf{bool}}{\circ_1}\ \overset{\mathsf{list}\ T}{\circ_1}$ and $[\,]\ \mathsf{++}\ \mathsf{filter}\ \overset{T\to\mathsf{bool}}{\circ_1}\ \overset{\mathsf{list}\ T}{\circ_1}$ are equivalent in any context; this follows directly from the definition of ++, available as part of $\mathcal{E}$. It is therefore acceptable to discard one of them without affecting completeness. TheSy does not discard terms—since they are merged in the e-graph, there is no need to—rather, it chooses the smaller term as representative when it needs one. This sort of *equivalence reduction* is present, in some way or another, in many automated reasoning and synthesis tools.

To formalize the procedure of generating and comparing the terms, in an attempt to discover new equality conjectures, we introduce the concept of *Syntax Guided Enumeration* (SyGuE). SyGuE is similar to Syntax Guided Synthesis (SyGuS for short [3]) in that they both use a formal definition of a language to find program terms solving a problem. They differ in the problem definition: while SyGuS is defined as a search for a correct program over the well-formed programs in the language, SyGuE is the sub-problem of iterating over *all distinct* programs in the language. SyGuS solvers may be improved using a smart search algorithm, while SyGuE solvers need an efficient way to eliminate duplicate terms, which may depend on the definition of program equivalence. We implement our variant of SyGuE, over the equivalence relation $\xleftrightarrow{\{\mathcal{R}_i\}^*}$, using the aforementioned e-graph: by applying and re-applying rewrite rules, provably equivalent terms are naturally *merged* into hyper-vertices, representing equivalence classes.

4.2 Conjecture Inference and Screening

Of course, in order to discover *new* conjectures, we cannot rely solely on term rewriting based on $\mathcal{E}$. To find more equivalent terms, TheSy carries on to generate a second set of terms, called *symbolic examples*, this time using only the constructors $\mathcal{C} \subset \mathcal{V}$ and uninterpreted symbols for leaves. This set is denoted $\mathcal{S}^\tau$, where τ is an algebraic datatype participating in $\mathcal{V}$ (if several such datatypes are present, one $\mathcal{S}^\tau$ per type is constructed). The depth of the symbolic examples (i.e. depth of applied constructors) is also bounded, but it is independent of the current term depth and does not increase during execution. For example, using the constructors of $\mathsf{list}\ T$ with an example depth of 2, we obtain the symbolic examples $\mathcal{S}^{\mathsf{list}\ T} = \{[\,], v_1 :: [\,], v_2 :: v_1 :: [\,]\}$, corresponding to lists of length up to 2 having arbitrary element values. Intuitively, if two terms are equivalent for all possible assignments of symbolic examples to $\overset{\mathsf{list}\ T}{\circ_i}$, then we are going *hypothesize* that they are equivalent for all list values. This process is very similar to observational equivalence as used by program synthesis tools [2,42], but since it uses the symbolic value terms instead of concrete values, we dub it *symbolic observational equivalence* (SOE).

Consider, for example, the simple terms $\overset{\mathsf{list}\ T}{\circ_1}$ and $\overset{\mathsf{list}\ T}{\circ_1}\ \mathsf{++}\ [\,]$. In placeholder form, none of the rewrite rules derived from $\mathcal{E}$ applies, so it cannot be determined that these terms are, in fact, equivalent. However, with the symbolic list examples above, the following rewrites are enabled:

$$[\,] \mathbin{+\!+} [\,] \xleftrightarrow{\{\mathcal{R}_i\}}{}^{*} [\,] \qquad v_1{::}[\,] \mathbin{+\!+} [\,] \xleftrightarrow{\{\mathcal{R}_i\}}{}^{*} v_1{::}[\,] \qquad v_2{::}v_1{::}[\,] \mathbin{+\!+} [\,] \xleftrightarrow{\{\mathcal{R}_i\}}{}^{*} v_2{::}v_1{::}[\,]$$

A similar case can be made for the two bottom terms in (1). For symbolic values $l_1, l_2 \in \mathcal{S}^{\text{list}\,T}$, it can be shown that

$$\text{filter} \overset{T\to\text{bool}}{\circ_1} (l_1 \mathbin{+\!+} l_2) \xleftrightarrow{\{\mathcal{R}_i\}}{}^{*} (\text{filter} \overset{T\to\text{bool}}{\circ_1} l_1) \mathbin{+\!+} (\text{filter} \overset{T\to\text{bool}}{\circ_1} l_2)$$

In fact, it is sufficient to substitute for $\overset{\text{list}\,T}{\circ_1}$, while *leaving* $\overset{\text{list}\,T}{\circ_2}$ *alone, uninterpreted*: e.g., $\text{filter} \overset{T\to\text{bool}}{\circ_1} ([\,] \mathbin{+\!+} \overset{\text{list}\,T}{\circ_2}) \xleftrightarrow{\{\mathcal{R}_i\}}{}^{*} (\text{filter} \overset{T\to\text{bool}}{\circ_1} [\,]) \mathbin{+\!+} (\text{filter} \overset{T\to\text{bool}}{\circ_1} \overset{\text{list}\,T}{\circ_2})$. This reduces the number of equivalence checks significantly, and is more than a mere heuristic: since we are going to rely on a prover that proceeds by applying induction to one of the arguments, it makes perfect sense to only bound that argument. If computation is blocked on the second argument, we would prefer to first infer an auxiliary lemma first, then use it to discover the blocked lemma later. See Example 1 below for an idea of when this situation arises.

The attentive reader may notice that the cases of $v_1{::}[\,]$ and $v_2{::}v_1{::}[\,]$ are a bit more involved: to proceed with the rewrite of filter, the expressions $\overset{T\to\text{bool}}{\circ_1}\ v_1$, $\overset{T\to\text{bool}}{\circ_1}\ v_2$ must be resolved to either *true* or *false*. However, the predicate $\overset{T\to\text{bool}}{\circ_1}$ as well as the arguments $v_{1,2}$ are uninterpreted. In this case, TheSy is required to perform a *case split* in order to enable the rewrites and unify the symbolic terms separately in each of the resulting four (2^2) cases. Notice that leaving $\overset{T\to\text{bool}}{\circ_1}$ uninterpreted means that the cases are only split when evaluation is blocked by one or more rewrite rule applications, potentially saving some branching. The following steps are then carried out for each case.

TheSy applies all the available rewrite rules to the entire e-graph, containing all the terms and symbolic examples. For every two terms t_1, t_2 such that for all viable substitutions σ of placeholders to symbolic examples of the corresponding types, $t_1\sigma$ and $t_2\sigma$ were shown equal—that is, ended up in the same equivalence class of the e-graph—the conjecture $t_1 \overset{?}{=} t_2$ is emitted. *E.g.*, in the case of the running example:

$$\text{filter} \overset{T\to\text{bool}}{\circ_1} (\overset{\text{list}\,T}{\circ_1} \mathbin{+\!+} \overset{\text{list}\,T}{\circ_2}) \overset{?}{=} (\text{filter} \overset{T\to\text{bool}}{\circ_1} \overset{\text{list}\,T}{\circ_1}) \mathbin{+\!+} (\text{filter} \overset{T\to\text{bool}}{\circ_1} \overset{\text{list}\,T}{\circ_2})$$

In the presence of multiple cases, the results are intersected, so that a conjecture is emitted only if it follows from all the cases.

Screening. Generating all the pairs according to the above criteria potentially creates many "obvious" equalities, which are valid propositions, but do not contribute to the overall knowledge and just clutter the prover's state. For example,

$$\text{filter} \overset{T\to\text{bool}}{\circ_1} (\overset{\text{list}\,T}{\circ_1} \mathbin{+\!+} \overset{\text{list}\,T}{\circ_2}) \overset{?}{=} \text{filter} \overset{T\to\text{bool}}{\circ_1} (\overset{\text{list}\,T}{\circ_1} \mathbin{+\!+} ([\,] \mathbin{+\!+} \overset{\text{list}\,T}{\circ_2}))$$

which follows from the definition of ++ and has nothing to do with filter. The synthesizer avoids generating such candidates, by choosing at most one term from every equivalence class of placeholder-form terms induced during the term generation phase. If both sides of the equality conjecture belong to the same equivalence class, the conjecture is dropped altogether.

The conjectures that remain are those equalities $t_1 \overset{?}{=} t_2$ where t_1 and t_2 got merged for all the assignments $\mathcal{S}^\tau$ to some $\overset{\tau}{\circ}_1$, and, furthermore, t_1 and t_2 themselves *were not* merged in placeholder form, prior to substitution. Such conjectures, if true, are guaranteed to increase the knowledge represented by $\mathcal{E}$ as (at least) the equality $t_1 = t_2$ was not previously provable using term rewriting and congruence closure.

4.3 Induction Prover

For practical reasons, the prover employs the following induction tactic:

– *Structural* induction based on the provided constructors ($\mathcal{C}$).
– The *first* placeholder of the inductive type is selected as the decreasing argument.
– Exactly *one* level of induction is attempted for each candidate.

The reasoning behind this design choice is that for every multi-variable term, *e.g.* $\overset{\text{list } T}{\circ_1} \mathbin{++} \overset{\text{list } T}{\circ_2}$, the synthesizer also generates the symmetric counterpart $\overset{\text{list } T}{\circ_2} \mathbin{++} \overset{\text{list } T}{\circ_1}$. So electing to perform induction on $\overset{\text{list } T}{\circ_1}$ does not impede generality.

In addition, if more than one level of induction is needed, the proof can (almost) always be revised by factoring out the inner induction as an auxiliary lemma. Since the synthesizer produces *all* candidate equalities, that inner lemma will also be discovered and proved with one level of induction. Lemmas so proven are added to $\mathcal{E}$ and are available to the prover, so that multiple passes over the candidates can gradually grow the set of provable equalities.

When starting a proof, the prover never needs to look at the base case, as this case has already been checked during conjecture inference. Recall that placeholders $\overset{\tau}{\circ}_1$ are instantiated with bounded-depth expressions using the constructors of τ, and these include all base cases (non-recursive constructors) by default. For the example discussed above, the case of $\text{filter } \overset{T\to\text{bool}}{\circ_1} \; ([\,] \mathbin{++} \overset{\text{list } T}{\circ_2}) = (\text{filter } \overset{T\to\text{bool}}{\circ_1} \; [\,]) \mathbin{++} (\text{filter } \overset{T\to\text{bool}}{\circ_1} \; \overset{\text{list } T}{\circ_2})$ has been discharged early on, otherwise the conjecture would not have come to pass. The prover then turns to the induction step, which is pretty routine but is included in Fig. 4 for completeness of the presentation.

It is worth noting that the conjecture inference, screening and induction phases utilize a common reasoning core based on rewriting and congruence closure. In situations where the definitions include conditions such as match $p\,x$ in Fig. 4 (in this case, desugared from if $p\,x$), the prover also performs automatic case split and distributes equalities over the branches. Details and specific optimizations are described in Sect. 5.

$$\begin{array}{lll}
\textit{Assume} & \multicolumn{2}{l}{\text{filter } p\ (xs +\!\!+ l_1) = \text{filter } p\ xs +\!\!+ \text{filter } p\ l_1} \\
\textit{Prove} & \multicolumn{2}{l}{\text{filter } p\ ((x :: xs) +\!\!+ l_1) = \text{filter } p\ (x :: xs) +\!\!+ \text{filter } p\ l_1} \\
\textit{via} & (1) & \text{filter } p\ ((x :: \text{xs}) +\!\!+ l_1) = \text{filter } p\ (x :: (\text{xs} +\!\!+ l_1)) \\
 & (2) & \quad = \text{match } (p\,x) \text{ with } \textit{true} \Rightarrow x :: \text{filter } p\ (xs +\!\!+ l_1) \\
 & & \qquad\qquad\qquad\qquad \textit{false} \Rightarrow \text{filter } p\ (xs +\!\!+ l_1) \\
(IH) & (3) & \quad = \text{match } (p\,x) \text{ with } \textit{true} \Rightarrow x :: (\text{filter } p\ xs +\!\!+ \text{filter } p\ l_1) \\
 & & \qquad\qquad\qquad\qquad \textit{false} \Rightarrow \text{filter } p\ xs +\!\!+ \text{filter } p\ l_1 \\
 & (4) & \text{filter } p\ (x :: xs) +\!\!+ \text{filter } p\ l_1 \\
 & & \quad = \big(\text{match } (p\,x) \text{ with } \textit{true} \Rightarrow x :: \text{filter } p\ xs \\
 & & \qquad\qquad\qquad\qquad \textit{false} \Rightarrow \text{filter } p\ xs\big) +\!\!+ \text{filter } p\ l_1 \\
 & (5) & \quad = \text{match } (p\,x) \text{ with } \textit{true} \Rightarrow x :: (\text{filter } p\ xs +\!\!+ \text{filter } p\ l_1) \\
\square & & \qquad\qquad\qquad\qquad \textit{false} \Rightarrow \text{filter } p\ xs +\!\!+ \text{filter } p\ l_1
\end{array}$$

Fig. 4. Example proof by induction based on congruence closure and case splitting.

Speculative Generalization. When the prover receives a conjecture with multiple occurrences of a placeholder, *e.g.* $\overset{\text{list } T}{\circ_1} +\!\!+ (\overset{\text{list } T}{\circ_2} +\!\!+ \overset{\text{list } T}{\circ_1}) \overset{?}{=} (\overset{\text{list } T}{\circ_1} +\!\!+ \overset{\text{list } T}{\circ_2}) +\!\!+ \overset{\text{list } T}{\circ_1}$, it is designed to first speculate a more general form for it by replacing the multiple occurrences with fresh placeholders. Recall that in Subsect. 4.1 we argued that two placeholders of each type is going to be sufficient; this is the mechanism that enables it. There is more than one way to generalize a given conjecture: for this example, there are two ways (up to alpha-renaming):

$$\overset{\text{list } T}{\circ_1} +\!\!+ (\overset{\text{list } T}{\circ_2} +\!\!+ \overset{\text{list } T}{\circ_3}) \overset{?}{=} (\overset{\text{list } T}{\circ_1} +\!\!+ \overset{\text{list } T}{\circ_2}) +\!\!+ \overset{\text{list } T}{\circ_3} \qquad \overset{\text{list } T}{\circ_1} +\!\!+ (\overset{\text{list } T}{\circ_2} +\!\!+ \overset{\text{list } T}{\circ_3}) \overset{?}{=} (\overset{\text{list } T}{\circ_3} +\!\!+ \overset{\text{list } T}{\circ_2}) +\!\!+ \overset{\text{list } T}{\circ_1}$$

The prover must attempt both. Failing that, it would fall back to the original conjecture. Formally, given an equality conjecture $s = t$ we can consider an assignment σ such that $r = s\sigma, q = t\sigma$; where the original conjecture uses an assignment with only two values per type. The prover thus must iterate through different assignments σ_i with more possible values per type, and attempt to prove a new conjecture $r\sigma_i = q\sigma_i$. This incurs more work for the prover but is well worth its cost compared to a-priori generation of terms with three placeholders.

4.4 Looping Back

The equations obtained from Subsect. 4.3 are fed back in four different but interrelated ways. The first, inner feedback loop is from the induction prover to itself: the system will attempt to prove the smaller lemmas first, so that when proving the larger ones, these will already be available as part of $\mathcal{E}$. This enables more proofs to go through. The second feedback loop uses the lemmas obtained to filter out proofs that are no longer needed. The third, outer loop is more interesting: as equalities are made into rewrite rules, additional equations may now pass the inference phase, since the symbolic evaluation core can equate more terms based on this additional knowledge. The fourth resonates with the third, applying the new rewrite rules acts as an equality reduction mechanism, reducing the number of hyperedges added to the e-graph during term generation.

It is worth noting that while concrete observational equivalence uses a trivially simple equivalence checking mechanism with the trade-off that it may generate many incorrect equalities, our *symbolic* observational equivalence is conservative in the sense that a symbolic value may represent infinitely many concrete inputs, and only if the synthesizer can *prove* that two terms will evaluate to equal values on *all* of them, by way of constructing a small proof, are they marked as equivalent. This means that some actually-equivalent terms may be "blocked" by the inference phase, which cannot happen when using concrete values—but also means that having additional inference rules ($\mathcal{E}$) can improve this equivalence checking, potentially leading to more discovered lemmas. This property of TheSy is appealing because it allows an explored theory to evolve from basic lemmas to more complex ones.

Example 1 (Lemma seeding). To understand this last point, consider the standard definition of list reversal for the list datatype:

$$\begin{aligned}\text{rev } [\,] &= [\,]\\ \text{rev } (x :: xs) &= \text{rev } xs +\!\!+ (x :: [\,])\end{aligned}$$

Given the terms $t_1 = \text{rev } (\overset{\mathsf{list}\,T}{\circ_1} +\!\!+ \overset{\mathsf{list}\,T}{\circ_2})$ and $t_2 = \text{rev } \overset{\mathsf{list}\,T}{\circ_2} +\!\!+ \text{rev } \overset{\mathsf{list}\,T}{\circ_1}$, symbolic observational equivalence with the assignments $\{\overset{\mathsf{list}\,T}{\circ_1} \mapsto \mathcal{S}^{\mathsf{list}\,T}\}$ fails to unify them. This is due to $+\!\!+$ being defined by induction on its first argument, hence, *e.g.*—

$$\begin{aligned}\text{rev } (v_2 :: v_1 :: [\,] +\!\!+ \overset{\mathsf{list}\,T}{\circ_2}) &\quad \rightarrow^* \quad (\text{rev } \overset{\mathsf{list}\,T}{\circ_2} +\!\!+ (v_1 :: [\,])) +\!\!+ (v_2 :: [\,])\\ \text{rev } \overset{\mathsf{list}\,T}{\circ_2} +\!\!+ \text{rev } v_2 :: v_1 :: [\,] &\quad \rightarrow^* \quad \text{rev } \overset{\mathsf{list}\,T}{\circ_2} +\!\!+ (v_1 :: v_2 :: [\,])\end{aligned}$$

Without the associativity property of $+\!\!+$, it would not be possible to show that these symbolic values are equivalent, so the conjecture $t_1 \overset{?}{=} t_2$ will not even be generated. Luckily, having proven $\overset{\mathsf{list}\,T}{\circ_1} +\!\!+ (\overset{\mathsf{list}\,T}{\circ_2} +\!\!+ \overset{\mathsf{list}\,T}{\circ_3}) \overset{?}{=} (\overset{\mathsf{list}\,T}{\circ_1} +\!\!+ \overset{\mathsf{list}\,T}{\circ_2}) +\!\!+ \overset{\mathsf{list}\,T}{\circ_3}$, these rewrites are "unblocked", so that the equality can be conjectured and ultimately proven.

One caveat is that whenever $\mathcal{E}$ is updated by the addition of a new lemma, some of the previously emitted conjectures may consequently become redundant. Moreover, conjectures that were passed to the prover before but failed validation may now succeed, and new ones may be emitted in the generation phase. To take these into account, the actual loop performed by TheSy is a bit more involved than has been described so far. For each term depth, TheSy performs all phases as described, but each time a lemma is discovered TheSy re-runs the conjecture generation, screening, and prover phases. Only when no more conjectures are available does TheSy increase the term depth and generate new terms.

5 Evaluation

We implemented TheSy in Rust, using the e-graph manipulation library *egg* [44]. TheSy accepts definitions in SMTLIB-2.6 format [6], based on the UF theory (uninterpreted functions), limited to universal quantifications. Type declarations occurring in the input are collected and comprise $\mathcal{V}$; universal equalities form $\mathcal{E}$ and are translated into rewrite rules (either uni- or bidirectional, as explained in Subsect. 3.1). Then SyGuE is performed on $\mathcal{V}$, generating candidate conjectures using SOE. SyGuE uses *egg* for equivalence reduction, and SOE uses it for comparing symbolic values. Conjectures are then dismissed using TheSy's induction-based prover. This is done in an iterative deepening loop.

Case Split. Both SOE and the prover use a case splitting mechanism; This mechanism detects when rewriting cannot match due to an opaque value (an uninterpreted symbol), and applies case splitting according to the constructors of relevant ADTs. However, doing so for every rule is too costly and, in most cases, redundant—TheSy generates a variety of terms, so if one term is blocked due to an uninterpreted symbol, another one exists with a symbolic example instead. A situation where this is *not* the case is when *multiple* uninterpreted symbols block the rewrite (recall that TheSy only substitutes one placeholder per term with symbolic examples). To illustrate, consider the case in Fig. 4 where both the list $x :: xs$ and $p\,x$ are used in match expressions, therefore a case split is needed by $p\,x \in \{\mathit{true}, \mathit{false}\}$. Therefore, TheSy only performs case splitting for rewrite rules that require multiple match patterns but only one is blocked.

The splitting mechanism itself, operates by copying the e-graph and applying the term rewriting logic separately for each case. Each copy then yields a partition of the existing equivalence classes. These partitions are intersected between all cases, and each of the resulting intersections lead to merging of equivalence classes in the original e-graph. It is worth noting that TheSy never needs to backtrack a case split it has elected to apply. As a consequence, execution time is not exponential in the total number of case splits performed, only in the nesting level of such splits (which is bounded by 2 in our experiments).

We compare TheSy to the most recent and closely related theory exploration system, Hipster [23]—which is based on random testing (backed by QuickSpec [38]) with proof automations from and frontend in Isabelle/HOL [33]. Hipster represents the culmination of several works on existing theory exploration (see Sect. 6). Both systems generate a set of proved lemmas as output, each such set encompassing a conceptual volume of knowledge that was discovered automatically. We note that the same knowledge can be represented in various ways, so directly comparing the sets of lemmas is going to be meaningless.

5.1 Evaluating Theory Exploration Quality

We define a comparison method for two theory exploration systems A and B starting from a common initial theory (defined as a set of closed formulas) $\mathcal{T}$. As a metric for the quality and efficacy of results obtained from theory exploration, and, therefore, their perceived usefulness, we use the notion of *knowledge*

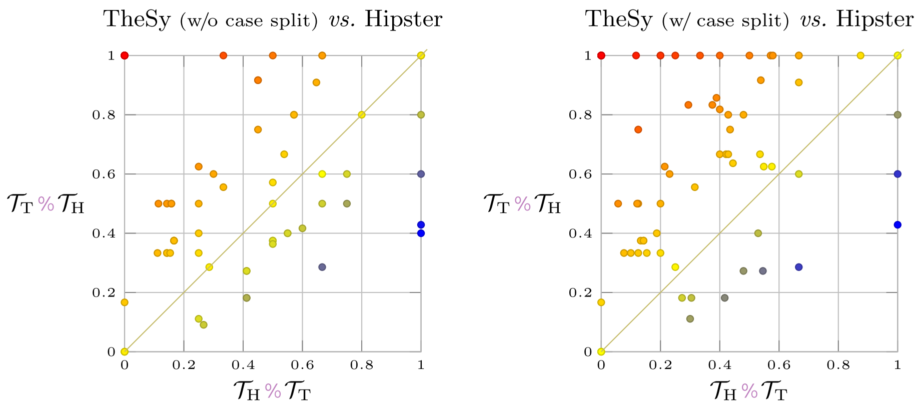


Fig. 5. A scatter plot showing the ratio of lemmas in theories discovered by each tool that were subsumed by the theory discovered by its counterpart (T = TheSy, H = Hipster). Each point represents a single test case. The vertical axis shows how many of the lemmas discovered by Hipster were subsumed by those discovered by TheSy, and the horizontal axis shows the converse.

(inspired by "knowledge base" in Theorema [8]). A theory $\mathcal{T}$ in a given logical proof system induces a collection of attainable knowledge, $\mathcal{K}_\mathcal{T} = \{\varphi \mid \mathcal{T} \vdash \varphi\}$, that is, characterized by the set of (true) statements that can be proven based on $\mathcal{T}$. In practice, a "pure" notion of knowledge based on provability is impractical, because most interesting logics are undecidable, and automated proving techniques cannot feasibly find proofs for all true statements. We, therefore, parameterize knowledge relative to a *prover*—a procedure that always terminates and can prove a subset of true statements. Termination can be achieved by restricting the space of proofs by either size or resource bounds. We say that $\mathcal{T} \overset{S}{\vdash} \varphi$ when a prover, S, is able to verify the validity of φ in a theory $\mathcal{T}$. A more realistic characterization of knowledge would then be $\mathcal{K}^S_\mathcal{T} = \left\{\varphi \mid \mathcal{T} \overset{S}{\vdash} \varphi\right\}$. Assuming that the prover S is fixed, a theory $\mathcal{T}'$ is said to *increase knowledge* over $\mathcal{T}$ when $\mathcal{K}^S_{\mathcal{T}'} \supset \mathcal{K}^S_\mathcal{T}$.

We utilize the notion of $\mathcal{K}^S_\mathcal{T}$ described above to test the knowledge gained by A against that of B, and vice versa. We take the set of lemmas $\mathcal{T}_A$ generated by A and check whether it is subsumed by $\mathcal{T}_B$, generated by B, by checking whether $\mathcal{T}_A \subseteq \mathcal{K}^S_{\mathcal{T} \cup \mathcal{T}_B}$; we then carry out the same comparison with the roles of A and B reversed. A working assumption is that both A and B include some mechanism for screening redundant conjectures. That is, a component that receives the current set of known lemmas T_i and a conjecture φ and decides whether the conjecture is redundant. It is important to choose S such that whenever A (or B) discards φ, due to redundancy, it holds that $\varphi \in \mathcal{K}^S_{T_i}$.

Incorporating the solver into the comparison makes the evaluation resistant to large amounts of trivial lemmas, as they will be discarded by A or B. It is still possible for some lemmas to be "better" than others, so knowledge is not uniformly distributed; this is hard to quantify, though. A few possible measures of usefulness come to mind, such as lemma utilization in a task (such as proof search), proof complexity, or matching to a given context, but given just the exploration task, there is not sufficient information to apply them. A first approximation is to consider the discovered lemmas themselves, *i.e.*, $\mathcal{T}_A \cup \mathcal{T}_B$, as representing proof objectives. In doing so, we pit A and B in direct contest with one another. We choose this avenue because it is straightforward to apply, admitting that it may be inaccurate in some cases.

To evaluate our approach and its implementation, we run both TheSy and Hipster on functional definitions collected from the TIP 2015 benchmark suite [11], specifically the IsaPlanner [21] benchmarks (85 benchmarks in total), for compatibility between the two systems. TIP benchmarks also contain goal propositions, but for the purpose of evaluating the exploration technique, these are redacted. This experiment uses the simple rewrite-driven congruence-closure decision procedure with a case split mechanism in the role of the solver, S, occurring in the definition of knowledge $\mathcal{K}$. Hipster uses Isabelle/HOL's simplifier as a conjecture redundancy filtering mechanism, which is in itself a simple rewrite-driven decision procedure, therefore S provides a suitable comparison. We compute the portion of lemmas found by Hipster that were provable (by S) from TheSy's results and vice versa. In other words, we check the ratio given by $|\mathcal{T}_A \cap \mathcal{K}^S_{\mathcal{T} \cup \mathcal{T}_B}| \,/\, |\mathcal{T}_A|$, which we denote $\mathcal{T}_B \,\%\, \mathcal{T}_A$, in both directions. Figure 5 displays the ratios, where each point represents a single test case. Points above the diagonal line represent test cases where TheSy's ratio was higher and for points under the line Hipster's ratio was higher. We conduct this experiment twice: Once with the case-splitting mechanism of TheSy turned off for its exploration, and once with it turned on. (Hipster does not have such a switch as it always generates concrete values.) The reason for this is that case splitting increases the running time significantly (as we show next), so we want to evaluate its contribution to the discovery of lemmas. Comparing the two charts, while TheSy performs reasonably well compared to Hipster without case splitting (in 48 out of the 85 TheSy's ratio was better and equal in 12), enabling it leads to a clear advantage (in 65 out of the 85 TheSy's ratio was better and equal in 6).

Performance. To compare runtime efficiency, we consider the time it took to fully explore the IsaPlanner test suite. We consider an exploration "full" when it has finished enumerating all the terms, and associated candidate conjectures, up to the depth bound ($k = 2$)[2] with TheSy or size bound with Hipster ($s = 7$), and check them; or when a timeout of one hour is reached, whichever is sooner. We then sort the benchmarks from shortest- to longest-running for each of the tools, and report the accumulated time to explore the first i benchmarks ($i = 1..85$). The results are shown in the graph in Fig. 6, for Hipster, TheSy with case split disabled, and TheSy with case split enabled. In both configurations,

[2] Our experience shows that choosing larger ks greatly affects the run-time, but does not lead to many useful lemmas.

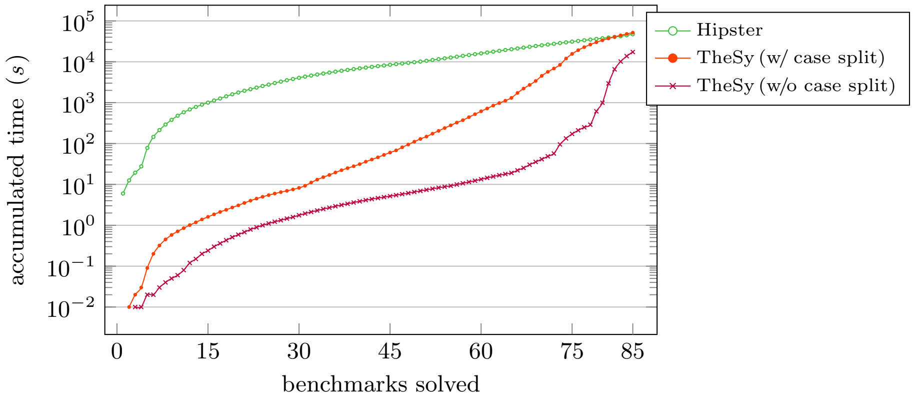


Fig. 6. Time to fully explore the 85 IsaPlanner benchmarks. A full exploration is considered one where either all terms up to the depth bound have been enumerated or a timeout of 1 h has been reached. The y axis shows the amount of time needed to complete the first x benchmarks, when they are sorted from shortest- to longest-running. (Time scale is logarithmic; lower is better.)

TheSy is very fast for the lower percentiles, but begins to slow down, due to case splitting, towards the end of the line. To illustrate, in the 25th percentile TheSy was ~380 times faster (0.48 s *vs.* 182.47 s); in the 50th percentile, ~57 times faster (5.28 s *vs.* 305.37 s); and in the 75th percentile, ~6 times faster (141.24 to 883.8). Overall TheSy took 51.6K seconds and Hipster 47.1K, meaning Hipster was ~1.1 times faster. It is evident from the chart that case splitting is largely responsible for the longer execution times. Without case splitting, TheSy is much faster, and completes all 85 benchmarks in less time than it takes Hipster. Of course, in that mode of operation, TheSy finds fewer lemmas (as shown in Fig. 5), but is still superior to Hipster. Future work needs to focus on improving the case-splitting mechanism, similar to their treatment in SAT and SMT, allowing TheSy to deal with such theories more efficiently.

5.2 Efficacy to Automated Proving

While the mission statement of TheSy is solely to provide lemmas based on core theories, we wish to claim that such discovered theories are beneficial toward proving theorems in general, based on the same core theory. We used a collection of benchmarks for induction proofs used by CVC4 [37], and conducted the following experiment: First, the proof goals are skipped and only the symbol declarations and provided axioms are used to construct an input to TheSy. Then, whenever a new lemma is discovered and passes through the prover, we also attempt to prove the goal—utilizing the same mechanism used for vetting conjectures. As soon as the latter goes through, the exploration process is aborted, and all lemmas collected are discarded. The experiments are thus independent across the individual benchmarks.

Table 1. Results of the CVC4 benchmark suite (number of successful proofs in each category).

	Total	Z3	CVC4	CVC4+ig	TheSy
clam	136	25	20	108	102
hipspec	42	6	7	33	29
isaplanner	87	35	34	79	47
leon	46	9	9	40	9
Total	311	75	70	260	187

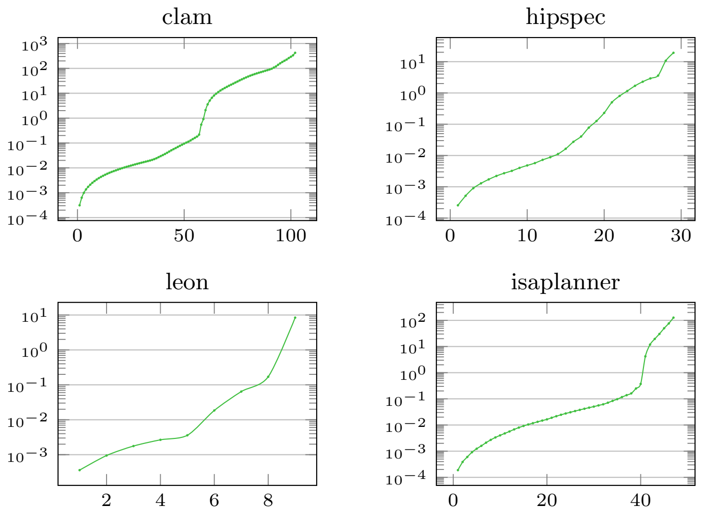


Fig. 7. Accumulated time-to-solve for each of the benchmark suites from the CVC4 collection. The y axis shows the amount of time needed to complete the first x (successful) proofs, when benchmarks are sorted from shortest- to longest-running.

Even though this setting is unfavorable to TheSy—because it does not take advantage of the fact that theory exploration can be done offline, then its results re-used for proofs over the same core theory—we report considerable success in solving these benchmarks. Out of the 311 benchmarks, our theory exploration + simple-minded induction was able to prove 187 (with a 5-min timeout, same as in the original CVC4 experiments). For comparison, Z3 and CVC4 (without conjecture generation) were able to prove 75 and 70 of them, respectively. This shows that the majority of instances were not solvable without the use of induction. CVC4 with its conjecture generation enabled was able to solve 260 of them. Table 1 shows the number of successful proofs achieved for each of the four suites. Figure 7 shows the accumulated time required for the benchmarks; the vast majority of the success cases occur early on, because in some cases a rather small auxiliary lemma is all that is needed to make the proof go through.

6 Related Work

Equality Graphs. Originally brought into use for automated theorem proving [15], e-graphs were popularized as a mechanism for implementing low-level compiler optimizations [41], under the name *PEGs*. These e-graphs can be used to represent a large program space compactly by packing together equivalent programs. In that sense they are similar to Version Space Algebras [26], but their prime objective is entirely different. While VSAs focus on efficient intersections, e-graphs are used to saturate a space of expressions with all equality relations that can be inferred. They have found use in optimizing expressions for more than just speed, for example to increase numerical stability of floating-point programs in Herbie [34]. There are two key differences in the way e-graphs are used in this work compared to prior: (i) equality laws are not hard-coded nor fixed, they are fertilized as the system proves more lemmas automatically; (ii) saturation cannot be guaranteed or even obtained in all cases, which we overcome by a bound on rewrite-rule application depth. (The latter point is an indirect consequence of the former.)

Automated Theorem Provers. Many systems rely on known theorems or are designed to support users in semi-automated proving. Congruence closure is also a proven method for tautology checking in automated theorem provers, such as Vampire [25], and is used as a decision procedure for reasoning about equality in leading SMT solvers Z3 [14] and CVC4 [5]. There, it is limited mostly to first-order reasoning, but can essentially be applied unchanged to higher-level scenarios such as ours.

Related to theory exploration, but using separate techniques, are Zipperposition [13], and the conjecture generation mechanism implemented as part of the induction prover in CVC4 [37]. It should be noted, that these are directed toward a specific proof goal, as opposed to theory exploration, which is presumed to be an offline phase. As such, the above two techniques incorporate generation of inductive hypotheses into the saturation proof search/SMT procedure, respectively.

Theory Exploration. IsaCoSy [22] pioneered the use of synthesis techniques for bottom-up lemma discovery. IsaCoSy combines equivalence reduction with counterexample-guided inductive synthesis (CEGIS [40]) for filtering candidate lemmas. This requires a solver capable of generating counterexamples to equivalence. Subsequent development was based on random generation of test values, as implemented in QuickSpec [38] for reasoning about Haskell programs, later combined with automated provers for checking the generated conjectures [10,20]. We have mentioned the deficiencies of using concrete values (as opposed to symbolic ones) and random testing in Sect. 1 and make an empirical comparison with Hipster, a descendent of IsaCoSy and QuickSpec, in Sect. 5.

Inductive Synthesis. In the area of SyGuS [3], tractable bottom-up enumeration is commonly achieved by some form of equivalence reduction [39]. When dealing with concrete input-output examples, observational equivalence [2,42] is very effective. The use of symbolic examples in synthesis has been suggested [17], but to the best of our knowledge, ours is the only setting where symbolic observational equivalence has been applied. Inductive synthesis, in combination with abduction [16], has also been used to infer specifications [1], although not as an exploration method but as a supporting mechanism for verification.

7 Conclusion

We described a new method for theory exploration, which differentiates itself from existing work by basing the reasoning on a novel engine based on term rewriting. The new approach differs from previous work, specifically those based on testing techniques, in that:

1. This lightweight reasoning is purely symbolic, supporting value abstraction and performs better then prior art.
2. Functions are naturally treated as first-class objects, without specific support implementation.
3. The only needed input is the code defining the functions involved, and no support code such as a specific theory solver or random value generators.
4. TheSy has a unique feedback loop between the prover and the synthesizer, allowing more conjectures to be found and proofs to succeed.

By creating a feedback loop between the four different phases, term generation, conjecture inference, conjecture screening and induction prover, this system manages to efficiently explore many theories. This goes beyond similar feedback loops in existing tools, aiming to reduce false and duplicate conjectures. As explained in Subsect. 4.2, this form is also present in TheSy, but TheSy utilizes this feedback in more phases of the computation.

Theory exploration carries practical significance to many automated reasoning tasks, especially in formal methods, verification and optimization. Complex properties lead to an ever-growing number of definitions and associated lemmas, which constitute an integral part of proof construction. These lemmas can be used for SMT solving, automated and interactive theorem proving, and as a basis for equivalence reduction in enumerative synthesis. The term rewriting-based method that we presented in this paper is simple, highly flexible, and has already shown results surpassing existing exploration methods. The generated lemmas allow even this simple method to prove conjectures that normally require sophisticated SMT extensions. Our main conclusion is that deductive techniques and symbolic evaluation can greatly contribute to theory exploration, in addition to their existing applications in invariant and auxiliary conjecture inference.

Acknowledgements. This research was supported by the Israeli Science Foundation (ISF) Grants No. 243/19 and 2740/19 and by the United States-Israel Binational Science Foundation (BSF) Grant No. 2018675.

References

1. Albarghouthi, A., Dillig, I., Gurfinkel, A.: Maximal specification synthesis. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, pp. 789–801. Association for Computing Machinery, New York (2016)
2. Albarghouthi, A., Gulwani, S., Kincaid, Z.: Recursive program synthesis. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 934–950. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_67
3. Alur, R., et al.: Syntax-guided synthesis. Dependable Softw. Syst. Eng. **40**, 1–25 (2015)
4. Barbosa, H., Fontaine, P., Reynolds, A.: Congruence closure with free variables. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10206, pp. 214–230. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54580-5_13
5. Barrett, C., et al.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
6. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa (2017). www.SMT-LIB.org
7. Blanc, R., Kuncak, V., Kneuss, E., Suter, P.: An overview of the Leon verification system: verification by translation to recursive functions. In: Proceedings of the 4th Workshop on Scala, SCALA 2013. Association for Computing Machinery, New York (2013)
8. Buchberger, B.: Theory exploration with theorema. Analele Universitatii Din Timisoara, ser. Matematica-Informatica **38**(2), 9–32 (2000)
9. Buchberger, B., et al.: Theorema: towards computer-aided mathematical theory exploration. J. Appl. Logic **4**(4), 470–504 (2006)
10. Claessen, K., Johansson, M., Rosén, D., Smallbone, N.: Automating inductive proofs using theory exploration. In: Bonacina, M.P. (ed.) CADE 2013. LNCS (LNAI), vol. 7898, pp. 392–406. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38574-2_27
11. Claessen, K., Johansson, M., Rosén, D., Smallbone, N.: TIP: tons of inductive problems. In: Kerber, M., Carette, J., Kaliszyk, C., Rabe, F., Sorge, V. (eds.) CICM 2015. LNCS (LNAI), vol. 9150, pp. 333–337. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-20615-8_23
12. The Coq Development Team: The Coq Proof Assistant Reference Manual, version 8.7 (October 2017)
13. Cruanes, S.: Superposition with structural induction. In: Dixon, C., Finger, M. (eds.) FroCoS 2017. LNCS (LNAI), vol. 10483, pp. 172–188. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66167-4_10
14. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
15. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM **52**(3), 365–473 (2005)
16. Dillig, I., Dillig, T., Li, B., McMillan, K., Sagiv, M.: Synthesis of circular compositional program proofs via abduction. Int. J. Softw. Tools Technol. Transf. **19**(5), 535–547 (2015). https://doi.org/10.1007/s10009-015-0397-7

17. Drachsler-Cohen, D., Shoham, S., Yahav, E.: Synthesis with abstract examples. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 254–278. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_13
18. Einarsdóttir, S.H., Johansson, M., Åman Pohjola, J.: Into the infinite - theory exploration for coinduction. In: Fleuriot, J., Wang, D., Calmet, J. (eds.) AISC 2018. LNCS (LNAI), vol. 11110, pp. 70–86. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99957-9_5
19. Feser, J.K., Chaudhuri, S., Dillig, I.: Synthesizing data structure transformations from input-output examples. In: Grove, D., Blackburn, S. (eds.) Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15–17, 2015, pp. 229–239. ACM (2015)
20. Johansson, M.: Automated theory exploration for interactive theorem proving. In: Ayala-Rincón, M., Muñoz, C.A. (eds.) ITP 2017. LNCS, vol. 10499, pp. 1–11. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66107-0_1
21. Johansson, M., Dixon, L., Bundy, A.: Case-analysis for rippling and inductive proof. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 291–306. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14052-5_21
22. Johansson, M., Dixon, L., Bundy, A.: Conjecture synthesis for inductive theories. J. Autom. Reason. **47**, 251–289 (2010)
23. Johansson, M., Rosén, D., Smallbone, N., Claessen, K.: Hipster: integrating theory exploration in a proof assistant. In: Watt, S.M., Davenport, J.H., Sexton, A.P., Sojka, P., Urban, J. (eds.) CICM 2014. LNCS (LNAI), vol. 8543, pp. 108–122. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08434-3_9
24. Knuth, D.E., Bendix, P.B.: Simple word problems in universal algebras. In: Siekmann, J.H., Wrightson, G. (eds.) Automation of Reasoning. Symbolic Computation (Artificial Intelligence). Springer, Heidelberg (1983). https://doi.org/10.1007/978-3-642-81955-1_23
25. Kovács, L., Voronkov, A.: First-order theorem proving and Vampire. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 1–35. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_1
26. Lau, T., Wolfman, S.A., Domingos, P., Weld, D.S.: Programming by demonstration using version space algebra. Mach. Learn. **53**(1–2), 111–156 (2003)
27. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20
28. Milder, P., Franchetti, F., Hoe, J.C., Püschel, M.: Computer generation of hardware for linear digital signal processing transforms. ACM Trans. Des. Autom. Electron. Syst. **17**(2), 1–33 (2012)
29. José, M.F., et al.: Spiral: Automatic implementation of signal processing algorithms. In: HPEC, HPEC 2000 (2000)
30. Nandi, C., et al.: Synthesizing structured cad models with equality saturation and inverse transformations. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2020, pp. 31–44. Association for Computing Machinery, New York (2020)
31. Nelson, G., Oppen, D.C.: Fast decision procedures based on congruence closure. J. ACM **27**(2), 356–364 (1980)
32. Nieuwenhuis, R., Oliveras, A.: Fast congruence closure and extensions. Inf. Comput. **205**(4), 557–580 (2007)

33. Nipkow, T., Wenzel, M., Paulson, L.C. (eds.): Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9
34. Panchekha, P., Sanchez-Stern, A., Wilcox, J.R., Tatlock, Z.: Automatically improving accuracy for floating point expressions. In: PLDI, vol. 50, pp. 1–11. ACM, New York (2015)
35. Polozov, O., Gulwani, S.: Flashmeta: a framework for inductive program synthesis. SIGPLAN Not. **50**(10), 107–126 (2015)
36. Ragan-Kelley, J., et al.: Halide: decoupling algorithms from schedules for high-performance image processing. Commun. ACM **61**(1), 106–115 (2018)
37. Reynolds, A., Kuncak, V.: Induction for SMT solvers. In: D'Souza, D., Lal, A., Larsen, K.G. (eds.) VMCAI 2015. LNCS, vol. 8931, pp. 80–98. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46081-8_5
38. Smallbone, N., Johansson, M., Claessen, K., Algehed, M.: Quick specifications for the busy programmer. J. Funct. Program. **27**, e18 (2017)
39. Smith, C., Albarghouthi, A.: Program synthesis with equivalence reduction. In: Enea, C., Piskac, R. (eds.) VMCAI 2019. LNCS, vol. 11388, pp. 24–47. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-11245-5_2
40. Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., Saraswat, V.: Combinatorial sketching for finite programs. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21–25, 2006, pp. 404–415 (2006)
41. Tate, R., Stepp, M., Tatlock, Z., Lerner, S.: Equality saturation: a new approach to optimization. In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, pp. 264–276. Association for Computing Machinery, New York (2009)
42. Udupa, A., Raghavan, A., Deshmukh, J.V., Mador-Haim, S., Martin, M.M.K., Alur, R.: Transit: specifying protocols with concolic snippets. ACM SIGPLAN Not. **48**(6), 287–296 (2013)
43. Valbuena, I.L., Johansson, M.: Conditional lemma discovery and recursion induction in hipster. ECEASST **72**, 1–15 (2015)
44. Willsey, M., Nandi, C., Wang, Y.R., Flatt, O., Tatlock, Z., Panchekha, P.: Egg: fast and extensible equality saturation. Proc. ACM Program. Lang. **5**(POPL), 1–29 (2021)
45. Xiong, J., Johnson, J., Johnson, R., Padua, D.: SPL: a language and compiler for DSP algorithms. In: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation, PLDI 2001, pp. 298–308. Association for Computing Machinery, New York (2001)

CoqQFBV: A Scalable Certified SMT Quantifier-Free Bit-Vector Solver

Xiaomu Shi[1], Yu-Fu Fu[2], Jiaxiang Liu[1(✉)], Ming-Hsien Tsai[3], Bow-Yaw Wang[3], and Bo-Yin Yang[3]

[1] Shenzhen University, Shenzhen, China
[2] Georgia Institute of Technology, Atlanta, USA
[3] Academia Sinica, Taipei City, Taiwan

Abstract. We present a certified SMT QF_BV solver CoqQFBV built from a verified bit blasting algorithm, Kissat, and the verified SAT certificate checker GratChk in this paper. Our verified bit blasting algorithm supports the full QF_BV logic of SMT-LIB; it is specified and formally verified in the proof assistant Coq. We compare CoqQFBV with CVC4, Bitwuzla, and Boolector on benchmarks from the QF_BV division of the single query track in the 2020 SMT Competition, and real-world cryptographic program verification problems. CoqQFBV surprisingly solves more program verification problems with certification than the 2020 SMT QF_BV division winner Bitwuzla without certification.

1 Introduction

Satisfiability Modulo Theories (SMT) solvers for the Quantifier-Free Bit-Vector (QF_BV) logic have been used to verify programs with bit-level accuracy [9, 10]. In such applications, a program verification problem is reformulated as an SMT QF_BV query. An SMT QF_BV solver is then invoked to compute a query result. The query result in turn decides the answer to the program verification problem. For cryptographic assembly programs, a missing carry or borrow flag will result in incorrect computation. Bit-accurate verification is thus necessary for cryptographic programs. SMT QF_BV solvers in fact have been employed to verify such programs [8,25]. These solvers nonetheless are very complex programs with possibly unknown bugs [7,18]. Since bugs in SMT QF_BV solvers may induce incorrect query results, program verification cannot be taken without a grain of salt when SMT QF_BV solvers are employed.

In order to check SMT QF_BV query results independently, SMT QF_BV solvers can generate certificates to validate their answers. In the LFSC certificates [14,23], for instance, an SMT QF_BV query result is certified by correct bit blasting and Boolean Satisfiability (SAT) solving. Such certificates demonstrate that the SMT QF_BV query is reduced to a Boolean SAT query correctly *and* the corresponding SAT query is solved correctly. Although one can certify SAT query results with certificates from SAT solvers [24], it is not always easy to certify correct bit blasting due to complex arithmetic operations in SMT QF_BV

A. Silva and K. R. M. Leino (Eds.): CAV 2021, LNCS 12760, pp. 149–171, 2021.
https://doi.org/10.1007/978-3-030-81688-9_7

queries. Developing correct and efficient checkers for SMT QF_BV certificates can be very challenging. Indeed, an LFSC certificate checker based on the proof assistant Coq has been developed to improve confidence [12]. Yet the Coq-based certificate checker does not fully support arithmetic operations and thus cannot certify results of SMT QF_BV queries with complicated arithmetic operations. Consequently, the correctness of cryptographic programs still relies on the correctness of SMT QF_BV solvers or their unverified certificate checkers.

In this paper, we take a more direct approach to ensure the correctness of SMT QF_BV query results. Instead of certifying correct bit blasting for every SMT QF_BV query, we specify a bit blasting algorithm and prove its correctness in the proof assistant Coq. In order to formalize the correctness of our bit blasting algorithm, we develop a formal bit-vector theory in Coq. Naturally, the formal theory has to support all arithmetic functions (addition, subtraction, multiplication, division, and remainder) for both signed and unsigned representations as needed in SMT-LIB [3]. Based on our new bit-vector theory, we give a formal semantics for SMT QF_BV queries in Coq. Our semantics follows the SMT-LIB semantics carefully. Particularly, division and remainder are total arithmetic operations even when the divisor is zero. Using our Coq bit-vector theory and semantics, we prove that our bit blasting algorithm always returns a corresponding Boolean formula correctly on any SMT QF_BV query. Since our algorithm has been formally verified, bit blasting is always correct and need not be certified. Through the OCaml program extracted from our verified bit blasting algorithm, a corresponding SAT query is obtained for each SMTQF_BV query and sent to a SAT solver. A SAT certificate checker suffices to validate SAT query results and hence the correctness of answers to SMT QF_BV queries. Since neither complicated SMT QF_BV solvers nor their certificate checkers are trusted, our work can improve the confidence of SMT QF_BV query results.

To our knowledge, our bit-vector theory is the first Coq formalization designed for bit blasting queries from the QF_BV logic of SMT-LIB. Our semantics is the first Coq formalization for full SMT QF_BV queries. We are not aware of any verified bit blasting algorithm or program for full SMT QF_BV queries of SMT-LIB at the time of writing. Even the correctness of its results could be ensured, our certified SMT QF_BV solver CoqQFBV would not be very useful if it were extremely inefficient. In order to evaluate its performance, we run CoqQFBV on benchmarks from the QF_BV division of the single query track in the 2020 SMT Competition. With the same memory and time limits in the competition, our solver successfully finishes 88.72% of the 6861 queries with certification. In comparison, CVC4 with its certificate checker solves 55.97% with certification, and the division winner Bitwuzla solves 98.22% of the benchmarks without certification. Our certified solver outperforms CVC4 with certification significantly. Generating and checking certificates make our certified solver finish about 10% of the queries less than the division winner. The price of accuracy perhaps is not unacceptable for the benchmarks in the competition. To further evaluate CoqQFBV, the certified solver is used to verify linear arithmetic assembly programs from various cryptography libraries such as OpenSSL [30].

CoqQFBV gives certified answers to 96.88% out of the 96 SMT QF_BV queries from real-world cryptographic program verification. CVC4 with its certificate checker certifies 19.79%. Compared with efficient SMT QF_BV solvers without certification, Boolector is able to solve 100% and Bitwuzla solves 91.67% of the queries. Intriguingly, our certified SMT QF_BV solver outperforms the 2020 division winner Bitwuzla in queries from real-world verification problems. Our certified solver is probably useful for real-world verification problems.

Related Work. As mentioned, SMT certificate generating and checking are challenging. There are few efforts developing SMT QF_BV certificate checkers, let alone verified ones. CVC4 is able to produce unsatisfiability certificates for QF_BV queries, and also equipped with an (unverified) certificate checker [14]. SMTCoq [12] is proposed to check certificates from SMT solvers veriT and CVC4. It supports fragments of several logics including the QF_BV logic. Moreover, its correctness is formally proved in Coq. However, the QF_BV logic is not fully supported by SMTCoq. Z3 also supports certificate generation for the QF_BV logic [19]. The proofs can be reconstructed, thus checked, within proof assistants HOL4 and Isabelle [6]. But the lack of details in Z3's generated certificates makes proof reconstruction particularly challenging.

With a similar approach in this paper, GL is a framework for bit blasting finitely bounded ACL2 theorems into SAT queries [28]. Its bit blasting algorithm is formally verified in ACL2. Though it is not designed for SMT-LIB, most of the operations defined in the QF_BV logic are supported, except division and concatenation for instance. A bit blasting algorithm is defined and verified in HOL4 as well [13]. Neither [28] nor [13] aims to develop a scalable SMT QF_BV solver. CoqQFBV accepts SMT-LIB inputs with fully supported QF_BV logic while adopting performance optimizations such as caches.

In Isabelle and HOL4, one can use the bit-vector libraries to conform SMT-LIB operations, see [17] for example. Under the frame of Coq, coq-bits is a formalization of logical and arithmetic operations on bit-vectors [15]. The library provides the mapping between bit-vector operations and abstract number operations. Different from our theory, it does not support division/remainder or signed operations. Why3 [11] provides a bit-vector theory which is formalized in Coq too. It defines the division by zero in a different way from SMT-LIB. Moreover, the operations are defined based on integer operations. Our new bit-vector theory instead defines bit-vector operations through bit manipulation. It is more suitable for the correctness proof of bit blasting algorithms.

We have the following organization. After the introduction, an overview is given in Sect. 2. Section 3 reviews preliminaries. Our formal bit-vector theory is presented in Sect. 4. It is followed by the formal semantics of SMT QF_BV queries (Sect. 5). The correctness of our bit blasting algorithm is established in Sect. 6. Section 7 outlines the construction of our certified SMTQF_BV solver. Experiments are presented in Sect. 8. Section 9 concludes our presentation.

2 Methodology Overview

Given an SMT QF_BV query, a bit blasting algorithm computes a Boolean formula such that the SMT QF_BV query is satisfiable if and only if the Boolean formula is satisfiable. The QF_BV logic contains arithmetic operations for bit-vectors. Computing an equi-satisfiable Boolean formula for an arbitrary SMT QF_BV query can be very complicated and susceptible to errors. Our goal is to construct a correct bit blasting program for every SMT QF_BV query. The correctness of the program moreover is verified by the proof assistant Coq to minimize gaps or even errors in hand-written proofs.

Our construction is based on a new formal bit-vector theory `coq-nbits` (Sect. 4). In `coq-nbits`, we define bit-vectors and their functions on top of the Coq data type for Boolean sequences. In order to support the QF_BV logic of SMT-LIB fully, five arithmetic bit-vector functions (addition, subtraction, multiplication, division, and remainder) are defined in our formal theory. To establish the correctness of our definitions, formal proofs are provided to relate bit-vector functions with their arithmetic counterparts. For instance, we show the number represented by the output of the bit-vector negation function is indeed the arithmetic negation of the number represented by the input bit-vector.

Using our `coq-nbits` theory, we then give a formal semantics for SMTQF_BV queries as defined in SMT-LIB (Sect. 5). In our formalization, a QF_BV predicate denotes a Boolean value; and a QF_BV expression denotes a bit-vector. An SMT QF_BV query is formalized as a Boolean combination of QF_BV predicates on QF_BV expressions over QF_BV variables and bit-vector constants. In order to demonstrate the correctness of our formal semantics for SMT QF_BV queries, formal proofs are provided to show that our formal semantics coincides with those defined in SMT-LIB.

Our bit blasting algorithm is given in Coq (Sect. 6). It extends Tseitin transformation for Boolean formulae to SMT QF_BV queries. More precisely, a QF_BV predicate is transformed to a literal with a Boolean formula; a QF_BV expression is transformed to a literal sequence with a Boolean formula. Using our formalization of SMT QF_BV queries, the correctness of bit blasting algorithm is established in Coq by mutual induction. To improve efficiency, our bit blasting algorithm is further optimized with more economic transformations and a cache. The optimized bit blasting algorithm is also verified with formal Coq proofs.

Our formally verified bit blasting algorithm is written in the Coq specification language. It is not yet a program compilable into executable binary codes. Using the code extraction mechanism in Coq, an OCaml program is extracted from our verified bit blasting algorithm. The OCaml program takes expressions in our formal SMT QF_BV query syntax as inputs and returns expressions in our formal syntax for Boolean formulae as outputs. SAT solvers can be employed to decide satisfiability of output Boolean formulae. Their certificates can be validated by SAT certificate checkers independently (Sect. 7).

3 Preliminaries

Let v be a Boolean *variable* with values *ff* and *tt*. A *literal* is of the form v or $\neg v$. A *clause* is a disjunction $l_0 \vee l_1 \vee \cdots \vee l_k$ of literals $l_0, l_1, \ldots, l_k$. A Boolean formula in the *conjunctive normal form (CNF)* is a conjunction $c_0 \wedge c_1 \wedge \cdots \wedge c_m$ of clauses $c_0, c_1, \ldots, c_m$. A SAT *query* is a Boolean CNF formula. An *environment* maps Boolean variables to their values. Given a SAT query, the *Boolean satisfiability problem* is to decide if the query evaluates to *tt* on some environments.

A bit-vector of *width* w is written as $\texttt{\#b}b_{w-1}b_{w-2}\cdots b_0$ with $b_i \in \{0, 1\}$ for $0 \leq i < w$. In the *unsigned* representation, the bit-vector $\texttt{\#b}b_{w-1}b_{w-2}\cdots b_0$ denotes the natural number (non-negative integer) $\sum_{0\leq i<w} b_i 2^i$; in *two's complement (signed)* representation, it denotes the integer $\sum_{0\leq i<w-1} b_i 2^i - 2^{w-1}b_{w-1}$. For instance, `#b1010` denotes 10 and -6 in the unsigned and two's complement representations respectively. We use $\mathit{bv2nat}(bv)$ for the natural number denoted by the bit-vector bv in the unsigned representation; and $\mathit{nat2bv}(w, i)$ stands for the bit-vector of width w representing the natural number i modulo 2^w.

Let $bv = \texttt{\#b}b_{w-1}b_{w-2}\cdots b_0$ and $cv = \texttt{\#b}c_{u-1}c_{u-2}\cdots c_0$ be bit-vectors of widths w and u respectively. The following QF_BV *operations* are defined in the QF_BV logic of SMT-LIB: $\mathit{concat}\ bv\ cv \triangleq \texttt{\#b}b_{w-1}b_{w-2}\cdots b_0c_{u-1}c_{u-2}\cdots c_0$ is the concatenation of bv and cv; $\mathit{extract}\ i\ j\ bv \triangleq \texttt{\#b}b_ib_{i-1}\cdots b_j$ extracts bits from bv where $0 \leq j \leq i < w$; $\mathit{bvnot}\ bv$, $\mathit{bvand}\ bv\ cv$, and $\mathit{bvor}\ bv\ cv$ are the bitwise complement, and, or operations respectively. Additionally, $\mathit{bvneg}\ bv \triangleq \mathit{nat2bv}(w, 2^w - \mathit{bv2nat}(bv))$ is the arithmetic negation operation; $\mathit{bvadd}\ bv\ cv \triangleq \mathit{nat2bv}(w, \mathit{bv2nat}(bv) + \mathit{bv2nat}(cv))$ is the arithmetic addition operation; and $\mathit{bvmul}\ bv\ cv \triangleq \mathit{nat2bv}(w, \mathit{bv2nat}(bv) \times \mathit{bv2nat}(cv))$ is the arithmetic multiplication operation. The arithmetic division and remainder operations are

$$\mathit{bvudiv}\ bv\ cv \triangleq \begin{cases} \mathit{nat2bv}(w, 2^w - 1) & \text{if } \mathit{bv2nat}(cv) = 0 \\ \mathit{nat2bv}(w, \mathit{bv2nat}(bv) \div \mathit{bv2nat}(cv)) & \text{otherwise} \end{cases}$$
$$\mathit{bvurem}\ bv\ cv \triangleq \begin{cases} bv & \text{if } \mathit{bv2nat}(cv) = 0 \\ \mathit{nat2bv}(w, \mathit{bv2nat}(bv) \bmod \mathit{bv2nat}(cv)) & \text{otherwise.} \end{cases}$$

Note that the arithmetic division and remainder operations are defined even when the divisor represents the number zero. Finally, the operations $\mathit{bvshl}\ bv\ cv \triangleq \mathit{nat2bv}(w, \mathit{bv2nat}(bv) \times 2^{\mathit{bv2nat}(cv)})$ shifts the bit-vector bv to the left by $\mathit{bv2nat}(cv)$ bits; $\mathit{bvlshr}\ bv\ cv \triangleq \mathit{nat2bv}(w, \mathit{bv2nat}(bv) \div 2^{\mathit{bv2nat}(cv)})$ shifts the bit-vector bv to the right by $\mathit{bv2nat}(cv)$ bits. In addition to bit-vector operations, the QF_BV logic of SMT-LIB defines QF_BV *predicates* on bit-vectors. The predicate $\mathit{bveq}\ bv\ cv$ is true when the bit-vectors bv and cv are equal; $\mathit{bvult}\ bv\ cv$ is true if $\mathit{bv2nat}(bv) < \mathit{bv2nat}(cv)$. In the QF_BV logic of SMT-LIB, both operands of binary operations and predicates must have the same width. Overall, seventeen bit-vector operations and predicates are defined in the QF_BV logic of SMT-LIB. Particularly, arithmetic division and remainder operations with operands in both unsigned and two's complement signed representations are defined in SMT-LIB.

A QF_BV *variable* denotes a bit-vector. A QF_BV *expression* is constructed from QF_BV operations over QF_BV variables and bit-vectors. An SMT QF_BV *query* is a Boolean combination of QF_BV predicates on QF_BV expressions. Let *stores* be mappings from QF_BV variables to bit-vectors. Given an SMT QF_BV query, the *satisfiability modulo* QF_BV *theory problem* is to decide if the query evaluates to *tt* on some stores.

4 Bit-Vector Theory

We present our formal Coq bit-vector theory `coq-nbits` in this section. The `coq-nbits` theory supports bit-vectors in both unsigned and two's complement signed representations. In `coq-nbits`, a bit-vector is represented by a Boolean sequence of the data type bits in the least significant bit-first order.

```
Definition bits : Set := seq bool.
```

In the definition, `bool` and `seq` are the data types for Boolean values (`false` and `true`) and sequences in Coq respectively. For instance, the bit-vector `#b100` is represented by [:: `false`; `false`; `true`] in `coq-nbits`.

Coq functions defined for sequences are applicable to bit-vectors. Particularly, size *bv* computes the width of the bit-vector *bv* and *bv* ++ *cv* is the concatenation of the bit-vectors *bv* and *cv*. It is also straightforward to define auxiliary bit-vector functions. For example, zeros n returns the bit-vector of n `false`'s; ones n returns the bit-vector of n `true`'s; extract i j *bv* returns the sub-sequence of the bit-vector *bv* with indices from j to i where $0 \leq j \leq i <$ size *bv*. Let a $\triangleq$ [:: `false`; `false`; `true`]. Then size a $= 3$ and extract 2 1 a = [:: `false`; `true`].

Bitwise functions are defined as easily. For instance, the bitwise inverse function maps each Boolean value to its complement:

```
Definition invB bv : bits := map (fun b => ~~b) bv.
```

Other bitwise functions are defined similarly. Specifically, bitwise and andB, bitwise or orB, logical left shift shlB, logical right shift shrB are all defined in `coq-nbits`. Let b $\triangleq$ [:: `false`; `true`; `true`]. We have invB b = [:: `true`; `false`; `false`], andB a b = [:: `false`; `false`; `true`], and shlB 1 b = [:: `false`; `false`; `true`].

Arithmetic bit-vector functions are slightly more complicated. To prove properties about arithmetic functions, `coq-nbits` provides conversion functions between bit-vectors and natural numbers.

```
Definition to_N (bv : bits) : N :=
  foldr (fun b res => N_of_bool b + res * 2) 0 bv.
```

In the definition, to_N *bv* converts the bit-vector *bv* to a natural number where `N_of_bool false` = 0 and `N_of_bool true` = 1. The to_N function multiplies the previous result by two and adds the least significant bit b. For instance, to_N a = to_N [:: `false`; `false`; `true`] = 4. The function from_N w n, on the other hand, converts any natural number n to a bit-vector of width w.

```
Fixpoint from_N (w : nat) (n : N) : bits :=
  match w with
  | O => [::]
  | S w' => (N.odd n)::(from_N w' (N.div n 2))
  end.
```

The function first checks the width w. If the width is zero, it returns the empty bit-vector. Otherwise, the function returns the bit-vector with the least significant bit `N.odd` n and the remaining $w - 1$ bits representing n divided by two. Observe that two Coq formalizations of natural numbers are used. The `nat` theory uses the unary representation suitable for inductive proofs; `N` uses the succinct binary representation. The following lemma is proved in Coq:

Lemma 1. *The following properties hold:*

1. $\forall bv, \mathsf{from\_N}\ (\mathsf{size}\ bv)\ (\mathsf{to\_N}\ bv) = bv.$
2. $\forall w\ n, n < 2^w \implies \mathsf{to\_N}\ (\mathsf{from\_N}\ w\ n) = n.$

The first property shows that bit-vectors can be converted to natural numbers and back to themselves. The second property shows that natural numbers can be converted to bit-vectors with sufficient widths and back to themselves. To see how they are used to prove properties about bit-vector functions in `coq-nbits`, consider the definition of the successor bit-vector function.

```
Fixpoint succB (bv : bits) : bits :=
  match bv with
  | [::] => [::]
  | hd::tl => if hd then false::(succB tl) else true::tl
  end.
```

If the input is the empty bit-vector, the function returns the empty bit-vector. Otherwise, succB checks the least significant bit of the input bit-vector. If the bit is `true`, the function computes the successor of the remaining bits and appends `false` as the least significant bit. If the least significant bit of the input is `false`, the function simply changes the least significant bit to `true` and copies the remaining bits. Using the conversion functions, the bit-vector successor is related to the arithmetic successor in the following lemma:

Lemma 2. $\forall bv, \mathsf{succB}\ bv = \mathsf{from\_N}\ (\mathsf{size}\ bv)\ ((\mathsf{to\_N}\ bv) + 1).$

Lemma 2 says that succB bv does compute the bit-vector representing the arithmetic successor of the natural number represented by the bit-vector bv. Observe that the successor bit-vector function is correct when the input bit-vector is empty. It is also correct when there is overflow. Indeed, both sides are zeros of width size bv when overflow occurs.

Other arithmetic bit-vector functions are defined and proved in `coq-nbits` similarly. Specifically, the arithmetic negation negB, addition addB, subtraction subB, unsigned multiplication mulB, unsigned division divB, and unsigned remainder remB functions are supported by `coq-nbits`. We give properties to relate the arithmetic functions for bit-vectors and natural numbers.

Lemma 3. *The following properties hold:*

1. $\forall bv\ cv, \mathsf{size}\ bv = \mathsf{size}\ cv \implies \mathsf{to\_N}\ (\mathsf{addB}\ bv\ cv) = (\mathsf{to\_N}\ bv + \mathsf{to\_N}\ cv) \mathtt{\ mod\ } 2^{\mathsf{size}\ bv}$.
2. $\forall bv\ cv, \mathsf{to\_N}\ (\mathsf{mulB}\ bv\ cv) = (\mathsf{to\_N}\ bv \times \mathsf{to\_N}\ cv) \mathtt{\ mod\ } 2^{\mathsf{size}\ bv}$.
3. $\forall bv\ n, \mathsf{divB}\ bv\ (\mathsf{zeros}\ n) = \mathsf{ones}\ (\mathsf{size}\ bv)$.
4. $\forall bv\ bv, \mathsf{size}\ bv = \mathsf{size}\ cv \implies cv \neq \mathsf{zeros}\ (\mathsf{size}\ cv) \implies \mathsf{to\_N}\ (\mathsf{divB}\ bv\ cv) = (\mathsf{to\_N}\ bv) \mathtt{\ div\ } (\mathsf{to\_N}\ cv)$.
5. $\forall bv\ n, \mathsf{remB}\ bv\ (\mathsf{zeros}\ n) = bv$.
6. $\forall bv\ cv, \mathsf{size}\ bv = \mathsf{size}\ cv \implies cv \neq \mathsf{zeros}\ (\mathsf{size}\ cv) \implies \mathsf{to\_N}\ (\mathsf{remB}\ bv\ cv) = (\mathsf{to\_N}\ bv) \mathtt{\ mod\ } (\mathsf{to\_N}\ cv)$.
7. $\forall bv\ n, \mathsf{to\_N}\ (\mathsf{shlB}\ n\ bv) = ((\mathsf{to\_N}\ bv) \times 2^n) \mathtt{\ mod\ } 2^{\mathsf{size}\ bv}$
8. $\forall bv\ n, \mathsf{to\_N}\ (\mathsf{shrB}\ n\ bv) = (\mathsf{to\_N}\ bv) \mathtt{\ div\ } 2^n$.

Let bv, cv be bit-vectors of width w. Lemma 3 shows that the natural number represented by the bit-vector addB $bv\ cv$ is equal to the modular sum of the natural numbers represented by bv and cv. Similarly, the natural number represented by mulB $bv\ cv$ is equal to the modular product of the natural numbers represented by bv and cv. The division and remainder functions in `coq-nbits` follow the SMT-LIB semantics. Specifically, the quotient of any bit-vector divided by zero is equal to the bit-vector of all `true`'s; the remainder of a bit-vector divided by zero is the bit-vector itself. For non-zero divisors, the division and remainder functions behave as expected. The natural number represented by the bit-vector divB $bv\ cv$ is the quotient of the number represented by bv divided by the number represented by cv; and the bit-vector remB $bv\ cv$ represents the remainder of the number represented by bv divided by the number represented by cv. Last but not least, the logical left (shlB) and right (shrB) shifts correspond to multiplication and division by powers of two respectively.

`coq-nbits` also provides comparison predicates. In addition to the equality predicate == inherited from Boolean sequences, ltB $bv\ cv$ and leB $bv\ cv$ compare the natural numbers represented by the bit-vectors bv and cv. Properties about comparison predicates have also been proved in Coq.

Lemma 4. *The following properties hold:*

1. $\forall bv\ cv, \mathsf{size}\ bv = \mathsf{size}\ cv \implies \mathsf{ltB}\ bv\ cv = (\mathsf{to\_N}\ bv < \mathsf{to\_N}\ cv)$.
2. $\forall bv\ cv, \mathsf{size}\ bv = \mathsf{size}\ cv \implies \mathsf{leB}\ bv\ cv = (\mathsf{to\_N}\ bv \leq \mathsf{to\_N}\ cv)$.

In addition to arithmetic functions and predicates in the unsigned representation, our formal bit-vector theory moreover defines arithmetic functions and predicates for bit-vectors in two's complement representation. For the signed representation, bit-vectors are converted to integers by the to_Z function. Arithmetic bit-vector functions and predicates in the signed representation are related to arithmetic integer functions and predicates as follows.

Lemma 5. *The following properties hold:*

1. $\forall bv, \neg(\mathsf{msb}\ bv \wedge \mathsf{dropmsb}\ bv = \mathsf{zeros}\ (\mathsf{size}\ bv - 1)) \implies \mathsf{to\_Z}\ (\mathsf{negB}\ bv) = -\mathsf{to\_Z}\ bv$.

2. $\forall bv\ n, 1 < \mathsf{size}\ bv \implies \mathsf{to\_Z}\ (\mathsf{sarB}\ n\ bv) = (\mathsf{to\_Z}\ bv)\ \texttt{quot}\ 2^n$.
3. $\forall bv\ cv, \mathsf{size}\ bv = \mathsf{size}\ cv \implies \mathsf{to\_Z}\ (\mathsf{mulB}\ (\mathsf{sext}\ (\mathsf{size}\ cv)\ bv)\ (\mathsf{sext}\ (\mathsf{size}\ bv)\ cv)) = \mathsf{to\_Z}\ bv \times \mathsf{to\_Z}\ cv$.
4. $\forall bv\ cv, 1 < \mathsf{size}\ bv \implies \mathsf{size}\ bv = \mathsf{size}\ cv \implies [\neg(\mathsf{msb}\ bv \wedge \mathsf{dropmsb}\ bv = \mathsf{zeros}\ (\mathsf{size}\ bv - 1)) \vee cv \neq \mathsf{ones}\ (\mathsf{size}\ cv)] \implies \mathsf{to\_Z}\ (\mathsf{sdivB}\ bv\ cv) = (\mathsf{to\_Z}\ bv)\ \texttt{quot}\ (\mathsf{to\_Z}\ cv)$.
5. $\forall bv\ cv, 1 < \mathsf{size}\ bv \implies \mathsf{size}\ bv = \mathsf{size}\ cv \implies \mathsf{to\_Z}\ (\mathsf{sremB}\ bv\ cv) = (\mathsf{to\_Z}\ bv)\ \texttt{rem}\ (\mathsf{to\_Z}\ cv)$.
6. $\forall bv\ cv, \mathsf{size}\ bv = \mathsf{size}\ cv \implies \mathsf{sltB}\ bv\ cv = (\mathsf{to\_Z}\ bv < \mathsf{to\_Z}\ cv)$.
7. $\forall bv\ cv, \mathsf{size}\ bv = \mathsf{size}\ cv \implies \mathsf{sleB}\ bv\ cv = (\mathsf{to\_Z}\ bv \leq \mathsf{to\_Z}\ cv)$.

In the lemma, sext n bv extends the bit-vector bv by n bits with the sign bit of bv, msb bv returns the sign bit of bv, and dropmsb bv drops the sign bit of bv. `quot` and `rem` are the quotient and remainder functions for Coq integers. Consider, for instance, the signed division function sdivB bv cv in `coq-nbits` (Lemma 5(4)). If the dividend bv is of width > 1, the widths of bv and the divisor cv are equal, and bv is not of the form `#b100`$\cdots$`0` or cv is not of the form `#b11`$\cdots$`1`, then the bit-vector sdivB bv cv represents the quotient of the integers represented by bv and cv. The condition may appear counter-intuitive. To see why it is necessary, consider bv = `#b100`$\cdots$`0` and cv = `#b11`$\cdots$`1` both of width w. bv and cv thus represent the integers -2^{w-1} and -1 respectively. Their quotient 2^{w-1} however cannot be represented by bit-vectors of width w in two's complement representation. The corner input case is hence excluded. The corner case is also excluded from the arithmetic negation function (Lemma 5(1)).

The `coq-nbits` theory has several important differences from the prior Coq formalization in [15]. Our formal bit-vector theory supports both unsigned and two's complement signed representations. It also provides the arithmetic division and remainder functions. Since these features are needed in the QF_BV logic of SMT-LIB, they are essential to the formalization of SMT QF_BV queries. Such important features unfortunately are lacking in the prior formalization. Another noted difference is the numeric representations used in theory developments. Since integers are needed for the QF_BV logic, `coq-nbits` naturally uses binary representations for integers and natural numbers in Coq. The prior formalization on the other hand is mainly based on the unary natural number representation but provides conversion to positive integers in the binary representation.

5 Theory for SMT QF_BV Queries

Using `coq-nbits`, we formalize SMT QF_BV queries. Our formalization consists of two parts: a syntactic representation for SMT QF_BV queries in Coq inductive types and a formal semantics in our bit-vector theory `coq-nbits`.

5.1 Syntax of SMT QF_BV Queries

An SMT QF_BV query is a Coq term of the data type `bexp`. It can be constants `Bfalse` or `Btrue`, a unary predicate `Bnot`, or binary predicates `Band` or `Bor` for

Boolean connectives. Additionally, `Bbveq` and `Bbvult` with two arguments of the data type `exp` are binary QF_BV predicates.

```
Inductive bexp : Type := Bfalse : bexp | Btrue : bexp
| Bnot : bexp -> bexp
| Band : bexp -> bexp -> bexp | Bor : bexp -> bexp -> bexp
| Bbveq : exp -> exp -> bexp  | Bbvult : exp -> exp -> bexp
(* other QF_BV predicates *)
end with exp : Type :=
| Evar : var -> exp           | Econst : bits -> exp
| Ebvnot : exp -> exp
| Ebvand : exp -> exp -> exp  | Ebvor : exp -> exp -> exp
| Ebvshl : exp -> exp -> exp  | Ebvlshr : exp -> exp -> exp
| Ebvneg : exp -> exp
| Ebvadd : exp -> exp -> exp  | Ebvmul : exp -> exp -> exp
| Ebvudiv : exp -> exp -> exp | Ebvurem : exp -> exp -> exp
| Eextract : nat -> nat -> exp -> exp
| Econcat : exp -> exp -> exp
(* other QF_BV operations *)
| Ebvsub : exp -> exp -> exp
end.
```

A Coq term of the data type `exp` represents a QF_BV expression. It can be a QF_BV variable `Evar` *vid* with a variable identifier *vid* : `var`, a bit-vector constant `Econst` *bv* with *bv* : bits, a bitwise-not operation `Ebvnot` e_0, a bitwise-and operation `Ebvand` e_0 e_1, a bitwise-or operation `Ebvor` e_0 e_1, a logical left-shift operation `Ebvshl` e_0 e_1, or a logical right-shift operation `Ebvlshr` e_0 e_1. For arithmetic operations, there are `Ebvneg` e_0 for negation, `Ebvadd` e_0 e_1 for addition, `Ebvmul` e_0 e_1 for multiplication, `Ebvudiv` e_0 e_1 for unsigned division, and `Ebvurem` e_0 e_1 for unsigned remainder with e_0, e_1 : `exp`. Finally, the extraction `Eextract` i j e_0 and the concatenation `Econcat` e_0 e_1 operations have the data type `exp` with i, j : `nat` and e_0, e_1 : `exp`.

5.2 Semantics of SMT QF_BV Queries

In our Coq formalization, an SMT QF_BV query is interpreted on stores. A *store* is a mapping from QF_BV variables to bits. Let σ be a store. The interpretation of *be* : `bexp` on σ is a Boolean value; the interpretation of e : `exp` on σ is a bit-vector. Semantic functions `eval_bexp` and `eval_exp` are as follows.

```
Fixpoint eval_bexp (be : bexp) (σ : store) : bool :=
  match be with
  | Bfalse => false
  | Btrue => true
  | Bnot be0 => ~~ (eval_bexp be0 σ)
  | Band be0 be1 => (eval_bexp be0 σ) && (eval_bexp be1 σ)
  | Bor be0 be1 => (eval_bexp be0 σ) || (eval_bexp be1 σ)
  | Bbveq e0 e1 => (eval_exp e0 σ) == (eval_exp e1 σ)
  | Bbvult e0 e1 => ltB (eval_exp e0 σ) (eval_exp e1 σ)
```

```
(* other QF_BV predicates *)
end with eval_exp (e : exp) (σ : store) : bits :=
match e with
| Evar v => Store.acc v σ
| Econst bv => bv
| Ebvnot e0 => invB (eval_exp e0 σ)
| Ebvand e0 e1 => andB (eval_exp e0 σ) (eval_exp e1 σ)
| Ebvor e0 e1 => orB (eval_exp e0 σ) (eval_exp e1 σ)
| Ebvshl e0 e1 => shlB (to_nat (eval_exp e1 σ)) (eval_exp e0 σ)
| Ebvlshr e0 e1 => shrB (to_nat (eval_exp e1 σ)) (eval_exp e0 σ)
| Ebvneg e0 => negB (eval_exp e0 σ)
| Ebvadd e0 e1 => addB (eval_exp e0 σ) (eval_exp e1 σ)
| Ebvmul e0 e1 => mulB (eval_exp e0 σ) (eval_exp e1 σ)
| Ebvudiv e0 e1 => divB (eval_exp e0 σ) (eval_exp e1 σ)
| Ebvurem e0 e1 => remB (eval_exp e0 σ) (eval_exp e1 σ)
| Eextract i j e0 => extract i j (eval_exp e0 σ)
| Econcat e0 e1 => (eval_exp e1 σ) ++ (eval_exp e0 σ)
(* other QF_BV operations *)
| Ebvsub e0 e1 => subB (eval_exp e0 σ) (eval_exp e1 σ)
end.
```

An SMT QF_BV query denotes a value in the Coq data type `bool`. `Bfalse` and `Btrue` denote `false` and `true` respectively. Boolean negation, conjunction, and disjunction correspond to `~~`, `&&`, and `||` in `bool` respectively. For QF_BV predicates, the bit-vector equality `Bbveq` is interpreted by the equality `==` for Boolean sequences. The `coq-nbits` function ltB is used to interpret `Bbvult`.

A QF_BV expression denotes a bit-vector. For basic cases, QF_BV variables are interpreted by corresponding bit-vectors in the store σ through the store access function `Store.acc`; bit-vector constants are interpreted by themselves. Bitwise logical operations `Ebvnot`, `Ebvand`, and `Ebvor` are interpreted by corresponding `coq-nbits` functions invB, andB, and orB respectively. For logical shift operations, the offset e_1 is first converted to a natural number through to_nat (eval_exp e_1 σ) and then passed to the corresponding logical shift functions shlB or shrB in `coq-nbits`. QF_BV arithmetic operations are interpreted by corresponding `coq-nbits` arithmetic functions as expected. Finally, the extraction `Eextract` and concatenation `Econcat` operations are interpreted by extract and ++ in `coq-nbits` respectively.

In an SMT QF_BV query, a QF_BV variable designates a bit-vector of a certain width. An SMT QF_BV query is hence associated with a *signature* Σ mapping QF_BV variables to their respective widths. A store σ *conforms* to a signature Σ if the interpretation of each QF_BV variable on σ has the same width as specified in Σ. Given an SMT QF_BV query *be* : `bexp` with its signature Σ, *be* is *satisfiable* if there is a store σ conforming to Σ and `eval_bexp` *be* σ = `true`.

5.3 Derived QF_BV Operations and Predicates

In the QF_BV logic of SMT-LIB, a number of QF_BV operations and predicates are derived from a small set of core operations and predicates. Consider the signed comparison predicate *bvslt bv cv* in SMT-LIB:

$$\begin{aligned} \mathit{bvslt}\ \mathit{bv}\ \mathit{cv} \triangleq (\mathit{or}\ (\mathit{and}\ &(=\ (\mathit{extract}\ (w-1)\ (w-1)\ \mathit{bv})\ \texttt{\#b1})\\ &(=\ (\mathit{extract}\ (w-1)\ (w-1)\ \mathit{cv})\ \texttt{\#b0}))\\ (\mathit{and}\ &(=\ (\mathit{extract}\ (w-1)\ (w-1)\ \mathit{bv})\\ &\qquad (\mathit{extract}\ (w-1)\ (w-1)\ \mathit{cv}))\\ &(\mathit{bvult}\ \mathit{bv}\ \mathit{cv}))). \end{aligned}$$

To compare two bit-vectors of width w in two's complement representation, the sign bits are checked. If *bv* is negative but *cv* is positive, *bvslt bv cv* is true. Otherwise, the signed predicate checks that both operands have the same sign and compares the operands using the unsigned comparison predicate. Interestingly, the arithmetic subtraction operation is actually a derived operation in SMT-LIB: $\mathit{bvsub}\ \mathit{bv}\ \mathit{cv} \triangleq \mathit{bvadd}\ \mathit{bv}\ (\mathit{bvneg}\ \mathit{cv})$. The arithmetic operation is defined to be the bit-vector sum of minuend and the negation of subtrahend. It is *not*, for instance, defined as $\mathit{nat2bv}(w, \mathit{bv2nat}(\mathit{bv}) - \mathit{bv2nat}(\mathit{cv}))$ because $\mathit{bv2nat}(\mathit{bv}) - \mathit{bv2nat}(\mathit{cv})$ may not be a natural number.

For derived operations and predicates, there is a subtle yet important difference between our formal semantics and those defined in SMT-LIB. In our formal bit-vector theory `coq-nbits`, most functions and predicates are defined directly. Particularly, the arithmetic subtraction function subB is defined by one-bit subtractors in `coq-nbits`. Our formal semantics for the QF_BV arithmetic operation *bvsub* therefore is defined by the corresponding bit-vector function subB. Since our formal semantics did not define *bvsub* by *bvadd* and *bvneg*, it could be different from those in SMT-LIB. In order to build a certified solver for the QF_BV logic of SMT-LIB, it is necessary to establish semantic equivalences between both semantic definitions for all derived QF_BV operations and predicates.

To justify our formal semantics, we show the semantics of our definitions and those of SMT-LIB indeed denote the same bit-vector functions or predicates. Consider again the subtraction operation. Recall the semantics of the arithmetic operations *bvadd* and *bvneg* are defined by the bit-vector functions addB and negB respectively. The next lemma is useful to show the semantic equivalence:

Lemma 6. $\forall bv\ cv, \mathsf{size}\ bv = \mathsf{size}\ cv \implies \mathsf{subB}\ bv\ cv = \mathsf{addB}\ bv\ (\mathsf{negB}\ cv)$.

For all derived QF_BV operations and predicates, we give Coq proofs for the equivalence between our formal semantics and those of SMT-LIB. Particularly, semantics of all QF_BV arithmetic operations and predicates over two's complement representation are equivalent to those in SMT-LIB. Our formal semantics for QF_BV queries is thus certified to be equivalent to SMT-LIB.

6 Certified Bit Blasting

Recall that a SAT query is a Boolean CNF formula. Given an SMT QF_BV query, a bit blasting algorithm computes a SAT query that is satisfiable if and only if the given SMT QF_BV query is satisfiable. Although it is the standard technique for solving SMT QF_BV queries, bit blasting can be very complex due to arithmetic operations and various optimizations. Bit blasting algorithms therefore can be tedious to construct and thus prone to errors. We verify a bit blasting algorithm for SMT QF_BV queries using our Coq formalization.

Let us start with a simple formalization of Boolean CNF formulae. In our formalization, a clause is represented by a sequence of literals; a CNF formula in turn is represented by a sequence of clauses. Let `bvar` be the data type for Boolean variables. We have the following data types in Coq:

```
Inductive lit : Set := Pos of bvar | Neg of bvar.
Definition clause : Set := seq lit.
Definition CNF : Set := seq clause.
```

Define an *environment* ϵ to be a mapping from `bvar` to `bool`. Given a literal ℓ, a CNF formula f, and an environment ϵ, it is straightforward to define the semantic functions `eval_lit` ℓ ϵ : `bool` and `eval_cnf` f ϵ : `bool`. A SAT query f is *satisfiable* if there is an environment ϵ such that `eval_cnf` f ϵ = `true`.

To illustrate how our Coq proof works, consider Tseitin transformation for the logical negation operation:

```
Definition bit_blast_Bnot ℓ : lit * CNF :=
  let r := a fresh literal in
  (r, [:: [:: r; ℓ]; [:: !r; !ℓ] ]).
```

Given a literal ℓ, `bit_blast_Bnot` ℓ returns a new literal r and the CNF formula $(r \vee \ell) \wedge (\neg r \vee \neg \ell)$. Tseitin transformation ensures the interpretations of ℓ and r are complementary on any environment ϵ evaluating the CNF formula to true. We give a formal proof using our formalization in Coq:

Lemma 7. $\forall r\ \mathit{cnf}\ \ell\ \epsilon, (r, \mathit{cnf})$ = `bit_blast_Bnot` $\ell \implies$ `eval_cnf` $\mathit{cnf}\ \epsilon$ = `true` $\implies$ `eval_lit` $r\ \epsilon$ = `~~` (`eval_lit` $\ell\ \epsilon$).

The idea is generalized to QF_BV operations naturally. For each QF_BV operation, we construct a literal sequence $\vec{r}$ and a Boolean CNF formula *cnf*. If *cnf* evaluates to true on an environment ϵ, the interpretation of $\vec{r}$ on ϵ needs to reflect the semantics of the QF_BV operation. For instance, a Coq proof is given for the QF_BV addition operation:

Lemma 8. $\forall \vec{r}\ \mathit{cnf}\ \vec{\ell_0}\ \vec{\ell_1}\ \epsilon, (\vec{r}, \mathit{cnf})$ = `bit_blast_Ebvadd` $\vec{\ell_0}\ \vec{\ell_1} \implies$ `eval_cnf` $\mathit{cnf}\ \epsilon$ = `true` $\implies$ `eval_lits` $\vec{r}\ \epsilon$ = `addB` (`eval_lits` $\vec{\ell_0}\ \epsilon$) (`eval_lits` $\vec{\ell_1}\ \epsilon$).

Given two literal sequences $\vec{\ell_0}$ and $\vec{\ell_1}$, `bit_blast_Ebvadd` $\vec{\ell_0}\ \vec{\ell_1}$ returns a literal sequence $\vec{r}$ and a CNF formula *cnf*. If *cnf* evaluates to true on an environment ϵ, then the interpretation of the literal sequence $\vec{r}$ on ϵ is indeed the bit-vector sum of the interpretations of $\vec{\ell_0}$ and $\vec{\ell_1}$ on ϵ. Bit blasting algorithms for other QF_BV operations are given and shown to reflect the semantics of corresponding functions defined in the bit-vector theory `coq-nbits`. Particularly, our bit blasting algorithms for arithmetic division and remainder correctly reflect corresponding arithmetic bit-vector functions in `coq-nbits`.

Recall that the semantics for SMT QF_BV queries is defined over stores for QF_BV variables. In order to prove the correctness of bit blasting algorithms, one has to relate stores for QF_BV variables with environments for Boolean variables. The relation is explicated through literal correspondences. A *literal correspondence* π is a mapping from QF_BV variables to sequences of literals. For each QF_BV variable v, the literal sequence $\pi(v)$ is meant to interpret v on environments for Boolean variables. More formally, let `eval_lits` $\vec{\ell}\ \epsilon$: bits be the bit-vector for the literal sequence $\vec{\ell}$ interpreted on the environment ϵ. The bit-vector `eval_lits` $\pi(v)\ \epsilon$ is hence the interpretation of the QF_BV variable v on the environment ϵ. Let σ be a store and π a literal correspondence. An environment ϵ is *consistent with* σ *through* π if the bit-vectors `eval_lits` $\pi(v)\ \epsilon$ and `Store.acc` $v\ \sigma$ are equal for every QF_BV variable v in σ. Thus, an environment is consistent with a store if their interpretations of variables coincide.

It is now straightforward to give our bit blasting algorithm for SMT QF_BV queries. For each QF_BV expression, our algorithm first computes literals and CNF formulae for operands recursively. It then invokes an auxiliary bit blasting algorithm to construct result literals and a CNF formula for the QF_BV operation. The literal correspondence is also updated when literals are allocated for QF_BV variables. Finally, the result literals and the updated literal correspondence are returned along with the concatenation of all CNF formulae.

```
Definition bit_blast_bexp Σ π b : lit * correspondence * CNF :=
match be with
| Bnot be0 =>
    let (r0, π', cnf0) := bit_blast_bexp Σ π be0 in
    let (r, cnf) := bit_blast_Bnot r0 in
    (r, π', cnf ++ cnf0)
(* other QF_BV predicates *)
end with bit_blast_exp Σ π e : seq lit * correspondence * CNF :=
match e with
| Evar v =>
    if π(v) is defined then (π(v), π, [::])
    else let r⃗ := fresh literals for v according to Σ in
         let π' := update π with v ↦ r⃗ in
         (r⃗, π', [::])
| Ebvadd e0 e1 =>
    let (r⃗0, π', cnf0) := bit_blast_exp Σ π e0 in
    let (r⃗1, π'', cnf1) := bit_blast_exp Σ π' e1 in
    let (r⃗, cnf) := bit_blast_Ebvadd r⃗0 r⃗1 in
    (r⃗, π'', cnf ++ cnf0 ++ cnf1)
(* other QF_BV operations *)
end.
```

The following Coq theorem establishes the connection between the output literals and the input SMT QF_BV query or expression of the algorithm.

Theorem 1. *Let be* : bexp *be an* SMT QF_BV *query with the signature* Σ_{be}, e : exp *a* QF_BV *expression with the signature* Σ_e, *and* π_0 *the empty literal correspondence.*

1. $\forall r\ \pi\ cnf\ \sigma\ \epsilon, (r, \pi, cnf) = \texttt{bit\_blast\_bexp}\ \Sigma_{be}\ \pi_0\ be \implies \sigma$ *conforms to* $\Sigma_{be} \implies \epsilon$ *is consistent with*σ *through* $\pi \implies \texttt{eval\_cnf}\ cnf\ \epsilon = \texttt{true} \implies \texttt{eval\_lit}\ r\ \epsilon = \texttt{eval\_bexp}\ be\ \sigma$.
2. $\forall \vec{r}\ \pi\ cnf\ \sigma\ \epsilon, (\vec{r}, \pi, cnf) = \texttt{bit\_blast\_exp}\ \Sigma_e\ \pi_0\ e \implies \sigma$ *conforms to* $\Sigma_e \implies \epsilon$ *is consistent with* σ *through* $\pi \implies \texttt{eval\_cnf}\ cnf\ \epsilon = \texttt{true} \implies \texttt{eval\_lits}\ \vec{r}\ \epsilon = \texttt{eval\_exp}\ e\ \sigma$.

Let *be* be an SMT QF_BV query with the signature Σ_{be}, r and *cnf* the literal and CNF formula returned by `bit_blast_bexp` respectively. Consider any store conforming to Σ_{be} and any environment consistent with the store. If the environment evaluates the formula *cnf* to true, Theorem 1 says that the literal r and the SMT QF_BV query *be* evaluate to the same Boolean value on the environment and store respectively. In other words, the algorithm `bit_blast_bexp` is a generalized Tseitin transformation for SMT QF_BV queries. Particularly, all QF_BV arithmetic operations (addition, subtraction, multiplication, division, and remainder in the unsigned and two's complement representations) are transformed to CNF formulae with formal proofs of correctness in Coq.

A useful corollary to Theorem 1 is the reduction of the satisfiability of SMT QF_BV queries to the satisfiability of SAT queries.

Corollary 1. *Let be* : `bexp` *be an* SMT QF_BV *query with the signature* Σ_{be} *and* π_0 *the empty literal correspondence. Then*

$$\begin{aligned}\forall r\ \pi\ cnf, (r, \pi, cnf) = \texttt{bit\_blast\_bexp}\ \Sigma_{be}\ \pi_0\ be \implies \\ [(\exists \sigma, \sigma \textit{ conforms to } \Sigma_{be} \land \texttt{eval\_bexp}\ be\ \sigma = \texttt{true}) \iff \\ (\exists \epsilon, \texttt{eval\_cnf}\ ([::\ [::\ r]]\ \texttt{++}\ cnf)\ \epsilon = \texttt{true})].\end{aligned}$$

Corollary 1 gives the formal proof of correctness for our bit blasting algorithm `bit_blast_bexp`. Let *be* be an arbitrary SMT QF_BV query, r and *cnf* the literal and the CNF formula returned by the algorithm. The corollary shows that the query *be* is satisfiable if and only of the SAT query $r \land cnf$ is satisfiable. An equi-satisfiable SAT query is indeed obtained from the bit blasting algorithm on every input SMT QF_BV query with a formal proof of correctness.

Recall that several QF_BV operations and predicates are derived from a small number of operations and predicates in SMT-LIB. A naïve bit blasting algorithm could expand derived operations or predicates, and then perform bit blasting on a small set of operations and predicates. Such an algorithm would have a simpler proof of correctness but generate more intermediate literals and clauses. For instance, the naïve algorithm for *bvsub* would perform bit blasting on *bvneg* followed by *bvadd* with intermediate literals and clauses. Our bit blasting algorithm for *bvsub* on the other hand reflects our semantics defined by the bit-vector function subB. Intermediate literals or clauses are not needed. Our bit blasting algorithm hence transforms *bvsub* more economically than the naïve algorithm.

To improve our bit blasting algorithm further, a cache for QF_BV expressions and predicates is added. In large queries, QF_BV expressions and predicates can occur a number of times. If a QF_BV expression has several occurrences, our basic bit blasting algorithm will generate result literals and CNF formulae for each occurrence. Consider the SMTQF_BV query

$$(\textit{and}\ (\textit{bvslt}\ \texttt{\#b1000}\ (\textit{bvadd}\ x\ y))\ (\textit{bvslt}\ (\textit{bvadd}\ x\ y)\ \texttt{\#b0111})).$$

The query checks whether the sum of the QF_BV variables x and y can be in a proper range. Since the Boolean predicate *and* has two operands, our basic algorithm invokes the auxiliary bit blasting algorithm for the two comparison predicates. It in turn blasts the same expression *bvadd* x y twice. Repeated bit blasting on the same expression or predicate is redundant. A hash function can detect repeated QF_BV expressions and predicates easily. When an expression or a predicate recurs, the previously computed literals with the empty CNF formula are returned from a cache as the result. More importantly, we give a formal Coq proof of Corollary 1 for the bit blasting algorithm with a cache.

7 A Certified SMT QF_BV Solver

We have so far built a formally verified bit blasting algorithm for SMT QF_BV queries. Using the code extraction mechanism in Coq, an OCaml program corresponding to the verified bit blasting algorithm is obtained. Using a SAT solver and a SAT certificate checker, a certified SMTQF_BV solver can be constructed. Figure 1 gives the flow of our certified solver.

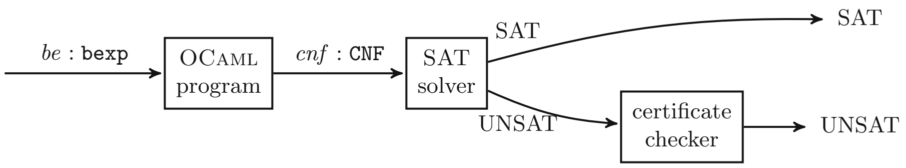


Fig. 1. Certified SMT QF_BV Solver

In the figure, the extracted OCaml program takes an OCaml expression *be* of the type `bexp` as an input (Sect. 5). The verified program performs bit blasting on the SMT QF_BV query and returns an OCaml expression *cnf* of the type `lit list list` representing a SAT query (Sect. 6). Precisely, an OCaml term of the type `lit` represents a literal. The OCaml type `lit list` corresponds to the data type for clauses; and the type `lit list list` corresponds to the data type for CNF formulae. The expression *cnf* is sent to a SAT solver to check satisfiability. If the SAT solver reports SAT, the SMT QF_BV query represented by *be* is satisfiable. Otherwise, the SAT solver reports UNSAT with a certificate. The certificate is sent to a SAT certificate checker for validation. If it is validated, the SMT QF_BV query *be* is unsatisfiable with certification.

8 Experiments

In order to evaluate the performance of our verified OCaml bit blasting program, we instantiate our SMT QF_BV solver CoqQFBV based on Fig. 1 as follows. We write an OCaml parser to translate a text file in the SMT-LIB format to an SMT QF_BV query in our formal syntax. The query is sent to the verified OCaml program for bit blasting. We then add an OCaml program to transform the output SAT query to a text file in the DIMACS format. The 2020 SAT Competition winner Kissat [5] is used to check the satisfiability of the SAT query. If the SAT solver reports UNSAT with a certificate in the DRAT format [31], the certificate is sent to the verified certificate checker GratChk [16] for validation. Certificate checkers for SAT solvers use much simpler algorithms than certificate checkers for SMT solvers. They are hence easier to build and prove correct. The correctness of GratChk is in fact verified by the proof assistant Isabelle [22]. We need not trust the certificate checker either.

We ran two experiments to evaluate our certified SMT QF_BV solver. The first experiment is the QF_BV division of the single query track in the 2020 SMT Competition [2]. The second experiment consists of verification problems from various assembly implementations for linear field arithmetic in cryptography libraries such as OpenSSL [30], RELIC [1], and BLST [29]. We compare CoqQFBV against three SMT QF_BV solvers: CVC4 [4] with an LFSC certificate checker [27], the 2020 SMTQF_BV division winner Bitwuzla [20], and the 2019 SMT QF_BV division winner Boolector [21]. Bitwuzla and Boolector are designed for efficiency without certification. CVC4 provides an LFSC certificate checker implemented in C [26]. The certificate checker can validate certificates from different theories but is itself not verified. All experiments were run on a Linux machine with a 3.20 GHz CPU and 1 TB memory.[1]

8.1 SMT QF_BV Competition

The first experiment is running our certified solver CoqQFBV on tasks from the QF_BV division of the 2020 SMT Competition. We set 60 GB memory limit and 20 min timeout for each task as in the competition. A task solves a single SMT-LIB file sequentially. The SMT QF_BV division contains 6861 files in the SMT-LIB format. All files are marked with *unsat*, *sat*, or *unknown* indicating expected query results. To save running time, we ran 10 tasks concurrently. The experimental results are summarized in Table 1.

In the table, the column N_{SC} indicates the number of solved tasks with certification. O_{SC} is the number of timeouts. E_{SC} shows the number of unsolved tasks due to tool errors. T_{SC} is the average time for solved tasks. CoqQFBV solves 6087 (88.72%) and CVC4 with its certificate checker solves 3840 (55.97%) with certification. We observe three stack overflow errors during bit blasting in CoqQFBV. These errors are induced by deep recursion. Among 328 errors from CVC4, 249 are segmentation faults raised by the LFSC certificate checker.

[1] CoqQFBV is available at https://github.com/fmlab-iis/coq-qfbv.git.

Table 1. Experimental results on the 2020 SMT QF_BV division

Tool	N_{SC}		O_{SC}	E_{SC}	T_{SC}	N_S		O_S	E_S	T_S
CoqQFBV	6087	(88.72%)	771	3	119.69	6169	(89.91%)	689	3	81.74
CVC4	3840	(55.97%)	2693	328	74.63	4255	(62.02%)	2544	62	56.87
Bitwuzla	–	–	–	–	–	6739	(98.22%)	122	0	16.09
Boolector	–	–	–	–	–	6719	(97.93%)	142	0	15.44

Table 2. Experimental results on the 2020 SMT QF_BV division by categories

Tool	N_{SC}		T_{SC}	P_{SU}	N_S		T_S
4238 *unsat* tasks							
CoqQFBV	3838	(90.56%)	146.72	291.35 MB	3920	(92.50%)	86.51
CVC4	1762	(41.58%)	86.68	266.61 MB	2177	(51.37%)	49.68
Bitwuzla	–	–	–	–	4188	(98.82%)	12.75
Boolector	–	–	–	–	4180	(98.63%)	11.72
2553 *sat* tasks							
CoqQFBV	–	–	–	–	2242	(87.82%)	73.26
CVC4	–	–	–	–	2078	(81.39%)	64.41
Bitwuzla	–	–	–	–	2524	(98.86%)	21.08
Boolector	–	–	–	–	2516	(98.55%)	21.31
70 *unknown* tasks							
CoqQFBV	5	(7.14%)	173.17	203.52 MB	7	(10.00%)	128.26
CVC4	–	–	–	–	0	(0.00%)	–
Bitwuzla	–	–	–	–	27	(38.57%)	66.36
Boolector	–	–	–	–	23	(32.86%)	48.58

The same table also compares against efficient but uncertified solvers. To evaluate the overhead from certificate checking, the two certified solvers CoqQFBV and CVC4 still generate certificates but do not validate them. The column N_S gives the number of solved tasks without certification. O_S is the number of timeouts. E_S indicates the number of errors, and T_S is the average time for solved tasks. Our certified solver CoqQFBV finishes 6169 (89.91%) tasks. The CVC4 solver finishes 4255 (62.02%) tasks. CoqQFBV and CVC4 solve 82(= 6169 – 6087) and 415(= 4255 – 3840) more tasks without certification respectively. Since our bit blasting algorithm is verified for all inputs, CoqQFBV does not certify bit blasting on each query and hence induces less overhead. The 2020 and 2019 SMT QF_BV division winners Bitwuzla and Boolector finish 6739 (98.22%) and 6719 (97.93%) tasks without certification respectively. CoqQFBV solves about 10% less tasks with certification than the 2020 track winner Bitwuzla without certification. It also performs significantly better than CVC4 with a general SMT certificate checker.

Table 2 compares the four solvers by tasks from the three expected query results. Among the 4238 *unsat* tasks, CoqQFBV and CVC4 give certified answers to 3838 (90.56%) and 1762 (41.58%) of them respectively. The column P_{SU} gives the average size of certificates. Efficient solvers Bitwuzla and Boolector give 4188 (98.82%) and 4180 (98.63%) uncertified answers respectively.

Among the 2553 *sat* tasks, Bitwuzla and Boolector finish 2524 (98.86%) and 2516 (98.55%) of them respectively. CoqQFBV and CVC4 solve only 2242 (87.82%) and 2078 (81.39%) *sat* tasks respectively. For the 70 tasks marked *unknown*, Bitwuzla and Boolector respectively answer 27 (38.57%) and 23 (32.86%) of them without certification. Our certified SMT QF_BV solver finds two *sat* and five *unsat* tasks. Answers to the five *unsat* tasks are all certified. CVC4 with its certificate checker fails to solve any *unknown* task. For the benchmarks from the 2020 SMT QF_BV division, our certified solver CoqQFBV appears to be more scalable than CVC4 with its general SMT certificate checker.

Table 3. Average time for CoqQFBV components

Task	T_{BB}	T_{SAT}	T_{Cert}
unsat	41.84	49.92	73.51
sat	37.08	62.09	–
unknown	32.34	121.99	62.86

Table 3 further decomposes the time spent on different components in CoqQFBV. The column T_{BB} gives the average time for our verified OCaml bit blasting program; T_{SAT} gives the average time used by the SAT solver Kissat; and T_{Cert} contains the average time for the certificate checker GratChk. For the tasks in the QF_BV division, the time for SAT solving and certificate checking are comparable. In comparison, the OCaml bit blasting program seems to take an unexpectedly large amount of time and hence can still be improved.

8.2 Linear Field Arithmetic in Cryptography

In this section, we evaluate our certified SMT QF_BV solver on benchmarks from real-world assembly implementations in various cryptography libraries such as OpenSSL [30], RELIC [1], and BLST [29]. In elliptic curve cryptography, arithmetic operations over large finite fields are needed. A field element is typically represented by hundreds of bits. A field arithmetic operation takes two field elements and returns a field element as the result. In the signature scheme Ed25519 used in OpenSSH, for instance, a field element belongs to the residue system modulo the prime number $2^{255} - 19$. Field sum of two field elements is obtained by the arithmetic sum modulo $2^{255} - 19$. Commodity processors however do not

Table 4. Experimental results on cryptographic assembly program verification

Tool	N_{SC}		T_{SC}	P_{SU}	N_S		T_S
CoqQFBV	93	(96.88%)	121.42	168.45 MB	93	(96.88%)	68.96
CVC4	19	(19.79%)	6.66	267.92 MB	46	(47.92%)	40.16
Bitwuzla	–	–	–	–	88	(91.67%)	16.07
Boolector	–	–	–	–	96	(100.00%)	18.25

support arithmetic instructions with operands in hundreds of bits natively. Field arithmetic has to be implemented by 32- or 64-bit instructions. The functional specification of the field addition used in Ed25519 may look as follows.

$$\{\textstyle\sum_{i=0}^{3} a_i \times 2^{64\times i} < 2^{255} - 19 \wedge \sum_{i=0}^{3} b_i \times 2^{64\times i} < 2^{255} - 19\}$$
$$\texttt{x25519\_fe64\_add}(r_0, r_1, r_2, r_3, a_0, a_1, a_2, a_3, b_0, b_1, b_2, b_3)$$
$$\left\{\begin{array}{c} \sum_{i=0}^{3} r_i \times 2^{64\times i} \equiv \sum_{i=0}^{3} a_i \times 2^{64\times i} + \sum_{i=0}^{3} b_i \times 2^{64\times i} (\text{mod } 2^{255} - 19) \\ \wedge \\ \sum_{i=0}^{3} r_i \times 2^{64\times i} < 2^{255} - 19 \end{array}\right\}$$

Let a_i, b_i, c_i be 64-bit variables (registers) for $0 \leq i \leq 3$. The specification says that the output field element represented by r_i's computed by the program `x25519_fe64_add` is the field arithmetic sum of the input elements represented by a_i's and b_i's. In finite field arithmetic programs, over- or under-flow in assembly instructions lead to incorrect results, and bit-accurate program verification is required. We obtain 46 implementations and generate 96 SMT QF_BV queries from verification conditions in order to evaluate our certified solver in this experiment.

Table 4 shows the verification results with the same memory and time limits in the 2020 SMT Competition. All SMT QF_BV queries are expected to be unsatisfiable. Boolector successfully solves all queries (100%) without certification. The 2020 QF_BV track winner Bitwuzla finishes 88 queries (91.67%) without certification. Surprisingly, CoqQFBV gives certified answers to 93 queries (96.88%). The verified SAT certificate checker GratChk used in CoqQFBV successfully validates all certificates for the real-world cryptographic program verification problems. In comparison, CVC4 solves 46 queries (47.92%) but certifies only 19 (19.79%). The CVC4 certificate checker raises segmentation faults on the 27 (= 46 − 19) solved but uncertified queries. These certificates are perhaps too complicated to be validated by the unverified LFSC certificate checker. For the SMT QF_BV queries from real-world program verification problems, our certified solver CoqQFBV seems to perform slightly better than the efficient but uncertified SMT QF_BV solver Bitwuzla. Our certified solver is probably scalable enough for certain bit-accurate program verification problems.

9 Conclusion

We combine algorithm design with interactive theorem proving to build a scalable certified SMT QF_BV solver CoqQFBV in this work. Our certified solver employs a verified OCaml bit blasting program and the verified certificate checker GratChk to improve the confidence in SMT QF_BV query results. Experiments on the QF_BV division of the 2020 SMT Competition and real-world cryptographic program verification suggest that CoqQFBV is useful.

For future work, we plan to specify and verify more heuristics to further optimize CoqQFBV. Particularly, cryptographic program verification requires more sophisticated range checks. More verified bit blasting algorithms for such checks will undoubtedly improve the confidence of bit-accurate program verification.

Acknowledgements. We thank all the anonymous reviewers for their insightful comments and suggestions. We thank the authors of SSReflect for its powerful language and libraries. We would like to give special thanks to Prof. Moshe Vardi for his encouragement. The work is supported by National Natural Science Foundation of China under the Grant Numbers 62002228, 61802259 and 61836005; the Guangdong Science and Technology Department under the Grant Number 2018B010107004; Ministry of Science and Technology of Taiwan under the Grant Numbers MOST108-2221-E-001-010-MY3 and MOST108-2221-E-001-009-MY2; Academia Sinica for Sinica Investigator Award AS-IA-109-M01; the Data Safety and Talent Cultivation Project AS-KPQ-109-DSTCP.

References

1. Aranha, D.F., Gouvêa, C.P.L., Markmann, T., Wahby, R.S., Liao, K.: RELIC is an Efficient LIbrary for Cryptography. https://github.com/relic-toolkit/relic
2. Barbosa, H., Hoenicke, J., Hyvarinen, A.: International Satisfiability Modulo Theories Competition (SMT-COMP) (2020). https://smt-comp.github.io/2020/
3. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa (2017). http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.6-r2017-07-18.pdf
4. Barrett, C., et al.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
5. Biere, A., Fazekas, K., Fleury, M., Heisinger, M.: CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT competition 2020. In: Balyo, T., Froleyks, N., Heule, M., Iser, M., Järvisalo, M., Suda, M. (eds.) SAT Competition 2020 - Solver and Benchmark Descriptions. B, vol. B-2020-1, pp. 50–53. University of Helsinki (2020)
6. Böhme, S., Fox, A.C.J., Sewell, T., Weber, T.: Reconstruction of Z3's bit-vector proofs in HOL4 and Isabelle/HOL. In: Jouannaud, J.-P., Shao, Z. (eds.) CPP 2011. LNCS, vol. 7086, pp. 183–198. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25379-9_15
7. Brummayer, R., Biere, A.: Fuzzing and delta-debugging SMT solvers. In: Dutertre, B., Strichman, O. (eds.) Satisfiability Modulo Theories (SMT), pp. 1–5. ACM (2009)

8. Chen, Y.F., et al.: Verifying Curve25519 software. In: Ahn, G.J., Yung, M., Li, N. (eds.) ACM Computer and Communications Security (CCS), pp. 299–309. ACM (2014)
9. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_15
10. Dockins, R., Foltzer, A., Hendrix, J., Huffman, B., McNamee, D., Tomb, A.: Constructing semantic models of programs with the software analysis workbench. In: Blazy, S., Chechik, M. (eds.) VSTTE 2016. LNCS, vol. 9971, pp. 56–72. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48869-1_5
11. Dross, C., Fumex, C., Gerlach, J., Marché, C.: High-Level Functional Properties of Bit-Level Programs: Formal Specifications and Automated Proofs. Research Report RR-8821, INRIA Saclay, December 2015. https://hal.inria.fr/hal-01238376
12. Ekici, B., Katz, G., Keller, C., Mebsout, A., Reynolds, A.J., Tinelli, C.: Extending SMTCoq, a certified checker for SMT (extended abstract). In: Electronic Proceedings in Theoretical Computer Science, vol. 210, pp. 21–29 (2016)
13. Fox, A.C.J.: LCF-style bit-blasting in HOL4. In: van Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) ITP 2011. LNCS, vol. 6898, pp. 357–362. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22863-6_26
14. Hadarean, L., Barrett, C., Reynolds, A., Tinelli, C., Deters, M.: Fine grained SMT proofs for the theory of fixed-width bit-vectors. In: Davis, M., Fehnker, A., McIver, A., Voronkov, A. (eds.) LPAR 2015. LNCS, vol. 9450, pp. 340–355. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48899-7_24
15. Kennedy, A., Benton, N., Jensen, J.B., Dagand, P.E.: Coq: the world's best macro assembler? In: Schrijvers, T. (ed.) Principles and Practice of Declarative Programming (PPDP), pp. 13–24. ACM (2013)
16. Lammich, P.: Efficient verified (UN)SAT certificate checking. In: de Moura, L. (ed.) CADE 2017. LNCS (LNAI), vol. 10395, pp. 237–254. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63046-5_15
17. Lochbihler, A.: Fast machine words in Isabelle/HOL. In: Avigad, J., Mahboubi, A. (eds.) ITP 2018. LNCS, vol. 10895, pp. 388–410. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94821-8_23
18. Mansur, M.N., Christakis, M., Wüstholz, V., Zhang, F.: Detecting critical bugs in SMT solvers using blackbox mutational fuzzing. In: Devanbu, P., Cohen, M., Zimmermann, T. (eds.) ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), pp. 701–712. ACM (2020)
19. de Moura, L.M., Bjørner, N.: Proofs and refutations, and Z3. In: Rudnicki, P., Sutcliffe, G., Konev, B., Schmidt, R.A., Schulz, S. (eds.) Proceedings of the LPAR 2008 Workshops, Knowledge Exchange: Automated Provers and Proof Assistants, and the 7th International Workshop on the Implementation of Logics. CEUR Workshop Proceedings, vol. 418. CEUR-WS.org (2008). http://ceur-ws.org/Vol-418/paper10.pdf
20. Niemetz, A., Preiner, M.: Bitwuzla at the SMT-COMP 2020. CoRR abs/2006.01621 (2020). https://arxiv.org/abs/2006.01621
21. Niemetz, A., Preiner, M., Biere, A.: Boolector 2.0. J. Satisfiability Boolean Modeling Comput. **9**(1), 53–58 (2014)
22. Nipkow, T., Wenzel, M., Paulson, L.C. (eds.): Isabelle/HOL – A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9

23. Oe, D., Reynolds, A., Stump, A.: Fast and flexible proof checking for SMT. In: Dutertre, B., Strichman, O. (eds.) Satisfiability Modulo Theories (SMT), pp. 6–13. ACM (2009)
24. Ozdemir, A., Niemetz, A., Preiner, M., Zohar, Y., Barrett, C.: DRAT-based bit-vector proofs in CVC4. In: Janota, M., Lynce, I. (eds.) SAT 2019. LNCS, vol. 11628, pp. 298–305. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-24258-9_21
25. Polyakov, A., Tsai, M.H., Wang, B.Y., Yang, B.Y.: Verifying arithmetic assembly programs in cryptographic primitives. In: Schewe, S., Zhang, L. (eds.) Concurrency Theory (CONCUR), pp. 4:1–4:16. LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2018)
26. Reynolds, A., Stump, A.: LFSC checker. https://github.com/CVC4/LFSC
27. Stump, A., Oe, D., Reynolds, A., Hadarean, L., Tinelli, C.: SMT proof checking using a logical framework. Formal Methods Syst. Des. **42**, 91–118 (2013)
28. Swords, S., Davis, J.: Bit-blasting ACL2 theorems. In: Hardin, D., Schmaltz, J. (eds.) The ACL2 Theorem Prover and its Applications (ACL2). EPTCS, vol. 70, pp. 84–102 (2011)
29. The blst Developers: The blst BLS12-381 signature library. https://github.com/supranational/blst
30. The OpenSSL Project: The OpenSSL repository. https://github.com/openssl/openssl
31. Wetzler, N., Heule, M.J.H., Hunt, W.A.: DRAT-trim: efficient checking and trimming using expressive clausal proofs. In: Sinz, C., Egly, U. (eds.) SAT 2014. LNCS, vol. 8561, pp. 422–429. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09284-3_31

Porous Invariants

Engel Lefaucheux[1], Joël Ouaknine[1], David Purser[1(✉)], and James Worrell[2]

[1] Max Planck Institute for Software Systems, Saarland Informatics Campus, Saarbrücken, Germany
dpurser@mpi-sws.org

[2] Department of Computer Science, Oxford University, Oxford, UK

Abstract. We introduce the notion of *porous invariants* for multipath (or branching/nondeterministic) affine loops over the integers; these invariants are not necessarily convex, and can in fact contain infinitely many 'holes'. Nevertheless, we show that in many cases such invariants can be automatically synthesised, and moreover can be used to settle (non-)reachability questions for various interesting classes of affine loops and target sets.

Keywords: Linear dynamical systems · Linear loops · Invariants · Reachability · Presburger arithmetic

1 Introduction

We consider the reachability problem for multipath (or branching) affine loops over the integers, or equivalently for nondeterministic integer linear dynamical systems. A (deterministic) integer linear dynamical system consists of an update matrix $M \in \mathbb{Z}^{d\times d}$ together with an initial point $x^{(0)} \in \mathbb{Z}^d$. We associate to such a system its infinite orbit $(x^{(i)})$ consisting of the sequence of reachable points defined by the rule $x^{(i+1)} = Mx^{(i)}$. The reachability question then asks, given a target set Y, whether the orbit ever meets Y, i.e., whether there exists some time i such that $x^{(i)} \in Y$. The nondeterministic reachability question allows the linear update map to be chosen at each step from a fixed finite collection of matrices.

When the orbit does eventually hit the target, one can easily substantiate this by exhibiting the relevant finite prefix. However, establishing non-reachability is intrinsically more difficult, since the orbit consists of an infinite sequence of points. One requires some sort of finitary certificate, which must be a relatively simple object that can be inspected and which provides a proof that the set Y is indeed unreachable. Typically, such a certificate will consist of an over-approximation I of the set R of reachable points, in such a manner that one can check both that $Y \cap I = \emptyset$ and $R \subseteq I$; such a set I is called an invariant.

Formally we study the following problem for *inductive invariants*:

The full version of this paper is available at http://arxiv.org/abs/2106.00662.

A. Silva and K. R. M. Leino (Eds.): CAV 2021, LNCS 12760, pp. 172–194, 2021.
https://doi.org/10.1007/978-3-030-81688-9_8

Meta Problem 1. *Consider a system with update functions $f_1, \ldots, f_n$. A set I is an inductive invariant if $f_i(I) \subseteq I$ for all i. Given a reachability query (x, Y) we search for a separating inductive invariant I such that $x \in I$ and $Y \cap I = \emptyset$.*

Meta Problem 1 is parametrised by the type of invariants and targets that are considered; that is, what are the classes of allowable invariant sets I and target sets Y, or equivalently how are such sets allowed to be expressed.

Fixing a particular invariant and target domain, a reachability query has three possible scenarios: (1) the instance is reachable, (2) the instance is unreachable and a separating invariant from the domain exists, or (3) the instance is unreachable but no separating invariant exists. Ideally, one would wish to provide a sufficiently expressive invariant domain so that the latter case does not occur, whilst keeping the resulting invariants as simple as possible and computable. For some classes of systems, it is known that distinguishing reachability (1) from unreachability (2, 3) is undecidable; it can also happen that determining whether a separating invariant exists (i.e., distinguishing (2) from (3)) is undecidable.

We note that the existence of *strongest* inductive invariants[1] is a desirable property for an invariant domain—when strongest invariants exist (and can be computed), separating (2) from (1, 3) is easy: compute the strongest invariant, and check whether it excludes the target state or not; if so, then you are done, and if not, no other invariant (from that class) can possibly do the trick either. However, unless (3) is excluded, computing the strongest invariant does not necessarily imply that reachability is decidable. Unfortunately, strongest invariants are not always guaranteed to exist for a particular invariant domain, although some separating inductive invariant may still exist for every target (or indeed may not).

In prior work from the literature, typical classes of invariants are usually convex, or finite unions of convex sets. In this paper we consider certain classes of invariants that can have infinitely many 'holes' (albeit in a structured and regular way); we call such sets *porous invariants*. These invariants can be represented via Presburger arithmetic[2]. We shall work instead with the equivalent formulation of semi-linear sets, generalising ultimately periodic sets to higher dimensions, as finite unions of linear sets of the form $\{b + p_1\mathbb{N} + \cdots + p_m\mathbb{N}\}$ (by which we mean $\{b + a_1p_1 + \cdots + a_mp_m \mid a_1, \ldots, a_m \in \mathbb{N}\}$, see Definition 2).

Let us first consider a motivating example:

Example 1 (Hofstadter's MU Puzzle [7]*).* Consider the following term-rewriting puzzle over alphabet $\{M, U, I\}$. Start with the word MI, and by applying the following grammar rules (where y and z stand for arbitrary words over our alphabet), we ask whether the word MU can ever be reached.

$$yI \to yIU \quad | \quad My \to Myy \quad | \quad yIIIz \to yUz \quad | \quad yUUz \to yz$$

[1] Given two invariants I and I', we say that I is *stronger* than I' iff $I \subseteq I'$; thus *strongest* invariants correspond to *smallest* invariant sets.

[2] Presburger arithmetic is a decidable theory over the natural numbers, comprising Boolean operations, first-order quantification, and addition (but not multiplication).

The answer is *no*. One way to establish this is to keep track of the number of occurrences of the letter 'I' in the words that can be produced, and observe that this number (call it x) will always be congruent to either 1 or 2 modulo 3. In other words, it is not possible to reach the set $\{x \mid x \equiv 0 \mod 3\}$. Indeed, Rules 2 and 3 are the only rules that affect the number of I's, and can be described by the system dynamics $x \mapsto 2x$ and $x \mapsto x - 3$. Hence the MU Puzzle can be viewed as a one-dimensional system with two affine updates,[3] or a two-dimensional system with two linear updates.[4] The set $\{1 + 3\mathbb{Z}\} \cup \{2 + 3\mathbb{Z}\}$ is an inductive invariant, and we wish to synthesise this. (The stability of this set under our two affine functions is easily checked: both components are invariant under $x \mapsto x - 3$, and $\{1 + 3\mathbb{Z}\} \mapsto \{2 + 6\mathbb{Z}\} \subseteq \{2 + 3\mathbb{Z}\}$ under $x \mapsto 2x$, and similarly $\{2 + 3\mathbb{Z}\} \mapsto \{4 + 6\mathbb{Z}\} \subseteq \{1 + 3\mathbb{Z}\}$.)

The problem can be rephrased as a safety property of the following multipath loop, verifying that the 'bad' state $x = 0$ is never reached, or equivalently that the above loop can never halt, regardless of the nondeterministic choices made.

```
    x = 1
while x ≠ 0
    x = 2 x || x = x−3        (where || represents nondeterministic branching)
```

The MU Puzzle was presented as a challenge for algorithmic verification in [4]; the tools considered in that paper (and elsewhere, to the best of our knowledge) rely upon the manual provision of an abstract invariant template. Our approach is to find the invariant fully automatically (although one must still abstract from the MU Puzzle the correct formulation as the program $x \mapsto 2x \mid\mid x \mapsto x - 3$).

Main Contributions. Our focus is on the automatic generation of porous invariants for multipath affine loops over the integers, or equivalently nondeterministic integer linear dynamical systems.

- We first consider targets consisting of a single vector (or 'point targets'), and present the classes of invariants and systems for which invariants can and cannot be automatically computed for the reachability question. A summary of the results for linear and semi-linear invariants for these targets is given in Table 1. For completeness we also consider $\mathbb{R}, \mathbb{R}_+$-(semi)-linear sets, where we complete the picture from prior work by showing that strongest $\mathbb{R}$-semi-linear invariants are computable.
  - We establish the existence of *strongest* $\mathbb{Z}$-linear invariants, and show that they can be found algorithmically (Theorem 2). These invariants may or may not separate the target under consideration.
  - If a $\mathbb{Z}$-linear invariant is not separating, we may instead look for an $\mathbb{N}$-semi-linear invariant (which generalises both $\mathbb{Z}$-semi-linear and $\mathbb{N}$-linear invariants), and we show that such an invariant can always be found

[3] One-dimensional affine updates are functions of the form $f(x) = ax + b$.

[4] $\begin{pmatrix} a & b \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ 1 \end{pmatrix} = \begin{pmatrix} ax + b \\ 1 \end{pmatrix}$ models affine functions using a matrix representation, holding one of the entries fixed to 1.

Table 1. Results for integer linear dynamical systems for a point target. Det/Non refers to deterministic or nondeterministic LDS. "Subsumed by ..." means that sufficient invariants can be generated, but of a more general type.

Dom	D/N	Linear	Semi-linear (SL)
$\mathbb{Z}$	Det	Strongest computable (Theorem 2)	No strongest (Sect. 4.1); subsumed by $\mathbb{N}$-SL
$\mathbb{Z}$	Non	Strongest computable (Theorem 2)	No strongest (Sect. 4.1)
$\mathbb{N}$	Det	No strongest (Sect. 4.1); subsumed by $\mathbb{N}$-SL	No strongest (Sect. 4.1), but sufficient computable (Theorem 4)
$\mathbb{N}$	Non	No strongest (Sect. 4.1)	1d-affine decidable (Theorem 6); undec. in general (Theorem 5)
$\mathbb{R}$	Det	Strongest: affine relations by Karr [17]	Strongest: affine closure on Zariski closure (Theorem 1)
$\mathbb{R}$	Non	Strongest: affine relations by Karr [17]	Strongest: affine closure on Zariski closure (Theorem 1)
$\mathbb{R}_+$	Det	No strongest (Sect. 4.1); subsumed by $\mathbb{R}_+$-SL	No strongest, but sufficient computable [8]
$\mathbb{R}_+$	Non	No strongest (Sect. 4.1)	Undecidable [8]

for any unreachable point target when dealing with *deterministic* integer linear dynamical systems (Theorem 4).

- However, for nondeterministic integer linear dynamical systems, computing an $\mathbb{N}$-semi-linear invariants is an undecidable problem in arbitrary dimension (Theorem 5). Nevertheless we show how such invariants can be constructed in a low-dimensional setting, in particular for affine updates in one dimension (Theorem 6). As an immediate consequence, this establishes that the multipath loop associated with the MU Puzzle belongs to a class of programs for which we can automatically synthesise $\mathbb{N}$-semi-linear invariants.

– For *full-dimensional*[5] $\mathbb{Z}$-linear targets we show that reachability is decidable, and, in the case of unreachability that a $\mathbb{Z}$-semi-linear invariant can always be exhibited as a certificate (Theorem 3). If the target is *not* full-dimensional then the reachability problem is Skolem-hard and undecidable for deterministic and nondeterministic systems respectively.

– In Sect. 6 we present our tool porous which handles one-dimensional affine systems for both point and $\mathbb{Z}$-linear targets, solving both the reachability problem and producing invariants. Inter alia, this allows one to handle the multipath loop derived from the MU Puzzle in fully automated manner.

1.1 Related Work

The reachability problem (in arbitrary dimension) for loops with a single affine update, or equivalently for deterministic linear dynamical systems, is decidable in polynomial time for point targets (that is $Y = \{y\}$), as shown by Kannan and Lipton [16]. However for nondeterministic systems (where the update matrix is chosen nondeterministically from a finite set at each time step), reachability is undecidable, by reduction from the matrix semigroup membership problem [22].

In particular this entails that for unreachable nondeterministic instances we cannot hope *always* to be able to compute a separating invariant. In some cases

[5] The affine span covers the entire space.

we may compute the strongest invariant (which may suffice if this invariant happens to be separating for the given reachability query), or we may compute an invariant in sub-cases for which reachability is decidable (for example in low dimensions). For some classes of invariants, it is also undecidable whether an invariant exists (e.g., polyhedral invariants [8]).

Various types of invariants have been studied for linear dynamical systems, including polyhedra [8,23], algebraic [15], and o-minimal [1] invariants. For certain classes of invariants (e.g., algebraic [15]), it is decidable whether a separating invariant exists, notwithstanding the reachability problem being undecidable. Other works (e.g., [5]) use heuristic approaches to generate invariants, without aiming for any sort of completeness.

Kincaid, Breck, Cyphert and Reps [18] study loops with linear updates, studying the closed forms for the variables to prove safety and termination properties. Such closed forms, when expressible in certain arithmetic theories, can be interpreted as another type of invariant and can be used to over-approximate the reachable sets. The work is restricted to a single update function (deterministic loops) and places additional constraints on the updates to bring the closed forms into appropriate theories.

Bozga, Iosif and Konecný's FLATA tool [2] considers affine functions in arbitrary dimension. However, it is restricted to affine functions with finite monoids; in our one-dimensional case this would correspond to limiting oneself to counter-like functions of the form $f(x) = x + b$.

Finkel, Göller and Haase [9], extending Fremont [10], show that reachability in a single dimension is **PSPACE**-complete for polynomial update functions (and allowing states can be used to control the sequences of updates which can be applied). The affine functions (and single-state restriction) we consider are a special case, but we focus on producing invariants to disprove reachability.

Other tools, e.g., AProVE [11] and Büchi Automizer [14] may (dis-)prove termination/reachability on *all* branches, but may not be able to prove termination/reachability on *some* branch.

Inductive invariants specified in Presburger arithmetic have been used to disprove reachability in vector addition systems [20]. A generalisation, 'almost semi-linear sets' [21] are also non-convex and can capture exactly the reachable points of vector addition systems. Our nondeterministic linear dynamical systems can be seen as vector addition systems over $\mathbb{Z}$ extended with affine updates (rather than only additive updates).

2 Preliminaries

We denote by $\mathbb{Z}$ the integers and $\mathbb{N}$ the non-negative integers. We say that $x, y \in \mathbb{Z}$ are congruent modulo $d \in \mathbb{N}$, denoted $x \equiv y \mod d$, if d divides $x - y$. Given an integer x and natural d we write $(x \mod d)$ for the number in $\{0, \ldots, d-1\}$ such that $(x \mod d) \equiv x \mod d$.

Definition 1 (Integer Linear Dynamical Systems). *A d-dimensional integer linear dynamical system (LDS) $(x^{(0)}, \{M_1, \ldots, M_k\})$ is defined by an initial point $x^{(0)} \in \mathbb{Z}^d$ and a set of integer matrices $M_1, \ldots, M_k \subseteq \mathbb{Z}^{d \times d}$. An LDS is* deterministic *if it comprises a single matrix ($k = 1$) and is otherwise* nondeterministic*.*

A point y is reachable *if there exists $m \in \mathbb{N}$ and $B_1, \ldots, B_m$ such that $B_1 \cdots B_m x^{(0)} = y$ and $B_i \in \{M_1, \ldots, M_k\}$ for all $1 \leq i \leq m$.*

The reachability set *$\mathcal{O} \subseteq \mathbb{Z}^d$ of an LDS is the set of reachable points.*

Definition 2 ($\mathbb{K}$-(semi)-linear sets). *A* linear set *L is defined by a base vector $b \in \mathbb{Z}^d$ and period vectors $p_1, \ldots, p_d \in \mathbb{Z}^d$ such that*

$$L = \{b + a_1 p_1 + \cdots + a_d p_d \mid a_1, \ldots, a_d \in \mathbb{K}\}.$$

For convenience we often write $\{b + p_1\mathbb{K} + \cdots + p_d\mathbb{K}\}$ for L. A set is semi-linear *if it is the finite union of linear sets.*

$\mathbb{N}$-semi-linear sets are precisely those definable in Presburger arithmetic ($\mathrm{FO}(\mathbb{Z}, +, \leq)$) [12]. However, we can also consider $\mathbb{Z}$-semi-linear sets (corresponding to $\mathrm{FO}(\mathbb{Z}, +)$ without order), and the real counterparts ($\mathbb{R}$ and $\mathbb{R}_+$). Note that even if $\mathbb{K} = \mathbb{N}$ we still allow $p_i \in \mathbb{Z}^d$.

Definition 3. *Given an integer linear dynamical system $(x^{(0)}, \{M_1, \ldots, M_k\})$, a set I is an* inductive invariant *if*

- *$x^{(0)} \in I$, and*
- *$\{M_i x \mid x \in I\} \subseteq I$ for all $i \in \{1, \ldots, k\}$.*

Note in particular that every inductive invariant contains the reachability set ($\mathcal{O} \subseteq I$). We are interested in the following problem:

Definition 4 (Invariant Synthesis Problem). *Given an invariant domain $\mathcal{D}$, an integer linear dynamical system $(x^{(0)}, \{M_1, \ldots, M_k\})$, and a target Y, does there exist an inductive invariant I in $\mathcal{D}$ disjoint from Y?*

In our setting, we are interested in classes $\mathcal{D}$ of invariants that are linear, or semi-linear. When a separating inductive invariant I exists, we also wish to compute it. Since (semi)-linear invariants are enumerable, the decision problem is, in theory, sufficient—although all of our proofs are constructive.

3 $\mathbb{R}$ Invariants: $\mathbb{R}$-linear and $\mathbb{R}$-semi-linear

Before delving into porous invariants, let us consider invariants over the real numbers, i.e., described as $\mathbb{R}$-(semi)-linear sets.

Strongest $\mathbb{R}$-linear invariants are given precisely by the affine hull of the reachability set, and can be computed using Karr's algorithm [17]. Moreover, we will show that strongest $\mathbb{R}$-semi-linear invariants also exist and can be computed by combining techniques for algebraic invariants [15] and $\mathbb{R}$-linear invariants.

$\mathbb{R}$-*linear.* Recall that a set L is $\mathbb{R}$-linear if $L = \{v_0 + v_1\mathbb{R} + \cdots + v_t\mathbb{R}\}$ for some $v_0, \ldots, v_t \in \mathbb{Z}^d$ that can be assumed to be linearly-independent[6] without loss of generality (and thus $t \leq d$). Given two distinct points of L, every point on the infinite line connecting them must also be in L. Generalising this idea to higher dimensions, given a set $S \subseteq \mathbb{R}^d$, let the affine hull be

$$\overline{S}^a = \left\{ \sum_{i=1}^{k} \lambda_i x_i \mid k \in \mathbb{N}, x_i \in S, \lambda_i \in \mathbb{R}, \sum_{i=1}^{k} \lambda_i = 1 \right\}.$$

Fix an LDS $(x^{(0)}, \{M_1, \ldots, M_k\})$ and consider its reachability set $\mathcal{O} = \{M_{i_m} \cdots M_{i_1} x^{(0)} \mid m \in \mathbb{N}, i_1, \ldots, i_m \in \{1, \ldots, k\}\}$. Then $\overline{\mathcal{O}}^a$ is precisely the strongest $\mathbb{R}$-linear invariant. Karr's algorithm [17,26] can be used to compute this strongest invariant in polynomial time. The next lemma follows from Theorem 3.1 of [26].

Lemma 1. *Given an LDS $(x^{(0)}, \{M_1, \ldots, M_k\})$ of dimension d, we can compute in time polynomial in d, k, and $\log \mu$ (where $\mu > 0$ is an upper bound on the absolute values of the integers appearing in $x^{(0)}$ and $M_1, \ldots, M_k$), a $\mathbb{Q}$-affinely independent set of integer vectors $R_0 \subseteq \mathcal{O}$ such that:*

1. *$x^{(0)} \in R_0$,*
2. *the affine span of R_0 and the affine span of $\mathcal{O}$ are the same ($\overline{R_0}^a = \overline{\mathcal{O}}^a$),*
3. *the entries of the vectors in R_0 have absolute value at most $\mu_0 := (d\mu)^d$.*

Let $R_0 = \{x^{(0)}, r_1, \ldots, r_{d'}\}$ be obtained as per Lemma 1, with $d' \leq d$. The $\mathbb{R}$-linear invariant of the LDS is the affine span $\overline{R_0}^a$, which can be written as the $\mathbb{R}$-linear set $L_0 = \{x^{(0)} + (r_1 - x^{(0)})\mathbb{R} + \cdots + (r_{d'} - x^{(0)})\mathbb{R}\}$.

$\mathbb{R}$-*semi-linear.* Let us now generalise this approach to $\mathbb{R}$-semi-linear sets. The collection of $\mathbb{R}$-semi-linear sets, $\{\bigcup_{i=1}^{m} L_i \mid m \in \mathbb{N}, L_1, \ldots, L_m \text{ are } \mathbb{R}\text{-linear sets}\}$, is closed under finite unions and arbitrary intersections[7]. Thus for any given set X, the smallest $\mathbb{R}$-semi-linear set containing X is simply the intersection of all $\mathbb{R}$-semi-linear sets containing X. Let us denote by $\overline{X}^{\mathbb{R}}$ this smallest $\mathbb{R}$-semi-linear set. We are interested in $\overline{\mathcal{O}}^{\mathbb{R}}$.

Theorem 1. *The strongest $\mathbb{R}$-semi-linear invariant $\overline{\mathcal{O}}^{\mathbb{R}}$ of $\mathcal{O}$ is computable.*

Algebraic sets are those that are definable by finite unions and intersections of zeros of polynomials. For example, $\{(x, y) \mid xy = 0\}$ describes the lines $x = 0$ and $y = 0$. The (real) Zariski closure $\overline{X}^z$ of a set X is the smallest algebraic subset of $\mathbb{R}^d$ containing the set X. The Zariski closure of the set of reachable points, $\overline{\mathcal{O}}^z$, can be computed algorithmically [15].

[6] $v_0, \ldots, v_m$ are linearly independent if there does not exist $a_0, \ldots, a_m \in \mathbb{R}$, not all 0, such that $a_0 v_0 + \cdots + a_m v_m = 0$.

[7] When intersecting a linear set with a semi-linear set, either the latter does not change, or one obtains a finite union of elements of smaller dimension. Thus, in an infinite intersection, only a finite number of intersections affects the original set.

An algebraic set A is *irreducible* if whenever $A \subseteq B \cup C$, where B and C are algebraic sets, then we have $A \subseteq B$ or $A \subseteq C$. Any algebraic set (and in particular a Zariski closure) can be written effectively as a finite union of irreducible sets [3].

Proposition 1. *Let* $\overline{X}^z = A_1 \cup \cdots \cup A_k$*, with* A_i*'s irreducible. Then* $\overline{X}^{\mathbb{R}} = \overline{\overline{X}^z}^{\mathbb{R}} = \overline{A_1}^{\mathbb{R}} \cup \cdots \cup \overline{A_k}^{\mathbb{R}} = \overline{A_1}^a \cup \cdots \cup \overline{A_k}^a$.

Proof. Since $A_i \subseteq \overline{X}^{\mathbb{R}} = \cup_j L_j$, and A_i is irreducible, we have $A_i \subseteq L_j$ for some j (as the L_j's are algebraic sets). Since L_j is $\mathbb{R}$-linear, and $\overline{A_i}^a$ is the smallest $\mathbb{R}$-linear set covering A_i, we have $\overline{A_i}^a \subseteq L_j$. Taking $\overline{X}^{\mathbb{R}} = \overline{A_1}^a \cup \cdots \cup \overline{A_k}^a$ is thus optimal. □

Thus $\overline{\mathcal{O}}^{\mathbb{R}}$ can be obtained by computing $\overline{A_i}^a$ for each irreducible A_i, where $\overline{\mathcal{O}}^z = A_1 \cup \cdots \cup A_k$. To complete the proof of Theorem 1 it remains to confirm that affine hulls of algebraic sets can be computed algorithmically. Let us fix an algebraic set A, and let W denote a set variable. Proceed as follows. Start with $W \leftarrow \{x\}$ for some point $x \in A$, and repeatedly let $W \leftarrow \overline{W \cup \{y\}}^a$, where $y \in A \setminus W$. Such a point y can always be found using quantifier elimination in the theory of the reals. Each step necessarily increases the dimension, which can occur at most d times, ensuring termination, at which point one has $\overline{A}^a = W$.

4 Strongest $\mathbb{Z}$-linear Invariants

Recall that a $\mathbb{Z}$-linear set $\{q + p_1\mathbb{Z} + \cdots + p_n\mathbb{Z}\}$ is defined by a base vector $q \in \mathbb{Z}^d$ and period vectors $p_1, \ldots, p_n \in \mathbb{Z}^d$. Equivalently, a $\mathbb{Z}$-linear set describes a *lattice*, i.e., $\{p_1\mathbb{Z} + \cdots + p_n\mathbb{Z}\}$, in d-dimensional space, translated to start from q rather than $\mathbf{0}$.

Theorem 2. *Given a d-dimensional dynamical system* $(x^{(0)}, \{M_1, \ldots, M_k\})$*, the strongest* $\mathbb{Z}$*-linear inductive invariant containing the reachability set* $\mathcal{O}$ *exists and can be computed algorithmically.*

The image of a $\mathbb{Z}$-linear set $L = \{q + p_1\mathbb{Z} + \cdots + p_n\mathbb{Z}\}$ by a matrix M is the $\mathbb{Z}$-linear set: $M(L) = \{Mq + (Mp_1)\mathbb{Z} + \cdots + (Mp_n)\mathbb{Z}\}$. The following lemma asserts that when two points are in a $\mathbb{Z}$-linear set, the direction between these two points can be applied from any reachable point, and hence this direction can be included as a period without altering the set.

Proposition 2. *Let* $L = \{q + a_1p_1 + \cdots + a_np_n \mid a_1, \ldots, a_n \in \mathbb{Z}\}$ *be a* $\mathbb{Z}$*-linear set. If* $x, y \in L$ *then for all* $z \in L$ *and all* $a' \in \mathbb{Z}$ *we have* $z + (y - x)a' \in L$*. In particular, we have* $L = \{q + a_1p_1 + \cdots + a_np_n + a'(y - x) \mid a_1, \ldots, a_n, a' \in \mathbb{Z}\}$.

Proof. If $x = q + a_1p_1 + \cdots + a_np_n$ and $y = q + b_1p_1 + \cdots + b_np_n$ then $y - x = q + b_1p_1 + \cdots + b_np_n - (q + a_1p_1 + \cdots + a_np_n) = (b_1 - a_1)p_1 + \cdots + (b_n - a_n)p_n$.

Then for any $z = q + c_1p_1 + \cdots + c_np_n$, we have $z + a'(y - x) = q + c_1p_1 + \cdots + c_np_n + a'((b_1 - a_1)p_1 + \cdots + (b_n - a_n)p_n) = q + (c_1 + a'(b_1 - a_1))p_1 + \cdots + (c_n + a'(b_n - a_n))p_n)$ where $(c_i + a'(b_i - a_i)) \in \mathbb{Z}$, so $z + a'(y - x) \in L$. □

Proposition 3. *Given two $\mathbb{Z}$-linear sets $L_1 = \{q + p_1\mathbb{Z} + \cdots + p_n\mathbb{Z}\}$ and $L_2 = \{s + t_1\mathbb{Z} + \cdots + t_m\mathbb{Z}\}$, there exists a smallest $\mathbb{Z}$-linear set L containing $L_1 \cup L_2$: the set $L = \{q + (s-q)\mathbb{Z} + p_1\mathbb{Z} + \cdots + p_n\mathbb{Z} + t_1\mathbb{Z} + \cdots + t_m\mathbb{Z}\}$.*

Proof. First we show $L_1 \cup L_2 \subseteq L$:

– If $x = q + a_1p_1 + \cdots + a_np_n \in L_1$, then $x = q + (s-q)0 + a_1p_1 + \cdots + a_np_n + 0t_1 + \cdots + 0t_m \in L$.
– If $x = s + b_1t_1 + \cdots + b_mt_m \in L_2$, then $x = q + (s-q)1 + 0p_1 + \cdots + 0p_n + b_1t_1 + \cdots + b_mt_m \in L$.

Next we show minimality as a straightforward consequence of Proposition 2.

Clearly the vectors $p_1, \ldots, p_n$ can be added by Proposition 2 because any two points of L_1 differing by p_i guarantees that adding p_i does not alter the resulting set. Similarly, $t_1, \ldots, t_m$ can also be included. Finally, by Proposition 2, the vector $s - q$ can be included because q and s both belong to $L_1 \cup L_2$. □

A d-dimensional lattice can always be defined by at most d vectors; and thus if d is the dimension of the matrices, no more than d period vectors are needed in total. However, Proposition 3 induces a representation which may over-specify the lattice by producing more than d vectors to define the lattice.

Example 2. Consider the lattice $\{(2,2)\mathbb{Z} + (0,6)\mathbb{Z} + (2,6)\mathbb{Z}\}$, specified with three vectors, which is equivalent to the lattice $\{(2,0)\mathbb{Z} + (0,2)\mathbb{Z}\}$. Note that one may not simply pick an independent subset of the periods, as none of the following sets are equal: $\{(2,2)\mathbb{Z} + (0,6)\mathbb{Z}\}$, $\{(2,2)\mathbb{Z} + (2,6)\mathbb{Z}\}$, $\{(0,6)\mathbb{Z} + (2,6)\mathbb{Z}\}$, and $\{(2,2)\mathbb{Z} + (0,6)\mathbb{Z} + (2,6)\mathbb{Z}\}$.

The *Hermite normal form* can be used to obtain a basis of the vectors that define the lattice. Consider a lattice $L_i = \{p_1\mathbb{Z} + \cdots + p_d\mathbb{Z}\}$. The lattice remains the same if p_i is swapped with p_j, if p_i is replaced by $-p_i$, or if p_i is replaced by $p_i + \alpha p_j$ where α is any fixed integer[8].

These are the unimodular operations. The Hermite normal form of a matrix M is a matrix H such that $M = UH$, where U is a unimodular matrix (formed by unimodular column operations) and H is lower triangular, non-negative and each row has a unique maximum entry which is on the main diagonal. Such a form always exists, and the columns of H form a basis of the same lattice as the columns of M, because they differ up to unimodular (lattice-preserving) operations. There are many texts on the subject; we refer the reader to the lecture notes of Shmonin [25] for more detailed explanations.

The columns of a matrix in Hermite normal form constitute a unique basis for the lattice (up to additional redundant zero columns). Hence a basis of minimal dimension can be obtained by computing the Hermite normal form of the matrix formed by placing the period vectors into columns.

[8] The last replacement is valid, since if $x = y + \beta p_i \in L$ then $x = y + \beta(p_i + \alpha p_j) - \beta\alpha p_j$ is in the new lattice.

We now prove the main theorem:

Proof (Proof of Theorem 2*).* We claim that Algorithm 1 returns the strongest $\mathbb{Z}$-linear invariant I.

Algorithm 1 proceeds in two phases:

– First find a necessary subset $L_0 \subseteq I$ of the invariant having already the same dimension as I.
– Then compute a growing sequence $L_0 \subsetneq L_1 \subsetneq \cdots \subsetneq L_{m-1} = L_m = I$, where at each step the algorithm merely increases the density of the attendant sets in order to 'fill in' missing points of the invariant.

Recall the set $R_0 = \{x^{(0)}, r_1, \ldots, r_{d'}\} \subseteq \mathcal{O}$, with $d' \leq d$, from Lemma 1. The resulting $\mathbb{Z}$-linear set $L_0 = \{x^{(0)} + (r_1 - x^{(0)})\mathbb{Z} + \cdots + (r_{d'} - x^{(0)})\mathbb{Z}\}$ is then a d'-dimensional porous subset of the d'-dimensional affine hull of the orbit ($L_0 \subseteq \overline{\mathcal{O}}^a$). Applying $M_1, \ldots, M_k$ can only increase the density, but not the dimension. As each r_i and $x^{(0)}$ are in $\mathcal{O}$, by Proposition 2 we can assume that each of the directions $(r_i - x^{(0)})$ must be represented in any $\mathbb{Z}$-linear set containing $\mathcal{O}$, and we therefore have that $L_0 \subseteq I$.

In the second phase, we 'fill in' the lattice as required to cover the whole of $\mathcal{O}$. To do this we repeatedly apply the covering procedure of Proposition 3. That is, L_{i+1} is the smallest $\mathbb{Z}$-linear set covering $L_i \cup M_1(L_i) \cup \cdots \cup M_k(L_i)$. To keep the number of vectors small, we keep the period vectors of the $\mathbb{Z}$-linear set in Hermite normal form.

The vectors $p_1 = (r_1 - x^{(0)}), \ldots, p_{d'} = (r_{d'} - x^{(0)})$ form a parallelepiped (hyper-parallelogram) that repeats regularly. There are a finite number of integral points inside this parallelepiped. If new points are added in some step, they are added to every parallelepiped. Thus we can add new points finitely many times before saturating or becoming fixed. The volume of the parallelepiped is bounded above by $|p_1| \cdots |p_{d'}|$.

At each step, the volume of the parallelepiped must at least halve, thus the volume at step t is $vol_t \leq |p_1| \cdots |p_{d'}|/2^t$. The procedure must saturate at or before the volume becomes 1, which occurs after at most $\log(|p_1| \cdots |p_{d'}|) = \sum_i \log(|p_i|)$ steps. At each step, for efficiency considerations, we convert the $\mathbb{Z}$-linear set into Hermite normal form to retain exactly d' period vectors.

Claim (I is the strongest invariant). For every invariant J, we have $I \subseteq J$.

By induction, let us prove that every invariant J must contain L_i. Clearly this is the case for L_0 because all points of $R_0 \subseteq \mathcal{O}$ must be in J and every period vectors in L_0 can be present, without loss of generality, thanks to Proposition 2. Assume $L_i \subseteq J$. Then it must be the case that J contains every $M_j(L_i)$, as otherwise it would not be an invariant. It therefore follows that J must contain L_{i+1}, since the latter is the minimal $\mathbb{Z}$-linear set containing L_i and $M_j(L_i)$ for all $j \leq k$. Finally, since I is itself one of the L_i's, we have $I \subseteq J$ as required. $\square$

Remark 1. Note that a $\mathbb{Z}$-linear set is not sufficient for the MU puzzle: both 1 and 2 are in the reachability set, thus $\{1 + 1\mathbb{Z}\} = \mathbb{Z}$ is the strongest $\mathbb{Z}$-linear invariant.

Algorithm 1: Strongest $\mathbb{Z}$-linear invariant for LDS $(x^{(0)}, M_1, \ldots, M_k)$

```
Input: $x^{(0)}, M_1, \ldots, M_k$
Compute $R_0 = \left\{x^{(0)}, r_1, \ldots, r_{d'}\right\} \subseteq \mathcal{O}$
Compute $p_i = r_i - x^{(0)}$ for $i \in \{1, \ldots, d'\}$
$L_0 = \left\{x^{(0)} + p_1\mathbb{Z} + \cdots + p_{d'}\mathbb{Z}\right\}$
while True do
    $L_i = \text{Covering}(L_{i-1} \cup M_1(L_{i-1}) \cup \cdots \cup M_k(L_{i-1}))$
    $H_i = \text{HermiteNormalForm}(L_i)$
    $L_i = \left\{x^{(0)} + h_1\mathbb{Z} + \cdots + h_{d'}\mathbb{Z} \mid h_j \text{ column of } H_i\right\}$
    if $L_i = L_{i-1}$ then
        return $L_i$
    end
end
```

4.1 Extensions of $\mathbb{Z}$-linear Sets Without Strongest Invariants

In this section we show that several generalisations of $\mathbb{Z}$-linear domains fail to admit strongest invariants.

$\mathbb{Z}$-semi-linear sets are unions of $\mathbb{Z}$-linear sets, and therefore can include singletons. Consider the deterministic dynamical system starting from point 1 and doubling at each step $\mathcal{M} = (1, (x \mapsto 2x))$. This system has reachability set $\mathcal{O} = \left\{2^k \mid k \in \mathbb{N}\right\}$, which is not even $\mathbb{N}$-semi-linear (our most general class). For this LDS we can construct the invariant $\left\{2, 4, 8, ..., 2^k\right\} \cup \left\{2^{k+1}p_1 \mid p_1 \in \mathbb{Z}\right\}$ for each k. For any proposed strongest $\mathbb{Z}$-semi-linear invariant, one can find a k for which the corresponding invariant is an improvement.

$\mathbb{N}$-linear sets generalise $\mathbb{Z}$-linear sets (observe that $\mathbb{Z}$-linear sets are a proper subclass, since $\{x + p_i\mathbb{Z}\}$ can be expressed as $\{x + (-p_i)\mathbb{N} + p_i\mathbb{N}\}$, but $\{x + p_i\mathbb{N}\}$ is clearly not $\mathbb{Z}$-linear). Consider the LDS $((x_1, x_2), \left(\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right))$, with a reachability set consisting of just two points $x = (x_1, x_2)$ and $y = (x_2, x_1)$. There are two incomparable candidates for the minimal $\mathbb{N}$-linear invariant: $\{x + (y - x)\mathbb{N}\}$ and $\{y + (x - y)\mathbb{N}\}$. Similarly for $\mathbb{R}_+$-linear invariants, the sets $\{y + (x - y)\mathbb{R}_+\}$ and $\{x + (y - x)\mathbb{R}_+\}$ are incomparable half-lines.

4.2 $\mathbb{Z}$-linear Targets

We have so far only considered invariants for point targets. We now turn to lattice-like targets, in particular targets specified as *full-dimensional* $\mathbb{Z}$-linear sets.

Theorem 3. *It is decidable whether a given LDS* $(x^{(0)}, \{M_1, \ldots, M_k\})$ *reaches a full-dimensional* $\mathbb{Z}$*-linear target* $Y = \{x + p_1\mathbb{Z} + \cdots + p_d\mathbb{Z}\}$*, with* $x, p_i \in \mathbb{Z}^d$*.*

Furthermore, for unreachable instances, a $\mathbb{Z}$*-semi-linear inductive invariant can be provided.*

Theorem 3 requires the targets to be *full-dimensional.* For nondeterministic systems reachability is undecidable for non-full-dimensional targets (in particular point targets) [22]. However, even for deterministic systems, when $\mathbb{Z}$-linear targets fail to be *full-dimensional* the reachability problem becomes as hard as the Skolem problem (see, e.g. [24]), for example by choosing as target the set $\{(0, x_2, \dots, x_d) \mid x_2, \dots, x_d \in \mathbb{Z}\} = \{\mathbf{0} + e_2\mathbb{Z} + \dots + e_d\mathbb{Z}\}$, where $e_i \in \{0,1\}^d$ is the standard basis vector, with $(e_i)_i = 1$ and $(e_i)_j = 0$ for $i \neq j$.

Towards proving Theorem 3, we first show that *full-dimensional* linear sets can be expressed as 'square' hybrid-linear sets. Hybrid-linear sets are semi-linear sets in which all the components share the same period vectors, and thus differ only in starting position (whereas semi-linear sets allow each component to have distinct period vectors). By square, we mean that all period vectors are the same multiple of standard basis vectors.

Lemma 2. *Let $Y = \{x + p_1\mathbb{Z} + \dots + p_d\mathbb{Z}\}$ be a full-dimensional $\mathbb{Z}$-linear set. Then there exists $m \in \mathbb{N}$ and a finite set $B \subseteq [0, m-1]^d$ such that $Y = \bigcup_{b \in B} \{b + me_1\mathbb{Z} + \dots + me_d\mathbb{Z}\}$.*

Proof. Suppose $p_1, \dots, p_d$ span a d-dimensional vector space. Let $P = \begin{pmatrix} p_1 \\ \vdots \\ p_d \end{pmatrix}$ be the matrix with rows $p_1, \dots, p_d$. Since P is full row rank it is invertible, hence there exists a rational matrix P^{-1} such that $e_i = P^{-1}_{i,1}p_1 + \dots + P^{-1}_{i,d}p_d$. In particular let m_i be such that $P^{-1}_{i,j}m_i$ is integral for all j. Then there is an integral combination of $p_1, \dots, p_d$ such that m_ie_i is an admissible direction in Y.

Let $m = \operatorname{lcm}\{m_1, \dots, m_d\}$. Then me_i is an admissible direction in Y. Hence by Proposition 2, Y is equivalent to $\big\{x + p_1\mathbb{Z} + \dots + p_d\mathbb{Z} + me_1\mathbb{Z} + \dots + me_d\mathbb{Z}\big\}$. By the presence of $me_1\mathbb{Z} + \dots + me_d\mathbb{Z}$ we have that $x \in Y$ if and only $x' \in Y$ where $x'_i = (x_i \mod m)$.

And therefore Y can be written as $\bigcup_{b \in B} \{b + me_1\mathbb{Z} + \dots + me_d\mathbb{Z}\}$, where $B = [0, m-1]^d \cap Y$. □

We now prove Theorem 3.

Proof (Proof of Theorem 3). Choose m and B as in Lemma 2, so that Y is of the form $\bigcup_{b \in B} \{b + me_1\mathbb{Z} + \dots + me_d\mathbb{Z}\}$. We build an invariant I of the form $\bigcup_{b \in B'} \{b + me_1\mathbb{Z} + \dots + me_d\mathbb{Z}\}$ for some $B' \subseteq [0, m-1]^d$.

We initialise the set $I_0 = \{x + me_1\mathbb{Z} + \dots + me_d\mathbb{Z}\}$, where $x \in [0, m-1]^d$ such that $x_j = (x^{(0)}_j \mod m)$. We then build the set I_1 by adding to I_0 the sets $\{y + me_1\mathbb{Z} + \dots + me_d\mathbb{Z}\}$ where for each choice of M_i, $y \in [0, m-1]^d$ is formed by $y_j = ((M_ix)_j \mod m)$ for some $x \in I_0$. We iterate this construction until it stabilises in an inductive invariant I. Termination follows from the finiteness of $[0, m-1]^d$ (noting in particular that if termination occurs with $B' = [0, m-1]^d$, then $I = \mathbb{Z}^d$ which is indeed an inductive invariant).

If there exists $y \in B \cap I$ then return Reachable. This is because the same sequence of matrices applied to $x^{(0)}$ to produce $y \in I$ would, thanks to the modulo step, wind up inside the set $\{y + me_1\mathbb{Z} + \cdots + me_d\mathbb{Z}\}$, which is a part of the target.

Otherwise, return Unreachable and I as invariant. By construction, I is indeed an inductive invariant disjoint from the target set. □

Remark 2. By the same argument, Theorem 3 extends to a restricted class of $\mathbb{Z}$-semi-linear targets: the finite union of *full-dimensional* $\mathbb{Z}$-linear sets.

5 $\mathbb{N}$-Semi-linear Invariants

We now consider $\mathbb{N}$-semi-linear invariants, our most general class. $\mathbb{N}$-semi-linear invariants gain expressivity thanks to the 'directions' provided by the period vectors. For example, the only possible $\mathbb{Z}$-semi-linear invariant for the LDS $(0, (x \mapsto x + 1))$ is $\mathbb{Z}$, yet the reachability set, $\mathbb{N}$, is captured exactly by an $\mathbb{N}$-linear invariant. We show that a separating $\mathbb{N}$-semi-linear invariant can *always* be found for unreachable instances of deterministic integer LDS, although the computed invariant will depend on the target. However, finding invariants is undecidable for nondeterministic systems, at least in high dimension. Nevertheless, we show decidability for the low-dimensional setting of the MU Puzzle—one dimension with affine updates.

5.1 Existence of Sufficient (but Non-minimal) $\mathbb{N}$-semi-linear Invariants for Point Reachability in Deterministic LDS

Kannan and Lipton showed decidability of reachability of a point target for deterministic LDS [16]. In this subsection, we establish the following result to provide a separating invariant in unreachability instances.

Theorem 4. *Given a deterministic LDS $(x^{(0)}, M)$ together with a point target y, if the target is unreachable then a separating $\mathbb{N}$-semi-linear inductive invariant can be provided.*

To do so, we will invoke the results from [8] to compute an $\mathbb{R}_+$-semi-linear inductive invariant, and then extract from it an $\mathbb{N}$-semi-linear inductive invariant. More precisely, the authors of [8] show how to build polytopic inductive invariants for certain deterministic LDS. Such polytopes are either bounded or are $\mathbb{R}_+$-semi-linear sets. In the first case, the polytope contains only finitely many integral points, which can directly be represented via an $\mathbb{N}$-semi-linear set. In the second case, we build an $\mathbb{N}$-semi-linear set containing exactly the set of integral points included in the $\mathbb{R}_+$-semi-linear invariant, thanks to the following lemma.

Lemma 3. *Given an $\mathbb{R}_+$-linear set $S = \{x + \sum_i p_i\mathbb{R}_+\}$, where the vectors p_i have rational coefficients and x is an integer vector, one can build an $\mathbb{N}$-semi-linear set N comprising precisely all of the integral points of S.*

Proof (Proof of Theorem 4). We note that every invariant produced in [8] has rational period vectors, as the vectors are given by the difference of successive point in the orbit of the system, and thus Lemma 3 can be applied. The authors of [8] build an inductive invariant in all cases except those for which every eigenvalue of the matrix governing the evolution of the LDS is either 0 or of modulus 1 and at least one of the latter is not a root of unity. This situation however cannot occur in our setting. Indeed, the eigenvalues of an integer matrix are algebraic integers, and an old result of Kronecker [19] asserts that unless all of the eigenvalues are roots of unity, one of them must have modulus strictly greater than 1 (the case in which *all* eigenvalues are 0 being of course trivial).

This concludes the proof of Theorem 4. □

5.2 Undecidability of $\mathbb{N}$-semi-linear Invariants for Nondeterministic LDS

If the enhanced expressivity of $\mathbb{N}$-semi-linear sets allows us always to find an invariant for deterministic LDS, it contributes in turn to making the invariant-synthesis problem undecidable when the LDS is not deterministic. We establish this through a reduction from the infinite Post correspondence problem (ω-PCP) that can be defined in the following way: given m pairs of non-empty words $\{(u^1, v^1), \ldots, (u^m, v^m)\}$ over alphabet $\{0, 2\}$, does there exist an infinite word $w = w_1 w_2 \ldots$ over alphabet $\{1, \ldots, m\}$ such that $u^{w_1} u^{w_2} \ldots = v^{w_1} v^{w_2} \ldots$. This problem is known to be undecidable when m is at least 8 [6,13].

Theorem 5. *The invariant synthesis problem for $\mathbb{N}$-semi-linear sets and linear dynamical systems with at least two matrices of size 91 is undecidable.*

Proof (Sketch). We first establish the result in the case of several matrices in low dimension; this can then be transformed in a standard way to two larger matrices (of size 91).

The proof is by reduction from the infinite Post correspondence problem. Given an instance of this problem the pair of words corresponding to each sequence of tiles has an integer representation, using base-4 encoding. An important property of our encoding is that the operation of appending a new tile to an existing pair of words can be encoded by matrix multiplication.

Recall that if the instance of ω-PCP is negative, then every generated pair of words will differ at some point. Our encoding is such that this difference of letters creates a difference in their numerical encodings that can be identified with an $\mathbb{N}$-semi-linear invariant. On the other hand, when there is a positive answer to the ω-PCP instance, there can be no $\mathbb{N}$-semi-linear invariant. □

5.3 Nondeterministic One-Dimensional Affine Updates

The previous section shows that point reachability for nondeterministic LDS is undecidable once there sufficiently many dimensions, motivating an analysis at lower dimensions. The MU Puzzle requires a single dimension with affine updates (or equivalently two dimensions in matrix representation, with the coordinate along the second dimension kept constant). We consider this one-dimensional affine-update case, and therefore, rather than taking matrices as input, we directly work with affine functions of the form $f_i(x) = a_i x + b_i$.

Theorem 6. *Given $x^{(0)}, y \in \mathbb{Z}$, along with a finite set of functions $\{f_1, \dots, f_k\}$ where $f_i(x) = a_i x + b_i$, $a_i, b_i \in \mathbb{Z}$ for $1 \le i \le k$, it is decidable whether y is reachable from $x^{(0)}$.*

Moreover, when y is unreachable, an $\mathbb{N}$-semi-linear separating inductive invariant can be algorithmically computed.

We note that decidability of reachability is already known [9,10]. We refine this result by exhibiting an invariant which can be used to disprove reachability. In fact our procedure will produce an $\mathbb{N}$-semi-linear set which can be used to decide reachability, and which, in instances of non-reachability, will be a separating inductive invariant. We have implemented this algorithm into our tool porous, enabling us to efficiently tackle the MU Puzzle as well as its generalisation to arbitrary collections of one-dimensional affine functions. We report on our experiments in Sect. 6.

We build a case distinction depending on the type of functions that appear:

Definition 5. *A function $f(x) = ax + b$...*

- *... is* redundant *if $f(x) = b$, (including possibly $b = 0$), or if $f(x) = x$.*
- *... is* counter-like *if $f(x) = x + b$, $b \neq 0$. Two counter-like functions, $f(x) = x + b$ and $g(x) = x + c$ are* opposing *if $b > 0$ and $c < 0$ (or vice-versa).*
- *... is* growing *if $f(x) = ax + b$ and $|a| \ge 2$. We say a growing function is* inverting *if $a \le -2$.*
- *... is* pure inverting *if $f(x) = -x + b$.*

Simplifying Assumptions

Lemma 4. *Without loss of generality, redundant functions are redundant; more precisely, we can reduce the computation of an invariant for a system having redundant functions to finitely many invariant computations for systems devoid of such functions.*

Proof. Clearly the identity function has no impact on the reachability set, and so can be removed outright. For any other redundant function, its impact on the reachability set does not depend on when the function is used, and we may therefore assume that it was used in the first step, or equivalently, using an alternative starting point. Hence the invariant-computation problem can be reduced to finitely many instances of the problem over different starting points, with redundant functions removed. Finally, taking the union of the resulting invariants yields an invariant for the original system. □

Lemma 5. *Without loss of generality, $x^{(0)} \ge 0$.*

Proof. We construct a new system, where each transition $f(x) = ax + b$ is replaced by $\overline{f}(x) = ax - b$. Then $x^{(0)}$ reaches y in the original system if and only if $-x^{(0)}$ reaches $-y$ in the new system. To see this, observe that if $f(x) = ax + b$, then $\overline{f}(-x) = -ax - b = -f(x)$. □

Lemma 6. *Suppose there are at least two distinct pure inverting functions (and possibly other types of functions). Then without loss of generality there are two opposing counters.*

Proof. Consider $f(x) = -x + b$, and $g(x) = -x + c$. Then $f(g(x)) = -(-x + c) + b = x + b - c$ and $g(f(x)) = -(-x + b) + c = x + c - b$. Since $b - c = -(c - b)$ and $b \neq c$ (as $f \neq g$) these two functions are opposing. □

Two Opposing Counters. Let us first observe that when there are two opposing counters, we essentially move in either direction by some fixed amount. This will entail that only $\mathbb{Z}$-(semi)-linear invariants can be produced, rather than proper $\mathbb{N}$-(semi)-linear invariants.

Lemma 7. *Suppose there are two opposing counters, $f(x) = x + b$, and $g(x) = x - c$. Then for any reachable x we have $\{x + d\mathbb{Z}\} \subseteq I$ for $d = \gcd(b, c)$.*

Therefore, starting with $\left\{x^{(0)} + d\mathbb{Z}\right\} \in I$ we can 'saturate' the invariant under construction using the following lemma:

Lemma 8. *Let $h(x) = x + d$ be chosen as a reference counter amongst the counters. If $\{x + d\mathbb{Z}\} \in I$, then $\{f(x) + d\mathbb{Z}\} \in I$ for every function f.*

Proof (Proof of Lemma 8*).* Consider the function $f(x) = ax+b$. If $x = y+dk \in I$, then $f(x) = ax + b = ay + adk + b = f(y) + adk \in I$.

Now thanks to the presence of counter $h(x) = x + d$, by choosing the initial $k \in \mathbb{Z}$ appropriately and applying $h(x)$ sufficiently many times (say $m \in \mathbb{N}$ times), one can reach $f(x) + adk + dm = f(x) + dn$ for any desired $n \in \mathbb{Z}$. □

Without loss of generality if $\{x + d\mathbb{Z}\}$ is in the invariant, then $0 \leq x < d$. We then repeatedly use Lemma 8 to find the required elements of the invariant. Since there are only finitely many residue classes (modulo d), every reachable residue class $\{c_1, \ldots, c_n\}$ can be found by saturation (in at most d steps), yielding invariant $\{c_1 + d\mathbb{Z}\} \cup \cdots \cup \{c_n + d\mathbb{Z}\}$.

Thanks to Lemma 6, in all remaining cases there is without loss of generality at most one pure inverter.

Only Pure Inverters. If there is exactly one pure inverter $f(x) = -x + b$ (and no other types of functions), then $f(x^{(0)}) = -x^{(0)} + b$ and $f(-x^{(0)} + b) = x^{(0)} - b + b = x^{(0)}$, thus the reachability set is finite, with exact invariant $\left\{x^{(0)}, -x^{(0)} + b\right\}$.

No Counters. If we are not in the preceding case and there are no counters, then there must be growing functions and by Lemma 6, without loss of generality at most one pure inverter. We show that all growing functions increase the modulus outside of some bounded region.

Lemma 9. *For every $M \geq 0$ and every growing function $f(x) = ax+b$, $|a| \geq 2$, there exists $C_f^M \geq 0$ such that if $|x| \geq C_f^M$ then $|f(x)| \geq |x| + M$.*

Proof. By the triangle inequality we have: $|f(x)| = |ax + b| \geq |a||x| - |b|$. Thus $|x| \geq \frac{|b|+|M|}{|a|-1} \implies |a||x| - |b| \geq |x| + |M| \implies |f(x)| \geq |x| + M$. $\square$

This is the only situation in which the invariant is not exactly the reachability set, and requires us to take an overapproximation.

Let $C = \max\left\{C_{f_1}^0, \ldots, C_{f_k}^0, |y| + 1\right\}$, for $f_1, \ldots, f_k$ growing functions. If there are no pure inverters then $\{-C - \mathbb{N}\} \cup \{C + \mathbb{N}\}$ is invariant (although may not yet contain the whole of $\mathcal{O}$). However, we can return the inductive invariant $\{-C - \mathbb{N}\} \cup \{C + \mathbb{N}\} \cup (\mathcal{O} \cap (-C, C))$. The set $\mathcal{O} \cap (-C, C)$ is finite and can elicited by exhaustive search, noting that once an element of the orbit reaches absolute value at least C, the remainder of the corresponding trajectory remains forever outside of $(-C, C)$.

If there is one pure inverter $g(x) = -x + d$ then observe that $-C$ is mapped to $C + d$ and $C + d$ is mapped to $-C$. Thus intuitively we want to use the interval $(-C, C + d)$. However two problems may occur: (a) since d could be less than 0 then $C + d$ may no longer be growing (under the application of the growing functions), and (b) an inverting growing function only ensures that $-C$ is mapped to a value greater than or equal to C, rather than $C+d$. Hence, we choose C' to ensure that $C' \pm d$ is still growing by at least $|d|$ (under the application of our growing functions). Let $C' = \max\left\{C_{f_1}^{|d|}, \ldots, C_{f_k}^{|d|}, |y| + 1\right\} + |d|$. Then the invariant is $\{-C' - \mathbb{N}\} \cup \{C' + d + \mathbb{N}\} \cup (\mathcal{O} \cap (-C', C' + d))$.

Non-opposing Counters. The only remaining possibility (if there do not exist two opposing counters, and not all functions are growing or pure inverters), is that there are counter-like functions, but they are all counting in the same direction. There may also be a single pure inverter, and possibly some growing functions.

Pick a counter $h(x) = x+d$ to be the reference counter; the choice is arbitrary, but it is convenient to pick a counter with minimal $|d|$. As a starting point, we have $\left\{x^{(0)} + d\mathbb{N}\right\} \subseteq I$.

Lemma 10. *If there is an inverter $g(x) = -ax + b$, with $a > 0, b \in \mathbb{Z}$, and we have $\{x + d\mathbb{N}\} \subseteq I$ then $\{g(x) + d\mathbb{Z}\} \subseteq I$.*

The crucial difference with Lemma 8 is the observation that now an $\mathbb{N}$-linear set has induced a $\mathbb{Z}$-linear set.

Proof. Let $r = g(x) + dm$ for $m \in \mathbb{Z}$. We show $r \in I$. Consider $x + dn$ for $n \in \mathbb{N}$, then $g(x + dn) = -a(x + dn) + b = -ax + b - adn = g(x) - adn$. Hence $g(x) - adn + dk$, $n, k \in \mathbb{N}$, is reachable by applying k times the function $h(x)$. Hence for any $m \in \mathbb{Z}$ there exists $k, n \in \mathbb{N}$ such that $k - na = m$, so that r is indeed reachable. □

Similarly to the situation with two opposing counters, whenever the invariant contains some $\mathbb{Z}$-linear set, Lemma 8 allows us to saturate amongst the finitely many reachable residue classes.

However, the invariant may contain subsets that are not $\mathbb{Z}$-linear. Consider $\{x + d\mathbb{N}\} \subseteq I$, which is not yet invariant. We repeatedly apply non-inverting functions to $\{x + d\mathbb{N}\}$ to obtain new $\mathbb{N}$-linear sets (not $\mathbb{Z}$-linear sets). When the function applied 'moves' in the direction of the counters this will ultimately saturate (in particular when applying other counter functions). However, in the opposite direction, we may generate infinitely many such classes.

Example 3. Consider the reference counter $h(x) = x+4$, with initial point 5. This yields an initial set $\{5 + 4\mathbb{N}\} \subseteq \mathcal{O}$, where 5 is the initial point and $4\mathbb{N}$ is derived from the counter increment. Now when applying $x \mapsto 2x + 6$ to $\{5 + 4\mathbb{N}\}$ we obtain $\{10 + 6 + 8\mathbb{N} + 4\mathbb{N}\} = \{16 + 4\mathbb{N}\}$, then $\{38 + 4\mathbb{N}\}$, and then $\{82 + 4\mathbb{N}\}$. However $\{82 + 4\mathbb{N}\} \subseteq \{38 + 4\mathbb{N}\}$ and we can therefore stop with the invariant $\{5 + 4\mathbb{N}\} \cup \{16 + 4\mathbb{N}\} \cup \{38 + 4\mathbb{N}\}$.

However, if the initial sequence is not moving in the direction of the reference counter, this saturation does not occur. Consider $\{5 + 4\mathbb{N}\}$ with the function $x \mapsto 2x - 6$. Then $\{5 + 4\mathbb{N}\}$ maps to $\{10 - 6 + 8\mathbb{N} + 4\mathbb{N}\} = \{4 + 4\mathbb{N}\}$, which maps to $\{2 + 4\mathbb{N}\}$, $\{-2 + 4\mathbb{N}\}$, $\{-10 + 4\mathbb{N}\}$, $\{-26 + 4\mathbb{N}\}$, and so on. However -2 and -10 are both 2 modulo 4 (and so is -26 as well). This means in the negative direction we can obtain arbitrarily large negative values congruent to 2 modulo 4 and then use the reference counter $h(x) = x + 4$ to obtain any value of $\{2 + 4\mathbb{Z}\}$. □

Clearly we can examine all reachable residue classes defined by our reference counter. Any residue class reachable after an inverting function induces a $\mathbb{Z}$-linear set. So it remains to consider those $\mathbb{N}$-linear sets reachable without inverting functions. The remaining case to handle occurs when we repeatedly induce $\mathbb{N}$-linear sets until they repeat a residue class in the direction opposite to that of the reference counter.

We consider the case for $h(x) = x + d$ with $d \geq 0$. The case with $h(x) = x - d$ is symmetric. It remains to detect when a set $\{x + d\mathbb{N}\}$ leads to $\{y + d\mathbb{N}\}$ by a sequence of non-inverting functions with $x \equiv y \mod d$. Then by repeated application of these functions one can reach sets $\{z + d\mathbb{N}\}$ with z arbitrarily small, hence we can replace $\{x + d\mathbb{N}\}$ by $\{x + d\mathbb{Z}\}$. We give further details in the full version.

Reachability. The above procedure is sufficient to decide reachability. In all cases apart from that in which there are no counters, the invariants produced coincide precisely with the reachability sets. A reachability query therefore reduces to asking whether the target belongs to the invariant.

In the remaining case, the invariant obtained is parametrised by the target via the bound C'. The target lies within the region $(-C', C'+d)$, within which we can compute all reachable points. Thus once again, the target is reachable precisely if it belongs to the invariant. However, for a new target of larger modulus, a different invariant would need to be built.

Complexity

Lemma 11. *Assume that all functions, starting point, and target point are given in unary. Then the invariant can be computed in polynomial time.*

Without the unary assumption, the invariant could have exponential size, and hence require at least exponential time to compute. That is because the invariant we construct could include every value in an interval, for example, $(-C, C)$, where C is of size polynomial in the largest value.

As shown in [10], the reachability problem is at least **NP**-hard in binary, because one can encode the integer Knapsack problem (which allows an object to be picked multiple times rather at most once). Moreover the Knapsack problem is efficiently solvable in pseudo-polynomial time via dynamic programming; that is, polynomial time assuming the input is in unary, matching the complexity of our procedure.

6 The POROUS Tool

Our invariant-synthesis tool porous[9] computes $\mathbb{N}$-semi-linear invariants for point and $\mathbb{Z}$-linear targets on systems defined by one-dimensional affine functions. porous includes implementations of the procedures of Theorem 3 (restricted to one-dimensional affine systems) and Theorem 6. porous is built in Python and can be used by command-line file input, a web interface, or by directly invoking the Python packages.

porous takes as input an instance (a start point, a target, and a collection of functions) and returns the generated invariant. Additionally it provides a proof that this set is indeed an inductive invariant: the invariant is a union of $\mathbb{N}$-linear sets, so for each linear set and each function, porous illustrates the application of that function to the linear set and shows for which other linear set in the invariant this is a subset. Using this invariant, porous can decide reachability; if the specific target is reachable the invariant is not in itself a proof of reachability (since the invariant will often be an overapproximation of the global reachability set). Rather, equipped with the guarantee of reachability, porous searches for a direct proof of reachability: a sequence of functions from start to target (a process which would not otherwise be guaranteed to terminate).

[9] Tool: invariants.davidpurser.net Code: github.com/davidjpurser/porous-tool.

Table 2. Results varying by size parameter (last row includes all instances tested). Times are given in seconds, with the average and maximum shown (except reachability proof time, which are all approximately 30 s due to instances that terminate just before the timeout).

Size	Invariant build time		Unreachable instances	Invariant proof time		Reachable instances	Reachable with proofs	Reachability proof time
	Avg	Max		Avg	Max		Within ≈30s	Avg
8	0.001	0.009	100 (9.84%)	0.005	0.261	916 (90.2%)	911 (99.5%)	0.033
16	0.001	0.020	122 (12.0%)	0.010	0.788	894 (88.0%)	885 (99.0%)	0.053
32	0.003	0.068	134 (13.2%)	0.020	0.911	882 (86.8%)	843 (95.6%)	0.203
64	0.008	0.261	150 (14.8%)	0.052	2.969	866 (85.2%)	766 (88.5%)	0.294
128	0.021	0.557	153 (15.1%)	0.096	2.426	863 (84.9%)	719 (83.3%)	0.464
256	0.088	2.838	166 (16.3%)	0.316	43.587	850 (83.7%)	620 (72.9%)	0.998
512	0.428	9.312	162 (15.9%)	0.899	21.127	854 (84.1%)	570 (66.7%)	1.120
1024	1.121	20.252	173 (17.0%)	3.275	65.397	843 (83.0%)	514 (61.0%)	1.646
All	0.209	20.252	1160 (14.3%)	0.584	65.397	6968 (85.7%)	5828 (83.6%)	0.499

Experimentation. POROUS was tested on all $2^7 - 1$ possible combinations of the following function types, with $a \geq 2, b \geq 1$: positive counters ($x \mapsto x + b$), negative counters ($x \mapsto x - b$), growing ($x \mapsto ax \pm b$), inverting and growing ($x \mapsto -ax \pm b$), inverters with positive counters ($x \mapsto -x + b$), inverters with negative counters ($x \mapsto -x - b$) and the pure inverter ($x \mapsto -x$). For each such combination a random instance was generated, with a size parameter to control the maximum modulus of a and b, ranging between 8 and 1024. The starting point was between 1 and the size parameter and the target was between 1 and 4 times the size parameter. Ten instances were tested for each size parameter and each of the $2^7 - 1$ combinations, with between 1 and 9 functions of each type (with a bias for one of each function type).

Our analysis, summarised in Table 2, illustrates the effect of the size parameter. The time to produce the proof of invariant is separated from the process of building the invariant, since producing the proof of invariant can become slower as $|I|$ becomes larger; it requires finding $L_k \in I$ such that $f_i(L_j) \subseteq L_k$ for every linear set $L_j \in I$ and every affine function f_i. In every case POROUS successfully built the invariant, and hence decided reachability very quickly (on average well below 1 s) and also produced the proof of invariance in around half a second on average. To demonstrate correctness in instances for which the target is reachable POROUS also attempts to produce a proof of reachability (a sequence of functions from start to target). Since our paper is focused on invariants as certificates of non-reachability, our proof-of-reachability procedure was implemented crudely as a simple breadth-first search without any heuristics, and hence a timeout of 30 s was used for this part of the experiment only.

Our experimental methodology was partially limited due to the high prevalence of reachable instances. A random instance will likely exhibit a large (often universal) reachability set. When two random counters are included, the chance that $\gcd(b_1, b_2) = 1$ (whence the whole space is covered) is around 60.8% and higher if more counters are chosen.

Overall around 86% of instances were reachable (of which 84% produced a proof within 30 s). Of the 14% of unreachable instances, all produced a proof, with the invariant taking around 0.2 s to build and 0.6 s to produce the proof. The 30-s timeout when demonstrating reachability directly is several orders of magnitudes longer than answering the reachability query via our invariant-building method.

A typical academic/consumer laptop was used to conduct the timing and analysis (a four-year-old, four-core MacBook Pro).

7 Conclusions and Open Directions

We introduced the notion of porous invariants, which are not necessarily convex and can in fact exhibit infinitely many ‘holes’, and studied these in the context of multipath (or branching/nondeterministic) affine loops over the integers, or equivalently nondeterministic integer linear dynamical systems. We have in particular focused on reachability questions. Clearly, the potential applicability of porous invariants to larger classes of systems (such as programs involving nested loops) or more complex specifications remains largely unexplored.

Our focus is on the boundary between decidability and undecidability, leaving precise complexity questions open. Indeed, the complexity of synthesising invariants could conceivably be quite high, except where we have highlighted polynomial-time results. On the other hand, the invariants produced should be easy to understand and manipulate, from both a human and machine perspective.

On a more technical level, in our setting the most general class of invariants that we consider are $\mathbb{N}$-semi-linear. There remains at present a large gap between decidability for one-dimensional affine functions, and undecidability for linear updates in dimension 91 and above. It would be interesting to investigate whether decidability can be extended further, for example to dimensions 2 and 3.

Acknowledgements. This work was funded by DFG grant 389792660 as part of TRR 248 (see perspicuous-computing.science). Joël Ouaknine was supported by ERC grant AVS-ISS (648701), and is also affiliated with Keble College, Oxford as emmy.network Fellow. James Worrell was supported by EPSRC Fellowship EP/N008197/1.

References

1. Almagor, S., Chistikov, D., Ouaknine, J., Worrell, J.: O-minimal invariants for discrete-time dynamical systems (2019, preprint, submitted). https://arxiv.org/abs/1802.09263

2. Bozga, M., Iosif, R., Konecný, F.: Fast acceleration of ultimately periodic relations. In: Touili, T., Cook, B., Jackson, P.B. (eds.) Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, 15–19 July 2010. Proceedings. Lecture Notes in Computer Science, vol. 6174, pp. 227–242. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_23. Extended VERIMAG technical report, TR-2012-10, 2012: http://www-verimag.imag.fr/TR/TR-2012-10.pdf
3. Chistov, A.: Algorithm of polynomial complexity for factoring polynomials and finding the components of varieties in subexponential time. J. Soviet Math. **34**(4), 1838–1882 (1986). https://doi.org/10.1007/BF01095643
4. Clarke, E.M., et al.: Abstraction and counterexample-guided refinement in model checking of hybrid systems. Int. J. Found. Comput. Sci. **14**(4), 583–604 (2003). https://doi.org/10.1142/S012905410300190X
5. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Aho, A.V., Zilles, S.N., Szymanski, T.G. (eds.) Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978, pp. 84–96. ACM Press (1978). https://doi.org/10.1145/512760.512770
6. Dong, J., Liu, Q.: Undecidability of infinite post correspondence problem for instances of size 8. RAIRO Theor. Informatics Appl. **46**(3), 451–457 (2012). https://doi.org/10.1051/ita/2012015
7. Douglas, R.H.: Gödel, Escher, Bach: An Eternal Golden Braid. Basic Books, New York (1979)
8. Fijalkow, N., Lefaucheux, E., Ohlmann, P., Ouaknine, J., Pouly, A., Worrell, J.: On the Monniaux problem in abstract interpretation. In: Chang, B.-Y.E. (ed.) SAS 2019. LNCS, vol. 11822, pp. 162–180. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32304-2_9
9. Finkel, A., Göller, S., Haase, C.: Reachability in register machines with polynomial updates. In: Chatterjee, K., Sgall, J. (eds.) MFCS 2013. LNCS, vol. 8087, pp. 409–420. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40313-2_37
10. Fremont, D.: The reachability problem for affine functions on the integers. CoRR abs/1304.2639 (2013). http://arxiv.org/abs/1304.2639
11. Giesl, J., et al.: Analyzing program termination and complexity automatically with AProVE. J. Autom. Reason. **58**(1), 3–31 (2016). https://doi.org/10.1007/s10817-016-9388-y
12. Ginsburg, S., Spanier, E.H.: Bounded ALGOL-like languages. Trans. Am. Math. Soc. **113**(2), 333–368 (1964). https://doi.org/10.1090/S0002-9947-1964-0181500-1
13. Halava, V., Harju, T.: Undecidability of infinite post correspondence problem for instances of size 9. RAIRO Theor. Informatics Appl. **40**(4), 551–557 (2006). https://doi.org/10.1051/ita:2006039
14. Heizmann, M., Hoenicke, J., Podelski, A.: Termination analysis by learning terminating programs. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 797–813. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_53
15. Hrushovski, E., Ouaknine, J., Pouly, A., Worrell, J.: Polynomial invariants for affine programs. In: Dawar, A., Grädel, E. (eds.) Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, 09–12 July 2018, pp. 530–539. ACM (2018). https://doi.org/10.1145/3209108.3209142
16. Kannan, R., Lipton, R.J.: Polynomial-time algorithm for the orbit problem. J. ACM **33**(4), 808–821 (1986). https://doi.org/10.1145/6490.6496

17. Karr, M.: Affine relationships among variables of a program. Acta Informatica **6**, 133–151 (1976). https://doi.org/10.1007/BF00268497
18. Kincaid, Z., Breck, J., Cyphert, J., Reps, T.W.: Closed forms for numerical loops. Proc. ACM Program. Lang. **3**(POPL), 55:1–55:29 (2019). https://doi.org/10.1145/3290368
19. Kronecker, L.: Zwei Sätze über Gleichungen mit ganzzahligen Coefficienten. Journal für die reine und angewandte Mathematik **57**(53), 173–175 (1857)
20. Leroux, J.: The general vector addition system reachability problem by presburger inductive invariants. Log. Methods Comput. Sci. 6(3) (2010). https://doi.org/10.2168/LMCS-6(3:22)2010
21. Leroux, J.: Vector addition system reachability problem: a short self-contained proof. In: Ball, T., Sagiv, M. (eds.) Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, 26–28 January 2011, pp. 307–316. ACM (2011). https://doi.org/10.1145/1926385.1926421
22. Markov, A.: On certain insoluble problems concerning matrices. Doklady Akad. Nauk SSSR. **57**, 539–542 (1947)
23. Monniaux, D.: On the decidability of the existence of polyhedral invariants in transition systems. Acta Informatica **56**(4), 385–389 (2018). https://doi.org/10.1007/s00236-018-0324-y
24. Ouaknine, J., Worrell, J.: Decision problems for linear recurrence sequences. In: Finkel, A., Leroux, J., Potapov, I. (eds.) RP 2012. LNCS, vol. 7550, pp. 21–28. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33512-9_3
25. Shmonin, G.: Lattices and Hermite normal form, February 2009. Lecture notes for the course Integer Points in Polyhedra at the Swiss Federal Institute of Technology Lausanne (EPFL)
26. Tzeng, W.: A polynomial-time algorithm for the equivalence of probabilistic automata. SIAM J. Comput. **21**(2), 216–227 (1992). https://doi.org/10.1137/0221017

JavaSMT 3: Interacting with SMT Solvers in Java

Daniel Baier, Dirk Beyer, and Karlheinz Friedberger

LMU Munich, Munich, Germany

Abstract. Satisfiability Modulo Theories (SMT) is an enabling technology with many applications, especially in computer-aided verification. Due to advances in research and strong demand for solvers, there are many SMT solvers available. Since different implementations have different strengths, it is often desirable to be able to substitute one solver by another. Unfortunately, the solvers have vastly different APIs and it is not easy to switch to a different solver (lock-in effect). To tackle this problem, we developed JavaSMT, which is a solver-independent framework that unifies the API for using a set of SMT solvers. This paper describes version 3 of JavaSMT, which now supports eight SMT solvers and offers a simpler build and update process. Our feature comparisons and experiments show that different SMT solvers significantly differ in terms of feature support and performance characteristics. A unifying Java API for SMT solvers is important to make the SMT technology accessible for software developers. Similar APIs exist for other programming languages.

Keywords: Satisfiability Modulo Theories · SMT Solver · Java · API

1 Introduction

SMT solvers [6, 21] are used in a multitude of applications, e.g., in formal software analysis, where automated test-case generation [7, 16, 29, 30], SMT-based algorithms for software verification [10, 34], and interactive theorem proving [27, 44] are used. Applications and users rely on efficiency and expressiveness (supported SMT theories) to compute reasonable results in time. For application developers, the usability and API of the solver are also important aspects, and some features needed in applications, such as interpolation or optimization, are not available in some solvers.

Using the solver's own API directly makes it difficult to switch to another solver without rewriting extensive parts of the application, as there is no standardized binary API for SMT solvers. The SMT-LIB2 standard [4] improves this issue by defining a common language to interact with SMT solvers. However, this communication channel does not define a solver interface for special features like optimization or interpolation.[1] Additionally, the application has to parse the data provided by the SMT solver on its own, and this of course slightly changes from solver to solver.

[1] A proposal for adding interpolation queries exists since 2012, see https://ultimate.informatik.uni-freiburg.de/smtinterpol/proposal.pdf .

JavaSMT [37] provides a common API layer across multiple back-end solvers to address these problems. Our Java-based approach creates only minimal overhead, while giving access to most solver features. JavaSMT is available under the Apache 2.0 License on GitHub.[2]

Contribution. Our contribution consists of three parts:

- We integrated more SMT solvers into the API framework JavaSMT (new: Boolector [43], CVC4 [5], and Yices2 [25]).
- We simplified the steps to get started using JavaSMT, by including support for more operating systems (new: MacOS and Windows) and more build techniques (new: Ant and Maven).
- We evaluated the performance of several algorithms for software verification to show that different SMT solvers have different strengths.

Outline. This paper first provides a brief overview of JavaSMT in Sect. 2, explaining the inner structure and features. Sect. 3 discusses the development since the previous publication [37]: more integrated SMT solvers and extended support for operating systems and build processes. Sect. 4 describes a case study, based on SMT-based algorithms [10] in a common verification framework.

Related Work. SMT-LIB2 [4] is the established standard format for exchanging SMT queries. It provides simple usage, is easy to debug, and widely known in the community. However, it requires extra effort to parse and transform formulas in the user application. Features like optimization, interpolation, and receiving nested parts of formulas are not defined by the standard, such that some SMT solvers provide their own individual solution for that. Alternatively, several SMT solvers already come with their special bindings for some programming languages. Most SMT solvers are written in C/C++, so interacting with them in these low-level languages is the easiest way. However, the support for higher-level languages is sparse. The most prominent language binding for several SMT solvers is Python, as it directly allows the access to C code and avoids automated memory management operations like asynchronous garbage collection. Bindings for Java are available for some SMT solvers, such as MathSAT5 and Z3, but missing, unsupported, or unmaintained for others, such as Boolector and CVC4.

In the following, we discuss libraries, similar to JavaSMT, that provide access to several underlying SMT solvers via a common user interface in different popular languages, and their binding mechanism, i.e., whether the solver interaction is based on a native interface or text-based on SMT-LIB2. With SMT-LIB2, an arbitrary SMT solver can be queried, but the interaction happens through communicating processes and the solver is mostly limited to features defined in the standard. Accessing a native interface directly allows to support more features of the underlying solver, e.g., using callbacks, simplifying formulas, or eliminating quantifiers.

Table 1 provides an overview of the libraries for interacting with SMT solvers. We enumerate several special features that are not available in some libraries,

[2] https://github.com/sosy-lab/java-smt

Table 1: Comparison of different interface libraries for SMT solvers

	Reference	Language	Native API	SMT-LIB2	Unsat Cores	Interpolation	Optimization	Formula Decomposition	Project - Forks	Project - Stars	Project - Year Latest Commit
JavaSMT	[37]	Java	✓	✗	✓	✓	✓	✓	22	90	2021
PySMT	[28]	Python	✓	✓	✓	✓	✓	✗	99	363	2021
SMT Kit		C/C++	✓	✗	✓	✗	✗	✗	4	36	2014
Smt-Switch	[38]	C/C++	✓	✗	✓	✓	✗	✓	15	40	2021
jSMTLIB	[20]	Java	✗	✓	✓	✗	✗	✗	15	21	2020
metaSMT	[45]	C/C++	✗	✓	✗	✗	✓	✓	19	43	2016
rsmt2		Rust	✗	✓	✓	✗	✗	✗	10	24	2021
SBV		Haskell	✗	✓	✓	✗	✓	✗	17	134	2021
Scala SMT-LIB		Scala	✗	✓	✓	✗	✗	✓	18	44	2021
ScalaSMT	[17]	Scala	✗	✓	✗	✗	✗	✓	1	4	2019
what4		Haskell	✗	✓	✓	✗	✗	✗	5	97	2021

such as unsat cores, interpolation, or optimization queries. Those features depend on the support by the underlying SMT solver, but can be provided in general by an API on top of them. Most libraries use their own formula representation and not just wrap the objects provided by the SMT solver. This potentially allows for easier formula decomposition and inspection, e.g., by using the visitor pattern. JavaSMT directly provides formula decomposition if available in the SMT solver. The provided numbers of forks and stars of the project repositories on GitHub or Bitbucket can be seen as a measurement of popularity.

PySMT [28] is a Python-based project and aims at rapid prototyping of algorithms using the native API of the installed SMT solvers. It has the ability to perform formula manipulation without a back-end SMT solver and additionally supports the conversion of boolean formulas to plain SAT problems and then apply a SAT solver or a BDD library. This approach comes with the drawback of a noticeable memory overhead and performance of an interpreted language. metaSMT [45], SMT Kit, and Smt-Switch [38] provide solver-agnostic APIs for interacting with various SMT solvers in C/C++ to focus on the application instead of the solver integration. jSMTLIB [20], Scala SMT-LIB, and ScalaSMT [17] are solver-independent libraries written in Java or Scala and interact via SMT-LIB2 with SMT solvers. Scala SMT-LIB and ScalaSMT allow to use an additional domain-specific language to interact with SMT solvers and rewrite Scala syntax into valid SMT-LIB2 and back. Both partially extend the SMT-LIB2 standard, e.g., by offering the ability to overload operators or receive interpolants. SBV and what4 are generic Haskell libraries based on process interaction via SMT-LIB2 and support several SAT and SMT solvers. rsmt2 offers a generic Rust library that currently supports three SMT solvers.

2 JavaSMT's Architecture and Solver Integration

In the following, we describe the architecture of JavaSMT and its main concepts. Afterwards, we give an overview of the integrated SMT solvers and their features. The architecture did not significantly change, but we added a few new SMT solvers, as shown in Fig. 1.

Architecture. JavaSMT provides a common API for various SMT solvers. The architecture, shown in Fig. 1, consists of several components: As common context, we use a `SolverContext` that loads the underlying SMT solver and defines the scope and lifetime of all created objects. As long as the context is available, we track memory regions of native SMT-solver libraries. When the context is closed, the corresponding memory is freed and garbage collection wipes all unused objects. Within a given context, JavaSMT provides `FormulaManager`s for creating formulas in various theories and `ProverEnvironment`s for solving SMT queries.

A `FormulaManager` allows to create symbols and formulas in the corresponding theories and provides a type-safe way to combine symbols and formulas in order to encode a more complex SMT query. We support the structural analysis (like splitting a formula into its components or counting all function applications in a formula) and transformations (like substituting symbols or applying equisatisfiable simplifications) of formulas.

Each `ProverEnvironment` represents a solver stack and allows to push/pop boolean formulas and check them for satisfiability (the hard part). This follows the idea of incremental solving (if the underlying SMT solver supports it). After a satisfiability check, the `ProverEnvironment` provides methods to receive a model, interpolants, or an unsatisfiable core for the given formula.

JavaSMT guarantees that formulas built with a single `FormulaManager` can be used in several `ProverEnvironment`s, e.g., the same formula can be pushed onto and solved within several distinct `ProverEnvironment`s. The interaction with independent `ProverEnvironment`s works from multiple threads. However, some SMT solvers require synchronization (e.g., locking for an interleaved usage) and other solvers do not require external synchronization (this allows concurrent usage).

SMT-Solver Integration and Bindings. Of the eight SMT solvers that are available in JavaSMT, only Princess [46] and SMTInterpol [18] were 'easy' to integrate, as they are written in Scala and Java, respectively. Those solvers also use the available memory management and garbage collection of the Java Virtual Machine (JVM). All other solvers are written in C/C++ and need a Java Native Interface (JNI) wrapper to interface with JavaSMT. Z3 [40] and CVC4 [5] provide their own Java wrappers, while the bindings used for MathSAT5 [19], Boolector [42], and Yices2 [25] are maintained by us. Those bindings are self-written or partially based on a version of the solver developers, extended with exception handling, and usable for debugging in JavaSMT. By providing language bindings for solvers in our library, we relieve the solver developers from this burden, and the implementation of exception handling and memory management is done in an efficient and common manner across several solvers.

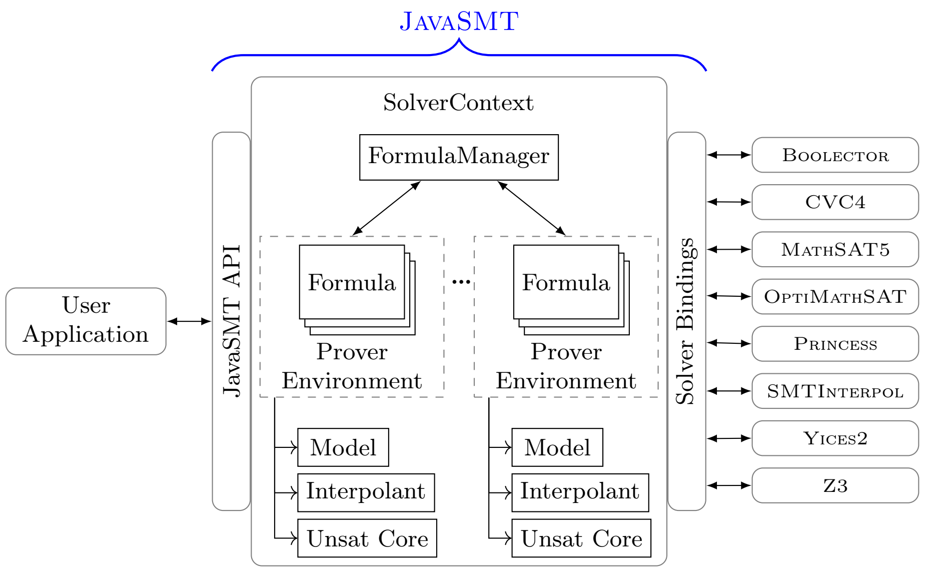


Fig. 1: Overview of JavaSMT

Table 2: Size (LOC) of the Java-based solver wrappers and native solver bindings

	Boolector	CVC4	MathSAT5	OptiMathSAT	Princess	SMTInterpol	Yices2	Z3
Java-based Wrapper	1644	1918	3229	3229	2042	2117	2728	2674
JNI Bindings	3136		1388	1508			1598	

Table 2 lists the size (lines of code) of the wrappers to integrate each solver in JavaSMT, in order to get a rough impression of the required effort to get a solver and its bindings usable in JavaSMT. The size information consists of two parts, namely the JNI bindings that are written in C/C++ and the Java code that implements the necessary interfaces of JavaSMT. An expressive solver API (like MathSAT5 or OptiMathSAT [47]) needs more code for their binding, with only a small increment in complexity compared to other solver bindings.

Note that the evolution of JavaSMT depends on the evolution of the underlying SMT solvers. Z3 is well-known, has a large user group, and an active development team. Yet, interpolation support for Z3 was dropped with release 4.8.1.[3] Bitwuzla [41] is the successor of the SMT solver Boolector, for which the developers still provide small fixes. Bitwuzla can be supported in JavaSMT in the future. CVC4 has been developed further to CVC5. However, the maintainers

[3] https://github.com/Z3Prover/z3/releases/tag/z3-4.8.1

dropped the existing Java API, partially because of issues with the Java garbage collection, and plan to replace it.[4] Yices2 is also actively maintained and adds new features regularly. For example, the developers added support for third-party SAT solvers such as CaDiCaL and CryptoMiniSat [48].

3 New Contributions in JavaSMT 3

This section describes the improvements over the JavaSMT version from five years ago [37], split into two parts. First, we describe newly integrated solvers and theory features. Second, we provide information about the build process.

Support for Additional SMT Solvers. JavaSMT 3 provides access to eight SMT solvers. Besides the solvers that were already integrated before, MathSAT5, OptiMathSAT, Z3, Princess, and SMTInterpol, the user can now additionaly use Boolector, CVC4, and Yices2. Table 3 lists available theories and important features supported by each individual solver. Boolector is specialized in Bitvector-based theories, but does not support the Integer theory. It is shipped with several back-end SAT solvers, from which the user can choose a favorite: CaDiCaL, CryptoMiniSat [48], Lingeling, MiniSat [26], and PicoSAT [13]. All solvers support the input of plain SMT-LIB2 formulas. However, the feature most requested by JavaSMT users is the input and output of SMT queries via the API, i.e., parsing and printing boolean formulas for a given context. This feature is required for (de-)serializing formulas to disk, for network transfer, and to translate formulas from one solver to another one. This feature is unfortunately missing for the newly integrated solvers, even though each solver internally already contains code for parsing and printing SMT-LIB2 formulas.

For formula manipulation, JavaSMT accesses the components of a formula, e.g., operators and operands. We do not require full access to the internal data structures of the SMT solvers, but only limited access to the most basic parts. Only Boolector does not provide the necessary API.

Build Simplification. JavaSMT 3 also supports more operating systems than before. Besides the existing support for Linux, we started to provide pre-compiled binaries for MacOS and Windows for more than half of the available solvers. This simplifies the initial steps for new users, which previously were required to compile and link the solvers on their own. This was an involving task, because of the diversity of build systems and dependencies of each solver.

In addition to this, we now offer direct support for two popular build systems for Java applications, namely Ant and Maven. JavaSMT comes with several examples and documentation, such that the mentioned build systems can be used to set up JavaSMT in a ready-to-go state on most systems. This eliminates the need for complex manual set up of dependencies and eases the use of JavaSMT and the SMT solvers.

[4] https://github.com/cvc5/cvc5/issues/5018

Table 3: SMT theories and features supported by SMT solvers in JavaSMT 3

		Boolector	CVC4	MathSAT5	OptiMathSAT	Princess	SMTInterpol	Yices2	Z3
SMT Theories	Integer	✗	✓	✓	✓	✓	✓	✓	✓
	Rational	✗	✓	✓	✓	✓	✓	✓	✓
	Array	✓	✓	✓	✓	✓	✓	✗	✓
	Bitvector	✓	✓	✓	✓	✓	✗	✓	✓
	Float	✗	✓	✓	✓	✗	✗	✗	✓
	UF	✓	✓	✓	✓	✓	✓	✓	✓
	Quantifier	✓	✓	✗	✗	✓	✗	✓	✓
Features	Incremental Solving	✓	✓	✓	✓	✓	✓	✓	✓
	Model	✓	✓	✓	✓	✓	✓	✓	✓
	Assumption Solving	✓	✗	✓	✓	✗	✗	✓	✓
	Interpolation	✗	✗	✓	✓	✓	✓	✗	✗
	Optimization	✗	✗	✗	✓	✗	✗	✗	✓
	UnsatCore	✗	✓	✓	✓	✓	✓	✓	✓
	UnsatCore with Assumptions	✗	✗	✓	✓	✗	✓	✓	✓
	SMT-LIB2 (plain text input)	✓	✓	✓	✓	✓	✓	✓	✓
	SMT-LIB2 (via API)	✗	✗	✓	✓	✓	✓	✗	✓
	Quantifier Elimination	✗	✓	✗	✗	✓	✗	✗	✓
	Formula Decomposition	✗	✓	✓	✓	✓	✓	✓	✓

4 Evaluation

Frameworks that provide a unified API to SMT solvers (such as JavaSMT, PySMT, and ScalaSMT) are necessary because the characteristics of the SMT solvers vary a lot. In the evaluation we provide support for this argument.

We inlined a discussion of the features already in the previous section. Table 3 provides the overview of supported theories and shows that certain theories are available only for a subset of SMT solvers. The table also shows that there are several features that restrict the choice of SMT solvers for certain applications.

In terms of performance, we evaluate JavaSMT 3 as a component of CPAchecker [11], which is an open-source software-verification framework[5] that provides a range of different SMT-based algorithms for program analysis [10] and encoding techniques for program control flow [8, 12]. We compare three well-known and successful SMT-based algorithms for software model checking and show that — when using the same algorithm and identical problem encoding — the performance result of an analysis depends on the used SMT solver. Some

[5] https://cpachecker.sosy-lab.org

algorithms depend on special features of the SMT solver, e.g., to provide a certain type of formula (such as interpolants) and operation on a formula (such as access to subformulas). There are SMT solvers that can not be used for some algorithms.

We aim to show that depending on the feature set of the SMT solvers, it is important to support a common API, and additionally, that using the text-based interaction via SMT-LIB2 is not an efficient solution, when it comes to formula analysis like adding additional information into a formula.

Benchmark Programs. We evaluate the usage of JavaSMT on a large subset of the SV-benchmark suite[6] containing over 1 000 verification tasks. To have a broad variation of benchmark tasks, we include reachability problems from the categories *BitVectors*, *ControlFlow*, *Heap*, and *Loops*.

BitVectors depends on bit-precise reasoning and thus, the SMT solver needs to support Bitvector logic. *Heap* depends on modeling heap memory access, e.g., which is either encoded in the theory of Arrays or as Uninterpreted Functions. The category *Loops* contains tasks where the state space is potentially quite large.

Experimental Setup. We run all our experiments on computers with Intel Xeon E3-1230 v5 CPUs with 3.40 GHz, and limit the CPU time to 15 min and the memory to 15 GB. We use CPAchecker revision r36714, which internally uses JavaSMT 3.7.0-73. The time needed for transforming the input program into SMT queries is rather small compared to the analysis time. Additionally, the progress of an algorithm depends on the result (e.g., model values or interpolants) returned from an SMT solver, thus we do not explicitly extract the run time required by the SMT solver itself for answering the satisfiability problem, but we measure the complete CPU time of CPAchecker for the verification run.

Analysis Configuration. We use three different SMT-based algorithms for software verification [10]. The first approach is bounded model checking (BMC) [14, 15], which is applied in software and hardware model checking since many years. In this approach, a verification problem is encoded as single large SMT query and given to the SMT solver. No further interaction with the SMT solver is required. In our evaluation, we use a loop bound $k = 10$, which limits the size of the SMT query.

The second approach is k-induction [9, 24], which extends BMC, and which uses auxiliary invariants to strengthen the induction hypothesis. In this approach, the algorithm generates several SMT queries (base case, inductive-step case, each with increasing loop bound) and uses an invariant generator that provides the auxiliary invariants. We use an interval-based invariant generator that provides not only the invariants, but also information about pointers and aliases, which must be inserted into the SMT formula using the formula visitor.

The third approach is predicate abstraction [3, 12, 31, 35], which uses Craig interpolation [22, 32, 39] to compute predicate abstractions of the program. This approach does not only query the SMT solver multiple times, but also uses (sequential) interpolation, which is currently supported only by MathSAT5, Princess, and SMTInterpol.

[6] https://github.com/sosy-lab/sv-benchmarks

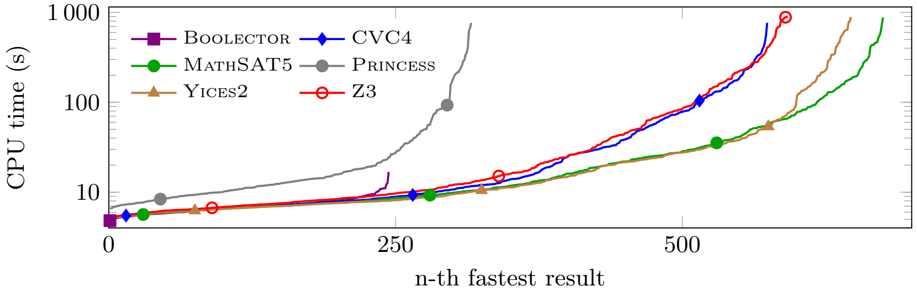


Fig. 2: Quantile plot for the runtime of k-induction with several SMT solvers

All approaches are executed in two configurations, depending on the used encoding of program statements: First, we apply a bitvector-based encoding that precisely models bit-precise arithmetics and overflows of the program. Second, an encoding based on linear integer arithmetic is used, which approximates the concrete program execution and is sufficient for some programs.

Solver Configuration. Overall, we aim to show that each solver provides a unique fingerprint of features and results. We aim for a precise program analysis and thus configure the SMT solvers to be as precise as possible, but with a reasonable configuration for each solver (i.e., without using a feature combination that is unsupported by the SMT solver).

SMTInterpol does not support efficient solving of SMT queries in Bitvector logic, thus, it is configured to use only Integer logic. Boolector misses Integer logic, thus, it is applied only to the bit-precise configurations. Additionally, this SMT solver does not support formula inspection and decomposition, which is required by several components in k-induction, e.g., to encode proper pointer aliasing for the program analysis. While the code for formula inspection is called quite often, its influence on the results for the selected benchmark tasks is small. In order to be comparable as far as possible, we deactivate pointer aliasing when using Boolector. Yices2 misses proper support for Array logic, thus, we use a UF-based encoding of heap memory as alternative for this solver, which results in a slightly unsound analysis, but a comparable formula size and run time.

Results and Discussion. Figure 2 provides the quantile plot for the results of k-induction configurations with bit-precise encoding using several SMT solvers. The plot shows the CPU time for valid analysis results, i.e., proofs or counterexamples found, for both expected results true and false. We aim for providing all result that are useful for a user and do not show results where the tool (or SMT solver) crashes or runs out of resources. We do not subtract the run time required for the framework CPAchecker itself (which starts a Java virtual machine), as we assume it to be comparable per program task; we are only interested in the asymptotics in this evaluation. The overall performance of SMT solvers is similar for simple verification tasks, i.e., those with a small run time in the analysis. For difficult tasks with harder SMT queries, the differences of the SMT solvers emerge. When applying k-induction, the analysis inserts additional constraints into the

Table 4: Run time for using different SMT solvers for bounded model checking ('BMC'), k-induction ('KI'), and predicate abstraction ('PA') with the theories of Bitvectors ('BV') and Integers ('Int'); CPU time given in seconds with two significant digits, ' TO' indicates timeouts (900 s), ' ERR' indicates errors, and empty cells indicate that the theory or interpolation was not supported

Verification Task	s3_srvr.blast.07.i.cil-2	byte_add_1-1	ps6-ll_valuebound100	s3	diamond_1-1	modulus-2	jain_5-2	s3_clnt_1.cil-2	diskperf_simpl1.cil	rule57_ebda_blast
Algorithm	BMC	BMC	KI	KI	KI	KI	PA	PA	PA	PA
Encoding	Int	BV	Int	Int	BV	BV	Int	Int	BV	BV
BOOLECTOR		**5.8**			ERR	ERR				
CVC4	340	6.4	TO	TO	110	TO				
MATHSAT5	17	7.8	200	53	60	54	TO	**11**	**16**	**7.1**
PRINCESS	TO	TO	530	TO	260	TO	**38**	160	TO	ERR
SMTINTERPOL	50		TO	140			TO	13		
YICES2	**14**	7.7	340	**23**	**34**	28				
Z3	15	6.7	**130**	66	43	**21**				

SMT formula and requires the SMT solver to allow access to components of existing formulas. As BOOLECTOR misses this specific feature, k-induction cannot be very effective here. Other SMT solvers are the preferred choice.

Table 4 contains some example tasks from all used algorithms and encodings, where the difference between distinct SMT solvers is noteworthy. Choosing the optimal SMT solvers for an arbitrary problem task is not obvious.

5 Conclusion

We contribute JAVASMT 3, the third generation of the unifying Java API for SMT solvers. The package now contains more SMT solvers, an improved build process, and support for MacOS and Windows. The project has over 20 contributors, 2 500 commits, and overall about 41 000 lines of code.[7] JAVASMT is used in Java applications (e.g., [23, 33, 36]) as a solution to combine convenience and performance for the interaction with SMT solvers, or to switch between different solvers and compare them [11, 49]. The most prominent application using JAVASMT is the verification framework CPACHECKER (a widely-used software

[7] `https://www.openhub.net/p/java-smt`

project[8] with 73 forks on GitHub alone), for which JavaSMT was originally developed. In the future, we plan to support more SMT solvers, operating systems, and hardware architectures, while keeping the user interface stable. We hope that even more researchers and developers of Java applications can benefit from SMT solving via a convenient and powerful API.

Data Availability Statement. All benchmark tasks for evaluation, configuration files, a ready-to-run version of our implementation, and tables with detailed results are available in our reproduction package on Zenodo as virtual machine [1] and as ZIP archive [2]. The source code of the open-source library JavaSMT [37] is available in the project repository; see `https://github.com/sosy-lab/java-smt`.

Funding. This project was supported by the Deutsche Forschungsgemeinschaft (DFG) – 378803395 (ConVeY).

References

1. Baier, D., Beyer, D., Friedberger, K.: Reproduction package (VM) for article 'JavaSMT 3: Interacting with SMT solvers in Java'. Zenodo (2021). https://doi.org/10.5281/zenodo.4708050
2. Baier, D., Beyer, D., Friedberger, K.: Reproduction package (ZIP) for article 'JavaSMT 3: Interacting with SMT solvers in Java'. Zenodo (2021). https://doi.org/10.5281/zenodo.4865175
3. Ball, T., Podelski, A., Rajamani, S.K.: Boolean and cartesian abstraction for model checking C programs. In: Proc. TACAS. pp. 268–283. LNCS 2031, Springer (2001). https://doi.org/10.1007/3-540-45319-9_19
4. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. In: Proc. SMT (2010)
5. Barrett, C.W., Conway, C.L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Proc. CAV. pp. 171–177. LNCS 6806, Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_14
6. Barrett, C., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Model Checking, pp. 305–343. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_11
7. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: Generating tests from counterexamples. In: Proc. ICSE. pp. 326–335. IEEE (2004). https://doi.org/10.1109/ICSE.2004.1317455
8. Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M.E., Sebastiani, R.: Software model checking via large-block encoding. In: Proc. FMCAD. pp. 25–32. IEEE (2009). https://doi.org/10.1109/FMCAD.2009.5351147
9. Beyer, D., Dangl, M., Wendler, P.: Boosting k-induction with continuously-refined invariants. In: Proc. CAV. pp. 622–640. LNCS 9206, Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_42
10. Beyer, D., Dangl, M., Wendler, P.: A unifying view on SMT-based software verification. J. Autom. Reasoning **60**(3), 299–335 (2018). https://doi.org/10.1007/s10817-017-9432-6

[8] `https://github.com/sosy-lab/cpachecker`

11. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Proc. CAV. pp. 184–190. LNCS 6806, Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_16
12. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: Proc. FMCAD. pp. 189–197. FMCAD (2010)
13. Biere, A.: PicoSAT Essentials. JSAT **4**(2-4), 75–97 (2008). https://doi.org/10.3233/SAT190039
14. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Proc. TACAS. pp. 193–207. LNCS 1579, Springer (1999). https://doi.org/10.1007/3-540-49059-0_14
15. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Advances in Computers **58**, 117–148 (2003). https://doi.org/10.1016/S0065-2458(03)58003-2
16. Cadar, C., Dunbar, D., Engler, D.R.: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. OSDI. pp. 209–224. USENIX Association (2008)
17. Cassez, F., Sloane, A.M.: ScalaSMT: Satisfiability modulo theory in Scala (tool paper). In: Proc. SCALA. pp. 51–55. ACM (2017). https://doi.org/10.1145/3136000.3136004
18. Christ, J., Hoenicke, J., Nutz, A.: SMTInterpol: An interpolating SMT solver. In: Proc. SPIN. pp. 248–254. LNCS 7385, Springer (2012). https://doi.org/10.1007/978-3-642-31759-0_19
19. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: Proc. TACAS. pp. 93–107. LNCS 7795, Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_7
20. Cok, D.R.: jSMTLIB: Tutorial, validation, and adapter tools for SMT-LIBv2. In: Proc. NFM. pp. 480–486. LNCS 6617, Springer (2011). https://doi.org/10.1007/978-3-642-20398-5_36
21. Cok, D.R., Déharbe, D., Weber, T.: The 2014 SMT competition. JSAT **9**, 207–242 (2016)
22. Craig, W.: Linear reasoning. A new form of the Herbrand-Gentzen theorem. J. Symb. Log. **22**(3), 250–268 (1957). https://doi.org/10.2307/2963593
23. Demarchi, S., Menapace, M., Tacchella, A.: Automating elevator design with satisfiability modulo theories. In: Proc. ICTAI. pp. 26–33. IEEE (2019). https://doi.org/10.1109/ICTAI.2019.00013
24. Donaldson, A.F., Haller, L., Kröning, D., Rümmer, P.: Software verification using k-induction. In: Proc. SAS. pp. 351–368. LNCS 6887, Springer (2011). https://doi.org/10.1007/978-3-642-23702-7_26
25. Dutertre, B.: Yices 2.2. In: Proc. CAV. pp. 737–744. LNCS 8559, Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_49
26. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Proc. SAT. pp. 502–518. LNCS 2919, Springer (2003). https://doi.org/10.1007/978-3-540-24605-3_37
27. Ernst, G., Huisman, M., Mostowski, W., Ulbrich, M.: VerifyThis: Verification competition with a human factor. In: Proc. TACAS. pp. 176–195. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_12
28. Gario, M., Micheli, A.: PySMT: A solver-agnostic library for fast prototyping of SMT-based algorithms. In: Proc. SMT (2015)
29. Godefroid, P., Sen, K.: Combining model checking and testing. In: Handbook of Model Checking, pp. 613–649. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_19

30. Godefroid, P., Levin, M.Y., Molnar, D.A.: Automated whitebox fuzz testing. In: Proc. NDSS. The Internet Society (2008)
31. Graf, S., Saïdi, H.: Construction of abstract state graphs with Pvs. In: Proc. CAV. pp. 72–83. LNCS 1254, Springer (1997). https://doi.org/10.1007/3-540-63166-6_10
32. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: Proc. POPL. pp. 232–244. ACM (2004). https://doi.org/10.1145/964001.964021
33. Ibrhim, H., Khattab, S., Elsayed, K., Badr, A., Nabil, E.: A formal methods-based rule verification framework for end-user programming in campus building automation systems. Building and Environment **181**, 106983 (2020). https://doi.org/10.1016/j.buildenv.2020.106983
34. Jhala, R., Majumdar, R.: Software model checking. ACM Computing Surveys **41**(4) (2009). https://doi.org/10.1145/1592434.1592438
35. Jhala, R., Podelski, A., Rybalchenko, A.: Predicate abstraction for program verification. In: Handbook of Model Checking, pp. 447–491. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_15
36. Joshaghani, R., Black, S., Sherman, E., Mehrpouyan, H.: Formal specification and verification of user-centric privacy policies for ubiquitous systems. In: Proc. IDEAS. pp. 31:1–31:10. ACM (2019). https://doi.org/10.1145/3331076.3331105
37. Karpenkov, E.G., Friedberger, K., Beyer, D.: JavaSMT: A unified interface for SMT solvers in Java. In: Proc. VSTTE. pp. 139–148. LNCS 9971, Springer (2016). https://doi.org/10.1007/978-3-319-48869-1_11
38. Mann, M., Wilson, A., Tinelli, C., Barrett, C.W.: SMT-Switch: A solver-agnostic C++ API for SMT solving. arXiv/CoRR (2007.01374) (2020), `https://arxiv.org/abs/2007.01374`
39. McMillan, K.L.: Interpolation and model checking. In: Handbook of Model Checking, pp. 421–446. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_14
40. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Proc. TACAS. pp. 337–340. LNCS 4963, Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
41. Niemetz, A., Preiner, M.: Bitwuzla at the SMT-COMP 2020. arXiv/CoRR (2006.01621) (2020), `https://arxiv.org/abs/2006.01621`
42. Niemetz, A., Preiner, M., Biere, A.: Boolector 2.0. J. Satisf. Boolean Model. Comput. **9**(1), 53–58 (2014). https://doi.org/10.3233/sat190101
43. Niemetz, A., Preiner, M., Wolf, C., Biere, A.: Btor2, BtorMC, and Boolector 3.0. In: Proc. CAV. pp. 587–595. LNCS 10981, Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_32
44. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. LNCS 2283, Springer (2002). https://doi.org/10.1007/3-540-45949-9
45. Riener, H., Haedicke, F., Frehse, S., Soeken, M., Große, D., Drechsler, R., Fey, G.: metaSMT: Focus on your application and not on solver integration. Int. J. Softw. Tools Technol. Transf. **19**(5), 605–621 (2017). https://doi.org/10.1007/s10009-016-0426-1
46. Rümmer, P.: A constraint sequent calculus for first-order logic with linear integer arithmetic. In: Proc. LPAR. pp. 274–289. LNCS 5330, Springer (2008). https://doi.org/10.1007/978-3-540-89439-1_20
47. Sebastiani, R., Trentin, P.: OptiMathSAT: A tool for optimization modulo theories. In: Proc. CAV. pp. 447–454. LNCS 9206, Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_27
48. Soos, M., Nohl, K., Castelluccia, C.: Extending SAT solvers to cryptographic problems. In: Kullmann, O. (ed.) Proc. SAT. pp. 244–257. LNCS 5584, Springer (2009). https://doi.org/10.1007/978-3-642-02777-2_24

49. Sprey, J., Sundermann, C., Krieter, S., Nieke, M., Mauro, J., Thüm, T., Schaefer, I.: SMT-based variability analyses in FeatureIDE. In: Proc. VaMoS. pp. 6:1–6:9. ACM (2020). https://doi.org/10.1145/3377024.3377036

Efficient SMT-Based Analysis of Failure Propagation

Marco Bozzano[1], Alessandro Cimatti[1], Anthony Fernandes Pires[1], Alberto Griggio[1], Martin Jonáš[1], and Greg Kimberly[2](✉)

[1] Fondazione Bruno Kessler, Trento, Italy
{bozzano,cimatti,griggio,mjonas}@fbk.eu
[2] The Boeing Company, Seattle, USA
greg.kimberly@boeing.com

Abstract. The process of developing civil aircraft and their related systems includes multiple phases of Preliminary Safety Assessment (PSA). An objective of PSA is to link the classification of failure conditions and effects (produced in the functional hazard analysis phases) to appropriate safety requirements for elements in the aircraft architecture. A complete and correct preliminary safety assessment phase avoids potentially costly revisions to the design late in the design process. Hence, automated ways to support PSA are an important challenge in modern aircraft design. A modern approach to conducting PSAs is via the use of abstract propagation models, that are basically hyper-graphs where arcs model the dependency among components, e.g. how the degradation of one component may lead to the degraded or failed operation of another. Such models are used for computing *failure propagations*: the fault of a component may have multiple ramifications within the system, causing the malfunction of several interconnected components. A central aspect of this problem is that of identifying the minimal fault combinations, also referred to as *minimal cut sets*, that cause overall failures.

In this paper we propose an expressive framework to model failure propagation, catering for multiple levels of degradation as well as cyclic and nondeterministic dependencies. We define a formal sequential semantics, and present an efficient SMT-based method for the analysis of failure propagation, able to enumerate cut sets that are minimal with respect to the order between levels of degradation. In contrast with the state of the art, the proposed approach is provably more expressive, and dramatically outperforms other systems when a comparison is possible.

1 Introduction

The process of developing civil aircraft and their related systems is guided by documents ARP4754A [17] and ARP4761 [16] produced by the engineering and standards organization SAE International. These documents describe a structured process for the safety assessment of these classes of platforms. An important stage is that of the Preliminary Aircraft Safety Assessment (PASA) and

A. Silva and K. R. M. Leino (Eds.): CAV 2021, LNCS 12760, pp. 209–230, 2021.
https://doi.org/10.1007/978-3-030-81688-9_10

Preliminary System Safety Assessment (PSSA). The PASA is followed by multiple PSSA, carried out at the level of the systems composing the aircraft. One important goal of these process stages is to link the classification of failure conditions and effects (produced in the aircraft functional hazard analysis phase) to appropriate safety requirements for elements in the aircraft architecture. These safety requirements drive, among other things, assignment of target Development Assurance Levels (DAL) for items within the architecture. A complete and correct preliminary safety assessment phase avoids potentially costly revisions to the design late in the design process. Hence, automated ways to support PSA are an important challenge in modern aircraft design [18].

An important goal of PSAs is to fully understand how faults of simple functions (e.g. providing electrical power, on-ground braking) interact and propagate to affect the overall behaviours (e.g. landing, take-off, taxiing). A modern approach to conducting such safety assessments is via propagation models [1,14,19], that model the dependency among components, e.g. how the degradation of one component may lead to the degraded or failed operation of another. Such models are used for computing *failure propagations*: the fault of a component may have multiple ramifications within the system, causing the malfunction of several interconnected components. A central problem is identifying the minimal fault combinations, also referred to as *minimal cut sets*, that cause overall failures [12].

Given that PSAs occur in the early stages of the development process when limited information regarding the design is available, reasoning is carried out at a very high level of abstraction. Therefore, instead of using behavioural models (e.g., infinite-state transition systems) adopted in formal verification, the system is more naturally modeled by a simpler formalism of propagation graphs. This does not make PSA any easier. There are in fact several aspects that must be taken into account. The first problem is the sheer size of propagation graphs, both in terms of nodes and hyper-paths to be explored, which make enumerative techniques completely inadequate.

Second, the propagation is non-Boolean [19]. That is, the degradation levels of the system functions are not binary (working vs not working) but the functions may be subject to different levels of degradation (e.g. fully operational, partly failed, completely failed), and fail in different ways (e.g. detected vs undetected, stuck open vs stuck closed), and different failures may be associated to different probabilities [19]. For example, the state of a component can be abstractly modeled into *working (w)*, *failed safe (fs)*, *failed detected (fd)*, or *failed undetected (fu)*, with degrees of degradation partially ordered as shown in Fig. 1.

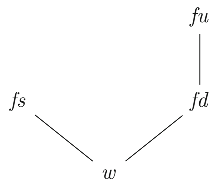


Fig. 1. Hasse diagram of the FDS W3F [14].

In this setting, the notion of minimality needs to take into account the order among the levels of degradation, and can not be simply considered in terms of minimality with respect to set-inclusion. Third, various forms of failure propagation may be possible, e.g., nondeterministic, temporally-constrained, cyclic. For example, the failure of a power generator may lead, within a certain amount of time, to a depleted battery and then to the loss of an engine. In turn, the loss of an engine may compromise the ability to generate power, which clearly requires the ability to deal with cyclic propagation graphs. Additionally, a failure of the control system might cause a pressure valve to become either stuck open or stuck closed; this requires the ability to deal with nondeterministic propagations.

In this paper we tackle the problem of analyzing failure propagation in the full generality required by real-world applications. We start from Finite Degradation Structures (FDS) [14], a recently-proposed modeling framework, which unifies various combinational models traditionally used in safety analysis (such as fault trees and minimal cut sets) and generalizes them to deal with different levels of degradation. We propose a framework, referred to as PGFDS (Propagation Graphs over FDS), that allows to model non-deterministic and cyclic propagation graphs. The framework is general and can be used in other safety-critical domains.

In order to deal with cyclic behaviours, PGFDS require a sequential semantics, expressed via symbolic transition systems. The computation of minimal cut sets over PGFDS can be carried out by means of techniques based on model checking, developed for the general case of behavioural models [6].

Then, we prove that it is possible to carry out the same analysis within a combinational setting, leveraging two widely adopted assumptions: that faults are persistent and that the fault propagation is monotone. These assumptions allow us to devise an efficient algorithm that can analyze fault propagations of realistic industrial benchmarks that are currently out of reach of state-of-the-art methods. The analysis of PGFDS is reduced to model enumeration for an SMT formula that does not require the explicit unrolling of the transition system. We tackle two key difficulties. The first one is to ensure causality and rule out self-supporting fault configurations in the combinational encoding. This is done by imposing cycle-breaking constraints requiring the existence of a partial order that is then constructed by the SMT solver during the analysis. The second one is to devise efficient enumeration techniques of models that are FDS-minimal, i.e., minimal with respect to *the severity of the degradation* given by the FDS. To this end, we propose an SMT-based enumerator of FDS-minimal models.

We have experimentally evaluated our approach on a comprehensive set of realistic benchmarks, also generating random systems that have a similar structure as our proprietary systems[1]. The results demonstrate substantial advances with respect to the state of the art. Our approach is clearly superior to the approach proposed in [14], that is limited to the case of acyclic deterministic PGFDS. For the cyclic PGFDSs, we contrast our approach against the sequential approach based on model-checking and show that our approach is able to scale to large PGFDS, dramatically outperforming the sequential approach.

This paper is structured as follows. In Sect. 2 we present the mathematical notation and background on FDS. In Sect. 3 we describe Propagation Graphs over FDS (PGFDS). In Sect. 4 we present the combinational encoding of PGFDS into SMT. In Sect. 5 we describe how to use the SMT encoding for the enumeration of FDS-minimal cut sets. In Sect. 6 we discuss some related work, and in Sect. 7

[1] Unfortunately the proprietary systems cannot be disclosed.

we present the experimental evaluation. In Sect. 8 we draw some conclusions and outline directions for future work.

2 Preliminaries

In the section, we explain the basic mathematical conventions that are used in the paper. We assume that the reader is familiar with the basic ideas of Satisfiability Modulo Theories (SMT) and in particular with the theory of linear integer arithmetic and the DPLL(T) procedure, as presented, e.g., in [2].

If convenient, we define unary functions with small domains in-place extensionally, e.g., $\{1 \mapsto 2, 2 \mapsto 3\}$ is a function with domain $\{1, 2\}$ that maps 1 to 2 and 2 to 3. We say that the n-ary function $f(x_1, x_2, \ldots, x_n)$ *depends* on its formal argument x_i if there are some values $v_1, v_2, \ldots, v_n, v_i'$ in the corresponding domains such that $f(v_1, v_2, \ldots, v_i, \ldots v_n) \neq f(v_1, v_2, \ldots, v_i', \ldots v_n)$. Given sets A and B, we denote as B^A the set of all functions from A to B. Given a partially ordered set $(A, \leq)$, its subset $B \subseteq A$ is called an *upper* (resp. *lower*) set if for all $b \in B$, $a \in A$, the condition $a \geq b$ (resp. $a \leq b$) implies $a \in B$.

A Finite Degradation Structure (FDS) [14] is a triple $(FM, \leq, \bot)$, where FM is a finite set of failure modes and $\leq$ is a partial order on FM with the least element $\bot$. For any set A and an FDS $B = (FM_B, \leq_B, \bot_B)$, the FDS B^A for the set of functions from A to FM_B is defined as $((FM_B)^A, \leq_{B^A}, \bot_{B^A})$, where $\bot_{B^A}(a) = \bot_B$ for all $a \in A$, and $f \leq_{B^A} f'$ if and only if $f(a) \leq_B f'(a)$ for all $a \in A$. We assume that each FDS contains at least two elements. We say that an FDS is *Boolean* if it is isomorphic to the structure $(\{\bot, \top\}, \bot \leq \top, \bot)$. In the following, for an FDS $D = (FM, \leq, \bot)$, we denote elements of the set FM with f, f' and call them *failure modes*.

Given a first-order formula φ over the language of the theory of linear integer arithmetic, an assignment μ that assigns a value $\mu(b) \in \{\mathbf{false}, \mathbf{true}\}$ to each free Boolean variable b of φ and a value $\mu(n) \in \mathbb{Z}$ to each free integer variable n of φ is called a model of φ (denoted $\mu \models \varphi$) if μ makes φ true. If B is a subset of free Boolean variables of φ, the model $\mu \models \varphi$ is called *subset-minimal with respect to* B if there is no model $\mu' \models \varphi$ such that $\{b \in B \mid \mu'(b) = \mathbf{true}\} \subsetneq \{b \in B \mid \mu(b) = \mathbf{true}\}$.

A *transition system* TS is a tuple (X, I, T) where X is a set of (state) variables, $I(X)$ is a formula representing the initial states, and $T(X, X')$ is a formula representing the transitions. A *state* of TS is an assignment to the variables X. A *trace* of M is a (possibly infinite) sequence $s_0, s_1, \ldots$ of states such that $s_0 \models I$ and, for all $i \geq 0$, $s_i, s_{i+1}' \models T$.

3 Propagation Graphs over FDSs

In this section, we introduce our model for fault propagation, which we call Propagation Graphs over FDSs (PGFDS), and provide a sequential semantics for it which can be used to encode PGFDSs into transition systems.

Intuitively, a Propagation Graph over FDS (PGFDS) consists of a set of components of the system and of the *next* function. In each step of the failure propagation, each component is in some failure mode from the underlying FDS. In the next step of the failure propagation, each component can either 1) stay in its previous failure mode or 2) switch to an arbitrary failure mode from the set of possible next failure modes. The set of possible next failure modes for each component is given by the function *next*, based on the current failure modes of all components in the system.

Definition 1 (Propagation Graph over FDS (PGFDS)). *Given a finite degradation structure $D = (FM, \leq, \bot)$, a* propagation graph over D *is a pair $S = (C, next)$, where*

- C *is a finite set of* system components*, and*
- $next\colon C \to (FM^C \to 2^{FM})$ *is a mapping that assigns to each component $c \in C$ a* next failure mode function $next(c)$*, which maps failure modes of all components in C to a set of possible next failure modes of c.*

A state *of S is a mapping $s\colon C \to FM$ that assigns a failure mode $f \in FM$ to each system component $c \in C$.*

Example 1. Consider a system with three components, H (hydraulic), E (electric), and G (control on ground), over the Boolean FDS $(\{\bot, \top\}, \bot \leq \top, \bot)$. Each of the components is either working correctly (represented by the failure mode $\bot$) or incorrectly ($\top$). Component G depends on the correct functionality of either E or H. Component E depends on H to function correctly and, symmetrically, H depends on E. The failure propagation of this system can be described by a PGFDS $S = (\{\text{G}, \text{E}, \text{H}\}, next)$, where

- $next(\text{G})(s) = \{\top\}$ if $s(\text{E}) = s(\text{H}) = \top$ and $next(\text{G})(s) = \emptyset$ otherwise;
- $next(\text{E})(s) = \{\top\}$ if $s(\text{H}) = \top$ and $next(\text{E})(s) = \emptyset$ otherwise;
- $next(\text{H})(s) = \{\top\}$ if $s(\text{E}) = \top$ and $next(\text{H})(s) = \emptyset$ otherwise.

Note that $next(c)(s) = \emptyset$ means that if the system is in the state s, the component c cannot change its current failure mode.

The structure is intuitively associated with the hypergraph depicted in Fig. 2. The dashed rectangles represent the fact that each component can fail on its own (*locally*); the hyper-arc from E and H to G is conjunctive, while the arcs incoming into a node are disjunctive. □

The important assumption of our approach is that we consider only fault-persistent propagations, i.e., fault propagations where each component can fail only once and after it does, it stays in the same failure mode forever. Note that this is a realistic assumption that is also used in other techniques for reliability analysis [5]. It is also implicitly used in other modeling techniques that are purely combinational (e.g., [19]) because they model the system only in a single time step, without considering any change in time whatsoever. Single propagation step of such computations can be described by a *fault-persistent transition relation*; the whole such computation as *fault-persistent failure propagation.*

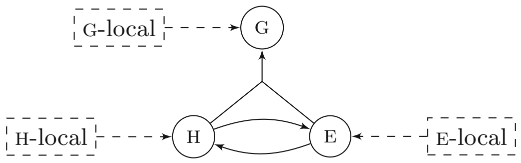


Fig. 2. The hypergraph view of a simple PGFDS.

Definition 2 (Fault-persistent transition relation). *Let $S = (C, next)$ be a* PGFDS *over an* FDS *with the least element $\bot$. The* fault-persistent transition relation *of S, denoted as R_s, is the binary relation between states of S such that for all states s, s', the relation $R_s(s, s')$ holds if and only if for each $c \in C$*

- *$s'(c) = s(c)$ or*
- *$s(c) = \bot$ and $s'(c) \in next(c)(s)$.*

Definition 3 (Fault-persistent failure propagation). *Given a* PGFDS *$S = (C, next)$, its fault-persistent transition relation R_s, and $k \in \mathbb{N}$, the sequence $(s_i)_{0 \le i \le k}$ of states of S is called a* fault-persistent failure propagation *if the relation $R_s(s_i, s_{i+1})$ holds for all $0 \le i < k$.*

Because we deal only with fault-persistent failure propagations in this paper, we from now on refer to the fault-persistent transition relation and the fault-persistent failure propagation only as *transition relation* and *failure propagation*, respectively.

Definition 4 (Cyclic PGFDS). *Let $S = (C, next)$ be a* PGFDS*. A component $c \in C$ depends on a component $d \in C$ iff $next(c)(s) \neq next(c)(s')$ for some $s, s' \colon C \to FM$ such that $s(d) \neq s'(d)$ and $s(c') = s'(c')$ for all $c' \neq d$. Let $deps(c) := \{d \in C \mid c \text{ depends on } d\}$, $D \subseteq C \times C$ be such that $D(c, c')$ if and only if $c' \in deps(c)$, and let D^+ be the transitive closure of D. Then we say that S is* cyclic *if and only if there exists $c \in C$ such that $D^+(c, c)$ holds.*

Example 2. In the PGFDS S from Example 1, the component G depends on components E and H, the component E depends on H, and the component H depends on E. The PGFDS S is therefore cyclic because E (and also H) transitively depends on itself. □

To analyze reliability of the modeled system, it is important to identify the failures of its components (i.e., assignment of failure modes to the components) which cause the system to reach a given set of dangerous states, usually called *top level event (TLE)*. Such assignments are called *cut sets*. Since the number of all cut sets can be prohibitively large, it is often enough to identify the least severe failures in terms of the underlying FDS that are sufficient to cause the TLE. Such cut sets are called FDS-*minimal*, or *minimal* for short. These concepts are formalized in the following definitions.

Definition 5 (Top Level Event). *Given a* PGFDS S*, a* Top Level Event *(TLE) is an arbitrary set of states of* S*.*

Definition 6 ((FDS-Minimal) Cut Set). *Given a* PGFDS $S = (C, next)$*, and a top level event* TLE*, a* cut set *is any state* s *for which there is a fault-persistent failure propagation that starts in* s *and ends in some* $s_k \in TLE$*. A cut set is called* FDS-minimal *(or* minimal *for short) if it is minimal with respect to the pointwise ordering* $\leq$ *of the underlying* FDS*.*

Given a system S and a top level event TLE, we denote the set of all corresponding cut sets as $CS(S, TLE)$ and the set of all minimal cut sets as $MCS(S, TLE)$. As a convention, when talking about cut sets, we will explicitly mention only the components to which the cut set assigns a failure mode different from $\bot$.

Example 3. Consider again the PGFDS S from Example 1 and the top level event $\text{TLE} = \{s\colon \{\text{G}, \text{E}, \text{H}\} \to \{\top, \bot\} \mid s(\text{G}) = \top\}$, which corresponds to the component G not working correctly. The minimal cut sets for the PGFDS S and the given top level event are

1. $\{\text{G} \mapsto \top\}$, witnessed by a failure propagation $(\{\text{G} \mapsto \top, \text{E} \mapsto \bot, \text{H} \mapsto \bot\})$ of length 1.
2. $\{\text{E} \mapsto \top\}$, witnessed by a failure propagation $(\{\text{G} \mapsto \bot, \text{E} \mapsto \top, \text{H} \mapsto \bot\}, \{\text{G} \mapsto \bot, \text{E} \mapsto \top, \text{H} \mapsto \top\}, \{\text{G} \mapsto \top, \text{E} \mapsto \top, \text{H} \mapsto \top\})$ of length 3.
3. $\{\text{H} \mapsto \top\}$, witnessed by a failure propagation $(\{\text{G} \mapsto \bot, \text{E} \mapsto \bot, \text{H} \mapsto \top\}, \{\text{G} \mapsto \bot, \text{E} \mapsto \top, \text{H} \mapsto \top\}, \{\text{G} \mapsto \top, \text{E} \mapsto \top, \text{H} \mapsto \top\})$ of length 3.

Note that besides these three minimal cut sets, there are other cut sets that are not minimal, such as $\{\text{E} \mapsto \top, \text{H} \mapsto \top\}$. □

Fault-persistent computations of a PGFDS can be easily represented as traces of a (symbolic) transition system.

Definition 7 (Fault-persistent transition system). *Given a* PGFDS $S = (C, next)$ *and an* FDS $D = (FM, \leq, \bot)$*, the corresponding* fault-persistent (symbolic) transition system *is given by* $TS_S = (X, \textbf{true}, T)$*, where:*

- $X = \{x_c \mid c \in C\}$ *is the set of state variables, with domain* FM*;*
- $T(X, X')$ *is a symbolic encoding of the fault-persistent transition relation of* S *as given in Definition 2. That is, for each assignment* $\mu\colon X \cup X' \to FM$*,* $\mu \models T$ *if and only if* $R_s(s, s')$ *holds, where* $s\colon C \to FM$ *is defined as* $s(c) = \mu(x_c)$ *(and similarly for* s'*).*

By definition, every fault-persistent computation of S has a corresponding trace (of the same length) in TS_S. Therefore, encoding PGFDSs as transition systems allows leveraging off-the-shelf algorithms for subset-minimal cut set enumeration, such as those given in [6]. However, this might be inefficient, particularly for TLEs that are triggered by long failure propagations (corresponding to equally-long traces of the induced transition system). Moreover, as we show later, enumerating FDS-minimal cut sets is more involved.

Fault propagation systems used in practice often have the property that no transition can be disabled by additional faults, i.e., by switching a failure mode of a component from $\bot$ to $f \neq \bot$. This is also the case for the PGFDS from Example 1. Such systems are called *subset-monotone* or *monotone* for short. This is formalized by the following definition.

Definition 8 (Subset-monotone PGFDS). *A* PGFDS $S = (C, \mathit{next})$ *is called* subset-monotone *if for all* $s, s' \colon C \to FM$*, the condition* $\forall c \in C.\ s(c) \neq \bot \to s(c) = s'(c)$ *implies* $\forall c \in C.\ \mathit{next}(c)(s) \subseteq \mathit{next}(c)(s')$*.*

4 From Sequential to Combinational

In this section, we describe a combinational encoding of fault-persistent computations of a PGFDS, which is guaranteed to be exact for subset-monotone PGFDSs and provides a useful overapproximation for general PGFDSs. In the rest of the section, let $S = (C, \mathit{next})$ be a PGFDS over the FDS $D = (FM, \leq, \bot)$, and TLE be a top level event. We show how to construct a first-order formula φ_{cs} over the theory of linear integer arithmetic whose models correspond to cut sets of S with respect to TLE. In the next section, we then use this formula to enumerate all FDS-minimal cut sets of S.

To encode the propagations of S, for each component $c \in C$ and each failure mode $f \in FM$ we introduce two Boolean variables: $I_{c,f}$ and $F_{c,f}$. The variable $I_{c,f}$ encodes whether c was in the failure mode f in the initial state of the propagation. The variable $F_{c,f}$ encodes whether c has been in the failure mode f at any time during the propagation. We can then encode TLE as a formula φ_{TLE} over variables $F_{c,f}$.[2]

Considering now a possible propagation, a component c can be in failure mode $f \neq \bot$ at some time during the propagation for two reasons: either it was already in f in the initial state of the propagation, or it transitions to f because of its *next* function. The first case is represented by $I_{c,f}$ being true. The second case can be encoded as follows (for each $c \in C$ and $f \in FM \setminus \{\bot\}$):

$$\bigvee_{\substack{s \colon C \to FM \\ f \in next(c)(s)}} \bigwedge_{\substack{d \in deps(c) \\ s(d) \neq \bot}} F_{d,s(d)}, \tag{1}$$

stating that there must exist a row in the truth table of $\mathit{next}(c)$, whose result includes f and which agrees with the current state on the failure modes of failed dependencies.[3] The above, however, would *not* work in the presence of cycles. This can already be seen on the simple cyclic PGFDS from Example 1.

[2] A naive encoding would be using the formula $\bigvee_{s \in TLE}(\bigwedge_{c \in C, s(c) \neq \bot} F_{c,s(c)} \wedge \bigwedge_{c \in C, s(c) = \bot} \bigwedge_{f \in FM \setminus \{\bot\}} \neg F_{c,f})$, but more compact representations are of course possible (particularly if TLE is given symbolically).

[3] This formula can again be encoded more compactly; particularly if the *next* function is given symbolically, which is usually the case in practice.

Example 4. Consider again the PGFDS S from Example 1. The above-described encoding of the propagations of S is

$$\begin{array}{lll}(F_{\mathrm{G},\top} & \rightarrow & (I_{\mathrm{G},\top} \vee (F_{\mathrm{E},\top} \wedge F_{\mathrm{H},\top}))) \quad \wedge \\ (F_{\mathrm{E},\top} & \rightarrow & (I_{\mathrm{E},\top} \vee F_{\mathrm{H},\top})) \quad \wedge \\ (F_{\mathrm{H},\top} & \rightarrow & (I_{\mathrm{H},\top} \vee F_{\mathrm{E},\top})).\end{array}$$

Although this encoding has a model μ such that $\mu \models \neg I_{\mathrm{G},\top} \wedge \neg I_{\mathrm{E},\top} \wedge \neg I_{\mathrm{H},\top} \wedge F_{\mathrm{G},\top} \wedge F_{\mathrm{E},\top} \wedge F_{\mathrm{H},\top}$, there is no propagation path of S in which both components E and H are initially in the state $\perp$ and switch to state $\top$ during the propagation. The problem is that the encoding allows models where a failure of E was caused by a failure of H, which was in turn caused by the same failure of E. □

In order to solve the problem, we introduce constraints imposing a *causal ordering* among the components, stating that the failure of a component can be caused only by other components that precede it in the causal order. We encode this by introducing one additional integer variable o_c for each component c, which intuitively corresponds to the time when the component c switched to a failure mode different from $\perp$, and modifying the formula (1) to take the causal ordering into account:[4]

$$\bigvee_{\substack{s:\, C \rightarrow FM \\ f \in next(c)(s)}} \bigwedge_{\substack{d \in deps(c) \\ s(d) \neq \perp}} \left(F_{d,s(d)} \wedge o_d < o_c\right). \tag{2}$$

Putting it all together, the encoding for the failure mode changes is given by the formula φ_{next} below:

$$\varphi_{next} = \bigwedge_{\substack{c \in C \\ f \in FM \setminus \{\perp\}}} (F_{c,f} \rightarrow (I_{c,f} \vee (2))) \wedge (I_{c,f} \rightarrow F_{c,f}).$$

Example 5. For the PGFDS S from Example 1, the correct encoding of the propagations of S is thus the following formula φ_{next}:

$$\begin{array}{lll}(F_{\mathrm{G},\top} & \rightarrow & (I_{\mathrm{G},\top} \vee ((F_{\mathrm{E},\top} \wedge o_{\mathrm{E}} < o_{\mathrm{G}}) \wedge (F_{\mathrm{H},\top} \wedge o_{\mathrm{H}} < o_{\mathrm{G}}))) \quad \wedge \\ (I_{\mathrm{G},\top} & \rightarrow & F_{\mathrm{G},\top}) \quad \wedge \\ (F_{\mathrm{E},\top} & \rightarrow & (I_{\mathrm{E},\top} \vee (F_{\mathrm{H},\top} \wedge o_{\mathrm{H}} < o_{\mathrm{E}}))) \quad \wedge \\ (I_{\mathrm{E},\top} & \rightarrow & F_{\mathrm{E},\top}) \quad \wedge \\ (F_{\mathrm{H},\top} & \rightarrow & (I_{\mathrm{H},\top} \vee (F_{\mathrm{E},\top} \wedge o_{\mathrm{E}} < o_{\mathrm{H}}))) \quad \wedge \\ (I_{\mathrm{H},\top} & \rightarrow & F_{\mathrm{H},\top}).\end{array}$$

Note that the constraints for causal ordering now rule out the spurious self-supporting propagation in which E fails because of H and H fails because of E.

[4] We remark that such ordering constraints are needed only if the input PGFDS is cyclic, and only between components in the same strongly connected component of the dependency graph.

This would require that $o_{\text{H}} < o_{\text{E}}$ and $o_{\text{E}} < o_{\text{H}}$ are both true, which is clearly impossible in the theory of linear integer arithmetic (or, more generally, in any theory in which $<$ is interpreted as a strict ordering relation).

The propagations of S mentioned in Example 3 correspond to the following assignments:

1. The propagation for the cut set $\{\text{G} \mapsto \top\}$ corresponds to an assignment μ such that $\mu \models I_{\text{G},\top} \wedge \neg I_{\text{E},\top} \wedge \neg I_{\text{H},\top} \wedge F_{\text{G},\top} \wedge \neg F_{\text{E},\top} \wedge \neg F_{\text{H},\top}$ and $\mu(o_{\text{G}}) = \mu(o_{\text{E}}) = \mu(o_{\text{H}}) = 0$.
2. The propagation for the cut set $\{\text{E} \mapsto \top\}$ corresponds to an assignment μ such that $\mu \models \neg I_{\text{G},\top} \wedge I_{\text{E},\top} \wedge \neg I_{\text{H},\top} \wedge F_{\text{G},\top} \wedge F_{\text{E},\top} \wedge F_{\text{H},\top}$ and $\mu(o_{\text{G}}) = 2$, $\mu(o_{\text{E}}) = 0$, $\mu(o_{\text{H}}) = 1$.
3. The propagation for the cut set $\{\text{H} \mapsto \top\}$ corresponds to an assignment μ such that $\mu \models \neg I_{\text{G},\top} \wedge \neg I_{\text{E},\top} \wedge I_{\text{H},\top} \wedge F_{\text{G},\top} \wedge F_{\text{E},\top} \wedge F_{\text{H},\top}$ and $\mu(o_{\text{G}}) = 2$, $\mu(o_{\text{E}}) = 1$, $\mu(o_{\text{H}}) = 0$.

These assignments are not unique; there are infinitely many choices for the values of the ordering variables o_c. Also note that there is no global causality ordering for the system: the causality ordering is different for different propagations. □

Finally, we encode the fault-persistence constraint by stating that no component can be in two failure modes either in the initial state of the propagation or at any time during the propagation:

$$\varphi_{once} = \bigwedge_{\substack{c \in C \\ f, f' \in FM \setminus \{\bot\} \\ f \neq f'}} (\neg I_{c,f} \vee \neg I_{c,f'}) \wedge (\neg F_{c,f} \vee \neg F_{c,f'})\,.$$

The final formula is then given by φ_{cs}:

$$\varphi_{cs} = \varphi_{TLE} \wedge \varphi_{next} \wedge \varphi_{once}.$$

As the following theorem shows, the formula φ_{cs} for general systems encodes an *overapproximation* of the set $CS(S, TLE)$. The reason for this is that the encoding does not enforce failure mode of dependencies that are working, i.e., are in the failure mode $\bot$. Note that even an overapproximation of $CS(S, TLE)$ is useful for safety analysis; it can be used, for example, for computing an upper bound on the probability of failure of the system. Moreover, if the system S is *subset-monotone*, which is often the case in practice, the formula φ_{cs} is guaranteed to encode the set $CS(S, TLE)$ exactly.

To formulate the relationship precisely, we define the function that provides the correspondence between the models of μ and the cut sets of S. Observe that thanks to φ_{once}, each model μ of φ_{cs} corresponds to a unique initial state *modelToState*(μ) of S as defined below:

$$modelToState(\mu)(c) = \begin{cases} f, & \text{if } \{f' \in FM \setminus \{\bot\} \mid \mu(I_{c,f'}) = \mathbf{true}\} = \{f\}, \\ \bot, & \text{if } \{f' \in FM \setminus \{\bot\} \mid \mu(I_{c,f'}) = \mathbf{true}\} = \emptyset. \end{cases}$$

```
MCS-enumeration($\varphi_{cs}$, $modelToState$):
1.  solver := SMT-solver()
2.  res := $\emptyset$
3.  assert-formula(solver, $\varphi_{cs}$)
4.  for $I_{c,f} \in vars(\varphi_{cs})$:
5.      add-preferred-var(solver, $I_{c,f}$, false)
6.  while check-sat(solver):
7.      $\mu$ := get-model(solver)
8.      $\psi$ := true
9.      for $I_{c,f} \in vars(\varphi_{cs})$:
10.         if $\mu(I_{c,f})$ = true:
11.             $\psi := \psi \wedge I_{c,f}$
12.     res := res $\cup$ {$modelToState(\mu)$}
13.     assert-formula(solver, $\neg\psi$)
14. return res
```

Fig. 3. SMT-based MCS enumeration algorithm.

Theorem 1. *For an arbitrary* PGFDS *S and a top level event TLE,*

$$CS(S, TLE) \quad \subseteq \quad \{modelToState(\mu) \mid \mu \models \varphi_{cs}\}.$$

Moreover, if S is subset-monotone, these sets are equal.

5 Enumeration of FDS-Minimal Cut Sets

In this section, we show how to efficiently enumerate FDS-minimal cut sets of subset-monotone systems using the formula φ_{cs} and an SMT solver. We first consider a simplified case, in which the underlying FDS D is Boolean. We then show how to generalize our solution to arbitrary FDSs.

5.1 Algorithm for Boolean FDSs

The pseudo-code of our procedure for the case when the underlying FDS is Boolean is shown in Fig. 3. Intuitively, the algorithm enumerates all the subset-minimal models of φ_{cs} with respect to the set of variables of form $I_{c,f}$. These models are enumerated one by one and each enumerated model is, together with all its supermodels, blocked by the assertion on line 13, until the formula becomes unsatisfiable. Each model of the formula is converted to a cut set by the function *modelToState*.

The algorithm makes use of a DPLL(T)-based SMT solver that provides the following functionalities:

1. An assert-formula method that allows to add constraints incrementally;
2. A check-sat method to determine the satisfiability of the current set of constraints;

3. A get-model method that returns a model for the current asserted set of constraints, in case they are satisfiable;
4. An add-preferred-var method that allows to control the branching heuristics of the internal SAT engine of the solver, such that whenever a SAT decision needs to be performed, variables in the preferred set are always considered before the other variables for branching, and are assigned the value specified in the add-preferred-var call.[5]

The correctness for our algorithm is formalized by the theorem below.

Theorem 2 (MCS enumeration over Boolean FDS). *For a subset-monotone* PGFDS *S over the Boolean* FDS*, the result of $MCS\text{-}enumeration(\varphi_{cs}, modelToState)$ is the set of all* FDS*-minimal cut sets of S.*

Proof. Let $S = (C, next)$ be a subset-monotone PGFDS. It was proven by Di Rosa et al. [15] that if branching heuristics of a CDCL-based SAT solver are modified to assign **false** to a subset V of variables before branching on other variables (lines 4–5 of our pseudocode), the produced model is subset-minimal with respect to the set of variables V. This claim straightforwardly extends to DPLL(T)-based SMT solvers. In every iteration, the algorithm thus finds one subset-minimal model μ of φ_{cs} with respect to the set of variables $I_{c,f}$ and adds a constraint that prevents enumerating any model μ' such that $\{I_{c,f} \in vars(\varphi_{cs}) \mid \mu(I_{c,f}) = \mathbf{true}\} \subseteq \{I_{c,f} \in vars(\varphi_{cs}) \mid \mu'(I_{c,f}) = \mathbf{true}\}$ in the following iterations. Therefore, the described algorithm enumerates, for each model $\overline{\mu}$ of the formula $\exists\{F_{c,f} \mid c \in C, f \in FM\}\,\exists\{o_c \mid c \in C\}\,(\varphi_{cs})$ that is subset-minimal with respect to the set of variables $I_{c,f}$, exactly one model μ of φ_{cs} that agrees with $\overline{\mu}$ on all variables $I_{c,f}$.

Note that $vars(\varphi_{cs})$ does not contain the variable $I_{c,\perp}$ for any $c \in C$. For a Boolean FDS and models $\mu, \mu' \models \varphi_{cs}$, we thus have $\{I_{c,f} \in vars(\varphi_{cs}) \mid \mu(I_{c,f}) = \mathbf{true}\} \subseteq \{I_{c,f} \in vars(\varphi_{cs}) \mid \mu'(I_{c,f}) = \mathbf{true}\}$ if and only if $modelToState(\mu) \leq modelToState(\mu')$. Therefore, Theorem 1 implies that for subset-monotone S, subset-minimal models of φ_{cs} with respect to the set of variables of form $I_{c,f}$ precisely correspond to FDS-minimal cut sets of S and the correspondence is given by the function $modelToState$. □

5.2 Extension to Arbitrary FDSs

The algorithm of Fig. 3 does not work in general for arbitrary FDSs, but only for the FDSs in which all the failure modes different from $\perp$ are incomparable. The problem is that the assumption that a cut set is FDS-minimal iff the corresponding model of φ_{cs} is subset-minimal with respect to the set of variables $I_{c,f}$ with $f \neq \perp$ does not hold in general with the encoding of Sect. 4, as can be seen on the following simple example.

[5] For example, calling add-preferred-var(solver, v, **true**) means that if the solver has to perform a case split, v will be assigned before all non-preferred variables, and it will always be assigned to true by the branching heuristic.

$$\begin{tikzcd}
 & & \{w, \mathit{fd}, \mathit{fu}\} \equiv w \land \neg \mathit{fs} \land \mathit{fd} \land \mathit{fu} \arrow[d, dash] \\
\{w, \mathit{fs}\} \equiv w \land \mathit{fs} \land \neg \mathit{fd} \land \neg \mathit{fu} \arrow[dr, dash] & & \{w, \mathit{fd}\} \equiv w \land \neg \mathit{fs} \land \mathit{fd} \land \neg \mathit{fu} \arrow[dl, dash] \\
 & \{w\} \equiv w \land \neg \mathit{fs} \land \neg \mathit{fd} \land \neg \mathit{fu} &
\end{tikzcd}$$

Fig. 4. Hasse diagram of the ordered set $(W3F\downarrow, \subseteq)$ together with the encoding of the elements as formulas.

Example 6. Consider the FDS $D = (\{\bot, m, \top\}, \bot \le m \le \top, \bot)$ and the PGFDS $S = (\{c\}, \mathit{next})$ with $\mathit{next}(c)(s) = \emptyset$ for all c and s. Intuitively, S contains one component that cannot change its failure mode during the computation. Consider further the top-level event $\mathit{TLE} = \{\{c \mapsto m\}, \{c \mapsto \top\}\}$.

Both $\{c \mapsto \top\}$ and $\{c \mapsto m\}$ are cut sets, but only the latter is FDS-minimal. However, the algorithm of Fig. 3 will return both, since they both correspond to subset-minimal models with respect to the set of variables $I_{c,f}$. □

We can adapt the procedure of Fig. 3 to arbitrary FDSs by using an encoding in which the ordering of assignments to the $I_{c,f}$ variables corresponds to the severity ordering $\le$ of the underlying FDS D. In order to do this, we exploit the isomorphism between $D = (\mathit{FM}, \le, \bot)$ and the poset $D\downarrow$ of its lower subsets generated by single elements defined as $D\downarrow = \{\{f' \in \mathit{FM} \mid f' \le f\} \mid f \in \mathit{FM}\}$ with partial order $\subseteq$ and the least element $\{\bot\}$. For example, the poset $(W3F\downarrow, \subseteq)$ for the FDS $W3F$ of Fig. 1 is shown in Fig. 4, together with an encoding of the elements as formulas.

With this isomorphism in mind, we define for each $c \in C$ and $f \in \mathit{FM}$ the formula $\psi_{c=f}$ that represents the failure mode f of component c by assigning the subset of variables $\{I_{c,\hat{f}} \mid \hat{f} \le f\}$ to true:

$$\psi_{c=f} \;=\; \bigwedge_{\hat{f} \in \mathit{FM}, \hat{f} \le f} I_{c,\hat{f}} \;\land \bigwedge_{\hat{f} \in \mathit{FM}, \hat{f} \not\le f} \neg I_{c,\hat{f}}.$$

The important property of this definition is that for all $c \in C$, $f, f' \in \mathit{FM}$ and assignments $\mu \models \psi_{c=f}$ and $\mu' \models \psi_{c=f'}$, we have $f \le f'$ if and only if $\{I_{c,\hat{f}} \mid \mu(I_{c,\hat{f}}) = \mathbf{true}\} \subseteq \{I_{c,\hat{f}} \mid \mu'(I_{c,\hat{f}}) = \mathbf{true}\}$.

We then modify the encoding φ_{cs} of Sect. 4 as follows:

1. First, we modify φ_{next} to encode the initial state by using $\psi_{c=f}$ instead of $I_{c,f}$. This ensures that the ordering of assignments to the initial variables reflects the ordering given by the underlying FDS. We also remove the mutual exclusion constraints on the variables $I_{c,f}$ from φ_{once}, because the mutual exclusion of initial failure modes is now guaranteed by the definition of $\psi_{c=f}$:

$$\varphi_{next} = \bigwedge_{\substack{c \in C \\ f \in FM \setminus \{\bot\}}} (F_{c,f} \rightarrow (\psi_{c=f} \vee (2))) \wedge (\psi_{c=f} \rightarrow F_{c,f}),$$

$$\varphi_{once} = \bigwedge_{\substack{c \in C \\ f, f' \in FM \setminus \{\bot\} \\ f \neq f'}} (\neg F_{c,f} \vee \neg F_{c,f'}) \,.$$

2. Then, we add domain constraints that ensure that the resulting formula represents only models with assignments to $I_{c,f}$ that correspond to elements of $D\!\downarrow$:

$$\varphi_{D\downarrow} = \bigwedge_{c \in C} \bigvee_{f \in FM} \psi_{c=f}.$$

The new encoding is then given by φ_{cs}^{FM}:

$$\varphi_{cs}^{FM} = \varphi_{TLE} \wedge \varphi_{next} \wedge \varphi_{once} \wedge \varphi_{D\downarrow}.$$

The modified encoding φ_{cs}^{FM} represents the cut sets in a different way: instead of representing the failure modes directly by $I_{c,f}$ as in φ_{cs}, they are now represented by the subformulas $\psi_{c=f}$. Therefore, to prove correctness of the modified encoding, the function *modelToState* that maps models to cut sets also has to be changed. We define the initial state $\mathit{modelToState}^{FM}(\mu)$ corresponding to the model μ by $\mathit{modelToState}^{FM}(\mu)(c) = \max\{f \in FM \mid \mu(I_{c,f}) = \mathbf{true}\}$. Note that the maximum is guaranteed to exist because of the $\varphi_{D\downarrow}$ constraint.

Theorem 3. *For an arbitrary* PGFDS *S and a top level event TLE,*

$$\mathit{CS}(S, \mathit{TLE}) \quad \subseteq \quad \{\mathit{modelToState}^{FM}(\mu) \mid \mu \models \varphi_{cs}^{FM}\}.$$

Moreover, if S is subset-monotone, these sets are equal.

Therefore, the algorithm MCS-enumeration from Fig. 3 can be used to enumerate FDS-minimal cut sets of a subset-monotone PGFDS, given as the inputs the modified encoding φ_{cs}^{FM} and the modified function $\mathit{modelToState}^{FM}$. This is formalized by the following theorem:

Theorem 4 (MCS enumeration for general FDS). *For a subset-monotone* PGFDS *S over an* FDS *D, the result of MCS-enumeration(φ_{cs}^{FM}, $\mathit{modelToState}^{FM}$) is the set of all* FDS*-minimal cut sets of S.*

Note that our encoding of FDS-minimality is general and does not depend on the algorithm for enumeration of subset-minimal models. Indeed, thanks to our encoding, any off-the-shelf minimal-model enumerator can be used to enumerate FDS-minimal models. Therefore, any improvements to minimal model enumeration directly translate to improved performance of our method for FDS-minimal cut set enumeration. From the opposite point of view, our encoding can in principle be employed by other tools to reduce FDS-minimal cut set enumeration to subset-minimal cut set enumeration.

6 Related Work

Finite Degradation Models (FDMs) [14] are an algebraic framework accommodating the concept of fault degradation, where faults may have different values organized into a semi-lattice. Using FDMs (probabilistic) safety analysis (fault trees and minimal cut sets) can be generalized from Boolean models to multi-state systems. Compared to FDMs, fault-persistent PGFDSs differ in two significant aspects: first, since the function *next* returns a set of possible next failure modes, PGFDSs allow non-determinism in the failure propagation, i.e., the failure of a component is not *uniquely* determined by the failure modes of its dependencies. Second, and more importantly, PGFDSs allow cyclic dependencies and give them well-defined and expected semantics. Since the work on FDMs is the closest to ours, we shall discuss it in detail below.

In [8] the authors present a framework for failure propagation which enables modeling sets of failure modes using a domain specific language. It is less expressive than FDMs, in that sets of failure modes cannot be related by degradation orders, which significantly simplifies the enumeration of MCSs. Finally, classical formalisms for failure propagation, but less expressive than FDS, include FPTN [9] and Hip-HOps [11].

TFPGs (Timed Failure Propagation Graphs) [1] extend fault propagation model by enabling the specification of time bounds and mode constraints on the propagation links. However, TFPGs do not consider degradation, and they do not support cyclic dependencies. Conversely, the PGFDS formalism can be easily extended to support time bounds, failure probabilities, mode constraints, and constraints on propagation delays similar to those available in TFPGs (e.g., following [5]). Moreover, once the minimal cut sets of a PGFDS are computed, the existing approach to computing probability of overall failure [5] can be used almost unchanged.

Finally, xSAP [3] is a safety analysis platform that supports library-based fault models and the generation of safety artifacts for fully general behavioral models, e.g., it can generate fault trees and minimal cut sets for arbitrary transition systems [6]. Currently, xSAP does not support FDS and degradation models.

6.1 Detailed Comparison with Finite Degradation Models

As outlined above, the formalism Finite Degradation Models (FDMs), introduced in [14], is closely related to our PGFDS. Here, we describe FDM in further detail and show that PGFDS are a strict generalization of FDS, obtained by (i) considering non-determinism in the propagation of failures, and (ii) by allowing cyclic dependencies among the components.

Each FDM has *state variables*, which correspond to the sources of failures in the system, and *flow variables*, which correspond to the propagated consequences of these failures. Each flow variable has an associated *equation*, which prescribes the failure mode of the corresponding flow variable based on the failure modes of state variables and other flow variables. We assume that the failure modes of all state and flow variables are modeled by the FDS $D = (FM, \leq, \bot)$.[6]

Definition 9 (Finite Degradation Model [14]**).** *Given an arbitrary* FDS $D = (FM, \leq, \bot)$*, a Finite Degradation Model (*FDM*) is a pair* $M = (\mathcal{V} = \mathcal{S} \uplus \mathcal{F}, \mathcal{E})$*, where*

- $\mathcal{S} = \{V_1, \dots, V_m\}$ is a finite set of *state variables*,
- $\mathcal{F} = \{W_{m+1}, \dots, W_{m+n}\}$ is a finite set of *flow variables*,
- $\mathcal{E} = \{W_{m+1} := \phi_{m+1}, \dots W_{m+n} := \phi_{m+n}\}$ is a finite set of *equations*, where each ϕ_{m+i} for $1 \leq i \leq n$ is a function of type $FM^{\mathcal{V}} \to FM$.

We say that a flow variable W_{m+i} *depends on* a variable v if the function ϕ_{m+i} depends on v. An FDM is called acyclic if there are no cyclic dependencies among its flow variables, i.e., no flow variable transitively depends on itself. We stress out that in contrast to our definitions of PGFDS, the original paper [14] only deals with acyclic FDMs and does not provide semantics and necessary definitions for cyclic FDMs. We thus assume in the rest of the section that all FDMs are acyclic.

An assignment $\sigma \colon \mathcal{V} \to FM$ is called *admissible* if the failure modes assigned to the flow variables satisfy all the corresponding equations, i.e., $\sigma(W_{m+i}) = \phi_{m+i}(\sigma)$ for each $1 \leq i \leq n$. The assumption of acyclicity of FDMs, together with the fact that all equations are deterministic functions and not general relations, guarantees that in each admissible assignment, failure modes of the flow variables are uniquely determined by the failure modes of the state variables. This defines a function $[\![M]\!](\sigma) = \overline{\sigma}_M$, which maps each state variable assignment σ to its unique admissible extension $\overline{\sigma}_M$ that assigns values to all variables. This is a stark contrast to PGFDS, where a single initial state can give rise to multiple different propagation paths.

A corresponding notion to our notion of *top level event* for FDM is the notion of *observer*. An observer is a pair (R, U), where R is a flow variable and $U \subseteq FM$ is a set of failure modes. Intuitively, the observer represents a set of dangerous failure modes of the given flow variable. A *cut set* is any assignment $\sigma \colon \mathcal{S} \to FM$ of failure modes to state variables such that $\overline{\sigma}(R) \in U$.

A notion related to our notion of *monotonicity* for FDM is *coherence*. The observer is coherent if for all assignments $\sigma, \sigma' \colon \mathcal{S} \to FM$ such that σ is a cut set and $\sigma \leq \sigma'$, the assignment σ' is also a cut set.

Each FDM M can be translated to a PGFDS S_M such that the cut sets of M correspond to the cut sets of S_M. Moreover, if the FDM M is coherent, the resulting PGFDS S_M is guaranteed to be subset-monotone. This enables efficient analysis of coherent FDMs by our SMT-based technique. Intuitively, the PGFDS S_M has one component for each state variable of M and an additional component R for the observer flow variable R. The *next* function is defined in a way that the failure modes of all the components that correspond to state variables cannot

[6] Both FDMs and our PGFDS can be defined over multiple different FDSs for different variables. Such generalization is straightforward, but it complicates the notation and the exposition significantly.

Table 1. Classes of PGFDS and their traces that each of the compared tools can handle precisely.

Tool	FDS	Cycles	Nondeterministic	Not fault-persistent
Emmy	**Arbitrary**	No	No	No
xSAP	Boolean	**Yes**	**Yes**	**Yes**
SMT-PGFPS	**Arbitrary**	**Yes**	**Yes**	No

change and that the component R can switch to a predefined set of failure modes if $\overline{\sigma}(R) \in U$. This is achieved by composing all equations for the flow variables. If local variables are used in the symbolic encoding[7], the size of the result is guaranteed to be polynomial.

7 Experimental Evaluation

To evaluate the performance and scalability of our approach, we have implemented the proposed algorithm MCS-enumeration in a simple Python tool that uses the solver MathSAT [7], which supports all the required functionalities that are described in Sect. 5.1. In this section, we refer to the tool as SMT-PGFPS.

As a comparison, we have used Emmy [13], a tool based on decision diagrams for the enumeration of FDS-minimal cut sets of FDMS, and xSAP [3], a tool for safety assessment for arbitrary transition systems. Each of these tools only supports a subset of the capabilities of our approach, as summarized in Table 1.

Emmy supports minimal cut set enumeration with respect to an arbitrary ordering of failure modes given by an FDS, but only for acyclic and deterministic FDMS;

xSAP supports analysis of arbitrary transition systems with cycles, given that it internally relies on the nuXmv model checker. However, it cannot enumerate FDS-minimal cut sets, but only subset-minimal ones. Note that for computation of subset-minimal cut sets, xSAP is more general than our approach, as it supports general transition systems and arbitrary temporal properties. However, we use xSAP as a baseline to compare performance and scalability of our approach for cyclic PGFDS because it is a subcase of general transition systems that is important in practice. In the evaluation, we use the IC3-based engine described in [6] (denoted as xSAP-IC3). Note that this algorithm assumes that the verified property is monotone and leverages this assumption for efficiency.

[7] For example, let-expressions of form `(let ((var definition) ...) body)` in SMT-LIB.

For the comparison, we have created three sets of benchmarks:

Scalable acyclic benchmarks consisting of linear structures extended by a triple modular redundancy scheme. The basic architecture of these structures is parameterized by its size n and the system contains $6n$ components: $3n$ modules and $3n$ voters. These benchmarks use the FDS W2F, which is a restriction of the FDS W3F of Fig. 1 to failure modes $\{w, fd, fu\}$, with the ordering $w < fd < fu$.
Note that FDS-minimal cut sets of these benchmarks cannot be enumerated by xSAP, as the benchmarks use a non-Boolean FDS.

Randomly generated systems with cycles over Boolean FDS which share some structural properties with real-world systems. In particular, we generated random systems that have a similar distribution of in-degrees and out-degrees of the components as our proprietary systems, which we cannot disclose. We have generated 950 such systems of sizes ranging between 50 and 1000 components. We have used the Boolean FDS for these benchmarks, so that they can be precisely analyzed also by xSAP.
Note that these benchmarks cannot be solved by Emmy, as they contain cyclic dependencies among the components.

Randomly generated systems over W2F which are created from the above-mentioned randomly generated systems by using the FDS W2F instead of the Boolean one. Although this does not change the overall structure of the system, it makes the transition relation more complicated and significantly increases number of minimal cut sets.
In the evaluation, we only used systems of size at most 400, as both the compared approaches timed out on the vast majority of larger systems.
Note that these benchmarks cannot be solved by Emmy, as they contain cyclic dependencies among the components. They can be solved by xSAP, but the generated cut sets are only subset-minimal with respect to fault variables, and not (in general) FDS-minimal.

For the scalable benchmarks, we have generated encodings in the SMT format described in this paper and in the FDS-ML format used by Emmy. For the randomly generated cyclic benchmarks, we have generated encodings in the SMT format and in the SMV format used by xSAP. The SMV encodings also include the assumption of *fault-persistence*. All the used benchmarks are *subset-monotone*, and therefore our SMT-based approach can be used to compute the set of minimal cut sets correctly.

We have used wall time limit of 30 min for each solver-benchmark pair. All experiments were performed on a Linux laptop with Intel Core i7-8665U CPU and 32 GiB of RAM.

A comparison of SMT-PGFPS and Emmy on the scalable acyclic benchmarks can be seen in Table 2. It shows that Emmy times out already on systems of size 5, i.e., on systems with 30 components. On the other hand, our approach is able to scale to systems with three thousand components.

A comparison against the sequential approach of xSAP on cyclic benchmarks can be seen in Fig. 5. Figures 5a and 5b show that on random systems over

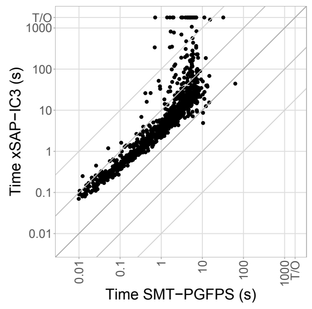
Table 2. Numbers of minimal cut sets (*#MCS*) and solving times for top level events *failed detected (fd)* and *failed undetected (fu)* on linear systems extended with triple-modular redundancy scheme. Note that the system with size n consists of $6n$ components (column *#Comp*).

		Failed detected			Failed undetected		
Size	#Comp	#MCS	Emmy (s)	SMT (s)	#MCS	Emmy (s)	SMT (s)
1	6	4	0.051	0.001	7	0.071	0.001
2	12	16	0.137	0.002	31	0.172	0.003
3	18	28	3.052	0.004	55	3.141	0.007
4	24	40	160.493	0.006	79	163.556	0.013
5	30	52	>1800	0.009	103	>1800	0.017
10	60	112	>1800	0.032	223	>1800	0.063
100	600	1192	>1800	3.456	2383	>1800	6.748
500	3000	5992	>1800	171.042	11983	>1800	328.737

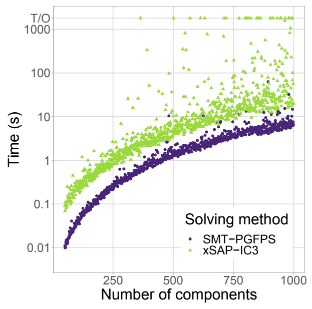
Boolean FDSs, our approach significantly outperforms the sequential approach of xSAP. As the size of the system grows, the difference can be up to several orders of magnitude. Both xSAP and SMT-PGFPS compute exactly the same minimal cut sets. Hence, the dramatic difference in performance can be justified by the reduction to the combinational case, which prevents the unrolling of the transition relation by implicitly encoding the propagations in the total ordering(s) found by the SMT solver.

The performance difference on the systems over the FDS W2F, shown in Figures 5c and 5d, is even more pronounced. This can be caused by two additional factors. First, the systems over the FDS W2F have more complicated transition relation, more minimal cut sets, and are in general harder. Thus, the unrolling performed by xSAP is even more costly. Second, xSAP has to enumerate more cut sets, because it is enumerating all subset-minimal cut sets and not only FDS-minimal cut sets. However, this cannot be the main source of the observed performance gap: on 35 from the 113 benchmarks on which both xSAP and SMT-PGFPS finished before timeout, the number of cut sets are the same; on the remaining 78 benchmarks, xSAP enumerates on average 6% more cut sets and at most 62% more cut sets. In order to obtain FDS-minimal cut sets from xSAP, the produced subset-minimal cut sets would have to be filtered or explicitly minimized, which would add yet another performance penalty for xSAP.

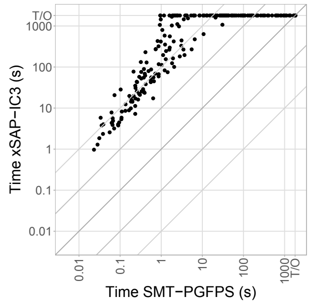
Overall, the SMT-based techniques presented in this paper yield a fundamental advancement with respect to the state of the art, both in terms of expressiveness as well as in terms of performance.



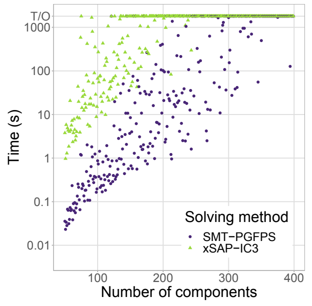
(a) Scatter plot of solving times over Boolean FDS.



(b) Dependence of solving time on the number of components over Boolean FDS.



(c) Scatter plot of solving times over FDS W2F.



(d) Dependence of solving time on the number of components over FDS W2F.

Fig. 5. Comparison of SMT-PGFPS and xSAP-IC3 on random cyclic systems.

8 Conclusions and Further Work

We tackled the problem of supporting the Preliminary Safety Assessment phase of aircraft design. Specifically, we defined an expressive framework for modeling failure propagation over components with multiple levels of degradation, with nondeterminism and cyclic dependencies. We presented a sequential semantics and proved that the problem can be tackled by means of minimal models enumeration in SMT. The framework is more expressive than the state of the art, and the proposed method outperforms the BDD-based techniques from [14] on acyclic benchmarks over generic FDSs, and the model checking techniques of [6] on cyclic benchmarks.

In the future, we are going to introduce timing constraints and analyze redundancy architectures. We also investigate ways to relax the monotonicity and fault-persistence assumptions to explore recovery mechanisms and to further extend the reach of our approach. We are also working on encoding the causality constraints in the frameworks of SAT modulo acyclicity [10] and ASP modulo acyclicity [4], which could improve the performance of our approach even further.

References

1. Abdelwahed, S., Karsai, G., Mahadevan, N., Ofsthun, S.C.: Practical implementation of diagnosis systems using timed failure propagation graph models. IEEE Trans. Instrum. Meas. **58**(2), 240–247 (2009)
2. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, volume 185 of Frontiers in Artificial Intelligence and Applications, pp. 825–885. IOS Press (2009)
3. Bittner, B., et al.: The xSAP safety analysis platform. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 533–539. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_31
4. Bomanson, J., Gebser, M., Janhunen, T., Kaufmann, B., Schaub, T.: Answer set programming modulo acyclicity. Fundamenta Informaticae **147**(1), 63–91 (2016)
5. Bozzano, M., Cimatti, A., Gario, M., Micheli, A.: SMT-based validation of timed failure propagation graphs. In: Bonet, B., Koenig, S. (eds.) Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, Austin, Texas, USA, 25–30 January 2015, pp. 3724–3730. AAAI Press (2015)
6. Bozzano, M., Cimatti, A., Griggio, A., Mattarei, C.: Efficient anytime techniques for model-based safety analysis. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 603–621. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_41
7. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 93–107. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_7
8. Delmas, K., Delmas, R., Pagetti, C.: SMT-based architecture modelling for safety assessment. In: 12th IEEE International Symposium on Industrial Embedded Systems, SIES 2017, Toulouse, France, 14–16 June 2017, pp. 1–8. IEEE (2017)
9. Fenelon, P., McDermid, J.A.: An integrated tool set for software safety analysis. J. Syst. Softw. **21**(3), 279–290 (1993)
10. Gebser, M., Janhunen, T., Rintanen, J.: SAT modulo graphs: acyclicity. In: Fermé, E., Leite, J. (eds.) JELIA 2014. LNCS (LNAI), vol. 8761, pp. 137–151. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11558-0_10
11. Papadopoulos, Y., McDermid, J.A.: Hierarchically performed hazard origin and propagation studies. In: Felici, M., Kanoun, K. (eds.) SAFECOMP 1999. LNCS, vol. 1698, pp. 139–152. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48249-0_13
12. Rauzy, A.: Mathematical foundations of minimal cutsets. IEEE Trans. Reliab. **50**(4), 389–396 (2001)
13. Rauzy, A., Yang, L.: Decision diagram algorithms to extract minimal cutsets of finite degradation models. Information **10**(12), 368 (2019)
14. Rauzy, A., Yang, L.: Finite degradation structures. FLAP **6**(6), 1447–1474 (2019)
15. Di Rosa, E., Giunchiglia, E., Maratea, M.: Solving satisfiability problems with preferences. Constraints Int. J. **15**(4), 485–515 (2010). https://doi.org/10.1007/s10601-010-9095-y
16. SAE: ARP4761 Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment, December 1996
17. SAE: ARP4754A Guidelines for Development of Civil Aircraft and Systems, December 2010

18. Wang, P.: Civil Aircraft Electrical Power System Safety Assessment: Issues and Practices. Butterworth-Heinemann, Oxford (2017)
19. Yang, L., Rauzy, A., Haskins, C.: Finite degradation structures: a formal framework to support the interface between MBSE and MBSA. In: IEEE International Systems Engineering Symposium (ISSE), Rome, Italy, pp. 1–6 (2018)

ddSMT 2.0: Better Delta Debugging for the SMT-LIBv2 Language and Friends

Gereon Kremer, Aina Niemetz(✉), and Mathias Preiner

Stanford University, Stanford, USA
{gkremer,niemetz,preiner}@cs.stanford.edu

Abstract. Erroneous behavior of verification back ends such as SMT solvers require effective and efficient techniques to identify, locate and fix failures of any kind. Manual analysis of large real-world inputs usually becomes infeasible due to the complex nature of these tools. Delta Debugging has emerged as a valuable technique to automatically reduce failure-inducing inputs while preserving the original erroneous behavior. We present ddSMT 2.0, the successor of the delta debugger ddSMT. ddSMT is the current de-facto standard delta debugger for the SMT-LIBv2 language. Our tool improves and extends core concepts of ddSMT and extends input language support to the entire family of SMT-LIBv2 language dialects. In addition to its *ddmin*-based main minimization strategy, it implements an alternative, orthogonal strategy based on hierarchical input minimization. We combine both strategies into a hybrid strategy and show that ddSMT 2.0 significantly improves over ddSMT and other delta debugging tools for SMT-LIBv2 on real-world examples.

1 Introduction

In recent years, a growing number of formal methods applications (e.g., [6,8]) rely on Satisfiability Modulo Theories (SMT) solvers as the back end. Current state-of-the-art SMT solvers are typically complex pieces of software, and debugging erroneous behavior requires effective and efficient techniques to analyze failure-inducing input with the purpose of identifying and locating the cause of the failure. Manual analysis of real-world problems that trigger a particular unwanted behavior is very often infeasible for large inputs, mainly due to the complex nature of these tools.

Erroneous behavior is never only triggered by a single unique input, but by a class of inputs that share a common trait. Extracting a *minimal working example*, i.e., an input that is *as small as possible* but still triggers the original faulty behavior, from such a class of inputs usually significantly decreases the time to identify and locate the cause of the failure. While ideally, the notion of size of an input directly correlates to the effort required to determine the failure

This work was supported in part by DARPA (award no. FA8650-18-2-7861) and ONR (award no. N68335-17-C-0558).

cause, in practice this is hard to quantify. We instead use metrics such as file size, number of language constructs, and solver runtime until the failure occurs.

Finding such minimal working examples, however, is a problem of its own. Manual minimization is typically infeasible in practice, simply due to the large number of possible simplifications that may even depend on each other. Delta debugging techniques, on the other hand, provide automated means to minimize failure-inducing inputs. This typically entails to first read some input, apply a set of rules to simplify the input, and then check that the modified input still triggers the original behavior. Delta debugging in its simplest form [24] extracts a minimal working example by omitting parts of the input that are irrelevant for triggering the original faulty behavior. More input language specific tools perform additional simplifications to further minimize the input. All of these simplifications are typically performed until a fixed point is reached.

For the design of a delta debugger, this process raises a number of questions: How does the debugging tool check for "same behavior" of a tool on some input? Which simplification rules should be employed and how should they be combined? To what (syntactic and semantic) degree should the delta debugger itself understand the input language? In this paper, we address these questions in the context of delta debugging for the SMT-LIBv2 language and its dialects with our delta debugger ddSMT 2.0, the successor of ddSMT [18]. In the following, we will refer to ddSMT 2.0 as ddSMTv2, and to its predecessor as ddSMTv1.

Related Work. Generic delta debugging tools that are agnostic to the input language can be surprisingly efficient for some use cases. For minimizing SMT-LIB input, however, their usefulness is usually rather modest. One such generic tool is linedd [4], which solely performs line-based simplifications. The first delta debugging tool specific to the SMT-LIB language was presented in [7] as deltaSMT and targeted SMT-LIBv1 [22]. Three years later, the SMT community adopted a new input language SMT-LIBv2. In 2013, an updated version of deltaSMT [10] extended the tool syntactically for SMT-LIBv2 compliance, but limited to the feature set of the SMT-LIBv1 language and without full SMT-LIBv2 support. Note that this updated version is not available anymore. In the same year, ddsexpr [5], a generic hierarchical delta debugger for S-expressions (and thus applicable to the SMT-LIB language family), and ddSMTv1 [18], a delta debugger specific to the SMT-LIBv2 language, were presented. The latter implements a variant of Zeller's *ddmin* algorithm [24] and is considered as the current de-facto standard delta debugger in the SMT community. The only other delta debugging tool specific to the SMT-LIBv2 language we are aware of is delta [15], a hierarchical delta debugger shipped together with the SMT solver SMT-RAT [9]. A reimplementation of delta in Python is available as pyDelta at [14].

Contributions. In this paper, we present ddSMTv2, a delta debugging tool for the SMT-LIBv2 [2] language and its dialects. It supports the entirety of the SMT-LIBv2 standard as well as non-standardized extensions and derived formats such as the SyGuS input language [21]. Our tool is agnostic to future extensions of the standard in the sense that it does not require any modifications for basic support. It is easy to extend, and extensions will only be required for simplifications that are specific to new language features or a certain dialect of the SMT-LIBv2 language. In this sense it will also immediately support the SMT-LIBv3 [1] language, which is currently under development.

ddSMTv2 is the successor of the delta debugger ddSMTv1 [18] and incorporates, improves and extends its core concepts. It also implements an improved variant of the hierarchical approach of pyDelta as an alternative, orthogonal strategy, and allows to combine these two strategies in a hybrid manner. ddSMTv2 is intended to overcome major weaknesses of ddSMTv1, which is limited to the SMT-LIBv2 language and does not support the full set of standardized background theories or language extensions to the point where it is even unable to parse the input file. ddSMTv2 further extends the set of theory-specific simplifications over both ddSMTv1 and pyDelta, which allows to exploit even more minimization opportunities.

ddSMTv2 is implemented in Python and can be installed via `pip3 install ddsmt`. Its documentation is available at [11], and its source code is available under version 3 of the GNU General Public License (GPLv3) at [13].

2 Detecting Failure-Inducing Inputs

An SMT solver is a fully automated tool to determine the satisfiability of a first order logic formula modulo some background theories and their combinations. For satisfiable inputs, SMT solvers optionally allow to query a model, whereas for unsatisfiable inputs, some optionally generate a proof of unsatisfiability. Additionally, SMT solvers usually provide a plethora of configuration options.

Within the SMT community, the notion of *failure* is generally defined as anything from abnormal termination or crashes (including segmentation faults and assertion failures), to performance regressions (one solver performs significantly worse on an input than a reference solver), unsoundness (answering sat instead of unsat and vice versa), incorrect models or incorrect proofs of unsatisfiability. In the following, we define a *failure-inducing input* to an SMT solver as an SMT-LIB input that triggers a failure. In particular, we do not consider options configured via command line as part of the input.

Strategies to determine if a minimized input still triggers the original faulty behavior typically differ depending on the kind of the failure. For *abnormal termination or crashes*, it is usually sufficient to compare the exit code of the solver call, optionally with additional comparisons of output on the standard output and error channels. For failures that generate error messages that include memory addresses, it is often useful to not compare the full output, but to only match against a specific phrase that occurs in the original error output.

By default, ddSMTv2 does exactly that: it determines if a simplified input has the same erroneous behavior as the original input by comparing the exit code and the output on the standard output and error channels for equality.

Standard output and error output can optionally be ignored or matched against user-defined strings via command line options.

Performance Regressions are more tricky and typically involve helper scripts that call two solver configurations with some time limit and return a specific exit code in case the performance regression is triggered. The delta debugger will then minimize the input based on this exit code. Inputs that trigger *unsoundness failures* can be dealt with in a similar way. For inputs that reveal performance regressions and unsound answers, ddSMTv2 provides easy-to-use wrapper scripts that can also be adapted to more specific use cases.

Incorrect models and *incorrect proofs* are more involved since they typically require some checking mechanism to determine if a generated model or proof is incorrect. Most SMT solvers implement such mechanisms and will throw an assertion failure in debug mode when such a failure is detected. For cases that are not detected by the solver itself, external checking tools are required. Implementing such checks is considered out of scope for a debugging tool due to their complex nature.

3 Simplification Rules and Staged Simplification

Historically, the set of simplification rules for delta debugging has been in general rather small and mainly limited to removing or reordering parts of the input. Adding *structural and semantic simplifications* on top of these basic transformations has proved successful for the SMT-LIB language, and greatly improves performance over language agnostic minimization techniques. The delta debuggers deltaSMT, delta and ddSMTv1 all support structural and semantic simplifications, albeit to a varying degree. Of these three, ddSMTv1 implements the largest set of language-specific simplifications. The SMT-LIB-agnostic delta debugger ddsexpr, on the other hand, performs structural simplifications only.

Additionally, it is beneficial to devise a strategy for *when* to apply *which kind* of simplification rules to *which part* of the input in order to avoid generating useless test cases. An example for a useless test case is when the declaration of a constant is removed before removing all occurrences of this constant. Such a test case is useless because it is almost guaranteed to fail due to a parse error in the solver instead of triggering the original faulty behavior. It is further beneficial to perform simplifications that promise larger overall reduction (e.g., removal of commands) early on, in order to reduce the burden of more local, theory-specific simplifications (e.g., replacing terms with default values of the same sort).

We require that applying a simplification rule indeed *simplifies* the input and that it is not possible to cycle between applications of simplification rules in order to ensure termination of the minimization procedure. Generally, we define *simplification* in terms of measuring the input size in bytes or in the number of S-expressions. We supplement this with specific syntactic and semantic properties, e.g., the number of variable binders in a quantified formula, or the degree of "sortedness" of children of an S-expression. Intuitively, we say that given an input $\mathcal{A}$, a simplification rule yields a simpler input $\mathcal{B}$ if the constructs in $\mathcal{B}$ are simpler according to some metric specific to the rule, or if $\mathcal{B}$ is smaller than $\mathcal{A}$ in terms of size. As an example for such a metric, consider a simplification rule that replaces a value with another value. Such a transformation is only interpreted as simpler if the value to be replaced does not already fall into the class of simpler values, e.g., for integer values we define the set of simpler values as $\{0, 1\}$. Thus, replacing value 1234 with 0 is a simplification, but replacing 1 with 0 is not.

In ddSMTv2, possible input simplifications are generated by so-called *mutators*, which implement simplification rules. They either perform small local changes to a given S-expression, or introduce global modifications on the input based on that S-expression. Each mutator implements a *filter* method, which checks if the mutator is applicable to the given S-expression. If this is the case, the mutator can be queried to suggest (a list of) possible local and global simplifications. Mutators are not required to be equivalence or satisfiability preserving. They may extract semantic information from the input when needed, e.g., to infer the sort of a term, to query the set of declared or defined symbols, to extract indices of indexed operators, and more. ddSMTv2 applies a considerably larger set of simplifications than ddSMTv1 and currently implements 48 mutators, which range from generic simplifications on S-expressions that require no understanding of SMT-LIB, to more theory-specific mutators that make full use of SMT-LIB semantics. Each of these mutators is enabled by default and can optionally be disabled. Extending ddSMTv2 with a new simplification boils down to implementing a filter method and methods to query local and/or global mutations in a new mutator class, and registering this class as an active mutator.

4 Parsing and Input Representation

While the question about the syntactic and semantic degree of understanding of the input language may seem silly at first glance, it is indeed warranted and actually crucial for the overall design of the delta debugger. The two extreme cases are aiming at *full understanding* of the language, and *no understanding*, i.e., treating the input as a sequence of bytes. The trade-off at hand is mainly between the ability to easily devise *language compliant* simplifications, and the burden of infrastructure required for *parsing* and *representing* the input, which is an additional burden on *maintenance* in case the input language changes.

Both deltaSMT and ddSMTv1 aim at full understanding, while most of the others try for some intermediate level of abstraction, i.e., a level that does not require full understanding of the input language but allows for smarter simplifications than just manipulating bytes. The line-based delta debugger linedd minimizes input by removing lines, whereas ddsexpr is syntax-aware in the sense that it understands S-expressions, but without any SMT-LIBv2 specific semantics. Both delta and pyDelta extend understanding of S-expressions with some semantic properties, however, in the case of delta only to a very basic degree (it is, e.g., not even aware of sorts). Outside of the context of the SMT-LIBv2 language, applying an intermediate abstraction approach was successful for the original *ddmin* algorithm [24], which considers change sets (e.g., commits or individual hunks of a commit), and in [23], where the authors use local semantics of certain C++ constructs. Another example is presented in [16], which exploits the hierarchical structure of an input, independent of the concrete semantics.

Our main target language is SMT-LIBv2, which is a hierarchically structured language where, to cite the SMT-LIBv2 standard [2], "every expression [...] is a legal S-expression of Common Lisp". In contrast to ddSMTv1, in ddSMTv2 we aim for an intermediate level of abstraction to ease the burden on infrastructure and maintenance and choose to use S-expressions as the main representation of the input, just like ddsexpr does. However, additionally, we extract a comprehensive set of semantic properties to allow for SMT-LIBv2 specific and compliant simplifications. Language compliant transformations are a requirement for the specific use case of minimizing SMT-LIBv2 input to debug erroneous behavior of SMT solvers. This is mainly to avoid generating nonsensical test cases, i.e., test cases that an SMT solver will refuse to parse. Even when such test cases are refused immediately, if the overwhelming majority of generated test cases is nonsensical it can significantly impact the efficiency of our debugging tool. Note that we explicitly do not disallow delta debugging non-compliant input.

ddSMTv2 features a simple S-expression parser and represents S-expressions as a lightweight wrapper around built-in Python tuples and strings. Semantic information is recovered in an ad-hoc manner after parsing. This allows for minimal infrastructure and maintenance overhead for input parsing and representation. The parser component of ddSMTv2 has less than 100 LOC, and the ad-hoc semantic analysis accounts for less than 400 LOC. Adding support for new versions, dialects or non-standardized extensions of the SMT-LIB language does not require any changes to the parser.

This is in stark contrast to deltaSMT and ddSMTv1, which both aim to get a full understanding of the input, with all its negative consequences: deltaSMT dedicates about 50% (more than 2000 LOC) of its Java code base and ddSMTv1 even over 80% (3000 LOC) of its Python code base to parsing and input representation. Note that the former targets SMT-LIBv1, whereas the latter provides full SMT-LIBv2 support for most of the standardized theories. In both tools, parsing is a disproportionate part of the code base and extending the tools to support new theories or language constructs usually requires extensive modifications to their input parsers. These modifications have significantly complicated or even inhibited the development of these tools in the past: adding support for the theory of floating-point arithmetic in ddSMTv1 required touching more than 1000 LOC; deltaSMT, on the other hand, has never seen full support of SMT-LIBv2 and fails to parse almost all inputs from our test set.

5 Delta Debugging Strategies

Our delta debugger ddSMTv2 implements two minimization strategies which we call ddmin and hierarchical. These two can be combined into a third strategy called hybrid, which aims to utilize the best of both worlds. All three strategies use the same input representation and have access to the same pool of available mutators. However, they differ in *how* they apply mutators to simplify the input.

Algorithm 1: Main loop of ddmin strategy.

```
   Input: S-Expression $input$
 1 do                                                          // run to fixed point
 2     $simplified := False$
 3     for $M \in mutators$ do
 4         $sexprs := \{e \mid e \in input \land \mathsf{filter}_M(e)\}$, $size := |sexprs|$
 5         while $size > 0$ do
 6             for $subset \in \mathsf{partition}(sexprs, size)$ do
 7                 $candidate := \mathsf{apply}_M(input, subset)$
 8                 if $\mathsf{check\_result}(candidate)$ then
 9                     $input := candidate$, $simplified := True$
10             $sexprs := \{e \mid e \in input \land \mathsf{filter}_M(e)\}$, $size := size/2$
11 while $simplified$
12 return $input$
```

Strategy ddmin. Our ddmin strategy implements a variant of the minimization strategy of ddSMTv1 and tries to perform simplifications on multiple S-expressions in the input in parallel. Algorithm 1 shows the main loop of this strategy. For each active mutator M, the algorithm first collects all S-expressions in the input that can be simplified by M (Line 4). Simplifications are applied and checked in a fashion similar to Zeller's original *ddmin* algorithm [24]: the set of S-expressions *sexprs* is partitioned into subsets of size *size*; each S-expression $e \in subset$ is substituted in *input* (Line 7) with a simplification suggested by M; the resulting simplified input *candidate* is then checked if it still triggers the original behavior (Line 8). Once all subsets of a given size are checked, *sexprs* is updated based on the current input and partitioned into smaller subsets. As soon as all subsets of size 1 were checked, the algorithm repeats these steps with the next available mutator. The main loop of strategy ddmin is run until a fixed point is reached, i.e., the input cannot be further simplified. Strategy ddmin applies mutators in two stages. The first stage targets top-level S-expressions (e.g., specific kinds of SMT-LIB commands) until a fixed point to aggressively simplify the input before applying more expensive mutators in the second stage.

Strategy hierarchical. The main loop of the hierarchical strategy performs a simple breadth-first traversal of the S-expressions in the input, and applies all enabled mutators to every S-expression, as shown in Algorithm 2. Once a simplification is found (Line 7), all pending checks for the current S-expression are aborted and the breadth-first traversal continues with the simplified S-expression *sexpr* (Line 9). This process is repeated until a fixed point is reached, i.e., until no further simplifications are found for any S-expression. The main simplification loop (Line 3) is applied multiple times, with varying sets of mutators. In the initial stages, strategy hierarchical aims for aggressive minimization using only a small set of selected mutators, in the next-to-last stage it employs all but a few mutators that usually only have cosmetic impact, and in the last stage it includes all mutators. We observed that breadth-first traversal yields significantly better results than

Algorithm 2: Core simplification loop of hierarchical strategy

```
Input: S-Expression $input$
1  do                                                    // run to fixed point
2      $simplified := False$
3      for $sexpr \in input$ do                          // BFS traversal
4          for $M \in mutators$ do
5              if $\neg \mathit{filter}_M(sexpr)$ then continue
6              for $candidate \in \mathit{apply}_M(input, sexpr)$ do
7                  if $\mathit{check\_result}(candidate)$ then
8                      $input := candidate$, $simplified := True$
9                      continue with simplified $sexpr$ in Line 3
10 while $simplified$
11 return $input$
```

a depth-first traversal, most probably since it tends to favor simplifications on larger subtrees of the input.

Strategy hybrid. This strategy combines strategies ddmin and hierarchical in a sequential portfolio manner. It first applies ddmin until a fixed point is reached, and then calls strategy hierarchical on the simplified input. We chose this order of strategies after observing in our experiments that ddmin is usually faster in simplifying input, while hierarchical often yields smaller inputs.

6 Experimental Evaluation

We compare the different strategies implemented in ddSMTv2 against the existing delta debuggers ddsexpr, ddSMTv1, delta, linedd, and pyDelta. For this purpose, we compiled a set of SMT-LIB and SyGuS test cases from different sources. Every test case consists of an input file, a solver binary and command line configuration options for that binary. Our set of test cases includes those used in [18] and instances reported in bug reports of the SMT solvers Bitwuzla [19], CVC4 [3], Yices [12], and Z3 [17]. The test cases from [18] include issues encountered with development versions of the SMT solvers Boolector [20] and CVC4. Note that we excluded 9 test cases from this set because they did not trigger any faulty behavior on our experimental setup. In total, we collected 244 test cases consisting of inputs that trigger assertion failures, unexpected behavior or wrong solver answers. We performed all experiments on a cluster with Intel Xeon E5-2620v4 CPUs with 2.1 GHz and 128 GB memory and used a 1 h wall-clock time limit and 8 GB of memory for each delta debugger/test case pair. Table 1 summarizes the results on all 244 test cases.

A first immediate observation is the value of a simpler and more generic parser: ddSMTv1 fails to parse more than 20% of the inputs, mostly due to the lack of support for newer standard and non-standard SMT-LIBv2 constructs. Examples include the `check-sat-assuming` command, algebraic datatypes, some operators of the theory of strings, the SyGuS language extension, and

Table 1. Results summarized over all 244 test cases.

	ddsexpr	ddSMTv1	delta	linedd	pyDelta	ddmin	hier.	hybrid
Parse Errors	0	54	1	0	0	0	0	0
Incorrect Output	0	0	1	1	1	0	0	0
Timeouts	155	81	128	3	126	6	122	6
Any Simplification	219	175	114	209	119	242	242	242
Smallest Output	2	10	0	3	58	89	59	168
Avg. Reduction (%)	-40	-63	+288	-26	-4	-75	+571	-77
Avg. Reduction w/o ERR (%)	-40	-80	+291	-26	-4	-75	+571	-77
Avg. Reduction w/o TO/ERR (%)	-32	-73	+617	-26	-57	-76	-59	-79

the non-standardized extension to encode problems of separation logic. We also observe that each strategy of ddSMTv2 simplifies significantly more inputs than any other tool. The only inputs that could not be simplified by ddSMTv2 were already very small (83 and 98 bytes). Strategy hybrid achieves the smallest output on 168 test cases (more than two thirds) and an average reduction in file size by 77% (79% not counting timeouts), while only timing out on 6 test cases.

Some debuggers increase the input size (in bytes), indicated by positive reductions. Eliminating let binders or inlining function definitions frequently increase the size of the input. A positive reduction occurs if the debugger times out while performing such simplifications, or if it is unable to find viable simplifications after the input size increased. In rare individual cases, incorrect outputs were produced that did not trigger the issue under investigation. This happened because of the unchecked removal of unused variables (delta), incorrect handling of timeouts (linedd) and defective handling of quoted symbols (pyDelta).

The hybrid strategy performs significantly better than ddSMTv1, even on the set of instances that both can reduce without any timeout or error. On these commonly reduced instances (107), the results from hybrid are smaller in most cases (99), and on average smaller by about a third.

On inputs that both ddmin and hybrid reduce without timeout or error (238), the hybrid strategy produces smaller outputs on 125 cases and never generates larger results. On average, over all 238 inputs the outputs are about 5% smaller. This may seem marginal, but can make a big difference for users in practice.

Figures 1–2 show the direct comparison of ddmin, hierarchical, hybrid and ddSMTv1 in terms of output size and overall runtime as scatter plots, where a dot represents a test case and dots on the "T" lines correspond to timeouts. While strategy hierarchical tends to produce smaller output files, it is considerably slower than ddmin and runs into the time limit on 116 more test cases. As a result of this observation, we combined both strategies into the hybrid strategy, which first uses ddmin to quickly reduce the input before applying hierarchical to achieve maximum reduction. Comparing hybrid to the best of strategies ddmin and hierarchical, we see that hybrid usually achieves the smallest output and is only slower on test cases that are comparably fast to minimize. If the runtime of ddSMTv2 exceeds a few minutes, there is no discernible performance penalty.

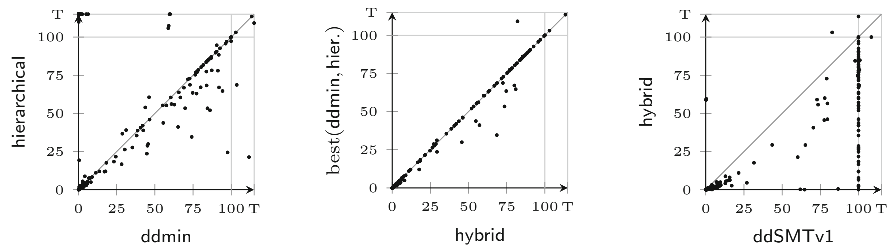


Fig. 1. Output size (in % of original size).

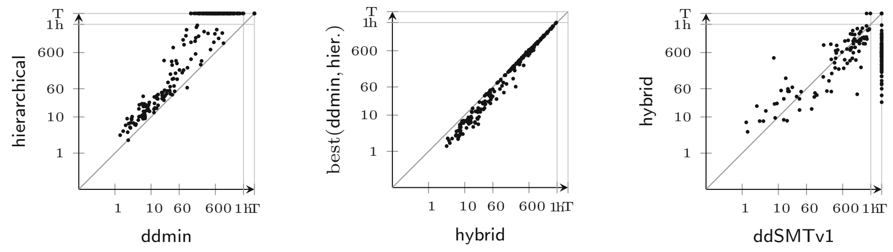


Fig. 2. Overall runtime (in seconds).

In comparison to ddSMTv1, strategy hybrid obtains significantly smaller output files on almost all inputs while having a similar runtime on inputs where ddSMTv1 terminates within the given time limit.

All strategies allow to use multiple worker processes to perform checks asynchronously. Though there is potential for significant runtime improvements, the current impact is rather limited. With 8 worker processes, hierarchical achieves on average a 2x speedup, and up to 6x speedup on a few instances. Both ddmin and hybrid, on the other hand, slow down on average (by 25% and 9%, respectively).

7 Conclusion

We have presented ddSMTv2, a delta debugger for the SMT-LIBv2 language and its dialects. Our tool improves substantially over its predecessor ddSMTv1, which is the current de-facto standard in the SMT community for delta debugging SMT-LIB input. We have shown how a more generic parser approach not only lowers the maintenance overhead of the tool itself, but also makes the delta debugger more robust and easier to extend for future SMT-LIB extensions. Our experimental evaluation has shown that ddSMTv2 significantly outperforms existing delta debugging tools on a variety of real-world test cases from different SMT solvers. Further, our experiments suggest that combining different minimization strategies is beneficial in practice to quickly obtain small output files.

References

1. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Version 3.0 - Preliminary Proposal (2021). http://smtlib.cs.uiowa.edu/version3.shtml
2. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB standard: version 2.0. In: Gupta, A., Kroening, D. (eds.) Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK) (2010)
3. Barrett, C., et al.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
4. Bayless, S.: linedd (2015). https://github.com/sambayless/linedd
5. Biere, A.: ddsexpr (2013). http://fmv.jku.at/ddsexpr
6. Bjørner, N.: SMT in verification, modeling, and testing at microsoft. In: Biere, A., Nahir, A., Vos, T. (eds.) HVC 2012. LNCS, vol. 7857, pp. 3–3. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39611-3_3
7. Brummayer, R., Biere, A.: Fuzzing and delta-debugging SMT solvers. In: Proceedings of the 7th International Workshop on Satisfiability Modulo Theories, pp. 1–5 (2009)
8. Cook, B.: Formal reasoning about the security of amazon web services. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 38–47. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_3
9. Corzilius, F., Kremer, G., Junges, S., Schupp, S., Ábrahám, E.: SMT-RAT: an open source C++ toolbox for strategic and parallel SMT solving. In: Heule, M., Weaver, S. (eds.) SAT 2015. LNCS, vol. 9340, pp. 360–368. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24318-4_26
10. Dobal, F.: DeltaSMT for SMT-LIBv2 (2013). updated version of [17], unavailable
11. Niemetz, A., Preiner, M., Kremer, G.: ddSMTv2 documentation. https://ddsmt.readthedocs.io
12. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 737–744. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_49
13. Niemetz, A., Preiner, M., Kremer, G.: ddSMTv2. https://github.com/ddsmt/ddsmt
14. Kremer, G.: pyDelta (2021). https://github.com/nafur/pydelta
15. Kremer, G., Nalbach, J.: delta. https://github.com/smtrat/smtrat/tree/master/src/delta, SMT-RAT's delta debugger
16. Misherghi, G., Su, Z.: HDD: hierarchical delta debugging. In: Osterweil, L.J., Rombach, H.D., Soffa, M.L. (eds.) 28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, 20–28 May 2006, pp. 142–151. ACM (2006). https://doi.org/10.1145/1134285.1134307
17. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
18. Niemetz, A., Biere, A.: ddSMT: a delta debugger for the SMT-LIB v2 format. In: Proceedings of the 11th International Workshop on Satisfiability Modulo Theories, SMT, pp. 8–9 (2013)
19. Niemetz, A., Preiner, M.: Bitwuzla at the SMT-COMP 2020. CoRR abs/2006.01621 (2020). https://arxiv.org/abs/2006.01621
20. Niemetz, A., Preiner, M., Biere, A.: Boolector 2.0. J. Satisf. Boolean Model. Comput. **9**(1), 53–58 (2014). https://doi.org/10.3233/sat190101

21. Raghothaman, M., Reynolds, A., Udupa, A.: The SyGuS Language Standard Version 2.0. Tech. rep. (2019). https://sygus.org/assets/pdf/SyGuS-IF_2.0.pdf
22. Ranise, S., Tinelli, C.: The SMT-LIB Standard: Version 1.2. Tech. rep., Department of Computer Science, The University of Iowa (2006). https://smtlib.cs.uiowa.edu/papers/format-v1.2-r06.08.30.pdf
23. Regehr, J., Chen, Y., Cuoq, P., Eide, E., Ellison, C., Yang, X.: Test-case reduction for C compiler bugs. In: Vitek, J., Lin, H., Tip, F. (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - 11–16 June 2012, pp. 335–346. ACM (2012). https://doi.org/10.1145/2254064.2254104
24. Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. IEEE Trans. Softw. Eng. **28**(2), 183–200 (2002). https://doi.org/10.1109/32.988498

Learning Union of Integer Hypercubes with Queries

(with Applications to Monadic Decomposition)

Oliver Markgraf[1](✉), Daniel Stan[1], and Anthony W. Lin[1,2]

[1] TU Kaiserslautern, Kaiserslautern, Germany
{markgraf,stan,lin}@cs.uni-kl.de
[2] Max Planck Institute for Software Systems, Kaiserslautern, Germany

Abstract. We study the problem of learning a finite union of integer (axis-aligned) hypercubes over the d-dimensional integer lattice, i.e., whose edges are parallel to the coordinate axes. This is a natural generalization of the classic problem in the computational learning theory of learning rectangles. We provide a learning algorithm with access to a minimally adequate teacher (i.e. membership and equivalence oracles) that solves this problem in polynomial-time, for any fixed dimension d. Over a non-fixed dimension, the problem subsumes the problem of learning DNF boolean formulas, a central open problem in the field. We have also provided extensions to handle infinite hypercubes in the union, as well as showing how subset queries could improve the performance of the learning algorithm in practice. Our problem has a natural application to the problem of monadic decomposition of quantifier-free integer linear arithmetic formulas, which has been actively studied in recent years. In particular, a finite union of integer hypercubes correspond to a finite disjunction of monadic predicates over integer linear arithmetic (without modulo constraints). Our experiments suggest that our learning algorithms substantially outperform the existing algorithms.

1 Introduction

Suppose that we are interested in finding a formula $\varphi(\bar{x})$ over some theory T (e.g. integer linear arithmetic) to "capture" a certain phenomenon, which in verification could be, for instance, an invariant that a program satisfies some safety property. The process of discovering φ can be captured by the notion of a *learning algorithm* by allowing certain types of queries as an interface to some teacher [3]. Most standard learning frameworks can be captured in this way. Here are some examples. Valiant's well-known notion of *PAC-learning* can be captured by an oracle that returns a new random sample from an unknown distribution. Angluin's well-known notion of *exact learning* [2,3] can be captured by an interaction with the so-called *minimally adequate teachers*, which

This research was supported by the ERC Starting Grant 759969 (AV-SMP) and Max-Planck Fellowship.

can answer membership and equivalence queries. This has many applications in verification, e.g., verification of parameterized systems [10,20,23] and compositional verification [9]. Another learning framework that has become very popular in verification is CEGIS (Counterexample Guided Inductive Synthesis) [21,27], wherein a learning algorithm can ask equivalence queries, but expect various types of "constraint-like" counterexamples (e.g. implication counterexamples) to be returned by the teacher. This is of course in contrast to Angluin's exact learning setting, wherein the teacher may return only a positive/negative counterexample (a point in the symmetric difference of the target concept and the hypothesis).

In this paper, we study the problem of learning sets of points over the d-dimensional integer lattice that can be expressed as a *finite union of integer (axis-aligned, a.k.a. rectilinear) hypercubes*, i.e., whose edges are parallel to the coordinate axes. Such a concept class of course forms a strict subclass of sets of points that are definable by a formula $\varphi(x_1, \ldots, x_d)$ in the integer linear arithmetic (a.k.a. *semilinear sets*), which have been addressed in several papers including [1,17,28], whose PAC-learnability is as hard as PAC-learning boolean formulas in DNF [16]—a long-standing open problem in learning theory—when binary representations are permitted (even over dimension one [1]). That said, finite unions of integer hypercubes are a concept class that naturally arises in computer science. Below we mention a few examples.

The problem of learning rectangles (2-cube) and generalization to d-dimension are a classic example in computational learning theory, e.g., see [16,22]. Maass and Turán [22] showed for example that the d-dimensional rectilinear cubes can be learned in polynomial-time with $O(\log n)$ queries, where the corners of the cubes are represented in binary. The authors posed as an open problem if one can learn a union of two (possibly overlapping) rectangles with only $O(\log n)$ equivalence queries. Chen [11] showed that this can be learned with 2 equivalence queries and $O(d. \log n)$ membership queries. Later Chen and Ameur [12] showed that there is a polynomial-time algorithm using at most $O(\log^2 n)$ queries. The same paper left as an open problem if there is a polynomial-time exact learning algorithm that learns finite unions of rectilinear cubes over a fixed dimension d. In this paper, *we answer this in the positive*, and further show that this can be extended to allow *infinite rectilinear hypercubes*, which in turn allow interesting applications in formal verification, as we discuss below.

Finite unions of rectilinear cubes arise naturally in program analysis and verification. Here we mention two examples. First, solving games over a large game graph has benefited from constraint-based approaches, where winning regions can be succinctly represented and checked efficiently [6]. For example, the discretization of the Cinderella-Stepmother problem [6] admits winning regions that may be represented by a union of a small number of cubes. Secondly, verification algorithms benefit from optimization techniques like monadic decomposition [29], where the aim is the rewriting of a given quantifier-free SMT formula $\varphi(x_1, \ldots, x_n)$ into an equivalent boolean combination of monadic predicates $\psi(x_i)$ in some special form, i.e., typically in DNF [5,7,15,19], or by an if-then-else formula [29], which could sometimes be exponentially smaller than the DNF equivalent representation. Veanes *et al.* [29] provided a generic semi-decision procedure for performing this monadic decomposition as an if-then-else formula, which works regardless of the base theory. The restriction of the problem to the quantifier-free theory of integer linear arithmetic (with and without extra modulo constraints) was studied in [15], wherein the problem was shown to be coNP-complete and a monadic decomposition could be exponentially large in general. For the subcase without modulo constraints, a monadic decomposition in DNF corresponds precisely to a finite union of (possibly infinite) rectilinear hypercubes, which is the subject of this paper. We describe below how oracles for memberships and equivalence (as well as more powerful queries like subsets) admit a fast implementation via an SMT-solver, which enable our learning algorithms to be applied to compute such a monadic decomposition.

Contributions. We study the problem of learning finite unions of rectilinear hypercubes (over $\mathbb{Z}^d$) in Angluin's exact learning framework with membership and equivalence queries [2,3]. Our result is a polynomial-time exact learning algorithm for learning finite unions of rectilinear hypercubes over $\mathbb{Z}^d$ for fixed d. This answers an open problem of [12]. As observed in [12], over non-fixed d, this problem generalizes DNF since each term can be seen as a hypercube over $\{0,1\}^d$. That is, without fixing d, the problem is as hard as learning unrestricted DNF, which is well-known to be a major open problem in computational learning theory [4].

In view of applying our learning algorithm to the monadic decomposition problem [15,29] for quantifier-free integer linear arithmetic formulas, we consider two extensions. Firstly, we allow *infinite hypercubes*. For example, over 1-dimension, these would include infinite intervals like $[7, \infty)$, which would correspond to the formula $x \geq 7$. Secondly, we observe that the *subset query* (i.e. checking if the target concept includes a given finite union H of hypercubes) is not an expensive query for performing monadic decomposition, i.e., it would correspond to a single satisfiability check of a quantifier-free integer linear arithmetic formula, which can be handled easily by an SMT-solver. Subset queries belong to one of the standard types of queries in Angluin's active learning framework, e.g., see [3]. For this reason, we provide an optimization of our learning algorithm by means of subset queries.

We implemented these learning algorithms (vanilla and various optimization including subset queries and "unary/binary acceleration"), using Z3 [26] as the backend for answering equivalence and subset queries (each a satisfiability check of a quantifier-free formula). We have performed a micro-benchmarking to stress-test our algorithms against the generic monadic decomposition procedure of [29], which also use Z3 as the backend, using various geometric objects over $\mathbb{Z}^d$ as benchmarks. Our experiments suggest that our algorithms substantially outperform the generic procedure.

Organization. Preliminaries are in Sect. 2. We present the *overshooting algorithm* that witnesses polynomial learnability of finite unions of rectilinear cubes over a fixed dimension d with membership and equivalence in Sect. 3. In Sect. 4, we provide two extensions: (1) how subset queries could help speed up the over-shooting algorithm, (2) how the algorithm could be extended to handle infinite cubes. Applications to monadic decomposition and experiments are presented in Sect. 5. We conclude in Sect. 6.

We refer the reader to the technical report [25] when proofs are omitted and to the artifact [24] for implementation and benchmark details.

2 Preliminaries

We introduce below some common mathematical notations: $\mathbb{N}$ and $\mathbb{Z}$ are the sets of natural numbers and integers, respectively. For $a, b \in \mathbb{Z}$, we write $[a, b] = \{i \mid a \leq i \leq b\}$; For any set X, we denote its power-set $\mathcal{P}(X)$ and its cardinal $|X| \in \mathbb{N} \uplus \{\infty\}$; Given two sets A, B, the *symmetric difference* is written $A \Delta B = A \backslash B \cup B \backslash A$;

When analyzing complexity of the presented algorithms, we assume binary encoding for any number $n \in \mathbb{Z}$, which is part of the input of the considered algorithms, namely, $\text{size}(n) = 1 + \lceil \log(|n|+1) \rceil$, where log is the base 2 logarithm.

Hypercubes. For a fixed *dimension* $d \in \mathbb{N}$, we consider the *discrete lattice* $\mathbb{Z}^d$. A *point* $\mathbf{v} \in \mathbb{Z}^d$ can be described by its coordinates $\mathbf{v}[k]$ for $k \in [1, d]$. Let $\mathbf{v}[k/\alpha]$ denote the vector $\mathbf{v}$ where the i-th coordinate has been replaced by $\alpha \in \mathbb{Z}$. The notation $\mathbf{0}^d = (0, \ldots, 0) \in \mathbb{Z}^d$ denotes the origin, or simply $\mathbf{0}$ when the dimension is clear from context. We use standard notation for component-wise additions and scalar multiplication. In particular, for $\alpha \in \mathbb{Z}$, $\mathbf{v} + \alpha \cdot \mathbf{v}'$ denotes the vector $\mathbf{v}'' \in \mathbb{Z}^d$ such that for all i, $\mathbf{v}''[i] = \mathbf{v}[i] + \alpha \cdot \mathbf{v}'[i]$. For $1 \leq i \leq d$, we write $\mathbf{e}_i$ for the i-th *elementary vector*, $\mathbf{e}_i = \mathbf{0}[i/1]$. We shall be mostly using the standard *component-wise order* $\leq$ over vectors in $\mathbb{Z}^d$: $\mathbf{v} \leq \mathbf{v}'$ iff for all i, $\mathbf{v}[i] \leq \mathbf{v}'[i]$. We finally denote the size of a vector as the sum of the sizes of its components: $\text{size}(\mathbf{v}) = \sum_{i=1}^{d} \text{size}(\mathbf{v}[i])$, for any $\mathbf{v} \in \mathbb{Z}^d$.

Our main study focuses on *rectilinear hypercubes* (*cubes* for short), i.e., any set of points of the form $C = \{\mathbf{v} \mid \underline{\mathbf{v}} \leq \mathbf{v} \leq \overline{\mathbf{v}}\}$ for some $\underline{\mathbf{v}} \leq \overline{\mathbf{v}} \in \mathbb{Z}^d$. The size of C is uniquely defined as $\text{size}(C) = \text{size}(\underline{\mathbf{v}}) + \text{size}(\overline{\mathbf{v}})$. On the contrary, an arbitrary finite set X has no unique representation as a finite union of cubes, therefore we define its size as the size of its best representation:

$$\text{size}(X) = \min \left\{ \sum_{i=1}^{n} \text{size}(\underline{\mathbf{v}}_i) + \text{size}(\overline{\mathbf{v}}_i) \;\middle|\; \exists n, \underline{\mathbf{v}}_1 \ldots \overline{\mathbf{v}}_n : X = \bigcup_{i=1}^{n} \text{Cube}(\underline{\mathbf{v}}_i, \overline{\mathbf{v}}_i) \right\}$$

We adopt here a worst-case analysis approach, where our later reasoning and complexity analysis are valid for any representation, they are in particular valid for its best representation.

Learning Model. We first recall some standard definition from computational learning theory; for more, see [16]. Fix a countable *base set* $\mathcal{D} = \bigcup_{i=1}^{n} \mathcal{D}_i$, where the sets $\mathcal{D}_i$'s are pairwise disjoint. The problem of learning boolean formulas in DNF uses $\mathcal{D}_i = \{0,1\}^i$, i.e., the set of all binary sequences of length i, which can be thought of as a set of all assignments to a boolean function over $x_1, \ldots, x_i$. The learning problem in this paper uses $\mathcal{D}_i = \mathbb{Z}^i$. A *concept* X is simply a subset of $\mathcal{D}_i$, for some $i \in \mathbb{Z}_{>0}$. For example, when $\mathcal{D}_i = \{0,1\}^i$, a concept is simply a boolean function over $x_1, \ldots, x_i$. When we speak of a learning problem, we always have a fixed set of representations in mind. For example, when we speak of learning boolean formulas in DNF (Disjunctive Normal Form), the representation φ_X of a boolean function X has to be a formula over $x_1, \ldots, x_i$ in DNF. For example, X could be a boolean function, whereas φ_X a DNF formula representing X. Note that a concept could admit many possible representations. A *concept class* $\mathcal{C} = \bigcup_{i=1}^{\infty} \mathcal{C}_i$ is a set of concepts, where $\mathcal{C}_i \subseteq \mathcal{P}(\mathcal{D}_i)$. For example, $\mathcal{C}_i$ could be the set of boolean functions over variables $x_1, \ldots, x_i$. When the set of representations for $\mathcal{C}$ is fixed (e.g. DNF for representing boolean functions), we could define size(X) of the concept X to be the size of the smallest representation of X. In this paper, we are dealing with the concept class $\mathcal{C}_d \subseteq \mathcal{P}(\mathbb{Z}^d)$ of sets of integer points that can be represented as a finite union of rectilinear hypercubes over $\mathbb{Z}^d$. Earlier in this section we have defined this concept, as well as the size of the representation. To avoid notational clutter, we will often denote the concept class $\mathcal{C}_d$ by $\mathcal{C}$ because our algorithm typically assumes that d is fixed.

In Angluin's active learning framework [2,3], the learner has access to oracles (a.k.a. teachers) that could provide hints about the target concept X to the learner. A *minimally adequate teacher* must be able to answer membership and equivalence queries.

Definition 1 (M+EQ Oracles). *Consider some target concept $X \in \mathcal{C}_d$ for some concept class $\mathcal{C} = \bigcup_{d=1}^{\infty} \mathcal{C}_d$ and let $\bot, \top \notin \mathcal{D}$ be two fresh symbols.*

- *A* membership oracle *(M) for X is a function $\Phi_X : \mathcal{D}_d \to \{\top, \bot\}$, which outputs $\top$ iff $\mathbf{v} \in X$.*
- *An* equivalence oracle *(EQ) for X is a function $\Psi_X : \mathcal{C}_d \to \mathcal{D}_d \uplus \{\top\}$ such that for all* hypothesis *$H \in \mathcal{C}$, $\Psi_X(H) \in (H \Delta X) \uplus \{\top\}$ and $\Psi_X(H) = \top$ implies $H = X$.*

Intuitively, an equivalence oracle tells, for any hypothesis $H \in \mathcal{C}$, whether $H = X$. If yes, $\top$ is returned; if not, it provides a *counterexample*, namely a point in the symmetric difference. Angluin has considered other types of queries as well in her framework including subset/superset queries and difference queries (e.g. see her excellent survey [3]). We will use the subset queries in Sect. 4.

A learning algorithm $\mathcal{A}$ is said to *learn* the concept class $\mathcal{C} = \bigcup_{d=1}^{\infty} \mathcal{C}_d$ if, given d as input and any unknown target concept X, it terminates and outputs a representation of X after a finite amount of interaction with the oracles. Assuming that the oracle always returns the shortest counterexamples, its running time is defined to be number of steps (measured in d and size(X)) that $\mathcal{A}$ takes to output a representation of X. The complexity *comp*$(d, \text{size}(X))$ of $\mathcal{A}$ measures the number of steps taken in the worst case for all d and size(X). It runs in polynomial time if *comp* is a polynomial function. It remains a long-standing open problem in computational learning theory if there is a learning algorithm for boolean formulas represented in DNF, which is true for almost all major models including exact learning and PAC (see [4]). Over geometric concepts including hypercubes and semilinear sets, the dimension d is sometimes considered a fixed parameter, e.g., see [1,12,17,22].

3 Minimally Adequate Teacher

We restrict first our attention to the minimally adequate teacher setting where only a membership and equivalence oracle are provided, and provide constructions for intermediate procedures that can be interpreted as oracles.

3.1 Corner Oracle

At the heart of our learning algorithm is the concept of corners:

Definition 2. *Given a set of points $X \subseteq \mathbb{Z}^d$, a* maximal corner *(resp* minimal corner*) of X is a point $\mathbf{v} \in X$ maximal (resp minimal) with respect to component-wise ordering $\leq$. We write $\overline{\text{Corners}}(X)$ and $\underline{\text{Corners}}(X)$ for the sets of maximal and minimal corners, respectively, and write* $\text{Corners}(X) = \overline{\text{Corners}}(X) \cup \underline{\text{Corners}}(X)$.

Given a membership oracle for some $X \in \mathcal{C}$ containing $\mathbf{0}$, Algorithm 1 returns *some* maximal corner of a given finite subset. Intuitively, for each coordinate i, a binary search is made until a border of X is eventually found. More precisely, we provide the following complexity analysis.

Algorithm 1. Binary search for a maximal corner, assuming $\mathbf{0} \in X$

```
Ensure: Returned value is a maximal corner of $X$
Require: $\mathbf{0} \in X$; $\Phi_X$ a membership oracle for $X$
  function FINDMAXCORNER($\Phi_X$)
    $i \leftarrow 0$;   $\mathbf{v} = \mathbf{0}$
    while $i < d$ do
      $i \leftarrow i + 1$;   $k \leftarrow 1$;   $l \leftarrow 1$;
      if $\Phi_X(\mathbf{v} + \mathbf{e}_i)$ then
        while $\Phi_X(\mathbf{v} + k \cdot \mathbf{e}_i)$ do
          $l \leftarrow k$;   $k \leftarrow 2k$
        while $k - l > 1$ do
          if $\Phi_X(\mathbf{v} + \lfloor (k+l)/2 \rfloor \cdot e_i)$ then
            $l \leftarrow \lfloor (k+l)/2 \rfloor$
          else
            $k \leftarrow \lfloor (k+l)/2 \rfloor$
        $\mathbf{v} \leftarrow \mathbf{v} + l \cdot \mathbf{e}_i$;   $i \leftarrow 0$
    return $\mathbf{v}$
```

Proposition 1. *Let Φ_X be a membership oracle for $X = \cup_{i=1}^{n} \mathrm{Cube}(\underline{\mathbf{v}}_i, \overline{\mathbf{v}}_i)$ and assume $\mathbf{0} \in X$. Then* findMaxCorner(Φ_X) *terminates after $O\left(\sum_{j=1}^{n} \mathrm{size}(\overline{\mathbf{v}}_j)\right)$ queries and returns some $\overline{\mathbf{v}} \in \overline{\mathrm{Corners}}(X)$.*

This algorithm provides a partial implementation of the following oracle:

Definition 3. *Given $X \in \mathcal{C}$, a* corner oracle *for X is any function $\Theta_X : X \to \underline{\mathrm{Corners}}(X) \times \overline{\mathrm{Corners}}(X)$.*

A complete implementation of this oracle is provided by noticing that membership oracles can easily be composed:

Remark 1. Assume Φ_A and Φ_B are two given membership oracles, respectively for two arbitrary sets A and B, and $f : \mathbb{Z}^d \to \mathbb{Z}^d$. One can build membership oracles for $A \cup B$, $A \cap B$, $A \Delta B$, $A \backslash B$ and $f(A)$. In particular:

- By instantiating $f : \mathbf{v} \mapsto -\mathbf{v}$, the previous procedure applied on $\Phi_{f(A)}$ returns some $\mathbf{v} \in \overline{\mathrm{Corners}}(-A)$, so $-\mathbf{v} \in \underline{\mathrm{Corners}}(A)$.
- For any $\mathbf{v}_0 \in A$ and $f : \mathbf{v} \mapsto \mathbf{v} - \mathbf{v}_0$, $\Phi_{f(A)}$ is a membership oracle for $A - \mathbf{v}_0 = \{\mathbf{v} \mid \mathbf{v} + \mathbf{v}_0 \in A\}$ containing $\mathbf{0}$, so findMaxCorner($\Phi_{f(A)}$) returns some $\mathbf{v} \in \overline{\mathrm{Corners}}(A - \mathbf{v}_0)$ so $\mathbf{v} + \mathbf{v}_0 \in \overline{\mathrm{Corners}}(A)$.

In both cases, notice that $\mathrm{size}(f(A)) \leq \mathrm{size}(A) + \mathrm{size}(\mathbf{v}_0) \leq 2\mathrm{size}(A)$.

In the sequel we write Φ_C for the membership oracle of any set C obtained by composing sets whose oracles are provided. We also assume having constructed the two procedures findMaxCorner($\mathbf{v}, \Phi_X$) and findMinCorner($\mathbf{v}, \Phi_X$).

3.2 Overshooting Algorithm

Algorithm 2 Overshooting algorithms

```
Require: $\Phi_X$ membership oracle for $X$, $\Psi_X$ equivalence oracle for $X$
  function LearnCubes($\Phi_X$, $\Psi_X$)
      $H \leftarrow \emptyset$
      repeat
          $\mathbf{v} \leftarrow \Psi_X(H)$
          $H \leftarrow$ Refine($H, \mathbf{v}, \Phi_X$)
      until $\mathbf{v} = \top$
  function RefineSym($H, \mathbf{v}, \Phi_X$)
      $\underline{\mathbf{v}} \leftarrow$ findMinCorner($\mathbf{v}, \Phi_{X \Delta H}$)
      $\overline{\mathbf{v}} \leftarrow$ findMaxCorner($\mathbf{v}, \Phi_{X \Delta H}$)
      return $H \Delta \mathrm{Cube}(\underline{\mathbf{v}}, \overline{\mathbf{v}})$

  function RefineAddRemove($H, \mathbf{v}, \Phi_X$)
      if $\Phi_X(\mathbf{v})$ then
          $\underline{\mathbf{v}} \leftarrow$ findMinCorner($\mathbf{v}, \Phi_{X \backslash H}$)
          $\overline{\mathbf{v}} \leftarrow$ findMaxCorner($\mathbf{v}, \Phi_{X \backslash H}$)
          return $H \cup \mathrm{Cube}(\underline{\mathbf{v}}, \overline{\mathbf{v}})$
      else
          $\underline{\mathbf{v}} \leftarrow$ findMinCorner($\mathbf{v}, \Phi_{H \backslash X}$)
          $\overline{\mathbf{v}} \leftarrow$ findMaxCorner($\mathbf{v}, \Phi_{H \backslash X}$)
          return $H \backslash \mathrm{Cube}(\underline{\mathbf{v}}, \overline{\mathbf{v}})$
```

The core loop of the learning algorithm is presented in the LearnCubes function of Algorithm 2. The hypothesis is initially empty, and is later refined, as long as a counterexample is returned. How to refine the hypothesis given a counterexample? Two implementations of Refine are provided namely RefineSym and RefineAddRemove, giving rise to two variants of the algorithm. In both cases, the refinement takes a counterexample as an input and uses the corner oracle to build a cube C. In the former variant, a symmetric difference between the current hypothesis and C is made, while in the latter, C is either added or removed from the hypothesis.

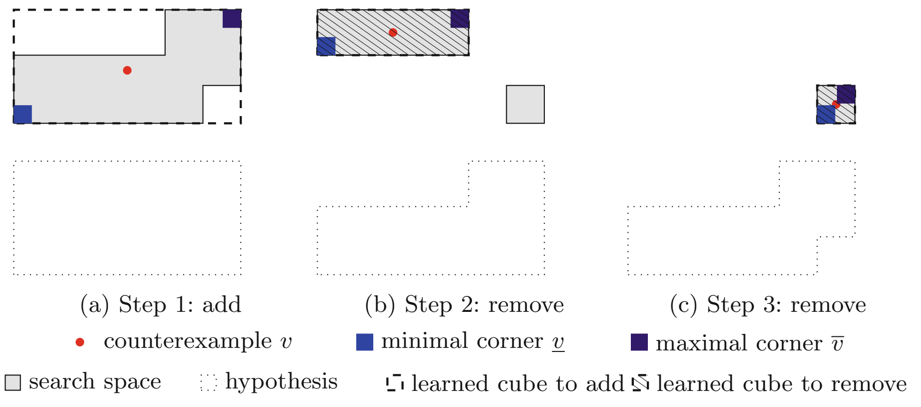


Fig. 1. Possible run of the overshooting algorithm on two cubes in 2 dimensions

An example run of the RefineAddRemove variant is depicted in Fig. 1. While the above diagrams represent the search space used by the corner oracles, the below diagrams depict the resulting hypothesis after refinement. Initially, the hypothesis is empty (not represented) so the search space coincides with the target set X, which can be represented as a union of two overlapping cubes. A counterexample $\mathbf{v} \in X \backslash H$ is therefore returned by the equivalence oracle. As $\mathbf{v} \in X$, the refinement procedure adds some cube by searching the state space $X \backslash H = X$ around $\mathbf{v}$. A too large cube is then added to the hypothesis, and a negative counterexample $\mathbf{v} \in H \backslash X$ is then returned. The search space is now $H \backslash X$ and the algorithm aims at removing some smaller cube from the hypothesis. After two removals, the final hypothesis coincides with the target.

Hypothesis Representation. Both variants are operating on the hypothesis by applying boolean operations. One can naturally wonder if hypothesis represented by union, symmetric differences and differences of cubes can be handled by oracles operating on the concept class of finite cubes. As a matter of fact, we will observe that $H \Delta X$, $H \backslash X$ and $X \backslash H$ can all be represented in $\mathcal{C}$:

Lemma 1 (Cube intersection and subtraction). *Let* $C_1 = \mathrm{Cube}(\underline{\mathbf{v}}_1, \overline{\mathbf{v}}_1)$ *and* $C_2 = \mathrm{Cube}(\underline{\mathbf{v}}_2, \overline{\mathbf{v}}_2)$ *two cubes.*

Then $C_1 \cap C_2$ *is a cube and* $C_2 \backslash C_1$ *can be written as the disjoint union of* $2d$ *cubes. Moreover, these computations are effective in* $2d$ *operations.*

Intuitively, one can think of a cube subtracted by a smaller cube results in a family of cubes, one for each face of the larger cube. There are $2d$ faces for a cube in dimension d.

3.3 Repetition-Free Complexity

In order to analyze the complexity of both variants of the algorithm, we fix a finite target set $X \in \mathcal{C}_d$ and one of its representation as a union of cubes:

$$X = \bigcup_{i=1}^{n} \mathrm{Cube}(\underline{\mathbf{v}}_i, \overline{\mathbf{v}}_i)$$

We prove by induction on the iteration step that H can be expressed as a union of cubes, whose corners are aligned on a particular set of points:

Definition 4 (Abstract grid). *For $1 \le k \le d$, we define the sets:*

$$\underline{B}_k = \{\overline{\mathbf{v}}_i[k] + 1 \mid 1 \le i \le C\} \cup \{\underline{\mathbf{v}}_i[k] \mid 1 \le i \le C\}$$
$$\overline{B}_k = \{\overline{\mathbf{v}}_i[k] \mid 1 \le i \le C\} \cup \{\underline{\mathbf{v}}_i[k] - 1 \mid 1 \le i \le C\}$$

For any $A \subseteq \mathbb{Z}^d$, we write $A \in \mathcal{B}$ whenever is a finite union of cubes of the form $\mathrm{Cube}(\mathbf{v}, \mathbf{v}')$ *such that for all k, $\mathbf{v}[k] \in \underline{B}_k$ and $\mathbf{v}'[k] \in \overline{B}_k$.*

Intuitively, $\underline{B}_k$ (resp $\overline{B}_k$) describes all the possible k-coordinate for minimal corners (resp maximal). A coordinate for a max corner, i.e. a constraint of the form $x_k \le \alpha$, can become a coordinate for a minimal corner, i.e. a constraint of the form $x_k \ge \alpha + 1$, when taking the complement during a difference operation, and vice versa.

We observe that $\mathcal{B}$ is stable by union, intersection and difference. In particular, the overshooting algorithms maintain $H \in \mathcal{B}$, namely the hypothesis always has minimal (resp maximal) corners that align with $\underline{B}_k$ (resp $\overline{B}_k$) on the k-th coordinate. Figure 2 provides an example of such points for a target made of the union of two cubes.

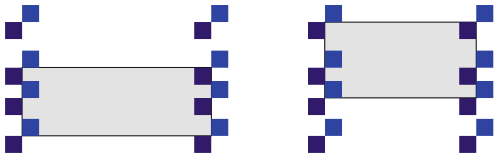


Fig. 2. Possible minimal and maximal corners for cubes appearing in the hypothesis, for a given target space

Since the sets $\underline{B}_k$ and $\overline{B}_k$ are of size at most $2n$ for every k, there are at most $(2n)^{2d}$ possible cubes, polynomial for a fixed d. Assuming $H \in \mathcal{B}$, we can ensure that Lemma 1 maintains a polynomial representation of the hypothesis throughout the algorithm until termination.

Although $\mathcal{B}$ is of polynomial size, proving $H \in \mathcal{B}$ is not sufficient to prove termination of the algorithm in polynomial time, especially if some cubes in $\mathcal{B}$ are added and removed several times. Consider for example Fig. 3 which depicts a possible run of the algorithm on three aligned cubes by its successive hypotheses:

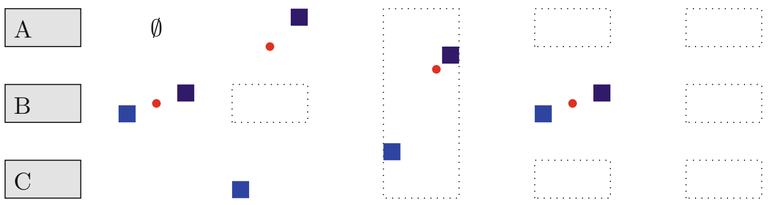


Fig. 3. Possible run on three cubes where cube B is added twice to the hypothesis.

cube B is added during the first step, but is later covered when the algorithm tries to learn A but overshoots. Another overshooting happens when trying to remove the space between A and B, which ends up removing all space between A and C. The cube C has then to be learned a second time, terminating the algorithm.

To circumvent this issue, we propose an optimization that prevents visiting twice the same minimal corner $\underline{\mathbf{v}}$. We base our reasoning on the following observations:

– If $\mathbf{v} \in X$, then $\underline{\mathbf{v}} \in X$, so $\underline{\mathbf{v}}$ should not be later removed.
– If $\mathbf{v} \notin X$, then $\underline{\mathbf{v}} \notin X$, so $\underline{\mathbf{v}}$ should not be later added back to H.

Algorithm 3 introduces an optimized refinement procedure to keep track of the already added maximal corners. Although an analogous optimization can be done on the symmetric difference variant, we only discuss here RefineAddRemove2.

Once a minimal corner $\underline{\mathbf{v}}$ for a candidate cube has been found, we continue the search of a maximal corner $\overline{\mathbf{v}}$ by avoiding points that will result in the removal (resp addition) of already added (resp removed) minimal corners.

Algorithm 3. Optimized refinement avoiding visited minimal corners

```
Let $V \leftarrow \emptyset$
**function** RefineAddRemove2($H, \mathbf{v}_e, \Phi_X$)
    **if** $\Phi_X(\mathbf{v}_e)$ **then**
        Let $\underline{\mathbf{v}} =$ findMinCorner($\mathbf{v}_e, \Phi_{X \setminus H}$)
        Let $\overline{\mathbf{v}} =$ findMaxCorner($\underline{\mathbf{v}}, \Phi_{X \setminus H \setminus \{\mathbf{v} \mid \exists \mathbf{v}' \in V : \underline{\mathbf{v}} \leq \mathbf{v}' \leq \mathbf{v}\}}$)
        $V \leftarrow V \uplus \{\underline{\mathbf{v}}\}$
        **return** $H \cup \mathrm{Cube}(\underline{\mathbf{v}}, \overline{\mathbf{v}})$
    **else**
        Let $\underline{\mathbf{v}} =$ findMinCorner($\mathbf{v}_e, \Phi_{H \setminus X}$)
        Let $\overline{\mathbf{v}} =$ findMaxCorner($\underline{\mathbf{v}}, \Phi_{H \setminus X \setminus \{\mathbf{v} \mid \exists \mathbf{v}' \in V : \underline{\mathbf{v}} \leq \mathbf{v}' \leq \mathbf{v}\}}$)
        $V \leftarrow V \uplus \{\underline{\mathbf{v}}\}$
        **return** $H \setminus \mathrm{Cube}(\underline{\mathbf{v}}, \overline{\mathbf{v}})$
```

Notice how only the maximal corner search benefits from the optimization, by tracking down minimal corners only. As a matter of fact, one could store the whole visited cubes in set V. However, when a search for maximal corner is carried, the resulting cube will intersect a previously visited cube as soon as the max corner crosses the minimal corner of the visited cube.

We exploit again Remark 1 to build an oracle for every mentioned membership oracle. Since V is a finite set, one can indeed build a membership oracle for the set $\{\mathbf{v} \mid \exists \mathbf{v}' \in V \backslash X : \underline{\mathbf{v}} \leq \mathbf{v}' \leq \mathbf{v}\}$. Due to this exclusion region, a finer analysis has to be conducted to prove $H \in \mathcal{B}$.

Lemma 2. *The two optimized variants maintain the following invariants:*

1. $V \cap X \subseteq H$*;*
2. $(V \backslash X) \cap H = \emptyset$*;*
3. for all $\mathbf{v} \in V$*, and any* k*,* $\mathbf{v}[k] \in \underline{B}_k$*;*
4. $H \in \mathcal{B}$*.*

Properties 1 and 2 ensure that every v added to V is never added twice. These also ensures correctness of the algorithm: remark that the search for a maximal corner is not started from the initial counterexample $\mathbf{v}_e$ but from $\underline{\mathbf{v}}$, which is indeed is in the search space since $\underline{\mathbf{v}} \notin \{\mathbf{v} \mid \exists \mathbf{v}' \in V : \underline{\mathbf{v}} \leq \mathbf{v}' \leq \mathbf{v}\}$ (no point added twice to V). Finally, property 3 ensures that only elements of $(\underline{B}_k)_k$ are added to V, hence a maximal number of $(2n)^d$ additions.

Proof. At the beginning of the algorithm, $V = H = \emptyset$, satisfying all given properties. We prove the result by induction on the iteration step:

1. By definition of corner oracles, namely FindMaxCorner, if $\underline{\mathbf{v}} \in X$ has been added to V during some previous iteration, it was added in the first branch (the oracle returns some point in the search region, which excludes X in the second branch). Therefore, it was also added to H during this iteration. Consider some later iteration removing elements from H, namely an iteration executing the second branch. Some cube $C = \text{Cube}(\underline{\mathbf{v}}', \overline{\mathbf{v}}')$ has been computed by the corner oracles in this branch such that $\overline{\mathbf{v}}' \in H \backslash X \backslash \{v \mid \exists \mathbf{v}' \in V : \underline{\mathbf{v}}' \leq \mathbf{v}' \leq \mathbf{v}\}$ In particular, since $\underline{\mathbf{v}} \in V$, we do not have $\underline{\mathbf{v}}' \leq \underline{\mathbf{v}} \leq \overline{\mathbf{v}}'$ hence $\underline{\mathbf{v}} \notin C$ and $\underline{\mathbf{v}}$ is not removed.
2. Similar to (1) (symmetric case).
3. For every $\underline{\mathbf{v}}$ added to V, it was produced by a (max) corner query made on $X \backslash H$ or $H \backslash X$. Both of these sets are in $\mathcal{B}$ since $H \in \mathcal{B}$ by induction hypothesis.
4. Let us prove that the cube $C = \text{Cube}(\underline{\mathbf{v}}, \overline{\mathbf{v}})$ currently added or removed satisfies $C \in \mathcal{B}$ (hence $H \cup C, H \backslash C \in \mathcal{B}$ which will conclude the induction). We already have proven that $\underline{\mathbf{v}} \in (\underline{B}_k)_k$. We prove now that $\overline{\mathbf{v}} \in (\overline{B}_k)_k$ which is searched over the restricted state space $B = A \backslash \{\mathbf{v} \mid \exists \mathbf{v}' \in V : \underline{\mathbf{v}} \leq \mathbf{v}' \leq \mathbf{v}\}$ for $A = X \backslash H \in \mathcal{B}$ or $A = H \backslash X \mathcal{B}$.
   For any $k \in [1, d]$, $\overline{\mathbf{v}} + \mathbf{e}_k \notin B$ so either:
   - $\overline{\mathbf{v}} + \mathbf{e}_k \notin A \in \mathcal{B}$ so $\overline{\mathbf{v}}[k] \in \overline{B}_k$;
   - or $\overline{\mathbf{v}} + \mathbf{e}_k \in \{\mathbf{v} \mid \exists \mathbf{v}' \in V : \underline{\mathbf{v}} \leq \mathbf{v}' \leq \mathbf{v}\}$ but since $\overline{\mathbf{v}}$ is not in the set, there exists $\mathbf{v}' \in V$ such that $\overline{\mathbf{v}}[k] + 1 = \mathbf{v}'[k]$. Since $\mathbf{v}'[k] \in \underline{B}_k$, we have $\overline{\mathbf{v}}[k] \in \overline{B}_k$.

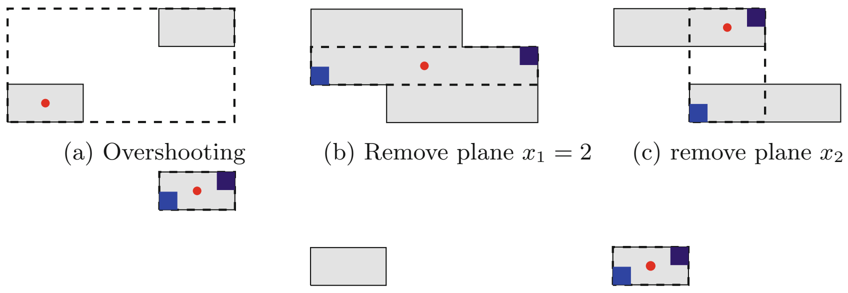
   This concludes the proof.

By combining Proposition 1 and Lemma 2, we summarize the complexity of our overshooting algorithms for a particular target $X = \cup_{i=1}^{n} \text{Cube}(\underline{\mathbf{v}}_i, \overline{\mathbf{v}}_i) \in \mathcal{C}_d$.

Theorem 1 (M+EQ). *Both variants of* LearnCubes *terminates in at most* $(2n)^d$ *iterations, where an iteration requires:*

1. *One equivalence query;*
2. *One corner query, or equivalently, a linear number* $O(\text{size}(X))$ *of membership queries.*

This algorithm terminates in polynomial time, for fixed d, in any representation of target X. In particular, the result holds in the worst-case where the representation of X as a finite union of cubes is minimal. As a matter of fact the presented exponential bound in d is tight: there exists a target $X \in \mathcal{C}$ and a pair of corner and equivalence oracles such that both algorithms terminate in exponential time.


(a) Overshooting (b) Remove plane $x_1 = 2$ (c) remove plane $x_2 = 2$

(d) Remove cube (e) Remove cube

Fig. 4. exponential blow-up, case $d = 2$

Example 1. Consider $X = \{\mathbf{0}, \sum_{i=1}^{d} 2\mathbf{e}_i\}$ composed of two cubes, then by learning $\text{Cube}(\mathbf{0}, \sum_{i=1}^{d} 2\mathbf{e}_i)$, then removing every middle plane of equation $x_k = 1$ for every $k \in [1, d]$, the resulting hypothesis is composed of $2^d - 2$ cubes to remove. An example with $d = 2$ is depicted in Fig. 4.

Whether finite unions of cubes can be learned in polynomial time in the dimension is left as an open problem, that we relate to DNF formula learning over d variables where each term can be interpreted as a cube over $\{0, 1\}^d$.

4 Extensions

In this section we introduce extensions to the overshooting algorithm from Sect. 3.2. While membership and equivalence queries are sufficient for learning finite sets, one natural extension of the minimal learner setting is to introduce a subset oracle [3]:

Definition 5 (Subset Oracle). *Consider some target concept* $X \in \mathcal{C}_d$ *for some concept class* $\mathcal{C} = \bigcup_{d=1}^{\infty} \mathcal{C}_d$ *and let* $\bot, \top \notin \mathcal{D}$ *be two fresh symbols.*

A subset oracle *(SUB) for* X *is a function* $\rho_X : \mathcal{C}_d \to \{\top, \bot\}$, *which outputs* $\top$ *iff* $H \subseteq X$.

The definition is similar to the membership oracle from Definition 1 except the oracle takes a set instead of a single point as input.

4.1 Maximal Cube Oracle

As opposed to the overshooting algorithm, using a subset oracle avoids the overshooting issue, that is to say, we can now search for cubes included in the target X. In order to increase the convergence speed, we nonetheless introduce a maximality criterion on the suitable cubes:

Definition 6 (Maximal Cubes). *A cube* $\mathrm{Cube}(\underline{\mathbf{v}}, \overline{\mathbf{v}})$ *is* maximal *w.r.t.* X *if*

1. $\mathrm{Cube}(\underline{\mathbf{v}}, \overline{\mathbf{v}}) \subseteq X$
2. *For all* i, $\mathrm{Cube}(\underline{\mathbf{v}}, \overline{\mathbf{v}} + \mathbf{e}_i) \nsubseteq X$
3. *For all* i, $\mathrm{Cube}(\underline{\mathbf{v}} - \mathbf{e}_i, \overline{\mathbf{v}}) \nsubseteq X$

Figure 5 provides examples of possible maximal cubes in dimension $d = 2$.

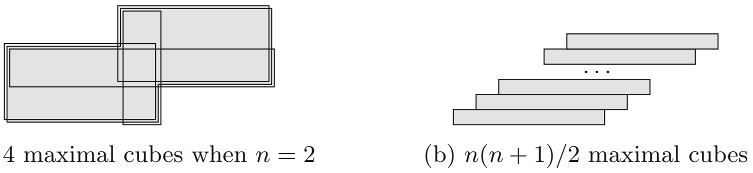

(a) 4 maximal cubes when $n = 2$
(b) $n(n+1)/2$ maximal cubes

Fig. 5. Example of maximal cubes w.r.t. to a union of n cubes

Next, we modify the corner oracle from Sect. 3.1 to use subset queries. Again, we only define the algorithm to find a max corner, the min corner algorithm can be implemented analogously. The algorithm first computes a lower and upper bound for the subsequent binary search. The computation is shown in the function computeMaxBounds. Given a cube defined by its minimal and a maximal corner, the value of coordinate i is increased as long as the resulting cube is still a subset of the target set X. The upper bound $\overline{\mathbf{v}}$ is the first negative reply by the oracle and the lower bound $\underline{\mathbf{v}}$ the last positive response. A binary search is made on $\underline{\mathbf{v}}$ and $\overline{\mathbf{v}}$ in the findMaxIncCorner function.

4.2 Maximal Cube Algorithm

Algorithm 5 presents a procedure that iteratively refines the hypothesis: for any point, the algorithm searches for a maximal cube contained by this point w.r.t. the target and adds it to the hypothesis. One can check that both procedure calls are valid, as $H \subseteq X$ is an invariant. At every iteration the counterexample $\mathbf{v}$ satisfies $\mathbf{v} \in X \setminus H$. The use of the subset oracle ensures that the function FindMaxIncCorner always returns a point $\overline{\mathbf{v}}$ such that $\mathrm{Cube}(\mathbf{v}, \overline{\mathbf{v}}) \subseteq X$. Similarly, the function FindMinIncCorner always returns a corner $\underline{\mathbf{v}}$ such that $\mathrm{Cube}(\underline{\mathbf{v}}, \overline{\mathbf{v}}) \subseteq X$. The resulting cube is then added to the hypothesis, ensuring point $\mathbf{v}$ is never visited again as a counterexample. This entails the termination of the algorithm, in at most $|X|$ iteration of the main loop. A better bound will be explored in Sect. 4.4.

Algorithm 4. Maximal corner of a maximal cube, in $O(\text{size}(X))$ subset queries

```
Ensure: Returned value is a maximal corner of $X$
  function findMaxIncCorner($\underline{\mathbf{v}}, \overline{\mathbf{v}}, \rho_X$)
    for $i \in [1, d]$ do
      $(\underline{b}, \overline{b}) = $ computeMaxBounds($\underline{\mathbf{v}}, \overline{\mathbf{v}}, i, \rho_X$)
      while $\underline{b} \neq \overline{b}$ do
        $m \leftarrow (\underline{b} + \overline{b}) \div 2$
        if $\rho_X(\text{Cube}(\underline{\mathbf{v}}, \overline{\mathbf{v}}[i/m]))$ then
          $\underline{b} \leftarrow m$
        else
          $\overline{b} \leftarrow m$
      $\overline{\mathbf{v}}[i] \leftarrow \overline{b}$
    return $\overline{\mathbf{v}}$
  function computeMaxBounds($\underline{\mathbf{v}}, \overline{\mathbf{v}}, i, \rho_X$)
    $\delta \leftarrow 1$
    while $\rho_X(\text{Cube}(\underline{\mathbf{v}}, \overline{\mathbf{v}} + \delta \cdot \mathbf{e}_i))$ do
      $\delta \leftarrow 2 \cdot \delta$
    return $(\overline{\mathbf{v}}[i] + \delta/2, \overline{\mathbf{v}}[i] + \delta)$
```

Algorithm 5. The maximal cube algorithm

```
  function LearnMaxCube($\rho_X$, $\Psi_X$)
    Let $H \leftarrow \emptyset$
    while $(\mathbf{v} \leftarrow \Psi_X(H)) \neq \top$ do
      Let $\overline{\mathbf{v}} \leftarrow$ findMaxIncCorner($\mathbf{v}, \mathbf{v}, \rho_X$)
      Let $\underline{\mathbf{v}} \leftarrow$ findMinIncCorner($\mathbf{v}, \overline{\mathbf{v}}, \rho_X$)
      $H \leftarrow H \cup \text{Cube}(\underline{\mathbf{v}}, \overline{\mathbf{v}})$
```

4.3 Extension to the Infinite Case

We discuss now one possible extension to the infinite case, namely when cubes are possibly unbounded and may contain infinitely many points.

We adapt our learning formalism to deal with infinite bounds: for the remainder of the section we extend the discrete lattice $\mathbb{Z}^d$ to $(\mathbb{Z} \uplus \{+\infty, -\infty\})^d$ and extend trivially $\leq$ over the newly introduced points. For $\underline{\mathbf{v}}, \overline{\mathbf{v}} \in (\mathbb{Z} \uplus \{+\infty, -\infty\})^d$, the definition of $C = \text{Cube}(\underline{\mathbf{v}}, \overline{\mathbf{v}})$ remains unchanged, in particular $C \subseteq \mathbb{Z}^d$ but may be infinite. The concept class $\mathcal{C}$, hence the domain of oracle functions, is augmented with all finite unions of cubes with (possibly) infinite bounds.

A possible approach to tackle this problem in the minimally adequate teacher (M+EQ) formalism consists in running the overshooting algorithm of Sect. 3 on the state space restricted to some cube of width 2^k centered in $\mathbf{0}$ and gradually increase k if counterexamples outside this restriction are found. This method is discussed in the extended version of the present article [25] but we focus here on a LearnMaxCube adaptation exploiting subset queries (SUB+EQ).

While Algorithm 5 remains unchanged, we need however to adjust the functions FindMaxIncCorner and FindMinIncCorner as those are not able to accelerate the search to infinity. Algorithm 6 achieves this goal by simply overriding the ComputeMaxBounds and ComputeMinBounds subroutines in order to check for possible $+\infty$ and $-\infty$ bounds. Whenever such bound is returned, no further binary search occurs for this coordinate (constant time).

Algorithm 6. Maximal bound overriding, checking for $+\infty$.

```
function computeMaxBounds($\underline{\mathbf{v}}, \overline{\mathbf{v}}, i, \rho_X$)
    if $\rho_X(\text{Cube}(\underline{\mathbf{v}}, \overline{\mathbf{v}}[i/+\infty]))$ then
        return $(+\infty, +\infty)$
    else                ▷ We refer to original ComputeMaxBounds of Algorithm 4
        return Super.ComputeMaxBounds($\underline{\mathbf{v}}, \overline{\mathbf{v}}, i, \rho_X$)
```

4.4 Complexity

Termination of LearnMaxCube was proved using cardinality arguments in Sect. 4.1. These arguments obviously don't apply in the case where the target set is infinite. Moreover, we are interested in finer complexity analysis.

As in Sect. 3.3, we fix a target representation $X = \cup_{i=1}^{n}\text{Cube}(\underline{\mathbf{v}}_i, \overline{\mathbf{v}}_i)$ and study the algorithm complexity with respect to $\sum_{i=1}^{n} \text{size}(\underline{\mathbf{v}}_i) + \text{size}(\overline{\mathbf{v}}_i) \in \mathcal{C}_d$. As some of the vectors $\mathbf{v}$ may contain infinite coordinates, we carefully specify $\text{size}(+\infty) = \text{size}(-\infty) = 1$ and keep the usual definition of $\text{size}(v)$.

Theorem 2 (SUB+EQ). LearnMaxCube *terminates in at most n^{2d} iterations, where an iteration requires:*

1. *One equivalence query;*
2. *One maximal cube query, or equivalently, a linear number $O(\text{size}(X))$ of subset queries.*

Proof. At every iteration, one equivalence query is performed then FindMaxIncCorner and FindMinIncCorner perform a binary search, resulting in a linear number of subset similar (proof similar to Proposition 1).

In order to analyze the number of iterations of the main loop, let us first remark that each added maximal cube is added only once: if we write $\mathbf{v}_k$ the k-th counterexample and C_k the learned maximal cube, then $\mathbf{v}_{k+1} \in X \setminus \cup_{i=1}^{k} C_i$ and $\mathbf{v}_{k+1} \in C_{k+1}$ so $C_{k+1} \neq C_i$ for every $i \in [1, k]$.

The number of iterations is therefore bounded by the number of maximal cubes. We proceed now to bound the number of maximal cubes: Let $C = \text{Cube}(\underline{\mathbf{v}}, \overline{\mathbf{v}})$ be a maximal cube w.r.t. X. For any $k \in [1, d]$ there exist $i, j \in [1, n]$ such that $\underline{\mathbf{v}}[k] = \underline{\mathbf{v}}_i[k]$ and $\overline{\mathbf{v}}[k] = \overline{\mathbf{v}}_j[k]$, hence at most n^2 possibilities for coordinate k.

As in Theorem 1 the number of iterations is polynomial in the number of cubes n but exponential in the dimension d. As opposed to the LearnCubes algorithm, the bound is not tight as the example Fig. 5b provides only a quadratic number of maximal number of cubes. As the maximal cube concept can be related to the notion of *prime implicant*, examples of DNF formula with an exponential of prime implicants (see for example [8]) can be translated into union of cubes with an exponential number of maximal 0–1 cubes.

From a practical perspective, one can nonetheless argue that LearnMaxCube is likely to perform well in practice, by avoiding the overshooting problem mentioned in Example 1 as $H \subseteq X$ is an invariant. In fact, one can easily check that if there are no adjacent[1] cubes, the number of iterations becomes linear.

5 Applications and Experiments

In this section, we describe an immediate application of our learning algorithms to monadic decomposition of quantifier-free Presburger formulas [15,29]. We then report on experimental comparisons between our algorithms and existing methods for the problem.

5.1 Application to Monadic Decomposition

Here we consider quantifier-free linear integer arithmetic formulas without modulo arithmetic:

$$\varphi ::= \alpha_1 \sim \alpha_2 \mid \varphi \wedge \varphi \mid \varphi \vee \varphi,$$

where $\sim\, \in \{\leq, \geq, =\}$, and α_1, α_2 are integer linear combinations of the variables $x_1, \ldots, x_n$, i.e., α_i is of the form $c_0 + \sum_{j=1}^{n} c_j.x_j$, where each $c_i \in \mathbb{Z}$. The formula $\varphi(\bar{x})$ is said to be *satisfiable* (written $\langle \mathbb{Z}; + \rangle \models \varphi$) if there exists an assignment σ of $\bar{x}$ to $\mathbb{Z}$ such that the formula becomes true. Of course, this is just a simple fragment of the first-order theory of integer linear arithmetic and the notion of $\langle \mathbb{Z}; + \rangle \models \varphi$ can be defined in the same way even with quantifiers [14,18]. A formula φ is said to be *monadic* if it has only one variable. Every monadic formula $\varphi(x)$ in this fragment can be easily transformed into a union integer intervals of the form: (1) $l \leq x \wedge x \leq u$ where $l, u \in \mathbb{Z}$, (2) $l \leq x$ where $l \in \mathbb{Z}$, (3) $x \leq u$ where $u \in Z$, or (4) $\top$ or $\bot$.

A *monadic decomposition* [29] of a formula $\varphi(\bar{x})$ is a boolean combination $\psi(\bar{x})$ of monadic formulas that is equivalent to φ over the theory, i.e., $\langle \mathbb{Z}; + \rangle \models \forall \bar{x}(\varphi \leftrightarrow \psi)$. Of course, not all formulas admit a monadic decomposition (e.g., $x = y$). It was shown in [15] that deciding if a formula in the theory be monadically decomposable is coNP-complete[2]. Veanes *et al.* [29] provides a generic semi-decision procedure for computing a monadic decomposition of a quantifier-free formula as an if-then-else formula that is applicable to pretty much all theories considered in SMT. Despite its genericity, the procedure runs rather well, e.g., as the authors showed on their benchmarking in [29].

[1] Two cubes C_1 and C_2 are *adjacent* if $\min \left\{ \sum_i |\mathbf{v}_1[i] - \mathbf{v}_2[i]| \mid \mathbf{v}_1 \in C_1, \mathbf{v}_2 \in C_2 \right\} \leq 1$.

[2] The proof in [15] uses modulo constraints to show that monadic decomposition of a two-variable formula $\varphi(x, y)$ is coNP-complete. Modulo constraints could be easily removed by allowing more integer variables.

The application of our learning algorithms to computing monadic decomposition arises from the following observation. Since each monadic decomposition can be transformed into DNF, a monadic decomposition of a formula $\varphi(\bar{x})$ over $\langle \mathbb{Z}; + \rangle$ can be constructed as a finite union of (possibly infinite) hypercubes, where an infinite hypercube arises when a variable is either not bounded from above or not bounded from below (or both). Conversely, a finite union H of possibly infinite hypercubes can also be easily transformed into a boolean combination of monadic formulas φ_H. For example, the formula $(0 \leq x \leq 5 \wedge 3 \leq y \leq 10) \vee (8 \leq x)$ corresponds to the union of hypercubes $\mathrm{Cube}((0,3),(5,10)) \cup$ $\mathrm{Cube}((8,-\infty),(+\infty,+\infty))$. Furthermore, all relevant oracles admit a straightforward implementation:

- A membership query $\bar{v}$ requires checking $\langle \mathbb{Z}; + \rangle \models \varphi(\bar{v})$, which can be checked in polynomial-time because φ is quantifier-free.
- An equivalence query H can be reduced to checking
$$\langle \mathbb{Z}; + \rangle \models (\varphi_H \wedge \neg\varphi) \vee (\varphi \wedge \neg\varphi_H).$$
This is a single satisfiability check of quantifier-free integer-linear arithmetic formula, for which highly-optimized solvers exist (e.g., Z3 [26]).
- A subset query H can similarly be reduced to checking
$$\langle \mathbb{Z}; + \rangle \models (\varphi_H \wedge \neg\varphi).$$
This is also a single satisfiability check over $\langle \mathbb{Z}; + \rangle$.

This allows us to apply both of our learning algorithms to the problem.

Monadic decomposition has numerous applications including quantifier elimination [29], string solving [15], and symbolic finite automata/transducers [13,29], among others. In the following example we illustrate how our learning algorithm(s) could be applied to improving quantifier elimination for the theory of linear integer arithmetic.

Example 2. Consider a formula of the form $\forall \bar{x} \exists y\ \varphi(\bar{x}, \bar{y})$, where φ is a formula in linear integer arithmetic without modulo constraints. Suppose that φ is monadically decomposable, and is equivalent to the formula $\bigvee_{i=1}^{n} D_i(\bar{x}, \bar{y})$, where each D_i is a disjunction of monadic predicates over the variables $\bar{x} \cup \bar{y}$. We assume w.l.o.g. that each D_i is satisfiable. Then, this formula is equisatisfiable (over linear integer arithmetic) to $\psi := \forall \bar{x} \left(\bigvee_{i=1}^{n} D_i(\bar{x}, \bar{c}_i) \right)$, where $\bar{y}$ in D_i are replaced by *fresh* constants $\bar{c}_i$ (i.e. two distinct D_i, D_i' use different constants). This can be proven by a simple application of skolemization, and observing that each occurrence of $f(\bar{x})$ in any disjunct is of the form $a < f(\bar{x}) < b$, where $a \in \{-\infty\} \cup \mathbb{Z}$ and $b \in \mathbb{Z} \cup \{\infty\}$, implying that $f(\bar{x})$ can be replaced by a single constant, which does not depend on $\bar{x}$. Finally, let D_i' be the conjuncts in D_i only involving variables in $\bar{x}$. Checking that ψ is true reduces to checking satisfiability of $\bigwedge_{i=1}^{n} \neg D_i'$.

To make this example concrete, we consider the formula $\forall x \exists y (x \geq 0 \rightarrow x + y \geq 5 \wedge y \geq 0)$. A monadic decomposition of the quantifier-free part is $x < 0 \vee \bigvee_{i=0}^{5} (x \geq i \wedge y \geq 5 - i)$. Therefore, checking the above formula can be reduced to satisfiability of $x \geq 0 \wedge \bigwedge_{i=0}^{5} x < i$ which is not satisfiable.

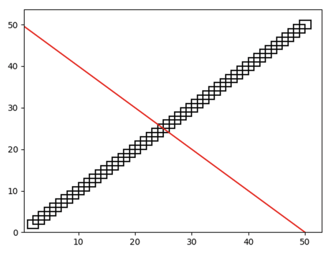
5.2 Experiments

In order to assess the performance of the algorithms FindMaxCorner and FindMinCorner respectively introduced in Sect. 3 and Sect. 4, we consider prototype implementations. The following prototypes and experiments can be found in [24].

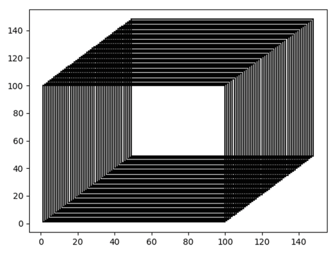
Variants. Although the methods were presented with binary search strategies in mind, we also implemented a more naive unary search procedure to obtain the corners. As later noticed in the experiments, unary search may be preferred for very small cubes and performs especially well for cubes which are based 0–1 integer programs, while binary search achieves better performance for larger cubes. Consequently, we refer to a third variation of the algorithm called "optimized", combining unary search for small instances and binary search for large values. More precisely two variants of the overshooting algorithm from Sect. 3 and three variants of the max cubes algorithm from Sect. 4 are presented, called respectively *overshoot_unary* and *overshoot_binary* and *max_unary*, *max_binary* and *max_optimized*.

Tool Comparison. Evaluation is performed against a generic monadic decomposition procedure *mondec*$_1$ from [29] by Veanes et al., which works over an arbitrary base theory and outputs an if-then-else formula, which could be exponentially more succinct than a formula in DNF. The algorithm, which exploits the python-Z3 framework [26], uses a kind of a decision tree search heuristics to split the input into monadic predicates.

Implementation. Similarly to mondec$_1$, our prototype is implemented in python using the python-Z3 framework, but is specialized in handling linear integer arithmetic formula, and that outputted formulas will be in DNF, unlike mondec$_1$. For monadic decomposition applications, oracles queries are converted to appropriate Z3 satisfaction queries since a (possibly non-monadic) representation of the target set is already known.



(a) 50 overlapping cubes and the diagonal $x + y = 50$.



(b) 100 big Cubes.

Fig. 6. Benchmarks for $\mathbb{Z}^2$.

5.3 Benchmark Suite

Our benchmark suite is restricted to the problem of monadic decomposition of linear integer arithmetic, and its purpose is to stress-test our learning algorithms and mondec$_1$ against various kinds of "extreme conditions". The suite consists of six classes of monadically decomposable example formulas, which were constructed to test five features (see below). Note that the given formulas themselves might contain non-monadic predicates.

The five features (left to right in Table 1) represent the presence of (1) a large amount of cube overlaps, (2) a large number of cubes, (3) a large cube, (4) large dimension, and (5) an unbounded cube. We hypothesized that these five features play important roles in how fast the algorithms perform, which are indeed validated in our experimental results. The six classes of formulas are elaborated below.

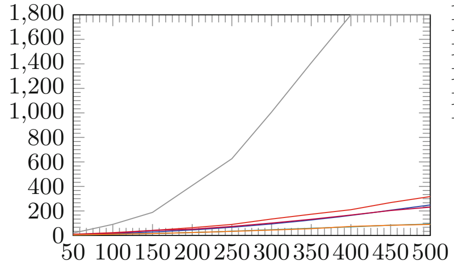
Table 1. Features of conducted benchmarks. A "+" (resp. "-") indicates a high (resp. low) presence of a feature.

	Overlap	# Cubes	|Cube|	Dimension	Unbounded
(a)	+	+	−	−	−
(b)	+	−	−	+	−
(c)	+	+	−	−	−
(d)	+	+	+	−	−
(e)	−	+	−	−	−
(f)	+	+	+	−	+

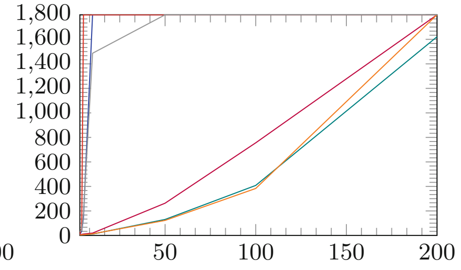
(a) *K Diagonal Restricted* consists of K overlapping cubes of length and width 2 and one diagonal as shown in Fig. 6a. The cubes overlap with at most two other cubes and stack up diagonally. The algorithms need to return all the cubes left of the diagonal.
(b) *10 cubes in* $\mathbb{Z}^d$ consists of $K = 10$ overlapping cubes of size 2^d stacking up diagonally similar to the benchmark K Diagonal Restricted without diagonal restriction.
(c) *K Diagonal Unrestricted* is a variation of Fig. 6a where the algorithms need to return *all* the cubes and all the points on the diagonal.
(d) *K Big Overlapping Cube* is a benchmark testing large cubes as depicted in Fig. 6b. It consists of K overlapping cubes of length and width 100 and are overlapping and stacking up diagonally like the benchmark K Diagonal Restricted.
(e) *K Diagonal* is built as the set of points along the diagonal $x = y \leq K$.
(f) *Example* 2 is generalized to any $K \in \mathbb{N}$ by $x \geq 0 \rightarrow x + y \geq K \wedge y \geq 0$. Its unbounded nature makes it tractable by max_optimized and mondec$_1$ only.

5.4 Results

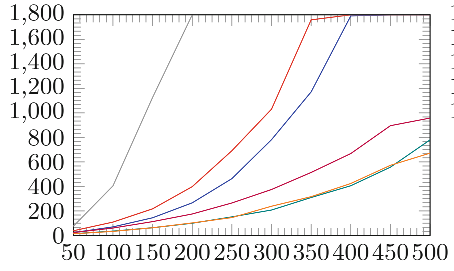
Experiments were conducted on an AMD Ryzen 5 1600 Six-Core CPU with 16 GB of RAM running on Windows 10. The results are summarized in Fig. 7 where each graph represents one benchmark comparing the run times of each algorithm.



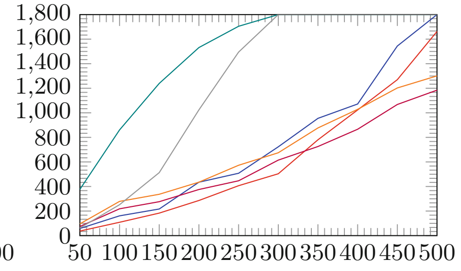
(a) Benchmark on K Diagonal Restricted in $\mathbb{Z}^2$.
The x-axis encodes the amount of cubes K.



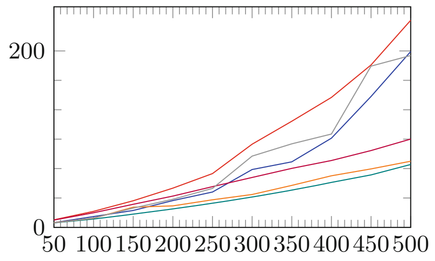
(b) Benchmark on 10 cubes in $\mathbb{Z}^d$.
The x-axis encodes the dimension d.



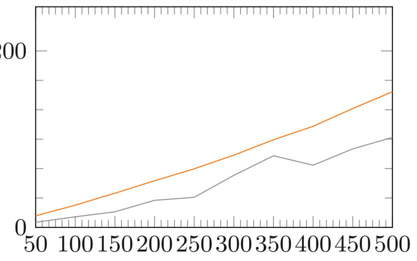
(c) Benchmark on K Diagonal Unrestricted in $\mathbb{Z}^2$.
The x-axis encodes the amount of cubes K.



(d) Benchmark on K Big Cubes in $\mathbb{Z}^2$.
The x-axis encodes the amount of cubes K.



(e) Benchmark on K Diagonal in $\mathbb{Z}^2$.
The x-axis encodes the maximal value for x and y.



(f) Benchmark on Example 2 in $\mathbb{Z}^2$.
The x-axis encodes parameter K.

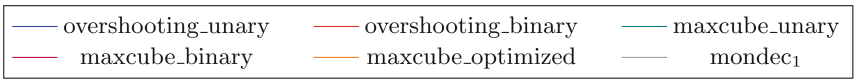


Fig. 7. Benchmark results. The y-axis encodes the time in seconds. The timeout is set to1800 s.

The overshooting phenomenon can be observed in Fig. 7c and Fig. 7e with its quadratic shape, as $d = 2$. In Fig. 7b, the running time quickly diverges as d increases, as anticipated by Example 1.

When the considered cubes are small, as in Fig. 7a and Fig. 7c, the unary search algorithms outperform their binary counterparts, meaning the few additional queries made by the binary search are more costly than a direct enumeration. The optimized variant is therefore a good compromise in all cases.

Figure 7d depicts a benchmark with many large cubes for a fixed dimension. While the impact of the overshooting phenomenon remains contained, the maxcube unary search variant is particularly slow. This can be explained by the size of the cubes making unary search inefficient, combined with the already expensive cost of every single inclusion query.

The mondec$_1$ algorithm is comparable to the overshooting algorithms in Fig. 7e. It also performs particularly well in Fig. 7f, which we conjecture is due to the conciseness of the solution in if-then-else form used by mondec$_1$.

Overall, the maxcube algorithm in its optimized form is the most stable algorithm for this benchmark set and should be preferred when an inclusion oracle is available. The extra cost of these queries are here taken into account and remain affordable when implemented with Z3 queries.

6 Conclusion and Future Work

We have presented a polynomial-time algorithm in Angluin's exact learning framework using membership and equivalence for learning a finite union of rectilinear cubes over $\mathbb{Z}^d$ over any fixed dimension d. By considering an additional subset oracle, learning possibly infinite cubes can be achieved with the same complexity, but a simpler and faster learning algorithm in practice. The technique enables the introduction of auxiliary oracles, namely the corner (resp. maximal cube) oracle when a membership (resp. subset) oracle is provided. While oracles for subset queries tend to be difficult to implement, this turns out not to be the case for our proposed application of computing monadic decompositions of quantifier-free integer linear arithmetic formulas without modulo constraints, which is successfully solved by our algorithm.

We mention three future research directions. First, extensions to modulo operations could be explored, by encoding periodicity on d additional coordinates and providing adequate oracles on the encoded target. A second direction consists in applying these learning techniques to the verification of systems by learning invariants which are monadically decomposable in a small number of cubes. Lastly, one promising direction to further improve our algorithms is to investigate how to leverage if-then-else formula representations as used in mondec$_1$ [29], which could be exponentially more succinct than formulas in DNF.

Acknowledgements. The authors would like to thank the reviewers and program chairs for their thoughtful comments and suggestions to improve the presentation, as well as the Simons Institute for the Theory of Computing and Christoph Haase for fruitful discussions.

References

1. Abe, N.: Characterizing PAC-Learnability of semilinear sets. Inf. Comput. **116**(1), 81–102 (1995)
2. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**(2), 87–106 (1987)
3. Angluin, D.: Queries and concept learning. Mach. Learn. **2**(4), 319–342 (1988)
4. Angluin, D., Kharitonov, M.: When won't membership queries help? J. Comput. Syst. Sci. **50**(2), 336–355 (1995)
5. Barceló, P., Hong, C., Le, X.B., Lin, A.W., Niskanen, R.: Monadic decomposability of regular relations. In: 46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, 9–12 July 2019, Patras, Greece, pp. 103:1–103:14 (2019). https://doi.org/10.4230/LIPIcs.ICALP.2019.103
6. Beyene, T.A., Chaudhuri, S., Popeea, C., Rybalchenko, A.: A constraint-based approach to solving games on infinite graphs. In: Jagannathan, S., Sewell, P. (eds.) The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20–21, 2014, pp. 221–234. ACM (2014). https://doi.org/10.1145/2535838.2535860
7. Carton, O., Choffrut, C., Grigorieff, S.: Decision problems among the main subfamilies of rational relations. ITA **40**(2), 255–275 (2006). https://doi.org/10.1051/ita:2006005
8. Chandra, A.K., Markowsky, G.: On the number of prime implicants. Discrete Math. **24**(1), 7–11 (1978). https://doi.org/10.1016/0012-365X(78)90168-1
9. Chen, Y.-F., Farzan, A., Clarke, E.M., Tsay, Y.-K., Wang, B.-Y.: Learning minimal separating DFA's for compositional verification. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 31–45. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00768-2_3
10. Chen, Y., Hong, C., Lin, A.W., Rümmer, P.: Learning to prove safety over parameterised concurrent systems. In: 2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2–6 2017, pp. 76–83 (2017). https://doi.org/10.23919/FMCAD.2017.8102244
11. Chen, Z.: An optimal algorithm for proper learning of unions of two rectangles with queries. In: Du, D.-Z., Li, M. (eds.) COCOON 1995. LNCS, vol. 959, pp. 334–343. Springer, Heidelberg (1995). https://doi.org/10.1007/BFb0030848
12. Chen, Z., Ameur, F.: The learnability of unions of two rectangles in the two-dimensional discretized space. J. Comput. Syst. Sci. **59**(1), 70–83 (1999). https://doi.org/10.1006/jcss.1999.1621
13. D'Antoni, L., Veanes, M.: The power of symbolic automata and transducers. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 47–67. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_3
14. Haase, C.: A survival guide to presburger arithmetic. ACM SIGLOG News **5**(3), 67–82 (2018). https://doi.org/10.1145/3242953.3242964
15. Hague, M., Lin, A.W., Rümmer, P., Wu, Z.: Monadic decomposition in integer linear arithmetic. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) IJCAR 2020. LNCS (LNAI), vol. 12166, pp. 122–140. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51074-9_8
16. Kearns, M.J., Vazirani, U.V.: An Introduction to Computational Learning Theory. MIT Press, Cambridge, MA, USA (1994)
17. Kopczynski, E., To, A.W.: Parikh images of grammars: complexity and applications. In: 2010 25th Annual IEEE Symposium on Logic in Computer Science, pp. 80–89 (2010). https://doi.org/10.1109/LICS.2010.21

18. Kroening, D., Strichman, O.: Quantified formulas. Decision Procedures. TTC-SAES, pp. 199–227. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-50497-0_9
19. Libkin, L.: Variable independence for first-order definable constraints. ACM Trans. Comput. Log. **4**(4), 431–451 (2003). https://doi.org/10.1145/937555.937557
20. Lin, A.W., Rümmer, P.: Liveness of randomised parameterised systems under arbitrary schedulers. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 112–133. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_7
21. Löding, C., Madhusudan, P., Neider, D.: Abstract learning frameworks for synthesis. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 167–185. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_10
22. Maass, W., Turán, G.: Algorithms and lower bounds for on-line learning of geometrical concepts. Mach. Learn. **14**(1), 251–269 (1994). https://doi.org/10.1023/A:1022653511837
23. Markgraf, O., Hong, C.-D., Lin, A.W., Najib, M., Neider, D.: Parameterized synthesis with safety properties. In: Oliveira, B.C.S. (ed.) APLAS 2020. LNCS, vol. 12470, pp. 273–292. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-64437-6_14
24. Markgraf, O., STAN, D., Lin, A.W.: (Artifact) Learning Union of Integer Hypercubes with Queries (with applications to monadic decomposition) (2021). https://doi.org/10.5281/zenodo.4742954
25. Markgraf, O., Stan, D., Lin, A.W.: Learning union of integer hypercubes with queries (technical report) (2021). https://arxiv.org/abs/2105.13071
26. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
27. Solar-Lezama, A.: Program Synthesis by Sketching. Ph.D. thesis, University of California at Berkele (2008)
28. Takada, Y.: Learning semilinear sets from examples and via queries. Theor. Comput. Sci. **104**(2), 207–233 (1992)
29. Veanes, M., Bjørner, N., Nachmanson, L., Bereg, S.: Monadic decomposition. J. ACM **64**(2), 14:1–14:28 (2017). https://doi.org/10.1145/3040488

Interpolation and Model Checking for Nonlinear Arithmetic

Dejan Jovanović(✉) and Bruno Dutertre

SRI International, Menlo Park, USA

Abstract. We present a new model-based interpolation procedure for satisfiability modulo theories (SMT). The procedure uses a new mode of interaction with the SMT solver that we call *solving modulo a model*. This either extends a given partial model into a full model for a set of assertions or returns an explanation (a model interpolant) when no solution exists. This mode of interaction fits well into the model-constructing satisfiability (MCSAT) framework of SMT. We use it to develop an interpolation procedure for any MCSAT-supported theory. In particular, this method leads to an effective interpolation procedure for nonlinear real arithmetic. We evaluate the new procedure by integrating it into a model checker and comparing it with state-of-art model-checking tools for nonlinear arithmetic.

Keywords: Satisfiability modulo theories · Craig interpolation · Nonlinear arithmetic

1 Introduction

Craig interpolation is one of the central reasoning tools in modern verification algorithms. Verification techniques such as model checking rely on Craig interpolation [11,39] as a symbolic learning oracle that drives abstraction refinement and invariant inference. Interpolation has been studied for many fragments of first-order logic that are useful in practice, such as linear arithmetic [23], uninterpreted functions [9,37], arrays [25,38], and sets [32]. In these fragments, a typical interpolation procedure constructs interpolants by traversing the clausal proof of unsatisfiability provided by an SMT solver [26,34,41] while performing interpolation locally at proof nodes. A major missing piece in the class of fragments supported by interpolating SMT solvers is nonlinear arithmetic,[1] as the

[1] By nonlinear arithmetic we mean Boolean combination of arithmetic constraints over arbitrary-degree polynomials.

This material is based upon work supported by the Defense Advanced Research Project Agency (DARPA) and Space and Naval Warfare Systems Center, Pacific (SSC Pacific) under Contract No. N66001-18-C-4011, and the National Science Foundation (NSF) grant 1816936. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA, SSC Pacific, or the NSF.

complex reasoning required for nonlinear arithmetic makes fine-grained symbolic proof generation extremely difficult.

We present an approach to interpolation that is driven by models rather than proofs. Given a pair of formulas A and B such that $A \wedge B$ is unsatisfiable, an interpolant is a formula I that is implied by A and inconsistent with B. Recent model-based decision procedures, specifically the ones developed within the MCSAT [13,28] framework for SMT, are internally naturally interpolating. But, rather than interpolating two formulas, they provide a way to interpolate a set of constraints against a partial model. We capitalize on this internal ability, and extend it so that a formula A can be checked and interpolated against a partial model (*model interpolation*). This is closely related to the ability of modern SAT solvers to perform solving modulo assumptions [17], a technique that can also been used to provide interpolation capabilities in finite-state model checking [3].

We take advantage of model interpolation to build a formula-interpolation procedure through a simple idea: we can compute an interpolant of formulas A and B by iteratively interpolating (and refuting) all models of B with model interpolants from A. We develop the interpolation procedure within the MCSAT framework. This immediately allows us to generate interpolants for any theory supported by the framework. As MCSAT provides efficient complete solvers for nonlinear real arithmetic [27,29], we develop the first complete interpolation procedure for real nonlinear arithmetic.

To show that this new interpolation procedure is an effective tool that can be used on real-world problems, we integrate it into a model checker that uses interpolation for inferring k-inductive invariants. We evaluate this model checker on a set of industrial benchmarks. Our evaluation shows that the new procedure is highly effective, both in terms of speed, and the ability to support the model checker in its quest for counter-examples and invariants.

Outline. Section 2 gives background on SMT, interpolation, and nonlinear arithmetic. Section 3 presents solving modulo a model and model interpolation, and develops the general interpolation procedure. In Sect. 4, we discuss the particular needs of nonlinear arithmetic. In Sect. 5 we evaluate our implementation on nonlinear model-checking problems. We conclude in Sect. 6 and provide future research directions.

2 Background

We assume that the reader is familiar with the usual notions and terminology of first-order logic and model theory (for an introduction see, e.g., [1]).

Nonlinear Arithmetic. As usual, we denote the ring of integers with $\mathbb{Z}$ and the field of real numbers with $\mathbb{R}$. Given a vector of variables $\boldsymbol{x}$ we denote the set of polynomials with integer coefficients and variables $\boldsymbol{x}$ as $\mathbb{Z}[\boldsymbol{x}]$. A polynomial $f \in \mathbb{Z}[\boldsymbol{y}, x]$ is of the form

$$f(\boldsymbol{y}, x) = a_m \cdot x^{d_m} + a_{m-1} \cdot x^{d_{m-1}} + \cdots + a_1 \cdot x^{d_1} + a_0,$$

where $0 < d_1 < \cdots < d_m$, and the coefficients a_i are polynomials in $\mathbb{Z}[\boldsymbol{y}]$ with $a_m \neq 0$. We call x the *top variable* and the highest power d_m is the *degree* of the polynomial f. As usual, we denote with $f^{(k)}$ the k-th derivative of f in its top variable. A number $\alpha \in \mathbb{R}$ is a *root of the polynomial* $f \in \mathbb{Z}[x]$ if $f(\alpha) = 0$.

A *polynomial constraint* C is a constraint of the form $f \triangledown 0$ where f is a polynomial and $\triangledown \in \{<, \leq, =, \geq, >\}$. If the polynomial $f = f(x)$ is univariate then we also say that C is univariate. An atom is either a polynomial constraint or a Boolean variable, and formulas are defined inductively with the usual Boolean connectives ($\wedge$, $\vee$, $\neg$). The symbols $\top$ and $\bot$ denote true and false, respectively. In addition to the basic polynomial constraints, we will also be working with extended polynomial constraints. An *extended polynomial constraint* F is of the form $x \triangledown_r \mathsf{root}(f, k, x)$ where $f \in \mathbb{Z}[\boldsymbol{y}, x]$ and $\triangledown_r \in \{<_r, \leq_r, =_r, \geq_r, >_r\}$. The semantics of this predicate is the following: Given an assignment that gives real values $\boldsymbol{v}$ to the variables $\boldsymbol{y}$, then the roots of $f(\boldsymbol{a}, x)$ can be ordered over $\mathbb{R}$. If the polynomial $f(\boldsymbol{a}, x)$ has at least k real roots and α_k is the k-th smallest root[2] then the constraint is equivalent to $x \triangledown \alpha_k$. Otherwise, the constraint evaluates to $\bot$. For example, the constraint $x < \mathsf{root}(x^2 - 2, 2, x)$ represents $x < \sqrt{2}$.

Given a formula $F(\boldsymbol{x})$ we say that a type-consistent variable assignment $M = \{\boldsymbol{x} \mapsto \boldsymbol{a}\}$ satisfies F if the formula F evaluates to $\top$ in the standard semantics of Booleans and reals. We call M a model of F and denote this with $M \models F$. If there is such a variable assignment, we say that F is *satisfiable*, otherwise it is *unsatisfiable*. If two models M_1 and M_2 agree on the values of their common variables, we denote the model that combines M_1 and M_2 with $M_1 \cup M_2$.

Definition 1 (Craig interpolant). *Given two formulas $A(\boldsymbol{x}, \boldsymbol{y})$ and $B(\boldsymbol{y}, \boldsymbol{z})$ such that $A \wedge B$ is unsatisfiable, a* Craig interpolant *is a formula $I(\boldsymbol{y})$ such that $A \Rightarrow I$ and $I \Rightarrow \neg B$. We call the pair (A, B) an* interpolation problem.

Model Checking. A *state-transition system* is a pair $\mathfrak{S} = \langle I, T \rangle$, where $I(\boldsymbol{x})$ is a state formula describing the initial states and $T(\boldsymbol{x}, \boldsymbol{x}')$ is a state-transition formula describing the system's evolution. Given a state formula P (*the property*), we want to determine whether all reachable states of $\mathfrak{S}$ satisfy P. If this is the case, P is an *invariant* of $\mathfrak{S}$. If P is not invariant, there is a concrete trace of the system, called a *counter-example*, that reaches $\neg P$.

The direct way to prove that a property P is an invariant of $\mathfrak{S}$ is to show that it is inductive. This requires showing that P holds in the initial states: $I \Rightarrow P$, and that it is preserved by transitions: $P(\boldsymbol{x}) \wedge T(\boldsymbol{x}, \boldsymbol{x}') \Rightarrow P(\boldsymbol{x}')$. As most invariants are not inductive, a key problem in model checking is to find am *inductive strengthening* of P, that is, a property P' such that $P' \Rightarrow P$ and P' is inductive.

[2] For example, $x^2 - 2$ has two roots. The first root $-\sqrt{2}$ is the smallest of the two and the second root is $\sqrt{2}$.

Example 1 (Cauchy–Schwarz inequality). We can frame the Cauchy–Schwarz inequality as a model-checking problem in nonlinear arithmetic. The inequality is the following

$$(\sum_{i=1}^{n} x_i y_i)^2 \leq (\sum_{i=1}^{n} x_i^2)(\sum_{i=1}^{n} y_i^2). \tag{1}$$

As shown in [21], many inequalities that involve a discrete parameter (such as n above) can be converted to model-checking problems. For inequality (1), we construct the transition system $\mathfrak{S}_{cs} = \langle I, T \rangle$ where

$$I \equiv (S_1 = 0) \wedge (S_2 = 0) \wedge (S_3 = 0),$$
$$T \equiv (S_1' = S_1 + xy) \wedge (S_2' = S_2 + x^2) \wedge (S_3' = S_3 + y^2).$$

The variables S_1, S_2, S_3 correspond to the sums in (1) in order. The two variables x and y of $\mathfrak{S}_{cs}$ model the variables x_i and y_i from (1) in each iteration of $\mathfrak{S}_{cs}$. Proving the inequality amounts to showing that property $P_{cs} \equiv (S_1^2 \leq S_2 S_3)$ is an invariant of $\mathfrak{S}_{cs}$. Property P_{cs} is not inductive on its own, but property $P_{cs}' \equiv P_{cs} \wedge (S_2 \geq 0) \wedge (S_3 \geq 0)$ is an inductive strengthening of P_{cs}.

Many modern model-checking techniques, specifically those based on SMT solving, use interpolation as a tool to automatically infer inductive invariants. In this context, an interpolant can be used to over-approximate a transition in the context of a spurious counter-example. In addition to interpolation, the recent class of techniques broadly termed *property-directed reachability* (PDR) (e.g., [24,30,33]), relies on *model generalization*, which converts a concrete counter-example state into a set of counter-examples.

Definition 2 (Generalization). *Given a formula $F(\boldsymbol{x}, \boldsymbol{y})$ such that F is true in a model M, we call a formula $G(\boldsymbol{x})$ a* generalization of M *if $G(\boldsymbol{x})$ is true in M and $G(\boldsymbol{x}) \Rightarrow \exists \boldsymbol{y} \,.\, F(\boldsymbol{x}, \boldsymbol{y})$.*

A PDR model-checking procedure for nonlinear arithmetic requires both an interpolation and a generalization procedure.

3 SMT Modulo Models and Interpolation

SMT solvers typically provide an API to assert formulas and to check the satisfiability of asserted formulas. We denote with SOLVER::ASSERT(F) the solver method that adds the formula F to the set of assertions to be checked by the solver. We denote with SOLVER::CHECK() the solver method for checking satisfiability, with the following contract.

SOLVER::CHECK(): Check satisfiability of asserted formulas A and

1. if there is a model M such that $M \vDash A$, return $\langle \textbf{sat}, M \rangle$;
2. otherwise return $\langle \textbf{unsat}, \emptyset \rangle$.

In this contract, the solver does not return any form of inconsistency certificate when the assertions are unsatisfiable.[3] We generalize the standard SMT satisfiability checking to *SMT modulo models* as follows.

SOLVER::CHECK(M_0): Check satisfiability of asserted formulas A and

1. if there is a model $M \supseteq M_0$ such that $M \models A$, return $\langle \mathbf{sat}, M, \top \rangle$;
2. otherwise return $\langle \mathbf{unsat}, \emptyset, I \rangle$ where $A \Rightarrow I$ and $M_0 \models \neg I$.

SMT modulo models allows one to check that a formula is satisfiable modulo a partial model M_0, by seeking a solution that extends M_0. If there is no such solution, the formula I returned as the certificate of unsatisfiability is a *model interpolant*: it is implied by the assertions and inconsistent with M_0 (i.e., I evaluates to $\bot$ in the model M_0). If we restrict ourselves to Boolean formulas, SMT modulo models reduces exactly to solving modulo assumptions [17] used in the SAT community. Although this idea is not completely new, it is the first time that it is used for interpolation in SMT, as far as we know.

3.1 Interpolation

Before diving into an approach that can support the above mode of satisfiability checking, we first show how model interpolation can be used to devise a general interpolation method.

Algorithm 1: INTERPOLATE(A, B)

```
1  $S_A$.assert($A$) ;
2  $S_B$.assert($B$) ;
3  $I \leftarrow \top$ ;
4  while true do
5      $\langle r_B, M_B \rangle \leftarrow S_B$.check() ;
6      if $r_B = \mathbf{unsat}$ then
7          return $\langle \mathbf{unsat}, I \rangle$
8      $\langle r_A, M_A, I_A \rangle \leftarrow S_A$.check($M_B$) ;
9      if $r_A = \mathbf{sat}$ then
10         return $\langle \mathbf{sat}, M_A \cup M_B \rangle$
11     $I \leftarrow I \wedge I_A$ ;
12     $S_B$.assert($I_A$)
```

Algorithm 1 shows the pseudocode of a procedure that checks satisfiability and interpolates two formulas A and B. The basic idea is simple: we enumerate

[3] Some solvers support proof generation. While proofs are fundamentally important, we are interested in certificates that can always be computed and are useful in supporting further analysis. For example, proof generation for nonlinear arithmetic is still a hard open problem.

models M_k of the formula B, and refute each model M_k with a model interpolant I_k from A. If the process converges and returns **unsat**, we collect the model interpolants and construct the final interpolant $I = \bigwedge I_k$. Each interpolant I_k is implied by A because it is a model interpolant, so $A \Rightarrow I$. Each model of B is refuted by some model interpolant I_k, and so $I \Rightarrow \neg B$. On the other hand, if the process returns **sat**, the procedure has found a common model for A and B. The procedure above is model-driven and modular, in that it checks the formulas A and B independently while only communicating models (from B to A) and model interpolants (from A to B).

Lemma 1 (Correctness). *If* INTERPOLATE(A, B) *returns* $\langle \mathbf{unsat}, I\rangle$ *then* $A \wedge B$ *is unsatisfiable and* I *is an interpolant for* (A, B)*. If* INTERPOLATE(A, B) *returns* $\langle \mathbf{sat}, M\rangle$ *then* $A \wedge B$ *is satisfiable and* M *is a model of both* A *and* B*.*

Note that Lemma 1 does not claim termination of the procedure. Termination depends on the ability of model interpolation to produce a finite number of model interpolants that can eliminate a potentially infinite number of models.

A naive approach to check a formula $A(\boldsymbol{x}, \boldsymbol{y})$ for satisfiability modulo a model $M_0 = \{\boldsymbol{y} \mapsto \boldsymbol{v}\}$ is to use an interpolating SMT solver. First, encode the model into a formula $F_M \equiv \bigwedge(y_i = v_i)$. If the formula $A \wedge F_M$ is satisfiable in a model M, so is A and $M \supseteq M_0$. Otherwise, we compute the interpolant I of A and F_M. This naive approach satisfies the requirements of SOLVER::CHECK(M_0), but it is limited for the following reasons. First, theories such as nonlinear arithmetic have complex models and the formula F_M can be hard to express. As an example, $x \mapsto \sqrt{2}$ can only be expressed by extending the constraint language to support algebraic numbers, or by using additional assertions such as $(x^2 = 2) \wedge (x > 0)$. More important, traditional interpolation provides no guarantees in terms of convergence of a sequence of interpolation problems. For example, as already noted in [42], $\neg F_M$ would be a valid interpolant for A and F_M. But such an interpolant only eliminates a single model and could, in general, lead to non-termination of INTERPOLATE(A, B). To tackle this issue, we require that the procedure SOLVER::CHECK() produces interpolants general enough to disallow such infinite sequences of model interpolants. We do this by adopting the convergence approach and terminology of [42] to model interpolation as follows.

Definition 3 (Model Interpolation Sequence). *Given a formula* $A(\boldsymbol{x}, \boldsymbol{y})$*, a sequence of models* (M_k) *of* $\boldsymbol{y}$*, and a sequences of formulas* (I_k) *over* $\boldsymbol{y}$*, we call* (I_k) *a* model interpolation sequence *for* A *and* (M_k) *if for all* k *it holds that*

1. *M_k is consistent with $\bigwedge_{i<k} I_i$;*
2. *M_k is inconsistent with A;*
3. *I_k is a model interpolant between A and M_k.*

Definition 4 (Finite Convergence). *We say that* SOLVER::CHECK() *has the* finite convergence property *if it does not allow infinite model interpolation sequences.*

Lemma 2 (Termination). *If* SOLVER::CHECK() *has the finite convergence property, then* INTERPOLATE(A, B) *always terminates.*

3.2 SMT Modulo Models with MCSAT

We build a procedure for solving SMT modulo models by modifying the satisfiability checking procedure of MCSAT. The MCSAT method for SMT solving was introduced in [13,28] and further extended in [27]. We give a brief overview of the MCSAT terminology and mechanics, and we describe the satisfiability procedure. We emphasize modifications to the original MCSAT procedure that are needed for solving SMT modulo models.

The architecture of an MCSAT solver consists of a core solver, an assignment trail, and reasoning plugins. The *core solver* drives the overall solving process, and is responsible for dispatching notifications and handling requests from the plugins. The *solver trail* is a chronological record that tracks assignments of terms to values. It is shared by the core solver and the reasoning plugins. The *reasoning plugins* are modules dedicated to handling specific theory terms and constraints (e.g., clauses for Booleans, polynomial constraints for arithmetic). A plugin reasons about the content of the solver trail with respect to the set of currently relevant terms. In the context of nonlinear arithmetic problems, the reasoning plugins are the arithmetic plugin and the Boolean plugin. The most important role of the core solver is to perform conflict analysis when one of the reasoning plugins detects a conflicting state.

When formulas $F_1, \ldots, F_n$ are asserted, by calling $\textsc{solver::assert}(F_i)$, the core solver notifies all plugins of the asserted formulas. The plugins analyze the formulas and report all *relevant terms* back to the core. The relevant terms are the variables and subterms of the formulas F_is that need to be consistently assigned to ensure a satisfying assignment. In nonlinear arithmetic, relevant terms are all variables, arithmetic constraints, and non-negated Boolean terms that appear in the input formula (or are part of a learnt clause). Once the relevant terms are collected, the core solver adds the assertions to the trail. The initial trail contains then the partial assignment $F_i \rightsquigarrow \top$ and the search for a full satisfying assignment starts from this trail.

Solver Trail and Evaluation. The assignment trail is the central data structure in the MCSAT framework. It is a generalization of the Boolean assignment trail used in modern CDCL SAT solvers. The trail records a partial (and potentially inconsistent) model that assigns values to relevant terms. If the satisfiability algorithm terminates with a **sat** answer, the full satisfying assignment can be read off the trail. At any point during the search, the trail can be used to evaluate any relevant compound term based on the values of its sub-terms. A term t (and $\neg t$, if Boolean) *can be evaluated* in the trail M if t itself is assigned in M, or if all closest relevant sub-terms of t are assigned in M (and its value can therefore be computed). As the search progresses, it is possible for some terms to *be evaluated in two different ways*, which can result in a conflict (i.e., a term assigned different values). In order to account for this ambiguity, we define an evaluation predicate $\mathsf{evaluates}[M](t, v)$ that returns **true** if the term t can evaluate to the value v in trail M.

Algorithm 2: MCSAT::CHECK($\boldsymbol{x} \mapsto \boldsymbol{v}$)

```
Data: solver trail $M$, relevant variables/terms to assign in $queue$
while true do
    unitPropagate() ;
    if a plugin detected a conflict and the conflict clause is $C$ then
        $\langle C, final \rangle \leftarrow$ analyzeConflict($M$, $C$, $\boldsymbol{x}$) ;
        if $final$ then
            $I \leftarrow$ analyzeFinal($M$, $C$) ;
            return $\langle$unsat, $I\rangle$
        else backtrackWith($M$, $C$) ;
    else
        if exists $x_i \in \boldsymbol{x}$ unassigned in $M$ then
            ownerOf($x_i$).decideValue($x_i$, $v_i$)
        else
            if $queue$.empty() then return $\langle$sat, $M\rangle$ ;
            $x \leftarrow queue$.pop() ;
            if $x$ is unassigned then ownerOf($x$).decideValue($x$) ;
```

Conflicts and Conflict Clauses. One of the main responsibilities of reasoning plugins is to ensure that the trail is consistent at any point in the search. A trail is *evaluation consistent* if no relevant term can evaluate to two different values, as described above. A trail is *unit consistent* if every relevant term can be given a value without making the trail evaluation inconsistent. If the trail is not evaluation consistent or unit consistent, the trail is *in conflict.*

Trail consistency is a generalization of the consistency that CDCL SAT solvers enforce during their search. By unit propagation, a SAT solver ensures that, if no conflict has been detected, no clause can be falsified by assigning a single variable (i.e., no clause evaluates to both $\top$ and $\bot$). In the MCSAT framework, the plugins do the same: they keep track of unit constraints and reason about the consistency of the trail. It is the responsibility of the plugin to report conflicts. Each conflict must be accompanied with a *valid* conflict clause that explains the inconsistency.[4] A clause $C \equiv (L_1 \vee \ldots \vee L_n)$ is a *conflict clause* in a trail M, if each literal L_i can evaluate to $\bot$ in M, i.e. if $\mathsf{evaluates}[M](L_i, \bot)$.

Example 2. Consider the constraint $C \equiv (x^2 + y^2 < 1)$ with the set of relevant terms $\{C, x, y\}$, and the following solver trails

$$M_1 = [\![\, C \mapsto \top, x \mapsto 0 \,]\!], \qquad M_2 = [\![\, C \mapsto \top, x \mapsto 0, y \mapsto 0 \,]\!],$$
$$M_3 = [\![\, C \mapsto \top, x \mapsto 1 \,]\!], \qquad M_4 = [\![\, C \mapsto \top, x \mapsto 1, y \mapsto 0 \,]\!].$$

The trails M_1 and M_2 are consistent, the trail M_3 is unit inconsistent (no consistent assignment for y exists), and M_4 is evaluation inconsistent (C evaluates to both $\top$ and $\bot$). A valid explanations for the inconsistency of M_3 is the conflict

[4] By valid here we mean that the clause is a universally true statement on its own.

clause $C_3 \equiv \neg C \vee (x < 1)$, while a valid explanation for the inconsistency of M_4 is the conflict clause $C_4 \equiv \neg C \vee C$. Although C_4 is a tautology, it is an acceptable conflict clause since both literals can evaluate to $\bot$ (because $\mathsf{evaluates}[M_4](C, \top)$ and $\mathsf{evaluates}[M_4](C, \bot)$).

Main Procedure. The implementation of the satisfiability checking procedure SOLVER::CHECK() is a generalization of the search-and-resolve loop of modern SAT solvers (see, e.g. [16,17]). The procedure is shown in Algorithm 2, where we emphasize the extensions needed for SMT modulo models in red. The overall procedure performs a direct search for a satisfying assignment and terminates either by finding an assignment that extends the given partial model, or deduces that the problem is unsatisfiable as certified by an appropriate model interpolant.

The main elements of the procedure are unit propagation and decisions, used for constructing the assignment, and conflict analysis for repairing the trail when it becomes inconsistent. The `unitPropagate()` procedure invokes the propagation procedures provided by the plugins. Propagation allows each plugin to add new assignments to the top of the trail. If, during propagation, a plugin detects an inconsistency, it reports the conflict to the core solver along with a valid conflict clause. The `decideValue(`x`)` procedure assigns a value of the given unassigned term x. Decisions are performed only after propagation has fully saturated with no reported conflicts, which means that the trail is unit consistent. In such a trail, an assignment for x is guaranteed to exist, but the choice of a particular value is delegated to the plugin responsible for x (e.g., the arithmetic plugin for real-typed terms).

Modification 1 (Decisions). *To support SMT modulo a model $\boldsymbol{x} \mapsto \boldsymbol{v}$, variables $x_i \in \boldsymbol{x}$ of the input model are decided before any other term, and are assigned the provided value v_i. The procedure that performs this decision is denoted with* `decideValue(`x_i`, `v_i`)`*. If a decision introduces an evaluation inconsistency, the plugin reports the conflict with a conflict clause.*

Detecting and explaining decision conflicts is straightforward: there must exist a single constraint C that can evaluate to both $\top$ and $\bot$ in the trail. Such conflicts can always be explained with a clause of the form $(\neg C \vee C)$.

If a conflict is reported, either during propagation or in a decision, the procedure invokes the conflict analysis procedure `analyzeConflict()`. This procedure takes the reported conflict clause C and finds the root cause of the conflict. The analysis backtracks the trail, element by element, so long as C is a conflict clause, while resolving any trail propagations from C. Once done, the analysis returns the clause along with the flag that indicates whether this conflict clause C is empty (indicating the final conflict). If the conflict is not final, the procedure calls `backtrackWith()` to backtrack the trail further, if possible, and add a new assignment to the trail, ensuring progress and fixing the conflict. The main invariant of the conflict resolution procedure is that the *conflict clause C is always implied by asserted formulas.*

Modification 2 (Conflict Analysis). *To support SMT modulo a model $\boldsymbol{x} \mapsto \boldsymbol{v}$, the analysis procedure* `analyzeConflict(`M`,` C`,` $\boldsymbol{x}$`)` *stops as soon as it encounters a variable $x_i \in \boldsymbol{x}$ to resolve, and returns $\langle C, \mathbf{true}\rangle$.*

This modification is based on the fact that the variables x_i have a fixed value given by the model. Assume that conflict analysis attempts to undo a variable x_i that is part of the provided model $\boldsymbol{x} \mapsto \boldsymbol{v}$. This can only happen when the trail consists of only variables from $\boldsymbol{x}$ and implications of asserted formulas. In other words, this particular conflict cannot be resolved unless we modify either the assertions themselves or the input model. The clause resulting from the analysis marked as final is our starting point for producing the model interpolant.

Modification 3 (Final Analysis). *To support SMT modulo a model $\boldsymbol{x} \mapsto \boldsymbol{v}$, the procedure* `analyzeFinal(`$M$`,` C`)` *resolves any remaining trail propagations in M from the clause C and returns the resulting clause I.*

The resolution of propagations in this final analysis is done in the same manner as in regular conflict analysis. This means that the resulting clause I is implied by the asserted formulas. In addition, resolving all propagations from the conflict clause ensures that all literals of I evaluate to false only because of the assignment $\boldsymbol{x} \mapsto \boldsymbol{v}$, making I an appropriate model interpolant.

Example 3. Consider two formulas $F_1 \equiv b$ and $F_2 \equiv \neg b \vee (x^2 + y^2 < 2)$. When asserting these two formulas to the MCSAT solver, the Boolean and arithmetic plugins will identify the set of terms relevant for satisfiability as $R = \{b, x, y, (x^2 + y^2 < 2)\}$. Additionally, the assertions will be added to the trail and propagated[5], resulting in the following initial trail

$$M_0 = [\![\, b \rightsquigarrow \top, F_2 \rightsquigarrow \top, (x^2 + y^2 < 2) \overset{F_2}{\rightsquigarrow} \top \,]\!].$$

We now apply our procedure to solve F_1 and F_2 modulo the partial model $\{x \mapsto 2\}$.

In the first iteration, no term in R is unit (with only one variable unassigned), and propagation does not infer any new facts or conflicts. The procedure thus perform a decision on the unassigned variable x of the model, resulting in the trail

$$M_1 = [\![\, b \rightsquigarrow \top, F_2 \rightsquigarrow \top, (x^2 + y^2 < 2) \overset{F_2}{\rightsquigarrow} \top, x \mapsto 2 \,]\!].$$

In the second iteration, as $(x^2 + y^2 < 2)$ is unit in the trail M_1, the arithmetic plugin examines the constraint and deduces that there is no potential solution for y. This constitutes a unit inconsistency that the plugin reports, along with the conflict clause[6]

$$C_0 \equiv \neg(x^2 + y^2 < 2) \vee \neg(x > \sqrt{2}).$$

[5] Notation $t \overset{F}{\rightsquigarrow} v$ denotes that t is assigned to v due to propagation, and F is the reason of the propagation.

[6] We use $(x > \sqrt{2})$ as a shorthand for the extended constraint $x >_r \mathsf{root}(x^2 - 2, 2, x)$.

Conflict analysis takes clause C_0 and starts the resolution process. As the top variable x on the trail M_1 is part of the input model, the analysis stops and reports that the clause C_0 is the final explanation. This clause is valid, but not yet a model interpolant as it contains a literal with variable y. We then proceed with the final analysis to remove such literals. First, we resolve $(x^2 + y^2 < 2)$ from C_0 using its reason clause F_1, which gives the clause $C_1 \equiv \neg b \vee \neg(x > \sqrt{2})$. Then, we resolve b from C_1 with an empty reason (b is an assertion), resulting in the final clause and model interpolant $I = \neg(x > \sqrt{2})$.

4 Nonlinear Arithmetic

The general approach to interpolation presented so far is not specific to nonlinear arithmetic. We now tackle two practical issues that arise in nonlinear arithmetic and we discuss the properties of our interpolation procedure in the context of nonlinear arithmetic. First, on nonlinear problems, as seen in Example 3, the interpolation procedure can return model interpolants that include extended polynomial constraints. This is an artifact of the underlying decision procedure (such as NLSAT [29]) that might use extended polynomial constraints to succinctly represent conflict explanations. While such constraints make decision procedures more effective, they are undesirable for interpolation: interpolants should be described in the language of the input formulas, if possible. Second, to use the interpolant procedure in the context of model checking, we also need to devise a generalization procedure for polynomial constraints.

This section uses concepts from cylindrical algebraic decomposition (CAD). We keep the presentation example-driven and focused on our particular needs, and refer the reader to the existing literature for further information [2,5,7]. Cylindrical algebraic decomposition is a general approach for reasoning about polynomials based on the following result due to Collins [10]. For any set of polynomials $f_1, \ldots, f_k \in \mathbb{Z}[x_1, \ldots, x_n]$ one can algorithmically decompose $\mathbb{R}^n$ into connected regions (called cells) such that all the polynomials f_j are sign-invariant in every cell C_i. This means that the cells also maintain the truth value of any polynomial constraints over the polynomials f_i, which is crucial in many reasoning techniques for polynomial constraints.

The theory and practice of CAD is heavily dependent on the ordering of variables involved. For this paper we always assume the CAD order to be the same as the order of the defined polynomials (e.g., $x_1 < x_2 < \ldots < x_n$). Every CAD cell is cylindrical in nature, and can be described by constraints where every dimension of the cell (called a level) can be completely defined by relying only on the previous dimensions. We illustrate this through an example.

Example 4. Consider the polynomial $f = x^2 + y^2 - 2 \in \mathbb{Z}[x, y]$. A CAD of f is depicted in Fig. 1 (left). The cell C_1 is defined by two constraints:

$$C_1^y \equiv y >_r \mathsf{root}(x^2 + y^2 - 2, 2, y),$$
$$C_1^x \equiv x >_r \mathsf{root}(x^2 - 2, 1, x) \wedge x <_r \mathsf{root}(x^2 - 2, 2, x).$$

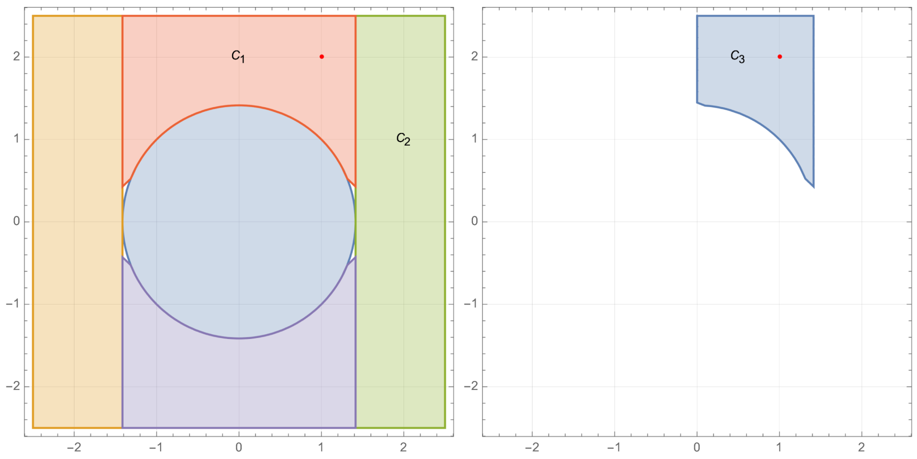


Fig. 1. CAD of the polynomial $f = x^2 + y^2 - 2$ from Example 4 (left). Computed cell capturing the model $(1, 2)$ of Example 5 (right).

Constraint C_1^x is at the first level (it's a constraint on x only), while constraint C_1^y is at the second level and relates variables x and y. The full cell description is then $C_1 \equiv C_1^x \wedge C_1^y$. The green cell C_2 can be described by $C_2^y \equiv \top$ and $C_2^x \equiv x >_r \mathsf{root}(x^2 - 2, 2, x)$, with the full description $C_2 \equiv C_2^y \wedge C_2^x$.

Model-based decision procedures such as NLSAT rely on CAD construction but do not construct the complete CAD decomposition. Instead, given a point in $\mathbb{R}^n$ they can construct a single cell of a CAD in a model-driven fashion. For more information about this approach, we refer the reader to [4,27]. For our purposes we abstract the cell construction, and denote with $\mathsf{describeCell}(F, M)$ the function that, given a set of polynomials F, returns a description of a CAD cell of F that contains the model M.

Following the terminology used in CAD, we say that a non-empty connected subset of $\mathbb{R}^k$ is a *region*. A set of polynomials $\{f_1, \ldots f_s\} \subset \mathbb{Z}[\boldsymbol{y}, x]$, with $\boldsymbol{y} = \langle y_1, \ldots, y_n \rangle$, is said to be *delineable* in a region $S \subseteq \mathbb{R}^n$ if for every f_i (and f_j) from the set, the following properties are invariant for any $\boldsymbol{\alpha} \in S$:

1. the *total number of complex roots* of $f_i(\boldsymbol{\alpha}, x)$;
2. the *number of distinct complex roots* of $f_i(\boldsymbol{\alpha}, x)$;
3. the *number of common complex roots* of $f_i(\boldsymbol{\alpha}, x)$ and $f_j(\boldsymbol{\alpha}, x)$.

Delineability has important consequences on the number and arrangement of real roots of polynomials f_i. As explained by the following theorem, if a set of polynomials F is delineable on a region S, then the number of real roots of the polynomials does not change on S. Moreover, these roots maintain their relative order on the whole of S.

Theorem 1 (Corollary 8.6.5 of [40]). *Let F be a set of polynomials in $\mathbb{Z}[\boldsymbol{y}, x]$, delineable in a region $S \subset \mathbb{R}^n$. Then, the real roots of F vary continuously over S, while maintaining their order.*

For a polynomial $f \in \mathbb{Z}[\boldsymbol{x}]$ and model $M = \{\boldsymbol{x} \mapsto \boldsymbol{v}\}$, we denote with $\mathsf{sgncstr}(f, M)$ the polynomial constraint that matches the sign of f in M, i.e.

$$\mathsf{sgncstr}(f, M) = \begin{cases} f < 0 & \text{if } \mathsf{sgn}(f(\boldsymbol{v})) < 0 \\ f > 0 & \text{if } \mathsf{sgn}(f(\boldsymbol{v})) > 0 \\ f = 0 & \text{if } \mathsf{sgn}(f(\boldsymbol{v})) = 0 \end{cases}$$

As described above, a CAD cell can be succinctly described by relying on extended polynomial constraints. We now show that the description of the cell can be reduced to basic polynomial constraints.

Lemma 3. *Let $f_i \in \mathbb{Z}[y_1, \ldots, y_n, x]$ be two polynomials of degrees m_i, and $F_i \equiv x \nabla_r \mathsf{root}(f_i, k_i, x)$ be extended polynomial constraints of a cell description. Let S be a region of $\mathbb{R}^n$ where $\{f_1, f_2\}$ are delineable and let $M = \{\boldsymbol{y} \mapsto \boldsymbol{v}, x \mapsto \alpha\}$ be a model such that $\boldsymbol{v} \in S$. Then, for all $\boldsymbol{y} \in S$ it holds that*

$$\bigwedge_{i=0}^{m_1-1} \mathsf{sgncstr}(f_1^{(i)}, M) \wedge \bigwedge_{i=0}^{m_2-1} \mathsf{sgncstr}(f_2^{(i)}, M) \Rightarrow F_1 \wedge F_2.$$

The proof of this lemma is relatively straightforward. The CAD cell description for level x represents an entry in the sign table of f_1 and f_2 (with no roots in between). A part of this sign table entry that contains M can be described with the signs of all the derivatives of f_1 and f_2 as long as we can guarantee that neither the arrangement nor the number of roots f_1 and f_2 change. But, this is guaranteed by f_1 and f_2 being delineable on S, so the lemma holds.

As a corollary to this lemma, in the context of CAD cell construction around a model M, we can replace any extended constraints describing a cell C with basic constraints stating that the signs of the polynomial derivatives are the same as in M. This results in a valid CAD subcell $C' \subseteq C$ for the same polynomials, that still contains the model M. We denote the function that constructs a basic CAD cell description of a set of polynomials F capturing the model M with $\mathsf{describeCellBasic}(F, M)$.

Example 5. Based on Example 4, we can construct a cell around the model $M = \{x \mapsto 1, y \mapsto 2\}$. Function $\mathsf{describeCellBasic}(F, M)$ will return the constraints

$$\begin{aligned} C_3^y &\equiv (x^2 + y^2 > 2) \wedge (y > 0), \\ C_3^x &\equiv (x^2 < 2) \wedge (x > 0). \end{aligned}$$

The full cell description is then $C_3 \equiv C_3^x \wedge C_3^y$. Note that this cell is smaller than the cell C_1 from Example 4. This reduction in size is generally undesirable, but it is a price to pay for having the description in a simpler language.

Interpolation Without Extended Constraints. We now show how the cell construction described above can be used to remove extended polynomial constraints from a model interpolant. Assume a clausal model interpolant

$$I = (L_1 \vee \ldots \vee L_i \vee \ldots \vee L_N)$$

that is implied by formula A and refutes a model $M = \{\boldsymbol{x} \mapsto \boldsymbol{v}\}$, i.e., all literals of I evaluate to $\bot$ in M. Assume also that some literal L_i contains an extended polynomial constraint $x_n \nabla_r \mathsf{root}(f, k, x_n)$, with $f \in \mathbb{Z}[\boldsymbol{x}]$. We aim to replace the extended literal L_i with literals over basic polynomial constraints. To do so, we need to find literals $L_i^1, \ldots, L_i^m$ such that $L_i \Rightarrow (L_i^1 \vee \ldots \vee L_i^m)$ and all literals L_i^j evaluate to $\bot$ in M. Then, the clause

$$I' = (L_1 \vee \ldots \vee L_i^1 \vee \ldots L_i^m \vee \ldots \vee L_N)$$

will also be a model interpolant implied by A that refutes the model M.

We can construct the literals L_i^j using single cell construction as follows. We create a description of the CAD cell of the polynomial f from L_i that captures the model M. Let $\mathsf{describeCellBasic}(\{f\}, M) = D_1 \wedge \ldots \wedge D_m$ be this description. Since the cell fully captures the behavior of f around M, we know that $D_1 \wedge \ldots \wedge D_m \Rightarrow \neg L_i$ and all literals D_j evaluate to $\top$. Therefore, we can use the cell description to eliminate the extended literal L_i, obtaining the clause

$$I' = (L_1 \vee \ldots \vee \neg D_i \vee \ldots \neg D_m \vee \ldots \vee L_n)$$

By continuing this process, we can replace all extended literals from a model interpolant, to obtain a model interpolant in the basic language of polynomial constraints.

Example 6. Consider the model interpolant $I = \neg(x >_r \mathsf{root}(x^2 - 2, 2, x)$ from Example 3 that refutes the model $M = \{x \mapsto 2\}$. To express I in terms of basic polynomials constraints we first construct a regular CAD cell of $f = x^2 - 2$ around M. In this case this cell is simply $x >_r \mathsf{root}(x^2 - 2, 2, x)$. Then, we use Lemma 3 to construct a basic CAD cell description as $(x^2 > 2) \wedge (x > 0)$. Finally, the simplified interpolant is $I' = \neg(x^2 > 2) \vee \neg(x > 0)$.

Termination. With the description of the interpolation procedure complete, we discuss the termination of the procedure. To do so, we fix the formula $A(\boldsymbol{x}, \boldsymbol{y})$ of Definition 3 and we assume a fixed order of variables that ensures $y_i < x_i$. Since the MCSAT decision procedure on which we rely is based on CAD, we can put a bound on the set of literals that can ever appear in a model interpolant from the formula A to an arbitrary model M. Let P_A be the set of polynomials appearing in A, and let $P = \mathsf{P}(P_A)$ denote the closure of the set P_A under the CAD projection operator used by the decision procedure. Finally, let P' be the closure of P under derivatives. The set of polynomial constraints that can appear in the interpolant I is limited to basic polynomial constraints over polynomials in P'. This means that the procedure MCSAT::CHECK() can only generate a finite number of model interpolants and therefore has the finite convergence property.

Lemma 4. *Assuming a fixed variable order, the* MCSAT::CHECK() *procedure has the finite convergence property for nonlinear arithmetic formulas.*

Together with Lemma 2, this lemma implies that our interpolation procedure for the theory of nonlinear arithmetic terminates.

Model Generalization. We now proceed to show how the CAD cell construction can be used in a natural way to provide model-driven generalization. As in Definition 2, assume a formula $F(\boldsymbol{x}, \boldsymbol{y})$ such that F is true in a model M. Our aim is to construct a formula $G(\boldsymbol{x})$ that generalizes the model M and still guarantees a solution to F.

Following the approach of [15], we do this in two steps. First, we construct an implicant B of F based on the model M. Then, we eliminate the variables $\boldsymbol{y}$ from B, again relying on the model M. The implicant B is a conjunction of literals that implies F and such that B is true in M. The implicant can be computed by a top-down traversal of the formula F while using the model M to evaluate the formula nodes (see, e.g., [15] for a detailed description). To find a formula G such that $G \Rightarrow \exists \boldsymbol{y} \ . \ B$, we use CAD cell construction as follows. Let $P \subseteq \mathbb{Z}[\boldsymbol{x}, \boldsymbol{y}]$ be the set of all polynomials appearing in B, and let the cell description of P around M be

$$\mathsf{describeCellBasic}(P, M) = D_{\boldsymbol{x}} \wedge D_{\boldsymbol{y}} .$$

Here, $D_{\boldsymbol{x}}$ denotes the description of cell levels of variables $\boldsymbol{x}$, while $D_{\boldsymbol{y}}$ denotes the description of cell levels of variables $\boldsymbol{y}$. Because of the cylindrical nature of CAD cells, and the order on variables y_i and x_i, we are guaranteed that every solution of $D_{\boldsymbol{x}}$ can be extended to a solution of $D_{\boldsymbol{y}}$. Therefore we set the final generalization $G(x) \equiv D_{\boldsymbol{x}}$.

Example 7 (Generalization). Consider the formula $F \equiv (x^2 + y^2 < 2)$ and the model $M = \{x \mapsto 1, y \mapsto 2\}$ that satisfies F, and let us compute a generalization $G(x)$ of M. First, we compute a CAD cell of $f = x^2 + y^2 - 2$ as shown in Example 5. Then we drop the description of cell level y, to obtain the model generalization $G(x) \equiv (x^2 < 2) \wedge (x > 0)$.

5 Evaluation

To the best of our knowledge, there is no clear metric for evaluating how good an interpolant is, or for comparing different interpolants. In this section, we first show two examples to illustrate the procedure and its applications. Then, we evaluate the effectiveness of our interpolation procedure on practical problems that arise from model-checking applications. To this end, we integrate the procedure into a model checker and evaluate whether the procedure is efficient, and can produce abstractions that help the model checker synthesize invariants and discover counter-examples.

We have implemented the reasoning procedures (solving modulo partial models and interpolation procedure) by extending the existing MCSAT implementation of the YICES2 SMT solver [14]. We used the LIBPOLY library [31] for computing the model generalization and simplification of algebraic cells. Since YICES2 is integrated into the SALLY model checker [30], we rely on the PDKIND method [30] as the model checking engine (the user of interpolation) in our evaluation.

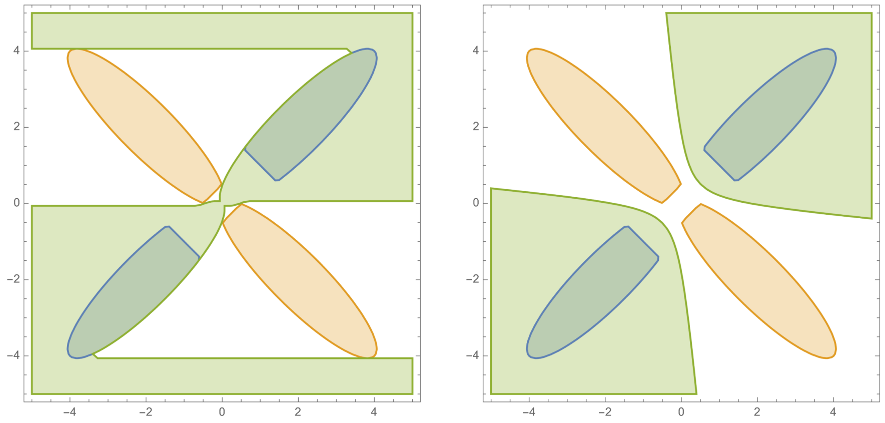


Fig. 2. Illustration of interpolants from Example 8. In blue and orange are the feasible space of the formulas A and B (projected on x and y). In green is the feasible space of the interpolant produced by our method (on the left) and the interpolant produced by [19] (on the right). (Color figure online)

Example 8. We compare the style of interpolants generated by our new procedure with the ones generated by numerical approaches such as [19]. Example 4 from [19] considers two formulas of the form

$$A(x, y, a_1, a_2, b_1, b_2) \equiv (f_1 \geq 0 \wedge f_2 \geq 0) \vee (f_3 \geq 0 \wedge f_4 \geq 0),$$
$$B(x, y, c_1, c_2, d_1, d_2) \equiv (g_1 \geq 0 \wedge g_2 \geq 0) \vee (g_3 \geq 0 \wedge g_4 \geq 0).$$

The polynomials f_i and g_i involved in A and B are of degree 2. The right-hand side of Fig. 2 shows the interpolant I_1 found by the approach in [19]. This interpolant is of the form $h(x, y) > 0$, where h is a polynomial degree two computed using semidefinite programming. Our approach, on the other hand, produces the interpolant I_2 shown on the left-hand side of Fig. 2. This interpolant consists of 12 clauses, each containing 6–8 polynomial constraints over 16 different polynomials (8 linear, 8 of degree 2). The interpolant I_2 is ultimately produced from fragments of a CAD so its edges touch upon the critical points of the shape they were produce from (formula A). Interpolant I_1, on the other hand, has a simple form dictated by the method [19]. Which form is ultimately more useful depends on a particular application.

	IC3-NRA			KIND			PDKIND		
problem set	solved	valid/invalid	time (s)	solved	valid/invalid	time (s)	solved	valid/invalid	time (s)
handcrafted (14)	10	9/1	381	3	2/1	0	**14**	13/1	4
hycomp (7)	2	2/0	15	4	1/3	796	**4**	2/2	792
hyst (65)	**39**	32/7	404	25	13/12	50	38	26/12	42
isat3 (1)	0	0/0	0	0	0/0	0	0	0/0	0
isat3-cfg (10)	8	6/2	14	9	6/3	9	**10**	7/3	8
nuxmv (2)	**2**	2/0	158	0	0/0	0	1	1/0	1118
sas13 (13)	10	5/5	13	5	0/5	0	**13**	8/5	7
tcm (2)	2	2/0	1	**2**	2/0	0	**2**	2/0	0
	73	58/15	986	48	24/24	855	**82**	59/23	1971

Fig. 3. Evaluation Results. For each tool, we report the number of solved problems, how many of the solved problems were valid and invalid, and the total time used to solve them. The rows correspond to different problem classes, and the bottom row reports the overall results for all 114 benchmarks.

Example 9 (Cauchy-Schwartz). As described in Example 1, we can model the computation of Cauchy-Schwarz inequality as a transition system $\mathfrak{S}_{cs}$. Then we can prove the inequality correct if we can prove that the property P_{cs} is valid in $\mathfrak{S}_{cs}$. The PDKIND model checking engine with the new interpolation procedure proves the property valid in 1 s.

Benchmarks. We run the evaluation on an existing set of nonlinear model-checking problems used by Cimatti, et al. [8]. This set consists of 114 benchmarks from various sources: handcrafted benchmarks, hybrid system verification, NUXMV benchmarks, C floating-point verification, and verification of Simulink models. The benchmark problems all contain transition systems with nonlinear behavior. For each problem, the goal is to prove or disprove a single invariant. We refer the reader to [8] for a more detailed description.

Evaluation. Cimatti, et al. [8] present an abstraction approach based on incrementally more precise linear approximations of nonlinear polynomials. They show that this approach, implemented in the IC3-NRA tool, is superior to other tools (such as, ISAT3 [36] and NUXMV [6] with upfront linear abstraction). Since our goal is to show the effectiveness of our interpolation procedure, rather than compare to many model checking engines, we keep the evaluation simple and only compare to IC3-NRA. In addition, we include the k-induction engine KIND of SALLY in the comparison to illustrate the importance of invariant inference and counter-example generation.[7]

We ran the tools on the benchmark set with a 1 h CPU timeout per problem. The results are shown in Fig. 3 and on the cactus plot in Fig. 4. A scatter plot comparison of PDKIND against IC3-NRA and KIND is shown in Fig. 5.

[7] KIND performs k-induction checks for increasing values of k and stops if either the property is shown k-inductive, or a counter-example is found.

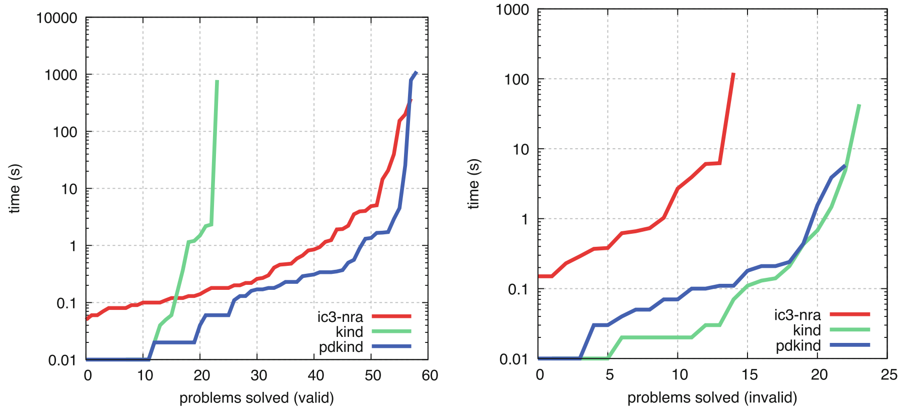


Fig. 4. Cactus Plots Comparing the Performance of IC3-NRA, KIND, and PDKIND. The x axis is the number of problems solved (valid on the left, invalid on the right) and the y axis is the time needed to solve the problem (log scale).

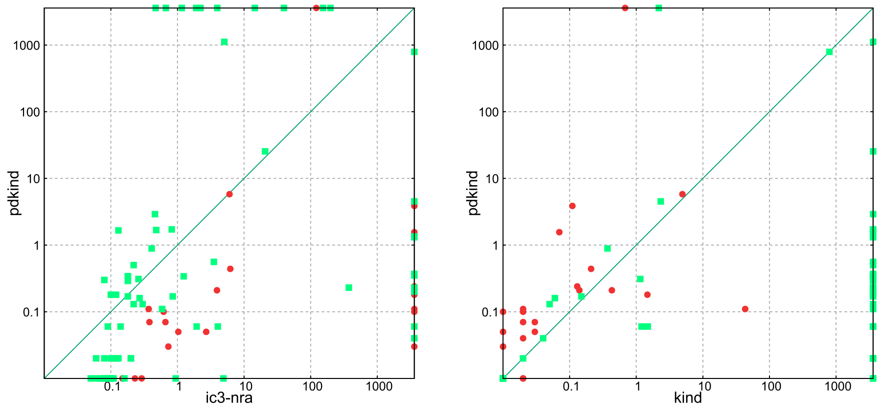


Fig. 5. Scatter Plots Comparing the Performance of IC3-NRA and KIND with PDKIND. Green squares represent problems that are valid. Red dots represent problems that are invalid. Each axis represents the time it took the tool to solve the problem (log scale). (Color figure online)

As can be seen from Fig. 3, the results are positive. The PDKIND engine with the new interpolation method can prove more properties and find more counter-examples than the state-of-the-art IC3-NRA.

Out of 59 properties that PDKIND shows correct, 36 cannot be proved by KIND. This means that these properties are likely not k-inductive and that the interpolants produced by our procedure are valuable abstractions in invariant inference. Similarly, IC3-NRA proves 37 properties that are not k-inductive. As can be seen from the scatter plot in Fig. 5, there are properties that PDKIND can prove than IC3-NRA cannot, and vice versa (11 and 10, respectively). This is to be expected from a difficult domain, but it also means that the interpolation and the abstraction approach (or other methods) can be used to complement each other.

As for the invalid properties, since our interpolation method (and thus PDKIND) is based on complete and precise reasoning, while IC3-NRA relies on abstraction, it is to be expected that PDKIND can prove more properties invalid. Furthermore, the comparison with KIND in Fig. 5 shows that PDKIND finds all but one counter-examples that KIND does in a similar amount of time. We see this as a confirmation that the interpolation and generalization methods are effective, i.e., they do not impede the search for counter-examples.

5.1 Related Work

There is ample literature on interpolation for different fragments of nonlinear arithmetic. Existing methods can roughly be classified into two categories: approaches based on interval reasoning, and approaches based on semidefinite programming. Interval reasoning techniques (e.g., [20,35,36]) construct a proof of unsatisfiability through interval slicing and propagation. From such a proof, interpolants can be built using proof-based interpolation techniques. While incomplete, interval-based techniques can be very effective on problems that are hard for complete techniques. Moreover they can support more polynomial functions (e.g., elementary functions, ODEs). Our procedure is complete, but it is limited to the theories supported by MCSAT. The approaches based on semidefinite programming [12,18,19] generally approach the interpolation problem by restricting both the fragment of arithmetic (e.g., bounded constraints, same set of variables, quadratic constraints) and the shape of the interpolant (a single polynomial constraint) so that the interpolant itself can be represented as a semidefinite optimization problem. When they apply, these procedures are also very effective but they suffer from numerical imprecision, requiring special care to account for these errors and making them difficult to use in formal verification. In contrast, out procedure applies to nonlinear arithmetic as a whole. It relies on symbolic techniques, which are not subject to numerical errors. It is precise and complete, and it produces clausal interpolants.

The core ideas beyond our model-based interpolation approach were presented at the Boolean level as SAT solving with assumptions [17]. Closest to our work is the work of Schindler and Jovanović [42] where a similar model-based approach to interpolation is applied to conjunctions of linear arithmetic constraints based on conflict resolution. Our work is more general as it applies to formulas other than conjunctions, and it is applicable to a wider range of theories.

6 Conclusion and Future Work

We have presented a general approach for interpolation in SMT. This novel approach relies on a mode of interaction with the SMT solver that can check a formula for satisfiability modulo a partial model and, if the formula is unsatisfiable, can return a model interpolant that refutes the model. This allows us to develop a first complete interpolation procedure for nonlinear arithmetic. We have implemented the new procedure in the YICES2 SMT solver and evaluated the interpolation procedure on model-checking problems. The new procedure seems to be effective in practice and opens new possibilities in the verification of systems that contain nonlinear behavior. Additionally, we show interesting examples of how the procedure can be used in automating induction proofs in mathematics.

The interpolation procedure that we presented can support other theories available in MCSAT (e.g., uninterpreted functions [28], bit-vectors [22], nonlinear integer arithmetic [27]). We plan to explore interpolation in these theories in more detail, and in the contexts where interpolation can be beneficial (e.g., model checking, quantified reasoning, termination, and proof generation).

References

1. Barrett, C., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Model Checking, pp. 305–343. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_11
2. Basu, S., Pollack, R., Roy, M.-F.: Algorithms in Real Algebraic Geometry. Springer, Heidelberg (2006). https://doi.org/10.1007/3-540-33099-2
3. Bayless, S., Val, C.G., Ball, T., Hoos, H.H., Hu, A.J.: Efficient modular SAT solving for IC3. In: Ray, S., Jobstmann, B. (eds.) 2013 Formal Methods in Computer-Aided Design, pp. 149–156. IEEE (2013)
4. Brown, C.W., Košta, M.: Constructing a single cell in cylindrical algebraic decomposition. J. Symb. Comput. **70**, 14–48 (2015)
5. Buchberger, B., Collins, G.E., Loos, R., Albrecht, R. (eds.): Computer Algebra. Symbolic and Algebraic Computation, Springer, Vienna (1982). https://doi.org/10.1007/978-3-7091-7551-4
6. Cavada, R., et al.: The NUXMV symbolic model checker. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 334–342. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_22
7. Caviness, B.F., Johnson, J.R. (eds.): Quantifier Elimination and Cylindrical Algebraic Decomposition. Texts and Monographs in Symbolic Computation, Springer, Vienna (2004). https://doi.org/10.1007/978-3-7091-9459-1
8. Cimatti, A., Griggio, A., Irfan, A., Roveri, M., Sebastiani, R.: Invariant checking of NRA transition systems via incremental reduction to LRA with EUF. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 58–75. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_4
9. Cimatti, A., Griggio, A., Sebastiani, R.: Efficient interpolant generation in satisfiability modulo theories. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 397–412. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_30
10. Collins, G.E.: Quantifier elimination for real closed fields by cylindrical algebraic decompostion. In: Brakhage, H. (ed.) GI-Fachtagung 1975. LNCS, vol. 33, pp. 134–183. Springer, Heidelberg (1975). https://doi.org/10.1007/3-540-07407-4_17

11. Craig, W.: Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. J. Symbolic Logic **22**(3), 269–285 (1957)
12. Dai, L., Xia, B., Zhan, N.: Generating non-linear interpolants by semidefinite programming. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 364–380. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_25
13. de Moura, L., Jovanović, D.: A model-constructing satisfiability calculus. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 1–12. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35873-9_1
14. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 737–744. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_49
15. Dutertre, B.: Solving exists/forall problems with Yices. In: 13th International Workshop on Satisfiability Modulo Theories (2015)
16. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24605-3_37
17. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electron. Notes Theor. Comput. Sci. **89**(4), 543–560 (2003)
18. Gan, T., Dai, L., Xia, B., Zhan, N., Kapur, D., Chen, M.: Interpolant synthesis for quadratic polynomial inequalities and combination with *EUF*. In: Olivetti, N., Tiwari, A. (eds.) IJCAR 2016. LNCS (LNAI), vol. 9706, pp. 195–212. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40229-1_14
19. Gan, T., Xia, B., Xue, B., Zhan, N., Dai, L.: Nonlinear Craig interpolant generation. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12224, pp. 415–438. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53288-8_20
20. Gao, S., Zufferey, D.: Interpolants in nonlinear theories over the reals. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 625–641. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_41
21. Gerhold, S., Kauers, M.: A procedure for proving special function inequalities involving a discrete parameter. In: Gao, X.-S., Labahn, G. (eds.) Proceedings of the 2005 International Symposium on Symbolic and Algebraic Computation, pp. 156–162 (2005)
22. Graham-Lengrand, S., Jovanović, D., Dutertre, B.: Solving Bitvectors with MCSAT: explanations from bits and pieces. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) IJCAR 2020. LNCS (LNAI), vol. 12166, pp. 103–121. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51074-9_7
23. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. ACM SIGPLAN Not. **39**(1), 232–244 (2004)
24. Hoder, K., Bjørner, N.: Generalized property directed reachability. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 157–171. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31612-8_13
25. Hoenicke, J., Schindler, T.: Efficient interpolation for the theory of arrays. In: Galmiche, D., Schulz, S., Sebastiani, R. (eds.) IJCAR 2018. LNCS (LNAI), vol. 10900, pp. 549–565. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94205-6_36
26. Huang, G.: Constructing Craig interpolation formulas. In: Du, D.-Z., Li, M. (eds.) COCOON 1995. LNCS, vol. 959, pp. 181–190. Springer, Heidelberg (1995). https://doi.org/10.1007/BFb0030832

27. Jovanović, D.: Solving nonlinear integer arithmetic with MCSAT. In: Bouajjani, A., Monniaux, D. (eds.) VMCAI 2017. LNCS, vol. 10145, pp. 330–346. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-52234-0_18
28. Jovanovic, D., Barrett, C., De Moura, L.: The design and implementation of the model constructing satisfiability calculus. In: Ray, S., Jobstmann, B. (eds.) 2013 Formal Methods in Computer-Aided Design, pp. 173–180. IEEE (2013)
29. Jovanović, D., de Moura, L.: Solving non-linear arithmetic. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS (LNAI), vol. 7364, pp. 339–354. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31365-3_27
30. Jovanović, D., Dutertre, B.: Property-directed k-induction. In: Piskac, R., Talupur, M., Veith, H. (eds.) 2016 Formal Methods in Computer-Aided Design (FMCAD), pp. 85–92. IEEE (2016)
31. Jovanović, D., Dutertre, B.: LibPoly: a library for reasoning about polynomials. In: Proceedings 15th International Workshop on Satisfiability Modulo Theories (SMT 2017) (2017)
32. Kapur, D., Majumdar, R., Zarba, C.G.: Interpolation for data structures. In: Young, M., Devanbu, P. (eds.) Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 105–116 (2006)
33. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-based model checking for recursive programs. Formal Methods Syst. Des. **48**(3), 175–205 (2016). https://doi.org/10.1007/s10703-016-0249-4
34. Krajíček, J.: Interpolation theorems, lower bounds for proof systems, and independence results for bounded arithmetic. J. Symbolic Logic **62**(2), 457–486 (1997)
35. Kupferschmid, S., Becker, B.: Craig interpolation in the presence of non-linear constraints. In: Fahrenberg, U., Tripakis, S. (eds.) FORMATS 2011. LNCS, vol. 6919, pp. 240–255. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24310-3_17
36. Mahdi, A., Scheibler, K., Neubauer, F., Fränzle, M., Becker, B.: Advancing software model checking beyond linear arithmetic theories. In: Bloem, R., Arbel, E. (eds.) HVC 2016. LNCS, vol. 10028, pp. 186–201. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-49052-6_12
37. McMillan, K.L.: An interpolating theorem prover. Theor. Comput. Sci. **345**(1), 101–121 (2005)
38. McMillan, K.L.: Quantified invariant generation using an interpolating saturation prover. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 413–427. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_31
39. McMillan, K.L.: Interpolation: proofs in the service of model checking. In: Hanbook of Model-Checking. Springer (2014)
40. Mishra, B.: Algorithmic Algebra. Springer, New York (1993). https://doi.org/10.1007/978-1-4612-4344-1
41. Pudlák, P.: Lower bounds for resolution and cutting plane proofs and monotone computations. J. Symbolic Logic **62**(3), 981–998 (1997)
42. Schindler, T., Jovanović, D.: Selfless interpolation for infinite-state model checking. In: VMCAI 2018. LNCS, vol. 10747, pp. 495–515. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-73721-8_23

An SMT Solver for Regular Expressions and Linear Arithmetic over String Length

Murphy Berzish[1(✉)], Mitja Kulczynski[2], Federico Mora[3], Florin Manea[4], Joel D. Day[5], Dirk Nowotka[2], and Vijay Ganesh[1]

[1] University of Waterloo, Waterloo, Canada
mtrberzi@uwaterloo.ca
[2] Kiel University, Kiel, Germany
[3] University of California, Berkeley, USA
[4] University of Göttingen and Campus-Institute Data Science, Göttingen, Germany
[5] Loughborough University, Loughborough, UK

Abstract. We present a novel length-aware solving algorithm for the quantifier-free first-order theory over regex membership predicate and linear arithmetic over string length. We implement and evaluate this algorithm and related heuristics in the Z3 theorem prover. A crucial insight that underpins our algorithm is that real-world regex and string formulas contain a wealth of information about upper and lower bounds on lengths of strings, and such information can be used very effectively to simplify operations on automata representing regular expressions. Additionally, we present a number of novel general heuristics, such as the prefix/suffix method, that can be used to make a variety of regex solving algorithms more efficient in practice. We showcase the power of our algorithm and heuristics via an extensive empirical evaluation over a large and diverse benchmark of 57256 regex-heavy instances, almost 75% of which are derived from industrial applications or contributed by other solver developers. Our solver outperforms five other state-of-the-art string solvers, namely, CVC4, OSTRICH, Z3seq, Z3str3, and Z3-Trau, over this benchmark, in particular achieving a speedup of 2.4× over CVC4, 4.4× over Z3seq, 6.4× over Z3-Trau, 9.1× over Z3str3, and 13× over OSTRICH.

Keywords: String solvers · SMT solvers · Regular expressions

1 Introduction

Satisfiability Modulo Theories (SMT) solvers that support theories over regular expression (regex) membership predicate and linear arithmetic over length of strings, such as CVC4 [25], Z3str3 [8], Norn [3], S3P [39], and HAMPI [22], have enabled many important applications in the context of analysis of string-intensive programs. Examples include symbolic execution and path analysis [11,32], as well as security analyzers that make use of string and regex constraints for input sanitization and validation [5,33,35]. Regular expression libraries in programming languages provide very intuitive and popular ways for developers to express input validation, sanitization, or pattern matching constraints. Common to all these program analysis applications is the requirement for a rich quantifier-free (QF) first-order theory over strings, regexes, and integer arithmetic over string length. Unfortunately, the QF first-order theory of strings containing regex constraints, linear integer arithmetic over string length, string-number conversion, and string concatenation (but no string equations[1]) is undecidable [7,9]. In a previous paper [19] we showed that a related QF first-order theory over word equations, linear integer arithmetic over string length, and string-number conversion predicate, but without regular expressions is also undecidable. It can also be shown that many non-trivial fragments of this theory are hard to decide (e.g., they have exponential-space lower bounds or are PSPACE-complete). Therefore, the task of creating efficient solvers to handle practical string constraints that belong to fragments of this theory remains a very difficult challenge.

Many modern solvers typically handle regex constraints via an automata-based approach [4]. Automata-based methods are powerful and intuitive, but solvers must handle two key practical challenges in this setting. The first challenge is that many automata operations, such as intersection, are computationally expensive, yet handling these operations is required in order to solve constraints that are relevant to real-world applications. The second challenge relates to the integration of length information with regex constraints. Length constraints derived from automata may imply a disjunction of linear constraints, which is often more challenging for solvers to handle than a conjunction.

As we demonstrate in this paper, the challenges of using automata-based methods can be addressed via prudent use of *lazy extraction of implied length constraints* and *lazy regex heuristics* in order to avoid performing expensive automata operations when possible. Inspired by this observation, we introduce a length-aware automata-based algorithm, Z3str3RE (and its implementation as part of the Z3 theorem prover [18]), for solving regex constraints and linear integer arithmetic over length of string terms. Z3str3RE takes advantage of the compactness of automata in representing regular expressions, while at the same time mitigating the effects of expensive automata operations such as intersection by leveraging length information and lazy heuristics.

Contributions: We make the following contributions in this paper.

Z3str3RE: An SMT Solver for Regular Expressions and Linear Integer Arithmetic over String Length. In Sect. 3, we present a novel decision procedure for the QF first-order theory over regex membership predicate and linear integer arithmetic over string length. We also describe its implementation, Z3str3RE, as part of the Z3 theorem prover [8,18]. The basic idea of our algorithm is that formulas obtained from practical applications have many implicit and explicit length constraints that can be used to reason efficiently about automata representing regexes. In Sect. 4 we present four heuristics that aid in solving regular expression constraints and that can be leveraged in general settings. Specifically, we present a heuristic to derive explicit length information directly from

[1] We use the terms "word" and "string" interchangeably in this paper.

regexes, a heuristic to perform expensive automata operations lazily, a heuristic to refine lower and upper bounds on lengths of string terms with respect to regex constraints, and a prefix/suffix over-approximation heuristic to find empty intersections without constructing automata. All heuristics are designed to guide the search and avoid expensive automata operations whenever possible. Our solver, Z3str3RE, handles the above theory as well as extensions (e.g. word equations and substring function) via the existing support in Z3str3. We focus on the core algorithm as it is the centerpiece of our regex solver. We also carefully distinguish the novelty of our method from previous work.

Empirical Evaluation and Comparison of Z3str3RE[2] Against CVC4, OSTRICH, Z3seq, Z3str3, and Z3-Trau: To validate the practical efficacy of our algorithm, we present a thorough and extensive evaluation of Z3str3RE in Sect. 5, where we compare it against CVC4 [24], OSTRICH [15], Z3's sequence solver [18], Z3str3 [42], and Z3-Trau [1] on 57256 instances across four regex-heavy benchmarks with connections to industrial security applications, including instances from Amazon Web Services and AutomatArk [16]. Z3str3RE significantly outperforms other state-of-the-art tools on the benchmarks considered, having more correctly solved instances in total, lower running time, and fewer combined timeouts/unknowns than other tools, and no soundness errors or crashes. We note that almost 75% of the benchmarks were obtained from industrial applications or other solver developers. Over all the benchmarks, we demonstrate a speedup of $2.4\times$ over CVC4, $4.4\times$ over Z3seq, $6.4\times$ over Z3-Trau, $9.1\times$ over Z3str3, and $13\times$ over OSTRICH.

2 Preliminaries

This section contains some basic definitions as well as a brief overview of the theoretical results which shape the landscape in which we state our contribution.

2.1 Basic Definitions

We first describe the syntax and semantics of the input language supported by our solver Z3str3RE (Algorithm 1).

Syntax: The core algorithm we present in Sect. 3 accepts formulas of the quantifier-free many-sorted first-order theory of regex membership predicates over strings and linear integer arithmetic over string length function. The syntax of this theory is shown in Fig. 1.

We denote the set of all string variables and all integer variables as Var_{str} and Var_{int} respectively, and the set of all string constants and all integer constants as Con_{str} and Con_{int} respectively. String constants are any sequence of zero or more characters over a finite alphabet (e.g., ASCII).

Atomic formulas are regular expression membership constraints and linear integer (in)equalities. Regex terms are denoted recursively over regex concatenation, union, Kleene star, and complement, and for a string constant w, the

[2] A reproduction package is available at https://figshare.com/s/5ae73a6f3c55f5c5e4c1.

$$\begin{array}{ll} F & ::= Atom \ \mid \ F \wedge F \ \mid \ F \vee F \ \mid \ \neg F \\ Atom & ::= t_{str} \in RE \ \mid \ A_{int} \\ A_{int} & ::= t_{int} = t_{int} \ \mid \ t_{int} < t_{int} \\ RE & ::= \text{“}w\text{”} \mid RE \cdot RE \mid RE \cup RE \mid RE^{*} \mid \overline{RE}, \text{ with } w \in \mathrm{Con}_{\mathrm{str}} \\ t_{int} & ::= m \mid v \mid len(t_{str}) \mid t_{int} + t_{int} \mid m \cdot t_{int}, \text{with } m \in \mathrm{Con}_{\mathrm{int}}, v \in Var_{int} \\ t_{str} & ::= s, \text{ with } s \in \mathrm{Var}_{\mathrm{str}} \cup \mathrm{Con}_{\mathrm{str}} \end{array}$$

Fig. 1. Syntax of the input language accepted by Algorithm 1. Z3str3RE accepts an extension of this syntax supporting word equations and other string terms.

regex term “w” represents the regular language containing w only. All regex terms must be grounded (i.e. cannot contain variables). Linear integer arithmetic terms include integer constants and variables, addition, and string length. Multiplication by a constant is expanded to repeated addition. String terms are either string variables or string constants. The length of a string S is denoted by $len(S)$, the number of characters in S. The empty string has length 0.

Our implementation Z3str3RE supports the theory in Fig. 1 extended with more expressive functions and predicates, including word equations (equality between arbitrary string terms) and functions such as `indexof` and `substr` that are needed for program analysis. Z3str3RE handles these terms via existing support in Z3str3. We focus on the above input language in the presentation of our algorithm in this paper and theoretical content.

Semantics: We refer the reader to [42] for a detailed description of the semantics of standard terms in this theory. We focus here on the semantics of terms which are less commonly known. The regex membership predicate $S \in R$, where S is a string term and R is a regex term, is defined by structural recursion as follows:

$$\begin{array}{ll} S \in \text{“}w\text{”} & \text{iff } S = w \text{ (where } w \text{ is a string constant)} \\ S \in R_1 \cdot R_2 & \text{iff there exist strings } S_1, S_2 \text{ with } S = S_1 \cdot S_2, S_1 \in R_1, S_2 \in R_2 \\ S \in R_1 \cup R_2 & \text{iff either } S \in R_1 \text{ or } S \in R_2 \\ S \in R^{*} & \text{iff either } S = \epsilon \text{ or there exists a positive integer } n \text{ such that} \\ & \quad S = S_1 \cdot S_2 \cdot \ldots \cdot S_n \text{ and } S_i \in R \text{ for each } i = 1 \ldots n \\ S \in \overline{R} & \text{iff } S \notin R \text{ (that is, } S \in R \text{ is false)} \end{array}$$

2.2 Theoretical Landscape

To put our contributions in context, we briefly discuss a series of (un)decidability and complexity results developed around the fragments and extensions of the theory supported by Z3str3RE.

In particular, we consider extensions which may have a string-number conversion predicate *numstr*[3] and/or string concatenation. Both extensions are

[3] We introduce *numstr*, which is not part of the SMT-LIB standard, in order to simplify presentation of the theoretical results. The predicate is no more expressive than the standard operators `str.to_int`/`str.from_int`, except that those terms handle decimal inputs. The results easily extend to other (finite) alphabets including decimal/hexadecimal digits with appropriate case analysis.

important to real-world program analysis. The predicate *numstr* has the syntax $numstr(t_{int}, t_{str})$ and the following semantics: $numstr(n, s)$ is true for a given integer n and string s iff s is a valid binary representation of the number n (possibly with leading zeros) and n is a non-negative integer. That is, s only contains the characters 0 and 1, and $\sum_{i=0}^{len(s)-1} s'[i]2^{len(s)-i-1} = n$, where $s'[i]$ is 0 if the ith character in s is '0' and 1 if that character is '1'. String concatenation has the syntax $t_{str} ::= t_{str} \cdot t_{str}$ and the usual semantics defined by SMT-LIB [10].

In the following, $T_{LRE,n,c}$ is the quantifier-free many-sorted first-order theory of linear integer arithmetic over string length function (L), regex (RE) membership predicates, string-number conversion (n), and string concatenation (c) [4]. The following quantifier-free fragments of $T_{LRE,n,c}$ are of interest: $T_{LRE,c}$, T_{LRE}, $T_{RE,n,c}$, $T_{RE,n}$, and T_{RE}. The fragment $T_{LRE,c}$ (respectively, T_{LRE}) has all functions and predicates of $T_{LRE,n,c}$ except the string-number conversion predicate (and, respectively, except the string concatenation function). The theory $T_{RE,n,c}$ (respectively, $T_{RE,n}$ and T_{RE}) has all functions and predicates of $T_{LRE,n,c}$ except the length function (and, respectively, the string concatenation function, and, in the case of T_{RE}, the string-number conversion predicate). Note that while all these theories allow equalities between terms of sort Int, they do not allow equalities between terms of sort Str and cannot express general word equations.

The theoretical landscape is laid out as follows. Firstly, following the results and techniques introduced in [3], we obtain that $T_{LRE,c}$ and, in particular, T_{LRE} is decidable. A procedure deciding a formula from $T_{LRE,c}$ would first construct for each variable (string or integer), based on the regular expression constraints and length constraints which involve it, a finite automaton, then reduce the problem of checking the satisfiability of the formula to checking whether the constructed automata accept at least one string. A similar approach shows that $T_{RE,n}$ is decidable. We observe that the presence of complements in regular expressions is an inherent source of complexity for these procedures. Indeed, we can easily encode the universality problem for regular expressions as a formula in the theory T_{RE}. Moreover, given a regex R of length n over an alphabet Σ, deciding whether $L(R) = \Sigma^*$ is equivalent to deciding the satisfiability of the formula φ of T_{RE} consisting of the atoms $x \in \overline{R}$ and $x \in \Sigma^*$. Accordingly, by the results from [37], if the choice for R is restricted to regular expressions with at least k stacked complements, then there exists a positive rational number c such that the considered problems are not contained in NSPACE$\left(\underbrace{2^{2^{2^{\cdots 2}}}}_{k-1\text{times}}{}^{cn}\right)$.
In other words, the depth of the stack of complements of the formula translates to the height of the tower of exponents in the complexity of deciding that formula φ. On the other hand, if we only consider regular expressions without stacked complements, then the decision problems for the considered theories are PSPACE-complete. Indeed, the automata-based approach described above can be implemented to work in nondeterministic polynomial space; strongly related complexity results are obtained in [26,27].

[4] Note that the fragments considered here do not include word equations.

Algorithm 1: Z3str3RE's length-aware algorithm for the theory T_{LRE} of regex and integer constraints

```
Input   : Conjunction $\phi$ of constraints of the form $S \in RE$, and conjunction $\psi$ of linear
          integer arithmetic constraints over string lengths
Output  : SAT or UNSAT
forall constraints $S \in RE$ in $\phi$ do
    $L_S \leftarrow$ ComputeLengthAbstraction($S$) ;
    $L_{RE} \leftarrow$ ComputeLengthAbstraction($RE$) ;
    if $\psi \cup L_S \cup L_{RE}$ inconsistent then
        return UNSAT
    end
    refine $L_S$ as tightly as possible with respect to $L_{RE}$;
end
forall strings $S_i$ occurring in $\phi$ do
    let $\mathcal{R}$ be the set of all regexes $RE$ in all terms $S_i \in RE$ ;
    Automaton $I \leftarrow$ intersection of all automata corresponding to regexes in $\mathcal{R}$ ;
    if $I$ is empty then
        return UNSAT
    else
        $L_I \leftarrow$ ComputeLengthAbstraction($I$) ;
    end
end
$\mathcal{L}_S \leftarrow$ the union of all length abstractions $L_S$;
$\mathcal{L}_{RE} \leftarrow$ the union of all length abstractions $L_{RE}$;
$\mathcal{L}_I \leftarrow$ the union of all length abstractions $L_I$;
if $\psi \cup \mathcal{L}_S \cup \mathcal{L}_{RE} \cup \mathcal{L}_I$ has any solution $M$ then
    forall strings $S$ occurring in $\phi$ do
        obtain $len(S)$ from $M$ ;
        let $\mathcal{A}$ be the set of all automata for all regexes $RE$ in all terms $S \in RE$ ;
        Automaton $J \leftarrow$ intersection of all terms in $\mathcal{A}$ ;
        $S \leftarrow$ any string of length $len(S)$ in $J$ ;
    end
    return SAT
else
    return UNSAT
end
```

At the opposite end of the spectrum is the theory $T_{LRE,n,c}$, which is undecidable. Indeed, one can show that the more specific theory $T_{RE,n,c}$ (i.e. disallowing arithmetic over length) has equivalent expressive power to the theory of word equations with regular constraints, a predicate allowing the comparison of the length of string terms, and the *numstr* predicate. Therefore, using the techniques from [17], one can show that the theory $T_{LRE,n,c}$, in which we additionally allow arithmetic over length, is undecidable [7].

3 Length-Aware Regular Expression Algorithm

This section outlines the high-level algorithm used by Z3str3RE to solve the satisfiability problem for T_{LRE}, and its extension based on length-aware heuristics.

3.1 High-Level Algorithm

The pseudocode presented in Algorithm 1 captures the essence of Z3str3RE regex solver. Implementation-specific details are omitted for clarity. Z3str3RE incorporates a version of this algorithm as part of a DPLL(T)-style interaction with a core solver for Boolean combinations of atoms and other theory solvers able to handle arithmetic constraints and other terms. The tool handles string concatenation, string equality, and other string terms and predicates besides regex membership and string length via existing support in Z3str3, and leverages Z3's integer arithmetic solver for arithmetic reasoning and model construction. This high-level presentation is expanded in Sect. 4, where we describe several heuristics used in our implementation as part of the Z3str3RE tool.

The algorithm takes as input a conjunction ϕ of regex membership constraints and a conjunction ψ of linear integer arithmetic constraints over the lengths of string variables appearing in ϕ. Without loss of generality, it is assumed that all constraints in ϕ are positive; negative constraints $S \notin RE$ can be replaced with the positive complement $S \in \overline{RE}$. The algorithm returns SAT iff there is a satisfying assignment to all string variables consistent with the regex constraints ϕ and length constraints ψ. It is assumed that the algorithm has access to a decision procedure for checking the consistency of linear integer arithmetic constraints and for obtaining satisfying assignments to these constraints (in our implementation, this is fulfilled by Z3's arithmetic solver).

Lines 1–8 check whether the length information implied by ϕ is consistent with ψ. The function `ComputeLengthAbstraction` takes as input either a string term S or a regex RE and computes a system of length constraints corresponding to derived length information from string constraints or possible lengths of words accepted by the regex RE. This abstraction is exact, not an over-approximation. For example, given the regex $(abc)^*$ as input, `ComputeLengthAbstraction` would construct the length abstraction $S \in (abc)^* \rightarrow len(S) = 3n, n \geq 0$ for a fresh integer variable n. If the length abstractions are inconsistent with the given length constraints, there can be no solution which satisfies both the length and regex constraints, and hence the algorithm returns UNSAT. Otherwise, line 7 refines the length abstraction L_S with respect to the regex RE. This improves the efficiency of finding solutions to the augmented system of length constraints later in the algorithm. In our implementation, the lower and upper bounds of the length of S are checked against the lengths of accepting paths in the automaton for RE. For instance, if L_S implies that $len(S) \geq 5$, but the shortest accepting path in the automaton has length 7, the lower bound is refined to $len(S) \geq 7$.

Lines 9–17 check that the intersection of all automata constraining each string variable is non-empty. Although intersecting automata is relatively expensive (as it runs in quadratic time w.r.t. the size of the intersected automata), it is still more efficient to do this before enumerating length assignments, and taking the intersection here is necessary to maintain soundness. (The heuristics in Sect. 4 illustrate some methods by which this computation can be made more efficient or even avoided.) If the length information is consistent, the algorithm adds a length abstraction constraint L_I encoding the lengths of all possible solutions to the intersection I.

By construction of $\psi \cup \mathcal{L}_S \cup \mathcal{L}_{RE} \cup \mathcal{L}_I$, the input formula is satisfiable iff this system of integer constraints has a solution. If such a solution M exists, lines 22–28 construct an assignment for each string variable with respect to its length assignment. A solution must exist as the lengths of strings considered are limited to those lengths for which the intersection of the corresponding automata is non-empty; the solution is consistent by construction with both the input length constraints and string constraints. If a solution M does not exist, then the constraints $\phi \wedge \psi$ are not jointly satisfiable, and the algorithm returns UNSAT.

We demonstrate soundness, completeness, and termination of Algorithm 1 as follows. On line 4 we check whether $\psi \cup L_S \cup L_{RE}$ is satisfiable. If not, we return UNSAT on line 5. Lines 9–17 check whether the intersection of regex constraints for each string variable is empty. If so, we return UNSAT; otherwise, we add an additional constraint encoding the lengths of all strings in this intersection. Therefore, $\psi \cup \mathcal{L}_S \cup \mathcal{L}_{RE} \cup \mathcal{L}_I$ has a solution iff there exists an assignment to each string variable that is consistent with the arithmetic constraints ψ and that corresponds to the length of a solution in the intersection of its regex constraints $\mathcal{L}_I$. Lines 22–28 construct this solution if it exists. Therefore, Algorithm 1 is a decision procedure for the QF first-order theory of regex constraints, string length, and linear integer arithmetic.

As previously mentioned, Z3str3RE supports other high-level operations that are not part of this theory via existing support in Z3str3. An extension to this algorithm provides support for including these operations, which may render the theory undecidable. These terms are not in Algorithm 1 because their inclusion would make the algorithm incomplete (see Sect. 2.2). Algorithm 1 describes the part of the implementation which is novel and complete.

4 Length-Aware and Prefix/Suffix Heuristics in Z3str3RE

In this section, we describe the length-aware heuristics that are used in Z3str3RE to improve the efficiency of regular expression reasoning. We present an empirical evaluation of the power of these heuristics in Sect. 5.6.

4.1 Computing Length Information from Regexes

The first length-aware heuristic is used when constructing the length abstraction on line 3. If the regex can be easily converted to a system of equations describing the lengths of all possible solutions (for instance, in the case when it does not contain any complements or intersections), this system can be returned as the abstraction without constructing the automaton for RE yet. As previously illustrated, for example, given the regex $(abc)^*$ as input, `ComputeLengthAbstraction` would construct the length abstraction $S \in (abc)^* \rightarrow len(S) = 3n, n \geq 0$ for a fresh integer variable n. Note that this can be done from the syntax of the regex without converting it to an automaton. Deriving length information from the automaton would be simple by, for example, constructing a corresponding unary automaton and converting to Chrobak normal form. However, performing automata construction lazily means we cannot rely on having an automaton in all cases; this technique also provides length information even when constructing an automaton would be expensive.

In cases where we cannot directly infer the length abstraction, the heuristic will fix a lower bound on the length of words in RE, and possibly an upper bound if it exists. Reasoning about the length abstraction early in the procedure gives our algorithm the opportunity to detect inconsistencies before expensive automaton operations are performed. This gives the arithmetic solver more opportunities to propagate facts discovered by refinement and potentially more chances to find inconsistencies or learn further derived facts.

4.2 Optimizing Automata Operations via Length Information

Similarly, computing the intersection I in line 11 is done lazily in the implementation of Z3str3RE and over several iterations of the algorithm. The most expensive intersection operations can be performed at the end of the search, after as much other information as possible has been learned. We use the following heuristics recursively to estimate the "cost" of each operation without actually constructing any automata:

- For a string constant, the estimated cost is the length of the string.
- For a concatenation or a union of two regex terms X and Y, the estimated cost is the sum of the estimates for X and Y.
- For a regex term X^*, the estimated cost is twice the estimate for X.
- For a regex term X under complement, the estimated cost is the product of the estimates obtained from subterms of X.

In essence, the constructions which "blow up" the least are expected to be the least expensive and are performed first. In the best-case scenario, this could mean avoiding the most expensive operations completely if an intersection of smaller automata ends up being empty. In the worst case, all intersections are computed eventually, as this is necessary to maintain the soundness of our approach.

4.3 Leveraging Length Information to Optimize Search

Our implementation communicates integer assignments and lower/upper bounds with the external arithmetic solver in order to prune the search space. Checking for length assignments is done in practice as an abstraction-refinement loop involving Z3's arithmetic solver. The arithmetic solver proposes a single candidate model for the system of arithmetic constraints; the regex algorithm checks whether that model has a corresponding solution over the regex constraints. If it does not, it asserts a conflict clause blocking that combination of length assignments and regex constraints from being considered again. This is necessary in a DPLL(T)-style solver such as Z3 in order to handle Boolean structure in the input formula.

4.4 Prefix/Suffix Over-Approximation Heuristic

As previously mentioned, computing automata intersections is expensive, but in many cases it is necessary in order to prove that a set of intersecting regex constraints has no solution. In some cases, this can be done "by inspection" from the syntax of the regex terms without constructing or intersecting any automata. From the structure of a regular expression, it is easy to determine the first letter of all possible accepted strings that it matches. If several regexes would be intersected over the same string term, this is used to check whether these regexes have a prefix of length one in common. If they do not, their intersection cannot contain any strings other than the empty string (and we can also check whether the empty string could be accepted by a similar syntactic approach). A similar construction for suffixes of length 1 is also used. In this way, the heuristic can infer that the intersection of several regex constraints is either empty, resulting in a conflict clause, or can only contain the empty string, resulting in a new fact and a simplification of the formula – without actually constructing the intersection or, in fact, constructing any automata for these regexes.

For example, consider the following regex constraints on a variable X:

$$X \in (abc)^*$$
$$X \in a^+ \mid b^+$$

In the first constraint, the pattern abc is matched zero or more times, and could be empty; therefore, either X is empty or it must start with a and end with c. In the second constraint, each pattern is matched at least once, and cannot be empty; therefore X must start with a or b, end with a or b, and cannot be the empty string. Observe that according to the prefix heuristic, these constraints are consistent, since a is a valid prefix of both regexes; however, according to the suffix heuristic, they are inconsistent, as the possible suffixes a and b of the second regex do not include c, and the empty string is not a solution to both constraints. Hence these constraints are not jointly satisfiable.

As demonstrated, all of these facts are derived from the syntax of the regular expression without constructing any automata. By constructing an over-approximation of the possible solutions of X allowed by regex constraints, the heuristic can determine that their intersection is empty (or can only contain the empty string) without computing it precisely using expensive automata-based reasoning. We limit this heuristic to the first letter as each additional letter requires exponentially more space.

5 Empirical Results

In this section, we describe the empirical evaluation of Z3str3RE, our implementation of the length-aware regular expression algorithm presented in Sect. 3, to validate the effectiveness of the techniques presented. We evaluate the correctness and efficiency of our tool against other solvers, as well as against different configurations of the tool in order to demonstrate the efficacy of our heuristics.

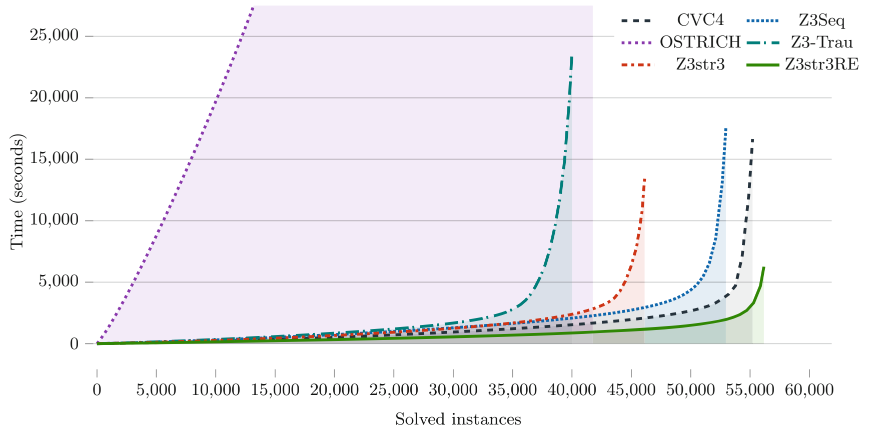


Fig. 2. Cactus plot summarizing performance on all benchmarks. Z3str3RE has the best overall performance.

Table 1. Combined results of string solvers on all benchmarks. **Z3str3RE** has the best overall performance on all benchmarks compared to CVC4, OSTRICH, Z3seq, Z3str3, and Z3-trau and the biggest lead with a score of 1.02.

	CVC4	Z3Seq	OSTRICH	Z3-Trau	Z3str3	Z3str3RE
Sat	33310	31550	22499	24133	27563	**33820**
Unsat	21897	21411	19281	21038	18566	**22339**
Unknown	**0**	**0**	10901	6504	1164	291
Timeout	2049	4295	4575	5581	9963	**806**
Soundness error	**0**	**0**	28	5325	13	**0**
Program crashes	**0**	**0**	**0**	2477	2	**0**
Total correct	55207	52961	41752	39846	46116	**56159**
Contribution score	95.99	19.87	–	–	–	**145.07**
Time (s)	57625.499	103487.844	305243.413	150288.386	213698.954	**23339.266**
Time w/o timeouts (s)	16645.499	17587.844	213743.413	38668.386	14438.954	**7219.266**

5.1 Empirical Setup and Solvers Used

We compare Z3str3RE against five other leading string solvers available today. CVC4 [24] is a general-purpose SMT solver which reasons about strings and regular expressions algebraically. Z3str3 [8] is the latest solver in the Z3-str family, and uses a reduction to word equations to reason about regular expressions. Z3str3RE is based on Z3str3 except for the length-aware algorithm and heuristics described in Sects. 3 and 4. Z3seq [36] is the Z3 sequence solver, implemented by Nikolaj Bjørner and others at Microsoft Research, as part of the Z3 theorem prover. Z3seq uses a new theory of derivatives for solving extended regular expressions. Z3-Trau [1] is also based on Z3 and uses an automata-based approach known as "flat automata" with both under- and over-approximations. OSTRICH [15] uses a reduction from string functions (including word equations) to a model-checking problem that is

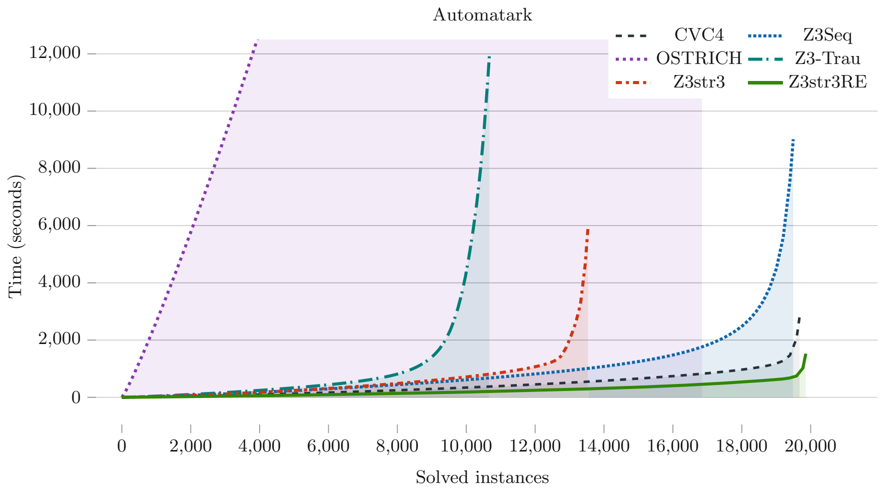


Fig. 3. Cactus plot summarizing detailed performance on Automatark benchmark.

solved using the SLOTH tool and an implementation of IC3. We used CVC4's binary version 1.8, commit `59e9c87` of Z3str3, the sequence solver included in Z3's binary version 4.8.9, Z3-Trau commit `1628747`, and OSTRICH version 1.0.1. All of these tools support the full SMT-LIB standard for strings. We did not compare against the Z3str2 [42] or Norn [3] solvers as neither tool supports the `str.to_int` or `str.from_int` terms which represent string-number conversion, which are used in some sanitizer benchmarks. Additionally, Norn does not support many of the other high-level string terms such as `indexof` or `substr` which are used in the benchmarks. The ABC [4] solver handles string and length constraints by conversion to automata. However, their method over-approximates the solution set of the input formula which may be unsound. Thus, we excluded ABC from our evaluation. We also were unable to evaluate against Trau [2] as the provided source code did not compile. All evaluations were performed on a server running Ubuntu 18.04.4 LTS with two AMD EPYC 7742 processors and 2TB of memory using the ZaligVinder [23] benchmarking framework. A 20 s timeout was used. We cross-verified the models generated by each solver for satisfiable instances against all competing solvers.

5.2 Benchmarks

The comparison was performed on four suites of regex-based benchmarks with a total of 57256 instances. In total, almost 75% of the instances in our evaluation came from previously published industrial benchmarks or other solver developers. Under 10% contain extended regular expressions (having either complement or intersection, or both) and 53% contain only regex predicates. Only 201 instances fall into the undecidable theory $T_{LRE,n,c}$. More details can be found in [7] where we analyse the benchmarks in greater detail. We briefly describe each benchmark's origin and composition.

Table 2. Detailed results for the Automatark benchmark. Z3str3RE has the biggest lead with a score of 1.01.

	CVC4	Z3Seq	OSTRICH	Z3-Trau	Z3str3	Z3str3RE
Sat	14376	14204	11461	8157	9151	**14437**
Unsat	5304	5290	5381	3817	4385	**5422**
Unknown	1	**0**	15	5045	406	**0**
Timeout	298	485	3122	2960	6037	**120**
Soundness error	**0**	**0**	**0**	1300	**0**	**0**
Program crashes	**0**	**0**	**0**	1063	2	**0**
Total correct	19680	19494	16842	10674	13536	**19859**
Contribution score	1.0	1.0	**2.0**	–	0.0	0.5
Time (s)	8789.425	18718.425	158910.126	80021.352	126825.967	**3925.150**
Time w/o timeouts (s)	2829.425	9018.425	96470.126	20821.352	6085.967	**1525.150**

AutomatArk is a set of 19979 benchmarks based on a collection of real-world regex queries collected by Loris D'Antoni from the University of Wisconsin, Madison, USA. We translated the provided regexes [16] into SMT-LIB syntax resulting in two sets of instances: a "simple" set with a single regex membership predicate per instance, and a "complex" set with 2–5 regex membership predicates (possibly negated) over a single variable per instance. The instances in this benchmark are evenly divided between simple and complex problems.

RegEx-Collected is a set of 22425 instances taken from existing benchmarks with the purpose of evaluating the performance of solvers against real-world regex instances. This benchmark includes all instances from the AppScan [41], BanditFuzz,[5] JOACO [38], Kaluza [33], Norn [3], Sloth [21], Stranger [40], and Z3str3-regression [8] benchmarks in which at least one regex membership constraint appears.[6] No additional restrictions are placed on which instances were chosen besides the presence of at least one regex membership predicate. This benchmark tests solvers against challenging instances from widely distributed benchmark suites. Additionally, these instances may contain regex terms in any context and with any other supported string operators. As a result, the benchmark is also exemplary of how string solvers perform in the presence of operations and predicates that are relevant to program analysis.

StringFuzz-regex-generated is a set of 4170 problems generated by the StringFuzz string instance fuzzing tool [12]. These instances only contain regular expression and linear arithmetic constraints. This benchmark isolates the regex performance of a string solver in the context of mixed regex and arithmetic constraints. Tools with better regex and arithmetic solvers should perform better. Fuzz testing, as performed in the **StringFuzz-regex-generated** benchmark, has been shown to be extremely productive in discovering bugs and performance

[5] The BanditFuzz benchmark is an unpublished suite obtained via private communication with the authors.

[6] Other benchmark suites available to us, including the PyEx, PISA, and Kausler benchmarks, did not include any regex membership constraints.

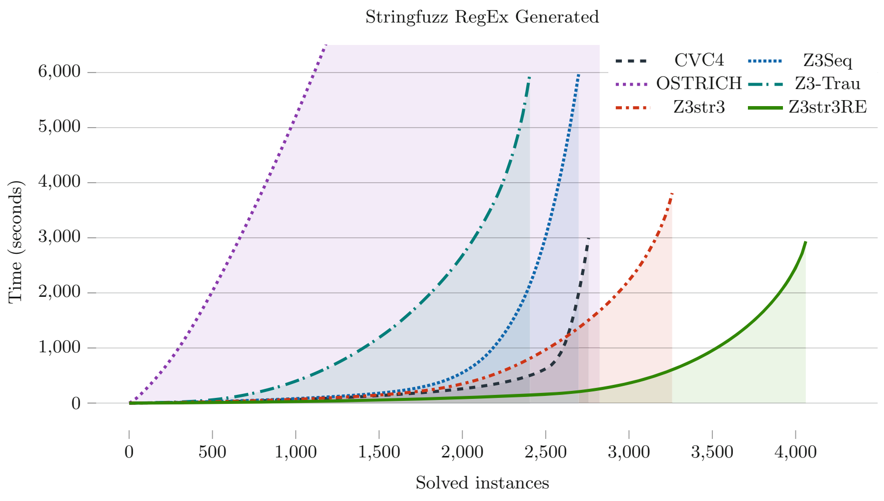


Fig. 4. Cactus plot showing detailed results for the StringFuzz-regex-generated benchmark.

Table 3. Detailed results for the StringFuzz-regex-generated benchmark. Z3str3RE has the biggest lead with a score of 1.25.

	CVC4	Z3Seq	OSTRICH	Z3-Trau	Z3str3	Z3str3RE
Sat	2316	2001	2005	1590	3227	**3231**
Unsat	442	697	819	824	32	**830**
Unknown	**0**	**0**	1	192	**0**	**0**
Timeout	1412	1472	1345	1564	911	**109**
Soundness error	**0**	**0**	**0**	8	**0**	**0**
Program crashes	**0**	**0**	**0**	192	**0**	**0**
Total correct	2758	2698	2824	2406	3259	**4061**
Contribution score	0.0	**3.17**	2.0	–	0.0	0.17
Time (s)	31236.207	35409.000	51571.800	37323.550	22031.636	**5116.456**
Time w/o timeouts (s)	2996.207	5969.000	24671.800	6043.550	3811.636	**2936.456**

issues in SMT solvers. We included these instances because they exercise the performance of the solver on regex-heavy constraints in a way that the industrial benchmarks or instances obtained from other solver developers cannot.

StringFuzz-regex-transformed is a set of 10682 instances which were produced by transforming existing industrial instances with StringFuzz. We applied StringFuzz's transformers to instances supplied by Amazon Web Services related to security policy validation, handcrafted instances inspired by real-world input validation vulnerabilities, and the regex test cases in Z3str3's regression test suite. The instances contain regex constraints, arithmetic and length constraints, string-number conversion (*numstr*), string concatenation, word equations, and other high-level string operations such as `charAt`, `indexof`, and `substr`. As is

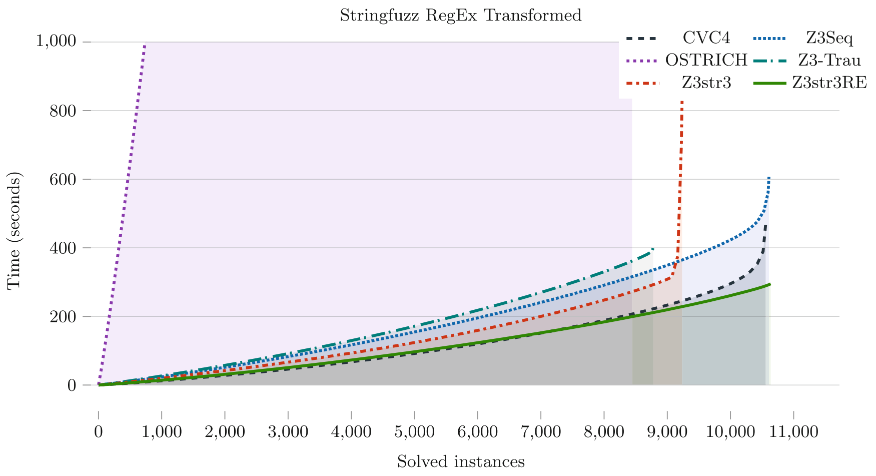


Fig. 5. Cactus plot showing detailed results for the StringFuzz-regex-transformed benchmark.

Table 4. Detailed results for the StringFuzz-regex-transformed benchmark. Z3str3RE has the biggest lead with a score of 1.0.

	CVC4	Z3Seq	OSTRICH	Z3-Trau	Z3str3	Z3str3RE
Sat	4541	**4633**	3899	3672	4417	4599
Unsat	6016	5976	4549	**6282**	4817	6037
Unknown	**0**	**0**	2233	721	**0**	6
Timeout	125	73	**1**	7	1448	40
Soundness error	**0**	**0**	5	1241	**0**	**0**
Program crashes	**0**	**0**	**0**	718	**0**	**0**
Total correct	10557	10609	8443	8713	9234	**10636**
Contribution score	0.5	0.0	–	–	0.0	**4.83**
Time (s)	2969.643	2066.935	23094.737	**722.545**	29788.245	1095.209
Time w/o timeouts (s)	469.643	606.935	23074.737	582.545	828.245	**295.209**

typical for fuzzing in software testing, the goal is to create a suite of tests from a given input that are similar in structure but that explore interesting behaviour not captured by a “typical” industrial instance. These transformed instances are often harder than the original industrial ones.

5.3 Comparison and Scoring Methods

We compare solvers directly against the total number of correctly solved cases, total time with and without timeouts, and total number of soundness errors and program crashes. We also computed the biggest lead winner and largest contribution ranking following the scoring system used by the SMT Competition [6]. Briefly, the biggest lead measures the proportion of correct answers of the leading tool to correct answers of the next ranking tool, and the contribution score measures what proportion of instances were solved the fastest by that solver. In accordance with the SMT Competition guidelines, a solver receives no contribution score (denoted as –) if it produces any incorrect answers on a given benchmark. In both cases, higher scores are better.

5.4 Analysis of Empirical Results

The cactus plot in Fig. 2 shows the cumulative time taken by each solver on all cases in increasing order of runtime. Solvers that are further to the right and closer to the bottom of the plot have better performance.

Overall Z3str3RE solves more instances and performs better than all competing solvers. Across all benchmarks, Z3str3RE is over 2.4× faster than CVC4, 4.4× faster than Z3seq, 6.4× faster than Z3-Trau, 9.1× faster than Z3str3, and 13× faster than OSTRICH (including timeouts). Additionally, Z3str3RE has fewer combined timeouts and unknowns than other tools considered, and no soundness errors or crashes. We summarize these results in Table 1. Notably, both Z3-Trau [1] and OSTRICH [15] had significant runtime issues in our experiments. Z3-Trau produced 5325 soundness errors and 2477 crashes on our benchmarks (13% of all instances), which is significantly higher than other tools used. OSTRICH produced 10901 “unknown” responses on the benchmarks (19% of all instances), due to both unsupported features and crashes, and also produced 28 soundness errors. Over all benchmarks, Z3str3RE produced 291 unknowns. There are several potential reasons for this; the solver may have encountered a resource limit and returned UNKNOWN, or it may have detected non-termination and returned UNKNOWN instead of looping forever. According to SMT Competition scoring, Z3str3RE won the division across all benchmarks with a lead of 1.02, and had the largest contribution to the division with a score of 145.07. CVC4 had a contribution score of 95.99, and Z3seq had a score of 19.87. OSTRICH, Z3-Trau, and Z3str3 received no contribution score as they each returned at least one incorrect answer. The presented results are typical of the performance of the evaluated tools over multiple runs. Results were cross-validated within runs and between multiple runs. For a random single instance, the sample variance in execution time for 100 runs is 0.001 (0.07% of average execution time). Over 57256 instances, this is negligible.

The empirical results make clear the efficacy of length-aware automata-based techniques for regular expression constraints when accompanied with length constraints (which is typical for industrial instances). The effectiveness of our technique is demonstrated particularly by comparing Z3str3RE with Z3str3, as the only differences between these tools are the length-aware regex algorithm and heuristics implemented in Z3str3RE and bug fixes. By improving the regex algorithm and applying our heuristics, we achieved a speedup of over 9x and solved over 10000 more cases than Z3str3.

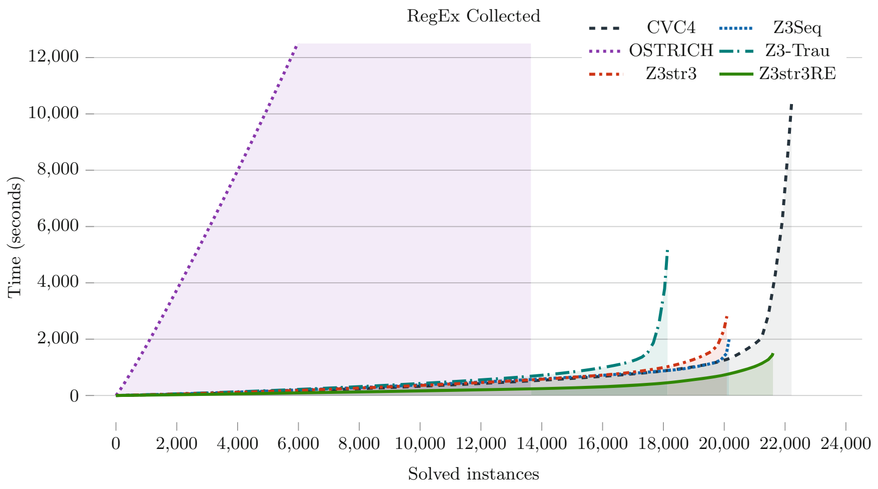


Fig. 6. Cactus plot showing detailed performance for the RegEx-Collected benchmark.

Table 5. Detailed results for the RegEx-Collected benchmark. CVC4 has the biggest lead with a score of 1.03.

	CVC4	Z3Seq	OSTRICH	Z3-Trau	Z3str3	Z3str3RE
Sat	**12077**	10712	5134	10714	10768	11553
Unsat	**10135**	9448	8532	10115	9332	10050
Unknown	**0**	**0**	8652	546	758	285
Timeout	213	2265	**107**	1050	1567	537
Soundness error	**0**	**0**	23	2776	13	**0**
Program crashes	**0**	**0**	**0**	504	**0**	**0**
Total correct	**22212**	20160	13643	18053	20087	21603
Contribution score	**91.06**	3.51	–	–	–	14.54
Time (s)	14610.224	47293.484	71666.750	32220.939	35053.106	**13202.451**
Time w/o timeouts (s)	10350.224	**1993.484**	69526.750	11220.939	3713.106	2462.451

5.5 Detailed Experimental Results

Figure 3 and Table 2 show the detailed results for the **AutomatArk** benchmark. In this benchmark, Z3str3RE solves more instances than all other solvers, has the fewest timeouts/unknowns, and has the fastest overall running time. Including timeouts, Z3str3RE is 2.2× faster than CVC4, 4.7× faster than Z3seq, 40.4× faster than OSTRICH, 20.4× faster than Z3-Trau, and 32.3× faster than Z3str3.

Figure 4 and Table 3 show the detailed results for the **StringFuzz-regex-generated** benchmark. Z3str3RE solves more instances than all other solvers, has over 90% fewer timeouts than other solvers, no unknowns, and has the fastest overall running time. Including timeouts, Z3str3RE is 6.1× faster than CVC4, 6.9× faster than Z3seq, 10× faster than OSTRICH, 7.3× faster than Z3-Trau, and 4.3× faster than Z3str3.

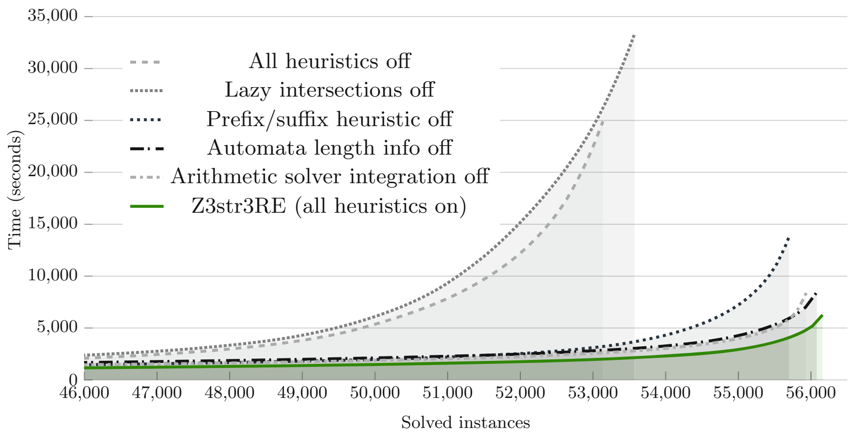


Fig. 7. Cactus plot comparing performance by disabling individual heuristics on all benchmarks.

Figure 5 and Table 4 show the detailed results for the **StringFuzz-regex-transformed** benchmark. Z3str3RE solves more instances in total than all other solvers and has the lowest total running time without timeouts. Including timeouts, Z3str3RE is 2.7× faster than CVC4, 1.9× faster than Z3seq, 21× faster than OSTRICH, and 27× faster than Z3str3. Although Z3-Trau is 1.5× faster than Z3str3RE on this benchmark, including timeouts, Z3-Trau also produces 1241 answers with soundness errors and crashes on 718 other cases. Z3str3RE produces no wrong answers or soundness errors on the benchmark. Z3-Trau also solves 1923 fewer cases correctly in total than Z3str3RE.

Figure 6 and Table 5 show the detailed results for the **RegEx-Collected** benchmark. Z3str3RE outperforms Z3seq, Z3str3, OSTRICH, and Z3-Trau on this benchmark and is competitive with CVC4 both in terms of total number of instances correctly solved and total running time. CVC4 solves 609 more instances than Z3str3RE on this benchmark, but Z3str3RE is 1.1× faster overall (including timeouts). Z3str3RE is 3.6× faster than Z3seq, 5.4× faster than OSTRICH, 2.4× faster than Z3-Trau, and 2.6× faster than Z3str3.

5.6 Analysis of Individual Heuristics and Results

To demonstrate the effectiveness of individual heuristics described in Sect. 4 and implemented in Z3str3RE, we evaluated different configurations of the tool in which one or more heuristics were disabled. Figure 7 and Table 6 show the results. The plot line "Z3str3RE" shows the performance of the tool with all heuristics enabled. The plot line "All heuristics off" shows the performance with all heuristics disabled. Each of the other plot lines shows the performance with the named heuristic disabled and all others kept enabled. From the plots and table, it is clear that Z3str3RE performs best with all heuristics enabled. Z3str3RE is 4.4× faster using all our heuristics than using none. Every other configuration of the

Table 6. Comparison of different heuristics in Z3str3RE on all benchmarks.

	All off	Lazy intersection off	Prefix/suffix off	Automata length info off	Arith. solver integ. off	Z3str3RE
Sat	31046	31486	33817	33816	33804	**33820**
Unsat	22090	22085	21880	22264	22131	**22339**
Unknown	313	323	287	285	**283**	291
Timeout	3807	3362	1272	891	1038	**806**
Soundness error	**0**	**0**	**0**	**0**	**0**	**0**
Program crashes	42	39	**0**	1	**0**	**0**
Total correct	53136	53571	55697	56080	55935	**56159**
Time (s)	102102.388	101799.263	40068.501	27178.746	30006.857	**23339.266**
Time w/o timeouts (s)	25962.388	34559.263	1462.8501	9358.746	9246.857	**7219.266**

tool performs significantly worse relative to the one with all heuristics enabled. Also, the length-aware and prefix/suffix heuristics provide significant boost over lazy intersections and the baseline. These results demonstrate empirically that each heuristic we introduce provides significant benefit in both total number of solved instances and total solver runtime, and that all of the heuristics can be used simultaneously for maximum efficacy.

6 Related Work

Comparison with Z3str3: Z3str3 [8] supports regex constraints via (incomplete) reduction to word equations. We have replaced this word-based technique with our automata-based approach introduced in this paper. As demonstrated by our evaluation, the length-aware automata-based approach used in Z3str3RE is more efficient at solving these constraints, and is sound and complete for the QF theory T_{LRE}.

Comparison with Z3's Sequence Solver: Z3's sequence solver [18] supports a more general theory of "sequences" over arbitrary datatypes, which allows it to be used as a string solver. Z3seq uses regular expression derivatives to reduce regex constraints without constructing automata. The experiments show Z3str3RE performs better than Z3seq overall.

Comparison with CVC4: The CVC4 solver [24] uses an algebraic approach to solving regex constraints. As shown in the experiments, Z3str3RE performs better than CVC4, widely considered as one of the best SMT solvers for strings as well as many other theories.

Comparison with Z3-Trau: The Z3-Trau [1] solver builds on Trau [2], re-implemented in Z3, and enriched with new ideas e.g. a more efficient handling of string-number conversion. The evaluation of Z3-Trau exposed 5325 soundness errors and 2477 crashes on our benchmarks.

Comparison with OSTRICH: The OSTRICH solver [15] implements a reduction from straight-line and acyclic fragments of an input formula to the emptiness problem of alternating finite automata. OSTRICH produced 10901 "unknown" responses and 4575 timeouts on our benchmarks, as well as 28 soundness errors.

Related Algorithms and Theoretical Results: The theory of word equations and various extensions have been studied extensively for many decades. In 1977, Makanin proved that satisfiability for the QF theory of word equations is decidable [28]; in 1999, Plandowski showed that this is in PSPACE [30,31]. Schulz [34] extended Makanin's algorithm to word equations with regex constraints. The satisfiability problem for the theory of word equations with length constraints still remains open [20,28,29,31], although the status of many other extensions of this theory was clarified [17]. Automata-based approaches were used to reason about string constraints enhanced with a ReplaceAll function [14] or transducers [21].

Liang et al. [25] present a formal calculus for a theory that extends T_{LRE} with string concatenation (but not word equations). However, in that paper the authors do not present experimental results regarding implementation of the string calculus proposed. We have implemented an algorithm based on fundamentals of the theory and standard automata-based constructions, and presented a thorough experimental evaluation of our implementation.

Abdulla et al. [3] present an automata-based solver called Norn built upon results involving construction of length constraints from regex constraints. This approach differs significantly from our method. In particular, Norn only uses automata in inferring length constraints implied by regular expressions, then uses an algebraic approach to solve the remainder of the formula. By contrast, our tool uses a hybrid approach that includes both algebraic solving and automata-based reasoning in a symbiotic loop. In addition, we present several novel heuristics using length information to guide the search and, in some cases, avoid constructing automata or computing intersections.

The prefix/suffix over-approximation heuristic is inspired partly by the work of Brzozowski on regex derivatives [13]. The heuristic we introduce is conceptually different as we examine possible prefixes (and suffixes) of strings that could be accepted by a regex in order to demonstrate unsatisfiability, rather than examining the set of all possible suffixes given a fixed prefix in order to demonstrate satisfiability. Our heuristic computes suffixes as well, whereas Brzozowski derivatives are traditionally computed with respect to prefixes of a string. Newer versions of Z3seq, including the one we evaluated, use a regex algorithm based on symbolic derivatives [36].

7 Conclusions and Future Work

In this paper, we empirically showcase the power of length-aware and prefix/suffix reasoning for regex constraints with our algorithm and its implementation in Z3str3RE via an extensive empirical comparison against five other state-of-the-art solvers (namely, CVC4, Z3seq, Z3str3, Z3-Trau, and OSTRICH) over a large and diverse benchmark of 57256 instances. Over this entire benchmark suite, we show that Z3str3RE has a speedup of 2.4× over CVC4, 4.4× over Z3seq, 6.4× over Z3-Trau, 9.1× over Z3str3, and 13× over OSTRICH. Our length-aware method is very general and has wide applicability in the broad context of string solving. In the future, we plan to explore further length-aware heuristics which include more expressive functions and predicates, including `indexof`, `substr`, and string-number conversion.

Acknowledgments. The work of Federico Mora is supported by NSF grants CNS-1739816 and CCF-1837132, by the DARPA LOGiCS project under contract FA8750-20-C-0156, by the iCyPhycenter, and by gifts from Intel, Amazon, and Microsoft. The work of Florin Manea is supported by the DFG grant 389613931.

References

1. Abdulla, P.A., et al.: Efficient handling of string-number conversion. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 943–957 (2020)
2. Abdulla, P.A., et al.: Flatten and conquer: a framework for efficient analysis of string constraints. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, pp. 602–617 (2017)
3. Abdulla, P.A., et al.: String constraints for verification. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 150–166. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_10
4. Aydin, A., Bang, L., Bultan, T.: Automata-based model counting for string constraints. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 255–272. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_15
5. Backes, J., et al.: One-click formal methods. IEEE Softw. **36**(6), 61–65 (2019)
6. Barbosa, H., Hoenicke, J., Hyvarinen, A.: 15th international satisfiability modulo theories competition (SMT-COMP 2020): rules and procedures (2020). https://smt-comp.github.io/2020/rules20.pdf
7. Berzish, M., et al.: String theories involving regular membership predicates: from practice to theory and back (2021)
8. Berzish, M., Ganesh, V., Zheng, Y.: Z3str3: a string solver with theory-aware heuristics. In: 2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, 2–6 October 2017, pp. 55–59 (2017)
9. Berzish, M., et al.: A length-aware regular expression SMT solver (2020). https://arxiv.org/abs/2010.07253
10. Bjørner, N., Ganesh, V., Michel, R., Veanes, M.: An SMT-LIB format for sequences and regular expressions. In: SMT workshop 2012 (2012)
11. Bjørner, N., Tillmann, N., Voronkov, A.: Path feasibility analysis for string-manipulating programs. In: Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2009, pp. 307–321 (2009). https://doi.org/10.1007/978-3-642-00768-2_27
12. Blotsky, D., Mora, F., Berzish, M., Zheng, Y., Kabir, I., Ganesh, V.: StringFuzz: a fuzzer for string solvers. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10982, pp. 45–51. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96142-2_6

13. Brzozowski, J.A.: Derivatives of regular expressions. J. ACM **11**(4), 481–494 (1964)
14. Chen, T., Chen, Y., Hague, M., Lin, A.W., Wu, Z.: What is decidable about string constraints with the replace all function. Proc. ACM Program. Lang. **2**(POPL), 3:1–3:29 (2018)
15. Chen, T., Hague, M., Lin, A.W., Rümmer, P., Wu, Z.: Decision procedures for path feasibility of string-manipulating programs with complex operations. In: Proceedings of the ACM on Programming Languages **3**(POPL), 1–30 (2019)
16. D'Antoni, L.: Automatark automata benchmark (2018). https://github.com/lorisdanto/automatark
17. Day, J.D., Ganesh, V., He, P., Manea, F., Nowotka, D.: The satisfiability of word equations: decidable and undecidable theories. In: Potapov, I., Reynier, P.-A. (eds.) RP 2018. LNCS, vol. 11123, pp. 15–29. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-00250-3_2
18. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
19. Ganesh, V., Berzish, M.: Undecidability of a theory of strings, linear arithmetic over length, and string-number conversion. CoRR abs/1605.09442 (2016). http://arxiv.org/abs/1605.09442
20. Ganesh, V., Minnes, M., Solar-Lezama, A., Rinard, M.: Word equations with length constraints: what's decidable? In: Biere, A., Nahir, A., Vos, T. (eds.) HVC 2012. LNCS, vol. 7857, pp. 209–226. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39611-3_21
21. Holík, L., Janku, P., Lin, A.W., Rümmer, P., Vojnar, T.: String constraints with concatenation and transducers solved efficiently. PACMPL **2**(POPL), 4:1–4:32 (2018)
22. Kiezun, A., Ganesh, V., Guo, P.J., Hooimeijer, P., Ernst, M.D.: HAMPI: a solver for string constraints. In: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, pp. 105–116 (2009)
23. Kulczynski, M., Manea, F., Nowotka, D., Poulsen, D.B.: The power of string solving: simplicity of comparison. In: 2020 IEEE/ACM 1st International Conference on Automation of Software Test (AST), pp. 85–88. IEEE/ACM (2020)
24. Liang, T., Reynolds, A., Tinelli, C., Barrett, C., Deters, M.: A DPLL(T) theory solver for a theory of strings and regular expressions. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 646–662. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_43
25. Liang, T., Tsiskaridze, N., Reynolds, A., Tinelli, C., Barrett, C.: A decision procedure for regular membership and length constraints over unbounded strings. In: Lutz, C., Ranise, S. (eds.) FroCoS 2015. LNCS (LNAI), vol. 9322, pp. 135–150. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24246-0_9
26. Lin, A.W., Majumdar, R.: Quadratic word equations with length constraints, counter systems, and Presburger arithmetic with divisibility. In: Lahiri, S.K., Wang, C. (eds.) ATVA 2018. LNCS, vol. 11138, pp. 352–369. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01090-4_21
27. Lin, A.W., Barceló, P.: String solving with word equations and transducers: towards a logic for analysing mutation XSS. In: Bodík, R., Majumdar, R. (eds.) Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, 20–22 January 2016, pp. 123–136. ACM (2016)
28. Makanin, G.: The problem of solvability of equations in a free semigroup. Math. Sbornik **103**, 147–236 (1977). English transl. in Math USSR Sbornik 32 (1977)

29. Matiyasevich, Y.: Word equations, fibonacci numbers, and Hilbert's tenth problem. In: Workshop on Fibonacci Words (2007)
30. Plandowski, W.: Satisfiability of word equations with constants is in PSPACE. J. ACM **51**(3), 483–496 (2004)
31. Plandowski, W.: An efficient algorithm for solving word equations. In: Proceedings of the 38th Annual ACM Symposium on Theory of Computing, STOC 2006, pp. 467–476 (2006)
32. Redelinghuys, G., Visser, W., Geldenhuys, J.: Symbolic execution of programs with strings. In: Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference, SAICSIT 2012, pp. 139–148 (2012)
33. Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.: A symbolic execution framework for JavaScript. In: Proceedings of the 2010 IEEE Symposium on Security and Privacy, SP 2010, pp. 513–528 (2010)
34. Schulz, K.U.: Makanin's algorithm for word equations-two improvements and a generalization. In: Schulz, K.U. (ed.) IWWERT 1990. LNCS, vol. 572, pp. 85–150. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-55124-7_4
35. Sen, K., Kalasapur, S., Brutch, T., Gibbs, S.: Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, pp. 488–498. ACM, New York (2013)
36. Stanford, C., Veanes, M., Bjørner, N.: Symbolic Boolean derivatives for efficiently solving extended regular expression constraints. Technical report. MSR-TR-2020-25, Microsoft, August 2020. https://www.microsoft.com/en-us/research/publication/symbolic-boolean-derivatives-for-efficiently-solving-extended-regular-expression-constraints/
37. Stockmeyer, L.J.: The Complexity of Decision Problems in Automata Theory and Logic. Ph.D. thesis, MIT (1974)
38. Thomé, J., Shar, L.K., Bianculli, D., Briand, L.: An integrated approach for effective injection vulnerability analysis of web applications through security slicing and hybrid constraint solving. IEEE TSE (2018)
39. Trinh, M.-T., Chu, D.-H., Jaffar, J.: Progressive reasoning over recursively-defined strings. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 218–240. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_12
40. Yu, F., Alkhalaf, M., Bultan, T.: Stranger: an automata-based string analysis tool for PHP. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 154–157. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12002-2_13
41. Zheng, Y., et al.: Z3str2: an efficient solver for strings, regular expressions, and length constraints. Formal Methods Syst. Des., 1–40 (2016)
42. Zheng, Y., Ganesh, V., Subramanian, S., Tripp, O., Dolby, J., Zhang, X.: Effective search-space pruning for solvers of string equations, regular expressions and length constraints. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 235–254. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_14

Counting Minimal Unsatisfiable Subsets

Jaroslav Bendík[1,2](✉) and Kuldeep S. Meel[2]

[1] Faculty of Informatics, Masaryk University, Brno, Czech Republic
xbendik@fi.muni.cz
[2] National University of Singapore, Singapore, Singapore

Abstract. Given an unsatisfiable Boolean formula F in CNF, an unsatisfiable subset of clauses U of F is called Minimal Unsatisfiable Subset (MUS) if every proper subset of U is satisfiable. Since MUSes serve as explanations for the unsatisfiability of F, MUSes find applications in a wide variety of domains. The availability of efficient SAT solvers has aided the development of scalable techniques for finding and enumerating MUSes in the past two decades. Building on the recent developments in the design of scalable model counting techniques for SAT, Bendík and Meel initiated the study of MUS counting techniques. They succeeded in designing the first approximate MUS counter, AMUSIC, that does not rely on exhaustive MUS enumeration. AMUSIC, however, suffers from two shortcomings: the lack of exact estimates and limited scalability due to its reliance on 3-QBF solvers.

In this work, we address the two shortcomings of AMUSIC by designing the first exact MUS counter, CountMUST, that does not rely on exhaustive enumeration. CountMUST circumvents the need for 3-QBF solvers by reducing the problem of MUS counting to projected model counting. While projected model counting is #NP-hard, the past few years have witnessed the development of scalable projected model counters. An extensive empirical evaluation demonstrates that CountMUST successfully returns MUS count for 1500 instances while AMUSIC and enumeration-based techniques could only handle up to 833 instances.

1 Introduction

Boolean formulas serve as a primary representation language to model the behaviour of systems and properties. Given an unsatisfiable Boolean formula F in Conjunctive Normal Form (CNF), i.e. a set of clauses $F = \{f_1, f_2, \ldots, f_n\}$, a subset $U \subseteq F$ is called Minimal Unsatisfiable Subset (MUS) of F iff U is unsatisfiable and for every $f \in U$, $U \setminus \{f\}$ is satisfiable.

MUSes serve as *explanations* or *reasons* for unsatisfiability of F, and have, consequently, found applications in a wide variety of domains such as diagnosis [24,56], constrained sampling and counting [28], equivalence checking [20], and the like [1,2,25,30,47,64]. While the early applications relied on identifying a single [3,6,7,51,53] or enumerating multiple [4,10,12,39,41,52] MUSes, the rapid adoption of MUSes lead researchers to investigate problem formulations and their corresponding applications that do not rely on explicit MUS

A. Silva and K. R. M. Leino (Eds.): CAV 2021, LNCS 12760, pp. 313–336, 2021.
https://doi.org/10.1007/978-3-030-81688-9_15

identification. These include, e.g., computing the union of all MUSes [45], deciding whether a given clause belongs to an MUS [31], or counting the number of MUSes. Especially, the counting of MUSes found many applications in the domain of diagnosis where the MUS count can be used to compute various inconsistency metrics [25,29,48–50,65] for general propositional knowledge bases.

A straightforward, and for many years the only available, approach for counting MUSes is to simply enumerate them. However, there can be up to exponentially many MUSes w.r.t. $|F|$ and hence the complete enumeration is often practically intractable [9,10,39,69]. Inspired by the development of model counting techniques in the context of SAT, which in its nascent stages also depended on complete model enumeration while contemporary techniques often need to explicitly identify just a fraction of models, Bendík and Meel [13] recently initiated an investigation of counting MUSes without their explicit enumeration. In this context, they succeeded by developing a hashing-based approximate counter, AMUSIC [13], that provides the so-called PAC guarantees, also known as (ε, δ)-guarantees, wherein the computed answer is within the $(1+\varepsilon)$-factor of the exact count with confidence at least $1 - \delta$. AMUSIC reduces the problem of MUS counting to logarithmically many calls to a Σ_3^P oracle (3-QBF solver, in practice) wherein every Σ_3^P query is constructed over a CNF formula conjuncted with XORs.

While AMUSIC achieved its stated goal of avoiding explicit enumeration, its scalability is significantly hampered by its reliance on a 3-QBF solver that can efficiently handle formulas conjuncted with XOR constraints. It is worth highlighting that the scalability of model counting techniques [17,60] in the context of SAT crucially relies on the availability of CryptoMiniSAT [61], a SAT solver with native support for CNF-XOR constraints. Despite significant advances in QBF solving over the years, the scalability remains a formidable challenge for 3-QBF solvers, and even more when XOR constraints are involved. As such, AMUSIC could scale to formulas involving few hundreds of variables and clauses.

In this work, we focus on addressing the scalability of MUS counting techniques. We begin our investigation by focusing on the observation of Bendík and Meel that their technique relied on a Σ_3^P oracle even though the problem of finding an MUS is in FP^{NP} [19,44]. Therefore, a natural direction is to investigate the design of an algorithmic framework that can circumvent reliance on oracles with high complexity. In this context, we rely on the observation of Durand, Hermann, and Koliatis [21] that the complexity of counting problems whose search problems have FP^{NP} complexity tend to be #NP (which contains #P class). Such an observation is timely given the recent surge of interest in designing efficient techniques for projected model counting, which is #NP-hard. Therefore, one wonders: *whether it is possible to design a MUS counting technique that can take advantage of projected model counters?*

The primary contribution of this paper is an affirmative answer to the above question. We design a new algorithmic framework, CountMUST, that reduces the problem of MUS counting to two projected model counting queries. In particular, CountMUST constructs a wrapper $\mathcal{W}$ and its remainder $\mathcal{R}$ such that the number of MUSes of F is $|\mathcal{W}| - |\mathcal{R}|$, i.e., the wrapper $\mathcal{W}$ over-approximates the set of MUSes while the remainder contains the spurious, non-MUS, subsets of F that emerge due to the over-approximation. We encode the wrapper $\mathcal{W}$ and the remainder $\mathcal{R}$ with Boolean formulas $\mathbb{W}$ and $\mathbb{R}$ such that the projected model counts for $\mathbb{W}$ and $\mathbb{R}$ (for a suitable projection set) equal to $|\mathcal{W}|$ and $|\mathcal{R}|$, respectively. An interesting (and perhaps surprising) aspect of our CountMUST is that we do not enumerate a single MUS in our process, which is in stark contrast to the design of AMUSIC that relies on the enumeration of a *small* number of MUSes.

We discuss several strategies to construct wrappers (and their corresponding remainders) that are efficient to compute and are tight over-approximations of the set of MUSes. We conduct a detailed empirical analysis over 2553 instances and observe that CountMUST successfully returns MUS count for 1500 instances while AMUSIC and enumeration-based techniques could only handle up to 833 instances. We observe interesting complementary nature of the exact and approximate MUS counting approaches: the scalability of AMUSIC is often impacted by the number of clauses and appears to be less impacted by the number of MUSes while, on the other hand, the scalability of CountMUST is less impacted by the number of clauses and appears to depend on the number of MUSes.

Finally, our empirical analysis showcases that our wrappers $\mathcal{W}$ approximate the set of MUSes very tightly. Motivated by the tightness of our wrappers, we discuss several interesting applications of our framework: approximate MUS counting [13], MUS enumeration [5,40], MUS Sampling, estimation of minimum and maximum MUS cardinality [27,38], and MUS membership testing [31].

The rest of the paper is organized as follows. We introduce preliminaries in Sect. 2 and discuss related work in Sect. 3. We then present the primary technical contribution of our work in Sect. 4. We present the empirical evaluation in Sect. 5 and then discuss the implications of the tightness of our wrappers in Sect. 6. We finally conclude in Sect. 7.

2 Preliminaries and Problem Definition

A Boolean formula F is built over Boolean values $\{1, 0\}$ and over a set $\mathit{Vars}(F)$ of Boolean variables connected via standard logical operators: $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$, $\neg$. A literal is either a variable $x \in \mathit{Vars}(F)$ or its negation $\neg x$; $\mathit{Lits}(F)$ denotes the set of all literals used in F. Given a set A of variables, a valuation $\pi : A \rightarrow \{1, 0\}$ assigns to each variable its Boolean value. $F[\pi]$ denotes the formula that emerges from F by substituting every variable x of F that is in the domain of π by $\pi(x)$; furthermore, trivial simplifications, e.g., $G \vee 0 = G$, $G \wedge 0 = 0$, $\neg 1 = 0$, $\neg 0 = 1$, are applied. Note that if $A \supseteq \mathit{Vars}(F)$, then $F[\pi]$ is simplified either to 1 or to 0. In the case when $A \supseteq \mathit{Vars}(F)$ and $F[\pi] = 1$, we call π a *model* of F and write $\pi \models F$; otherwise, when $F[\pi] = 0$, we write $\pi \not\models F$. A formula F is *satisfiable* if it has a model; otherwise, F is *unsatisfiable*. We write M_F to denote the set of all models of F. Moreover, given a set $A \subseteq \mathit{Vars}(F)$ of variables, we write $M_{F\downarrow A}$ to denote the projection of M_F on A, and for every $\pi \in M_F$, we write $\pi_{\downarrow A}$ to denote the projection of π on A. Finally, given two variable sets, $A = \{a_1, \ldots, a_k\}$ and $B = \{b_1, \ldots, b_k\}$, such that $A \subseteq \mathit{Vars}(F)$, we write $F_{[A/B]}$ to denote the formula that originates from F by substituting each variable $a_i \in A$ by $b_i \in B$.

A formula in conjunctive normal form, shortly a *CNF formula*, is a conjunction of *clauses* where a clause is a disjunction of literals. When suitable, a CNF formula can also be viewed as a multiset of clauses where a clause is a set of literals; we use the two representations interchangeably based on the context. Throughout the whole text, let us by $F = \{f_1, \ldots, f_n\}$ denote the input CNF formula of interest. Furthermore, capital letters, e.g., S, K, N, or blackboard bold letters, e.g., $\mathbb{W}$, $\mathbb{R}$, are used to denote other formulas, small letters, e.g., f, f_1, f_i, are used to denote clauses, and small letters, e.g., x, x', y, are used to denote variables. Finally, given a set X, $\mathcal{P}(X)$ denotes the power-set of X, and $|X|$ denotes the cardinality of X.

Definition 1 (MUS). *A subset N of F is a* minimal unsatisfiable subset *(MUS) of F iff N is unsatisfiable and for every $f \in N$ it holds that $N \setminus \{f\}$ is satisfiable.*

Definition 2 (MSS). *A subset N of F is a* maximal satisfiable subset *(MSS) of F iff N is satisfiable and for every $f \in F \setminus N$ it holds that $N \cup \{f\}$ is unsatisfiable.*

Definition 3 (MCS). *A subset N of F is a* minimal correction subset *(MCS) of F iff $F \setminus N$ is satisfiable and for every $f \in N$ it holds that $F \setminus (N \setminus \{f\})$ is unsatisfiable. Equivalently, N is an MCS iff $F \setminus N$ is an MSS.*

Note that the Boolean satisfiability is monotone w.r.t. the (clause) subset inclusion, i.e., all subsets of a satisfiable set of clauses are satisfiable. Consequently, all proper subsets of an MUS are in fact satisfiable, and, dually, all proper supersets of an MSS are unsatisfiable. Also, note that the minimality/maximality concept used here is a *set minimality/maximality* and not a *minimum/maximum cardinality.* Consequently, there can be up to $\binom{|F|}{|F|/2}$ MUSes/MCSes/MSSes of F (intuitively, this is the number of pair-wise incomparable subsets of F; see the Sperner's theorem [62]). We write *maximum* and *minimum* MUS to denote an MUS with the maximum and the minimum cardinality, respectively. Note that there can also be exponentially many maximum and minimum MUSes. We write $\texttt{MUS}_F$ to denote the set of all MUSes of F, and $\texttt{SS}_F$ to denote the set of all satisfiable subsets of F.

Example 1. Let us demonstrate the concepts of MUSes, MSSes and MCses on an example. Assume that $F = \{f_1 = \{x_1\}, f_2 = \{\neg x_1\}, f_3 = \{x_2\}, f_4 = \{\neg x_1, \neg x_2\}\}$. There are 2 MUSes: $\texttt{MUS}_F = \{\{f_1, f_3, f_4\}, \{f_1, f_2\}\}$, 3 MSSes: $\{\{f_2, f_3, f_4\}, \{f_1, f_4\}, \{f_1, f_3\}\}$, and thus also 3 MCSes: $\{\{f_1\}, \{f_2, f_3\}, \{f_2, f_4\}\}$. For illustration, see Fig. 1.

In this paper, we are concerned with the following two problems.

Name: `#MUS`
Input: A CNF formula F.
Output: The number $|\texttt{MUS}_F|$ of MUSes of F.

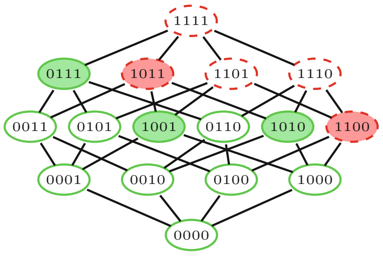


Fig. 1. Illustration of $\mathcal{P}(F)$ from the Example 1. Individual subsets are represented as bit-vectors, e.g., $\{f_1, f_2\}$ is written as 1100. The subsets with a dashed border are the unsatisfiable subsets, and the others are satisfiable subsets. MUSes and MSSes are filled with a background colour.

Name: `proj-#SAT`
Input: A formula G and a set of variables $S \subseteq \mathit{Vars}(G)$.
Output: The number $|M_{G \downarrow S}|$ of models of G projected on S.

Our goal is to solve the `#MUS` problem, and to do that, we propose a *strong subtractive reduction* to the `proj-#SAT` problem.

Definition 4 (Strong Subtractive Reductions). *[21] Let Σ be an alphabet and let Q_1 and Q_2 be two binary relations over Σ. Let $\# \cdot Q_1$ and $\# \cdot Q_2$ represent the corresponding counting problems. Then, $\# \cdot Q_1$ reduces to $\# \cdot Q_2$ via a strong subtractive reduction, if there exist polynomial-time computable functions f and g such that for every string $z \in \Sigma^*$:*

1. $Q_2(f(z)) \subseteq Q_2(g(z))$
2. $|Q_1(z)| = |Q_2(g(z))| - |Q_2(f(z))|$.

3 Related Work

MUS Counting. A straight-forward approach to count the MUSes is to simply enumerate them via an MUS enumeration algorithm, e.g. [4,5,8,10,12,39,41,52]. However, since there can be up to exponentially many MUSes w.r.t. $|F|$, the complete enumeration is often practically intractable. An alternative approach to identify the MUS count is based on a so-called *minimal hitting set duality* between MUSes and MCSes that states that every MUS is a *minimal hitting set* of the set of all MCSes [32,56]. Consequently, one can determine the MUS count by first identifying all MCSes and then counting their minimal hitting sets [40]. However, there can be in general up to exponentially many MCSes, which makes this approach also often practically intractable [11,52].

The study of MUS counting without relying on exhaustive enumeration was initiated just recently by Bendík and Meel [13], who proposed an (ε, δ)-approximation scheme called AMUSIC. AMUSIC extends a prior hashing-based model counting framework [15,18,63] to MUS counting. Briefly, AMUSIC divides the power-set $\mathcal{P}(F)$ into *nCells* small *cells*, then pick one of the cells and count the number *inCell* of MUSes in the cell, and estimate the overall MUS count as $\mathit{nCells} \times \mathit{inCell}$. The approach requires to perform logarithmically many calls to a Σ_3^P oracle (3-QBF solver) wherein each query consists of a CNF formula conjuncted with XOR constraints. The lack of solvers with native support for such constraints presents the major hindrance to the scalability of AMUSIC.

It is worth remarking on a recent work by Bendík and Meel [14] that focuses on exact counting of maximal satisfiable subsets (MSSes). While MUSes and MSSes are closely related concepts, to the best of our knowledge, there does not exist any efficient reduction from MUS counting to MSS counting, or vice versa. Note that the best known upper-bound on the problem of finding an MUS is $\mathrm{FP}^{\mathrm{NP}}$ [19], whereas for findind an MSS a tighter upper-bound $\mathrm{FP}^{\mathrm{NP}}[\mathit{wit}, \mathit{log}]$ is known [44], which suggests that counting MUSes is practically harder than counting MSSes. It would be an interesting question for future work if the counter developed in this work can be employed to perform MSS counting.

Model Counting. The complexity-theoretic study of model counting was initiated by Valiant [67] who showed that `proj-#SAT` is #P-complete when $S = \mathit{Vars}(G)$. Subsequently, Durand, Hermann, and Koliatis [21] showed that the general problem of `proj-#SAT` is #NP-hard. A significant conceptual contribution of Durand et al. was to show the importance of subtractive reductions for problems in #NP; this idea has been applied for reductions to projecting counting [14].

Our work relies on the recent progress in the development of efficient projected model counters; in particular, we employ GANAK [59], a state-of-the-art *search-based* exact model counter; the entry based on GANAK won the projected model counting track in 2020 Model Counting Competition [23]. Search-based model counters build on three core ideas: (1) for a formula G and $x \in S$, we have $|M_{G\downarrow S}| = |M_{G(x\mapsto 0)\downarrow S}| + |M_{G(x\mapsto 1)\downarrow S}|$, (2) if G can be partitioned into subset of clauses $\{C_1, C_2, \dots C_k\}$ such that $\forall i, j.\ \mathit{Vars}(C_i) \cap \mathit{Vars}(C_j) = \emptyset$, then we have $|M_{G\downarrow S}| = \prod_{i=1}^{k} |M_{C_i\downarrow S}|$, and (3) finally, component caching is employed to cache the components. Consequently, the model count can be often determined by explicitly identifying just a fraction of all models. GANAK is built on top of earlier search-based model counters, sharpSAT [66] and Cachet [57,58].

4 MUS Counting via a Projected Model Counter

We now gradually introduce several subtractive reductions of the MUS counting problem to the projected model counting, starting with the base idea in Sect. 4.1, and following with the particular reductions in Sects. 4.2–4.11.

4.1 Basic MUS Counting Idea

Definition 5 (wrapper and remainder). *A set $\mathcal{W}$ of subsets of F is a* wrapper *iff* $\mathtt{MUS}_F \subseteq \mathcal{W} \subseteq \mathtt{MUS}_F \cup \mathtt{SS}_F$*. Furthermore, the* remainder *of $\mathcal{W}$ is the set $\mathcal{R} = \mathcal{W} \cap \mathtt{SS}_F$.*

Proposition 1. *Let $\mathcal{W}$ be a wrapper and $\mathcal{R}$ its corresponding remainder. Then* $|\mathtt{MUS}_F| = |\mathcal{W}| - |\mathcal{R}|$.

Proof. Since $\mathcal{R} = \mathcal{W} \cap \mathtt{SS}_F$, then $\mathtt{MUS}_F \cap \mathcal{R} = \emptyset$, and hence $|\mathcal{W}| = |\mathtt{MUS}_F| + |\mathcal{R}|$.

Our approach to determine the MUS count $|\mathtt{MUS}_F|$ consists of the following steps. First, we define a wrapper $\mathcal{W}$ and its corresponding remainder $\mathcal{R}$. Subsequently, we encode the wrapper $\mathcal{W}$ with a Boolean formula $\mathbb{W}$ such that each projected model of $\mathbb{W}$ (for a suitable projection set) corresponds to an element of $\mathcal{W}$. Similarly, we construct a Boolean formula $\mathbb{R}$ such that each projected model of $\mathbb{R}$ corresponds to an element of the remainder $\mathcal{R}$. Finally, we employ a projected model counter to determine the projected model counts of $\mathbb{W}$ and $\mathbb{R}$, i.e., $|\mathcal{W}|$ and $|\mathcal{R}|$, and hence we obtain the MUS count $|\mathtt{MUS}_F| = |\mathcal{W}| - |\mathcal{R}|$.

In the following, we first describe in Sect. 4.2 how to build a simple wrapper $\mathcal{W}_1$ and its remainder $\mathcal{R}_1$ and how to encode them via Boolean formulas $\mathbb{W}_1$ and $\mathbb{R}_1$, respectively. Subsequently, in Sects. 4.3–4.11, we propose several additional wrappers (and their remainders) that improve upon the base wrapper $\mathcal{W}_1$ by exploiting various observations about MUSes. Finally, in Sect. 4.12, we show how to combine the individual wrappers.

4.2 $\mathcal{W}_1$ - the Base Wrapper and Its Reminder

Our base wrapper, $\mathcal{W}_1$, is simply the set of all satisfiable subsets and all MUSes of F, i.e., $\mathcal{W}_1 = \mathtt{SS}_F \cup \mathtt{MUS}_F$. The corresponding remainder $\mathcal{R}_1$ is thus the set $\mathtt{SS}_F$ of all satisfiable subsets of F. In the following, we describe how to encode the wrapper $\mathcal{W}_1$ and the remainder $\mathcal{R}_1$ via Boolean formulas $\mathbb{W}_1$ and $\mathbb{R}_1$ whose projected models correspond to elements of $\mathcal{W}_1$ and $\mathcal{R}_1$, respectively.

Let us start with encoding the remainder $\mathcal{R}_1 = \mathtt{SS}_F$. Given the unsatisfiable formula $F = \{f_1, \ldots, f_n\}$, we introduce a set $\mathcal{A} = \{a_1, \ldots, a_n\}$ of *activation variables*. Note that every valuation π of $\mathcal{A}$ one-to-one maps to an *activated* subset $\pi_{\mathcal{A},F}$ of F defined as $\pi_{\mathcal{A},F} = \{f_i \in F \,|\, \pi(a_i) = 1\}$. Using the activation variables, we build the formula $\mathbb{R}_1$ as follows:

$$\mathbb{R}_1 = \bigwedge_{f_i \in F} a_i \rightarrow f_i \tag{1}$$

Intuitively, if we set a_i to 0 then the formula $a_i \rightarrow f_i$ is trivially satisfied, and if we set a_i to 1 then f_i has to be satisfied to satisfy $a_i \rightarrow f_i$. Hence, the models of $\mathbb{R}_1$ projected on $\mathcal{A}$ map to satisfiable subsets of F; formally:

Proposition 2. *For every valuation π of $\mathcal{A}$, $\pi \in M_{\mathbb{R}_1 \downarrow \mathcal{A}}$ iff $\pi_{\mathcal{A},F} \in \mathcal{R}_1 = \mathtt{SS}_F$. Consequently, $|M_{\mathbb{R}_1 \downarrow \mathcal{A}}| = |\mathcal{R}_1|$.*

Let us note that the concept of activation variables (or alternatively *relaxation variables*) and the idea behind the formula $\mathbb{R}_1$ is not novel and it appeared also in several MUS/MSS/MCS related studies such as [14,31,42]. However, we are the first who apply it in the context of MUS counting.

To build a formula $\mathbb{W}_1$ that represents the wrapper $\mathcal{W}_1 = \texttt{SS}_F \cup \texttt{MUS}_F$, we will proceed similarly, i.e., we build $\mathbb{W}_1$ using the activation variables $\mathcal{A}$ in such a way that a valuation π of $\mathcal{A}$ is a projected model of $\mathbb{W}_1$ iff $\pi_{\mathcal{A},F} \in \mathcal{W}_1$. A straightforward approach to encode $\mathcal{W}_1$ is to directly express that we are interested either in satisfiable subsets or MUSes of F. Such an encoding might look as $\mathbb{R}_1(\mathcal{A}) \vee \mathsf{isMUS}(\mathcal{A})$ where $\mathbb{R}_1(\mathcal{A})$ is the formula from Eq. 1 encoding that $\pi_{\mathcal{A},F}$ is satisfiable and $\mathsf{isMUS}(\mathcal{A})$ is a formula encoding that $\pi_{\mathcal{A},F}$ is an MUS. However, encoding that a set S is an MUS is quite expensive since one has to express that all subsets of S are satisfiable and that S is unsatisfiable (Definition 1). Especially, encoding that a set S is unsatisfiable requires to assume all the exponentially many valuations of $\mathit{Vars}(S)$. Several MUS related studies used various QBF encodings for the property of being an MUS, e.g., [13,31]. In particular, to express that a set S is an MUS, one can use the following, intuitively described, $\forall\exists$-QBF encoding: "**for every** valuation τ of $\mathit{Vars}(S)$ the valuation τ models $\neg S$ (i.e., S is unsatisfiable) **and for every** subset S' of S there **exists** a valuation τ' of $\mathit{Vars}(S')$ that satisfies S'". One could convert the $\forall\exists$-QBF encoding into a plain Boolean formula by explicitly enumerating all the possible valuations of $\mathit{Vars}(S)$ and all the subsets of S, however, this yields an exponentially large, and thus intractable, formula. Hence, instead of directly expressing that every element of the wrapper $\mathcal{W}_1$ is either a satisfiable subset or an MUS of F, we propose another approach based on a novel concept of an *evidence*.

Definition 6 (evidence). *Let A be a subset of $F = \{f_1, \dots, f_n\}$. An* evidence *for A is a tuple $(\rho_1, \dots, \rho_n)$ such that for every $1 \leq i \leq n$ it holds that:*

1. $\rho_i : \mathit{Vars}(F) \rightarrow \{1, 0\}$ is truth assignment, and
2. $\rho_i \models A \setminus \{f_i\}$.

Crucially, we observe the following:

Proposition 3. *For every subset A of F it holds that $A \in \texttt{SS}_F \cup \texttt{MUS}_F = \mathcal{W}_1$ iff there exists an evidence for A.*

Our formula $\mathbb{W}_1$ (Eq. 2) that encodes the wrapper $\mathcal{W}_1$ captures every set $A \subseteq F$ for which there exists an evidence $(\rho_1, \dots, \rho_n)$. To represent the set A, we use the activation variables $\mathcal{A} = \{a_1, \dots, a_n\}$. To represent the truth assignments $\rho_1, \dots, \rho_n$, we introduce variable sets $\mathcal{I}_1, \dots, \mathcal{I}_n$ where $\mathcal{I}_i$ is a fresh copy of $\mathit{Vars}(F)$ for every $i \in \{1, \dots, n\}$.

$$\mathbb{W}_1 = \bigwedge_{a_i \in A} a_i \rightarrow \Big(\bigwedge_{j \in \{1,\dots,n\} \setminus \{i\}} (a_j \rightarrow f_{j[\mathit{Vars}(F)/\mathcal{I}_i]}) \Big) \tag{2}$$

Intuitively, let π' be a valuation of $\mathit{Vars}(\mathbb{W}_1)$ and $\pi'_{\mathcal{A},F} = \{f_i \in F \mid \pi'(a_i) = 1\}$ the subset of F activated by $\mathcal{A}$. For every activated clause $f_i \in \pi'_{\mathcal{A},F}$, the formula expresses that $\pi'_{\downarrow \mathcal{I}_i}$ is a model of $\pi'_{\mathcal{A},F} \setminus \{f_i\}$ where the variable set $\mathit{Vars}(F)$ is substituted by $\mathcal{I}_i$.

Proposition 4. *For every valuation π of $\mathcal{A}$, $\pi \in M_{\mathbb{W}_1 \downarrow \mathcal{A}}$ iff $\pi_{\mathcal{A},F} \in \mathcal{W}_1 = \texttt{SS}_F \cup \texttt{MUS}_F$. Consequently, $|M_{\mathbb{W}_1 \downarrow \mathcal{A}}| = |\mathcal{W}_1|$.*

Based on Propositions 2 and 4, we can now employ a projected model counter to obtain the model counts $|M_{\mathbb{W}_1 \downarrow \mathcal{A}}|$ and $|M_{\mathbb{R}_1 \downarrow \mathcal{A}}|$, which yields $|\mathcal{W}_1|$ and $|\mathcal{R}_1|$, and hence also $|\texttt{MUS}_F|$ (Proposition 1). However, the concern here is the tractability of obtaining the model counts. There are mainly two criteria that affect the practical tractability of projected model counting. One criterion is the number of projected models, i.e. the cardinality of the wrapper (and the remainder), and the other criterion is the cardinality of the projection set, i.e., $|\mathcal{A}|$. The wrapper $\mathcal{W}_1$ is not very efficient w.r.t. these two criteria. Especially, $\mathcal{W}_1$ contains all satisfiable subsets of F, and there are often exponentially many satisfiable subsets of F w.r.t. $|F|$. Therefore, in the following, we will present nine additional wrappers, $\mathcal{W}_2, \dots, \mathcal{W}_{10}$, and their corresponding remainders. Each of the wrappers captures a property of MUSes that allows us to provide a better description of MUSes, and hence reduce the cardinality of the wrapper and/or the cardinality of the projection set. Similarly as in the case of $\mathcal{W}_1$, we will use the activation variables $\mathcal{A}$ to represent the elements of the wrappers/remainders. Moreover, every of the following wrappers $\mathcal{W}_i$ will be encoded by a Boolean formula $\mathbb{W}_i$ such that for every valuation π of $\mathcal{A}$, $\pi \in M_{\mathbb{W}_i \downarrow \mathcal{A}}$ iff $\pi_{\mathcal{A},F} \in \mathcal{W}_i$ (and similarly for the remainders).

4.3 $\mathcal{W}_2$ - the Intersection of MUSes

Our second wrapper $\mathcal{W}_2$ is based on a simple observation: every MUS of F has to contain the intersection $\texttt{IMUS}_F$ of all MUSes of F. Hence, we define the wrapper as $\mathcal{W}_2 = \{N \in \mathcal{W}_1 \mid N \supseteq \texttt{IMUS}_F\}$ and encode it via $\mathbb{W}_2$ as follows:

$$\mathbb{W}_2 = \mathbb{W}_1 \wedge \bigwedge_{f_i \in \texttt{IMUS}_F} a_i \tag{3}$$

Proposition 5. *For every valuation π of $\mathcal{A}$, $\pi \in M_{\mathbb{W}_2 \downarrow \mathcal{A}}$ iff $\pi_{\mathcal{A},F} \in \mathcal{W}_2$. Consequently, $|M_{\mathbb{W}_2 \downarrow \mathcal{A}}| = |\mathcal{W}_2|$.*

The remainder $\mathcal{R}_2$ of $\mathcal{W}_2$ is by Definition 5 the set $\mathcal{W}_2 \cap \texttt{SS}_F$. To build the formula $\mathbb{R}_2$ that encodes $\mathcal{R}_2$, observe that we already have an encoding for the set $\mathcal{W}_2$ (Eq. 3), and we also have an encoding for the set $\texttt{SS}_F$ since $\texttt{SS}_F = \mathcal{R}_1$. Hence, we can build $\mathbb{R}_2$ as a conjunction of the two encodings: $\mathbb{R}_2 = \mathbb{W}_2 \wedge \mathbb{R}_1$. Note that this construction of the remainder and the formula that encodes it is purely mechanical and does not involve any specific property of the particular wrapper. Therefore, for every wrapper $\mathcal{W}_i$ and its encoding $\mathbb{W}_i$ that are presented in the following sections, we define the reminder as $\mathcal{R}_i = \mathcal{W}_i \cap \mathcal{R}_1$ and encode it as $\mathbb{R}_i = \mathbb{W}_i \wedge \mathbb{R}_1$. Proposition 6 witnesses the soundness of this construction:

Proposition 6. *For every valuation π of $\mathcal{A}$, $\pi \in M_{\mathbb{R}_i \downarrow \mathcal{A}}$ iff $\pi_{\mathcal{A},F} \in \mathcal{R}_i$. Consequently, $|M_{\mathbb{R}_i \downarrow \mathcal{A}}| = |\mathcal{R}_i|$.*

This section's final question is how to compute the intersection $\texttt{IMUS}_F$. It is well-known that a clause $f \in F$ belongs to $\texttt{IMUS}_F$ iff $F \setminus \{f\}$ is satisfiable (see, e.g., [32,40,56]). Hence, a straightforward way would be to perform such satisfiability check for each $f \in F$, however, that might be very expensive. Fortunately, there has been recently proposed [13] a quite efficient algorithm to compute $\texttt{IMUS}_F$ which usually requires only few satisfiability checks, so we implemented that algorithm and use it while building the wrapper.

4.4 $\mathcal{W}_3$ - The Union of MUSes

Our next wrapper, $\mathcal{W}_3$, is very similar to the previous wrapper. Observe that every MUS of F is necessarily a subset of the union $\texttt{UMUS}_F$ of all MUSes of F. Consequently, also a weaker observation holds: every MUS of F is a subset of every over-approximation of $\texttt{UMUS}_F$. We define the wrapper as $\mathcal{W}_3 = \{N \in \mathcal{W}_1 \mid N \subseteq U\}$ where U is either the exact union $\texttt{UMUS}_F$ or its over-approximation ($U \supseteq \texttt{UMUS}_F$). Details on obtaining U are provided below. The encoding $\mathbb{W}_3$ of $\mathcal{W}_3$ is analogical to $\mathbb{W}_2$:

$$\mathbb{W}_3 = \mathbb{W}_1 \wedge \bigwedge_{f_i \notin U} \neg a_i \tag{4}$$

Proposition 7. *For every valuation π of $\mathcal{A}$, $\pi \in M_{\mathbb{W}_3 \downarrow \mathcal{A}}$ iff $\pi_{\mathcal{A},F} \in \mathcal{W}_3$. Consequently, $|M_{\mathbb{W}_3 \downarrow \mathcal{A}}| = |\mathcal{W}_3|$.*

The computation of the union $\texttt{UMUS}_F$ has been examined in two recent studies [13,45] that provided two different approaches for that task. Unfortunately, due to the problem's hardness, both the studies showed that the proposed approaches can usually handle only relatively small input formulas. Namely, the approach from [13] requires $\mathcal{O}(|F|)$ calls of a Σ_2^P oracle. Fortunately, it is often possible to cheaply compute a good over-approximation of $\texttt{UMUS}_F$ via the concepts of *autark variables* and a *lean kernel*. Briefly, a subset V of $\mathit{Vars}(F)$ is an *autark* [46] of F iff there exists a valuation χ of V such that for every clause $f \in F$ that contains a variable from V it holds that $\chi \models f$. Since a union of two autark sets is also an autark set, there exists a unique maximum autark set [33,34]. The *lean kernel* K of F is the set of clauses that do not use any variable from the maximum autark set. It has been shown (e.g. [33,34]), that the lean kernel is an over-approximation of $\texttt{UMUS}_F$. Hence, when building the wrapper $\mathcal{W}_3$, we use the lean kernel K as the over-approximation U of $\texttt{UMUS}_F$, i.e., $\mathcal{W}_3 = \{N \in \mathcal{W}_1 \mid N \subseteq K\}$. There have been proposed several algorithms to compute the lean kernel, e.g. [36,43]; we have implemented the algorithm by Marques-Silva et al. [43] using a MaxSAT solver UWrMaxSat [54] as a back-end.

Few words are in order to the effect of the two wrappers, $\mathcal{W}_2$ and $\mathcal{W}_3$, on the tractability of the projected model counting. Observe that in both cases ($\mathbb{W}_2$ and $\mathbb{W}_3$), we fix values of some variables from the projection set $\mathcal{A}$. Hence, before passing the formulas to the projected model counter, we first propagate the fixed values of $\mathcal{A}$ to simplify the formulas. By doing so, we effectively reduce the size of the projection set $\mathcal{A}$ by $|\texttt{IMUS}_F|$ and $|U| = |K|$, respectively.

Finally, let us note that the fact that an MUS has to be a subset of the union of all MUSes and a superset of the intersection of all MUSes is well-known and it has been already exploited in various ways in several MUS related studies (see, e.g., [10,11,45]). Especially, the approximate MUS counting algorithm AMUSIC [13] utilizes $\texttt{UMUS}_F$ in its preprocessing phase, and $\texttt{IMUS}_F$ to simplify 3-QBF queries while searching for MUSes.

4.5 $\mathcal{W}_4$ - Minimum MUS Cardinality

Assume we can somehow compute the cardinality of a minimum MUS or at least its lower-bound $\texttt{minMUS}$. Knowing this number, we define our next wrapper as $\mathcal{W}_4 = \{N \in \mathcal{W}_1 \mid |N| \geq \texttt{minMUS}\}$. To encode this wrapper via a formula $\mathbb{W}_4$, we employ a Boolean cardinality constraint $\texttt{atLeast}(\mathcal{A}, \texttt{minMUS})$ expressing that at least $\texttt{minMUS}$ variables from $\mathcal{A}$ are set to 1:

$$\mathbb{W}_4 = \mathbb{W}_1 \wedge \texttt{atLeast}(\mathcal{A}, \texttt{minMUS}) \tag{5}$$

Proposition 8. *For every valuation π of $\mathcal{A}$, $\pi \in M_{\mathbb{W}_4 \downarrow \mathcal{A}}$ iff $\pi_{\mathcal{A},F} \in \mathcal{W}_4$. Consequently, $|M_{\mathbb{W}_4 \downarrow \mathcal{A}}| = |\mathcal{W}_4|$.*

There have been proposed several algorithms for computing an MUS with the minimum cardinality, e.g. [26,27,38]. However, since the task of computing a minimum MUS is in $\mathrm{FP}^{\Sigma_2^P}$ [27,37], computing exactly a minimum MUS is too expensive for our scenario (empirically experienced). Instead, we propose an approach for cheaply computing a lower-bound on the minimum MUS cardinality.

Our method is based on a well-known relationship between MUSes and MCSes called *minimal hitting set duality* [32,56]. Given a collection $\mathcal{C}$ of sets, a set X is a *hitting set* of $\mathcal{C}$ iff $C \cap X \neq \emptyset$ for every $C \in \mathcal{C}$. Furthermore, a hitting set X of $\mathcal{C}$ is *minimal* if none of its proper subsets is a hitting set. The duality relation states that a set N is an MUS of F iff N is a minimal hitting set of the set $\texttt{MCS}_F$ of all MCSes of F. Dually, a set M is an MCS of F iff M is a minimal hitting set of the set $\texttt{MUS}_F$. Consequently, one can identify all the MCSes and then compute their *minimum minimal* hitting set to get an MUS with the minimum cardinality. However, there can be up to exponentially many MCSes of F, and thus their complete enumeration is often practically intractable. Our approach to obtain a lower-bound on the minimum MUS cardinality is the following. First, we employ a recent MCS enumeration algorithm RIME [11] to generate a subset $\mathcal{M}$ of $\texttt{MCS}_F$. Subsequently, we compute a minimum minimal hitting set of $\mathcal{M}$ and use it as the lower-bound $\texttt{minMUS}$ on the minimum MUS cardinality while building the wrapper $\mathcal{W}_4$. Note that since $\mathcal{M} \subseteq \texttt{MCS}_F$, it holds that every hitting set of $\texttt{MCS}_F$ is also a hitting set of $\mathcal{M}$, and hence $\texttt{minMUS}$ is indeed a sound lower-bound on the cardinality of a minimum hitting set of $\texttt{MCS}_F$.

Let us also briefly describe an algorithm for computing the minimum MUS by Ignatiev et al. [27], since it works on a similar principle as our approach. Their algorithm iteratively maintains a set *kMCSes* of known MCSes; initially *kMCes* $= \emptyset$. In each iteration, the algorithm computes a minimum minimal hitting set X of *kMCSes* and checks X for satisfiability. If X is unsatisfiable, then it is guaranteed to be a minimum MUS. Otherwise, X is enlarged to an MSS using a single MSS extraction subroutine, the complement of the MSS (i.e., an MCS) is added to *kMCSes*, and the algorithm proceeds with a next iteration. Observe that one can also terminate their approach after a given time limit and use the last computed X as a lower-bound on the minimum MUS cardinality. The main difference between our and their approach is that we employ a dedicated MCS enumerator in the first step and then compute just a single minimum minimal hitting set, whereas they alternate single MCS extraction with minimum minimal hitting set computation.

4.6 $\mathcal{W}_5$ - Maximum MUS Cardinality

Assuming that we can somehow compute an upper-bound $\mathtt{maxMUS}$ on the maximum cardinality of an MUS of F, we define our next wrapper as $\mathcal{W}_5 = \{N \in \mathcal{W}_1 \mid |N| \leq \mathtt{maxMUS}\}$. Similarly as in the case of $\mathbb{W}_4$, to build the formula $\mathbb{W}_5$ that encodes $\mathcal{W}_5$, we introduce a Boolean cardinality constraint $\mathtt{atMost}(\mathcal{A}, \mathtt{maxMUS})$ expressing that at most $\mathtt{maxMUS}$ variables from $\mathcal{A}$ are set to 1:

$$\mathbb{W}_5 = \mathbb{W}_1 \wedge \mathtt{atMost}(\mathcal{A}, \mathtt{maxMUS}) \tag{6}$$

Proposition 9. *For every valuation π of $\mathcal{A}$, $\pi \in M_{\mathbb{W}_5 \downarrow \mathcal{A}}$ iff $\pi_{\mathcal{A},F} \in \mathcal{W}_5$. Consequently, $|M_{\mathbb{W}_5 \downarrow \mathcal{A}}| = |\mathcal{W}_5|$.*

We are not aware of any prior work on computing the cardinality of the maximum MUS nor of a reasonable approach for computing at least its upper-bound. Hence, we propose a custom approach to compute such an upper-bound $\mathtt{maxMUS}$. The base idea is to exploit our concept of wrappers:

Proposition 10. *Let $\mathcal{W}$ be a wrapper, i.e. $\mathcal{W} \subseteq \mathtt{MUS}_F \cup \mathtt{SS}_F$, $\mathcal{A}$ the set of activation variables, and $\mathbb{W}$ a formula such that for every valuation π of $\mathcal{A}$, $\pi \in M_{\mathbb{W} \downarrow \mathcal{A}}$ iff $\pi_{\mathcal{A},F} \in \mathcal{W}$. Furthermore, let $\mathtt{maxOnes} = \mathtt{max}(\{\mathit{ones}(\pi) \mid \pi \in M_{\mathbb{W} \downarrow \mathcal{A}}\})$ where $\mathit{ones}(\pi) = |\{a_i \in \mathcal{A} \mid \pi(a_i) = 1\}|$. Then $\mathtt{maxOnes}$ is an upper-bound on the maximum MUS cardinality.*

We use $\mathtt{maxOnes}$ as the value $\mathtt{maxMUS}$ while constructing wrapper $\mathcal{W}_5$. Any of the wrappers and its encoding presented in this paper can be used as $\mathcal{W}$ and $\mathbb{W}$, respectively. To determine the value $\mathtt{maxOnes}$, we define a partial MaxSAT problem using the formula $\mathbb{W} \wedge \bigwedge_{a_i \in \mathcal{A}} a_i$, where $\mathbb{W}$ are the hard clauses and $\bigwedge_{a_i \in \mathcal{A}} a_i$ are the soft clauses. To solve the problem, we employ the MaxSAT solver UWrMaxSat [54].

4.7 $\mathcal{W}_6$ - Component Partitioning

It is often the case that the clauses of F can be partitioned into several *components*, i.e. disjoint subsets of clauses, such that every MUS of F consists only of clauses from a single component. In particular:

Definition 7 (components). *Given a clause $f_i \in F$, the component $\mathcal{C}(f_i)$ of f_i is the minimal subset of F satisfying:*

1. $f_i \in \mathcal{C}(f)$, and
2. for every $l \in f_i$ and every $f_j \in F$ with $\neg l \in f_j$, $\mathcal{C}(f_i) = \mathcal{C}(f_j)$.

Example 2. Assume that $F = \{\{x_1\}, \{\neg x_1\}, \{x_2\}, \{\neg x_1, \neg x_2\}, \{x_3\}, \{\neg x_3\}, \{x_4\}, \{x_4, x_5\}\}$. There are four components: $C_1 = \{\{x_1\}, \{\neg x_1\}, \{x_2\}, \{\neg x_1, \neg x_2\}\}$, $C_2 = \{\{x_3\}, \{\neg x_3\}\}$, $C_3 = \{\{x_4\}\}$, and $C_4 = \{\{x_4, x_5\}\}$. C_1 has two MUSes: $\{\{x_1\}, \{\neg x_1\}\}$ and $\{\{x_1\}, \{x_2\}, \{\neg x_1, \neg x_2\}\}$, C_2 has one MUS: $\{\{x_3\}, \{\neg x_3\}\}$, and C_3 and C_4 have no MUSes.

Proposition 11. *Let N be an MUS. Then for every two clauses $f_i, f_j \in N$, it holds that $\mathcal{C}(f_i) = \mathcal{C}(f_j)$.*

The wrapper $\mathcal{W}_6$ captures the partition of MUSes into components, and it is defined as $\mathcal{W}_6 = \{N \in \mathcal{W}_1 \mid \forall_{f_i, f_j \in N}.\, \mathcal{C}(f_i) = \mathcal{C}(f_j)\}$ and encoded via $\mathbb{W}_6$:

$$\mathbb{W}_6 = \mathbb{W}_1 \wedge \bigwedge_{a_i \in \mathcal{A}} (a_i \rightarrow \bigwedge_{f_j \in F \setminus \mathcal{C}(f_i)} \neg a_j) \tag{7}$$

Proposition 12. *For every valuation π of $\mathcal{A}$, $\pi \in M_{\mathbb{W}_6 \downarrow \mathcal{A}}$ iff $\pi_{\mathcal{A},F} \in \mathcal{W}_6$. Consequently, $|M_{\mathbb{W}_6 \downarrow \mathcal{A}}| = |\mathcal{W}_6|$.*

To partition the input formula F into individual components, we construct an undirected graph whose vertices are the clauses of F and every two vertices, f_i and f_j, are connected via an edge iff there exists $l \in f_i$ such that $\neg l \in f_j$. The components of F then correspond to connected components of the graph (which can be identified in linear time w.r.t. the size of F by traversing the graph). Note that a similar *flip graph* has been used in a study [68] on *model rotation* and its usage during single MUS extraction.

4.8 $\mathcal{W}_7$ - Minimal Hitting Set Duality

We again exploit the minimal hitting set duality between MUSes and MCSes (Sect. 4.5). Recall that if a set M is an MCS of F then $M \cap N \neq \emptyset$ for every $N \in \mathtt{MUS}_F$. We define the wrapper $\mathcal{W}_7$ as $\{N \in \mathcal{W}_1 \mid \forall_{M \in \mathcal{M}} M \cap N \neq \emptyset\}$ where $\mathcal{M}$ is a set of MCSes. To obtain $\mathcal{M}$, we run an MCS enumeration algorithm RIME [11] constrained by a user-defined time limit. The encoding $\mathbb{W}_7$ of $\mathcal{W}_7$ is:

$$\mathbb{W}_7 = \mathbb{W}_1 \wedge \bigwedge_{M \in \mathcal{M}} \bigvee_{f_i \in M} a_i \tag{8}$$

Proposition 13. *For every valuation π of $\mathcal{A}$, $\pi \in M_{\mathbb{W}_7 \downarrow \mathcal{A}}$ iff $\pi_{\mathcal{A},F} \in \mathcal{W}_7$. Consequently, $|M_{\mathbb{W}_7 \downarrow \mathcal{A}}| = |\mathcal{W}_7|$.*

4.9 $\mathcal{W}_8$ - Literal Negation Cover

Our next wrapper captures the following observation about MUSes.

Proposition 14. *Let N be an MUS of F, $f_i \in N$ a clause of N, and $l \in f_i$ a literal of f_i. Then there exists a clause $f_j \in N$ such that $\neg l \in f_j$.*

Based on the above proposition, we define the wrapper $\mathcal{W}_8$ as $\mathcal{W}_8 = \{N \in \mathcal{W}_1 \,|\, \forall_{f_i \in N}.\, \forall_{l \in f_i}.\, \exists_{f_j \in N}.\, \neg l \in f_j\}$, and encode it as follows:

$$\mathbb{W}_8 = \mathbb{W}_1 \wedge \bigwedge_{a_i \in \mathcal{A}} a_i \rightarrow (\bigwedge_{l \in f_i} (\bigvee_{f_j \in \{f_j \in F \,|\, \neg l \in f_j\}} a_j)) \tag{9}$$

Proposition 15. *For every valuation π of $\mathcal{A}$, $\pi \in M_{\mathbb{W}_8 \downarrow \mathcal{A}}$ iff $\pi_{\mathcal{A},F} \in \mathcal{W}_8$. Consequently, $|M_{\mathbb{W}_8 \downarrow \mathcal{A}}| = |\mathcal{W}_8|$.*

4.10 $\mathcal{W}_9$ - Non-extendable Evidence Models

Assume that N is an MUS and $(\rho_1, \dots, \rho_n)$ is its evidence. By Definition 6, it holds that $\rho_i \models N \setminus \{f_i\}$ for every $1 \leq i \leq n$. Observe that since N is unsatisfiable, then it is also necessarily the case that $\rho_i \models \neg f_i$ for every $1 \leq i \leq n$. Hence, we define our next wrapper, $\mathcal{W}_9$, as $\mathcal{W}_9 = \{N \in \mathcal{W}_1 \,|\, \exists \rho_1, \dots, \rho_n.\, \forall_{1 \leq i \leq n}.\, \rho_i \models N \setminus \{f_i\} \,\text{and}\, \rho_i \models \neg f_i\}$. Note that the above-stated property applies *universally* to every evidence of an MUS, and yet we require in the definition of the wrapper only an *existence* of one such evidence. The reason is that there can be up to exponentially many evidences for an MUS w.r.t. $|\mathit{Vars}(F)|$ and hence it is intractable to reason about all of them in the Boolean encoding of the wrapper.

$$\mathbb{W}_9 = \mathbb{W}_1 \wedge \bigwedge_{a_i \in \mathcal{A}} a_i \rightarrow \neg f_{i[\mathit{Vars}(F)/\mathcal{I}_i]} \tag{10}$$

Proposition 16. *For every valuation π of $\mathcal{A}$, $\pi \in M_{\mathbb{W}_9 \downarrow \mathcal{A}}$ iff $\pi_{\mathcal{A},F} \in \mathcal{W}_9$. Consequently, $|M_{\mathbb{W}_9 \downarrow \mathcal{A}}| = |\mathcal{W}_9|$.*

4.11 $\mathcal{W}_{10}$ - Enforced Evidence Models

Our final wrapper, $\mathcal{W}_{10}$, again builds on the variable valuations $\rho_1, \dots, \rho_n$ that form an evidence of an MUS N of F. In the previous wrapper, $\mathcal{W}_9$, we have exploited that none of the variable valuations can be a model of N. Here, we express that none of the valuations can be *easily modified* to be a model of N. In particular, if $f_i \in N$, then by the definition of an evidence, $\rho_i \models N \setminus \{f_i\}$. Assume that we pick a literal $l \in f_i$ and turn ρ_i into a valuation ρ'_i by flipping the assignment to l so that $\rho'_i \models f_i$. Since N is an MUS (i.e., unsatisfiable), there necessarily exists a clause $f_j \in N$ such that $\rho'_i \not\models f_j$, i.e., f_j *forces* ρ_i to satisfy $\neg l$ and hence prevents from flipping ρ_i to a model ρ'_i of the whole N. Formally:

Proposition 17. *Let N be an MUS, $f_i \in N$ a clause of N, and ρ_i a model of $N \setminus \{f_i\}$. Then for every literal $l \in f_i$, there exists a clause $f_j \in N$ such that $\neg l \in f_j$ and $\rho_i \not\models f_j \setminus \{\neg l\}$.*

Similarly as in the case of $\mathcal{W}_9$, observe that Proposition 17 applies *universally* to every evidence of an MUS, however, since there can be exponentially many such evidences, it is expensive to reason about all of them. Hence, in the wrapper, we capture just an *existence* of such an evidence: $\mathcal{W}_{10} = \{N \in \mathcal{W}_1 \mid \exists \rho_1, \ldots, \rho_n . \forall_{1 \le i \le n} . \rho_i \models N \setminus \{f_i\}$ and if $f_i \in N$ then $\forall_{l \in f_i} . \exists_{f_j \in N} . \neg l \in f_j$ and $\rho_i \not\models f_j \setminus \{\neg l\}\}$. Equation 11 shows the corresponding encoding via $\mathbb{W}_{10}$:

$$\mathbb{W}_{10} = \mathbb{W}_1 \wedge \bigwedge_{a_i \in \mathcal{A}} a_i \rightarrow \bigwedge_{l \in f_i} (\bigvee_{f_j \in \{f_j \in F \mid \neg l \in f_j\}} a_j \wedge \neg(f_j \setminus \{\neg l\})_{[Vars(F)/\mathcal{I}_i]}) \tag{11}$$

Proposition 18. *For every valuation π of $\mathcal{A}$, $\pi \in M_{\mathbb{W}_{10} \downarrow \mathcal{A}}$ iff $\pi_{\mathcal{A},F} \in \mathcal{W}_{10}$. Consequently, $|M_{\mathbb{W}_{10} \downarrow \mathcal{A}}| = |\mathcal{W}_{10}|$.*

4.12 Combining Wrappers and Their Remainders

In the previous sections, we have presented multiple wrappers, each of which captures a different property of MUSes. In this section, we show that the individual wrappers can be easily combined and, hence, form wrappers that provide a more accurate description of the set $\texttt{MUS}_F$.

Proposition 19. *Let $\mathcal{A}$ be the set of activation variables, $\mathcal{W}^k$ and $\mathcal{W}^l$ wrappers, and $\mathcal{R}^k$ and $\mathcal{R}^l$ the remainders of $\mathcal{W}^k$ and $\mathcal{W}^l$. Furthermore, for every $m \in \{k, l\}$, let $\mathbb{W}^m$ and $\mathbb{R}^m$ be formulas such that:*

- *for every valuation π of $\mathcal{A}$, $\pi \in M_{\mathbb{W}^m \downarrow \mathcal{A}}$ iff $\pi_{\mathcal{A},F} \in \mathcal{W}^m$, and*
- *for every valuation π of $\mathcal{A}$, $\pi \in M_{\mathbb{R}^m \downarrow \mathcal{A}}$ iff $\pi_{\mathcal{A},F} \in \mathcal{R}^m$.*

Then all the following hold:

1. *$\mathcal{W}^k \cap \mathcal{W}^l$ is a wrapper and $\mathcal{R}^k \cap \mathcal{R}^l$ is its reminder.*
2. *For every valuation π of $\mathcal{A}$, $\pi \in M_{(\mathbb{W}^k \wedge \mathbb{W}^l) \downarrow \mathcal{A}}$ iff $\pi_{\mathcal{A},F} \in \mathcal{W}^k \cap \mathcal{W}^l$. Consequently, $|M_{(\mathbb{W}^k \wedge \mathbb{W}^l) \downarrow \mathcal{A}}| = |\mathcal{W}^k \cap \mathcal{W}^l|$.*
3. *For every valuation π of $\mathcal{A}$, $\pi \in M_{(\mathbb{R}^k \wedge \mathbb{R}^l) \downarrow \mathcal{A}}$ iff $\pi_{\mathcal{A},F} \in \mathcal{R}^k \cap \mathcal{R}^l$. Consequently, $|M_{(\mathbb{R}^k \wedge \mathbb{R}^l) \downarrow \mathcal{A}}| = |\mathcal{R}^k \cap \mathcal{R}^l|$.*

Note that although Proposition 19 discusses only a combination of two wrappers, it can be applied repeatedly on already combined wrappers. Hence, we can combine any subset of the wrappers $\mathcal{W}_1, \ldots, \mathcal{W}_{10}$ we proposed. Also, note that all the formulas $\mathbb{W}_2, \ldots, \mathbb{W}_{10}$ subsume the formula $\mathbb{W}_1$, and hence if we combine multiple wrappers, we duplicate some clauses. In our implementation, we first remove all the duplicates and apply other straightforward model preserving simplifications before we pass the encoding to a projected model counter.

5 Experimental Evaluation

We have implemented our approach for counting MUSes in a python-based tool[1], using the projected model counter GANAK [59] to count the models of wrappers and remainders, and also using several auxiliary tools as described above.

We presented 10 *base* wrappers $\mathcal{W}_1, \dots, \mathcal{W}_{10}$ and shown how to combine them. Since $\mathcal{W}_1$ is subsumed by all the wrappers $\mathcal{W}_2, \dots, \mathcal{W}_{10}$, there are 2^9 combined wrappers. Due to the large number of the combinations, we were able to evaluate only some of them. In particular, we evaluated the combination $\mathcal{W}_1 \cap \dots \cap \mathcal{W}_{10}$, denoted as *Wall*, of all wrappers since it provides the most precise description of MUSes. We also evaluated 6 wrappers that emerge from Wall by excluding individual base wrappers or combinations of similar base wrappers, and also the most basic wrapper $\mathcal{W}_1$. The table below shows the names and definitions of the evaluated combinations:

name	definition
W1	$\mathcal{W}_1$
Wno23	$\bigcap_{i \in \{4,5,6,7,8,9,10\}} \mathcal{W}_i$
Wno4	$\bigcap_{i \in \{2,3,5,6,7,8,9,10\}} \mathcal{W}_i$
Wno5	$\bigcap_{i \in \{2,3,4,6,7,8,9,10\}} \mathcal{W}_i$
Wno6	$\bigcap_{i \in \{2,3,4,5,7,8,9,10\}} \mathcal{W}_i$
Wn o7	$\bigcap_{i \in \{2,3,4,5,6,8,9,10\}} \mathcal{W}_i$
Wno8910	$\bigcap_{i \in \{2,3,4,5,6,7\}} \mathcal{W}_i$
Wall	$\bigcap_{i \in \{2,\dots,10\}} \mathcal{W}_i$

We also evaluated two contemporary MUS enumerators, MARCO[2] [39] and UNIMUS[3] [10]. Moreover, we evaluated the approximate MUS counter AMUSIC[4] [13] using its default guarantees, i.e., the provided MUS count estimates are within 1.8 multiplicative factor of the true count with 80% confidence.

Our benchmark suite consists of the 2553 instances previously employed in the prior MUS and MSS literature, including those released by authors of AMUSIC [13]. The formulas contain from 78 to 1000 clauses and from 40 to 996 variables. The MUS count varies from 1 to 1.7×10^9 MUSes.

We focus on three comparison criteria: 1) the number of benchmarks solved by the evaluated tools (i.e. benchmarks where the tools provided the MUS count), 2) the scalability of the tools w.r.t. the number of MUSes in the benchmarks, and 3) we examine the *accuracy* of our wrappers.

All experiments were run using a time limit of 3600 s per benchmark on a Linux machine with AMD 16-Core Processor and 20 GB memory limit. When using wrappers $\mathcal{W}_4$ and $\mathcal{W}_7$, we used a combined limit of 300 s (included in the 3600 s) and 100000 MCSes for the MCS enumeration while building the wrappers; if both wrappers were used, we run the MCS enumeration just once. Finally, while constructing a combined wrapper of the form $\mathcal{W}_* \cap \mathcal{W}_5$, we used $\mathcal{W}_*$ to compute the value `maxMUS` for creating $\mathcal{W}_5$.

[1] https://github.com/jar-ben/exactMUSCounter.
[2] https://sun.iwu.edu/~mliffito/marco/.
[3] https://github.com/jar-ben/unimus.
[4] https://github.com/jar-ben/amusic.

Table 1. Number of solved benchmarks by individual tools.

			Our Wrapper-Remainder Based Tools							
AMUSIC	UNIMUS	MARCO	W1	Wno23	Wno4	Wno6	Wno7	Wno8910	Wall	Wno5
623	833	799	403	1475	1498	1486	1445	1058	1486	**1500**

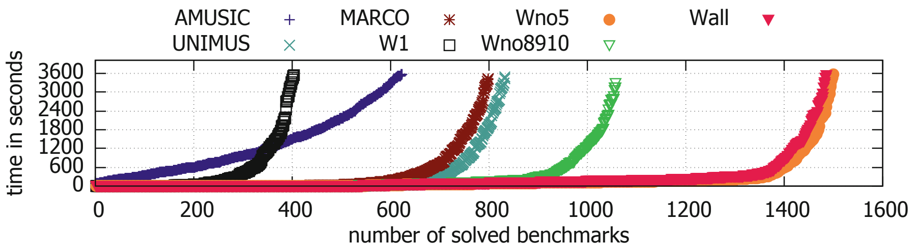


Fig. 2. The number of solved benchmarks in time.

5.1 Solved Benchmarks

In Table 1, we show the number of benchmarks that were solved by the individual evaluated tools. The worst performance was achieved by the basic wrapper W1 ($\mathcal{W}_1$), which is not surprising since it does not provide a good description of MUSes. AMUSIC solved 623 benchmarks, whereas UNIMUS and MARCO solved 833 and 799 benchmarks, respectively. Except for Wno8910 (and W1), which solved *only* 1058 benchmarks, all the remaining combined wrappers solved around 1450–1500 benchmarks and hence significantly dominated both AMUSIC and the two MUS enumerators. Maybe surprisingly, Wall that combines all the base wrappers ended up at the third position; the highest number (1500) of solved benchmarks was achieved by Wno5, and the second-highest (1498) by Wno4. Note that Wno5 and Wno4 exclude encoding of the minimum and maximum MUS cardinality via Boolean cardinality constraints. In general, solving Boolean cardinality constraints is often quite hard, and hence even though a presence of the two wrappers might provide a better description of MUSes, the constraints increase the hardness of the generated instances.

Figure 2 compares the time needed to solve the benchmarks by a subset (for a better clarity) of the evaluated tools. A point with coordinates $[x, y]$ means that x benchmarks were solved (by the corresponding tool) within the first y seconds.

5.2 Scalability W.r.t the MUS Count

In Fig. 3, we compare the scalability of the evaluated tools w.r.t. the number of MUSes in the benchmarks. In particular, a point with coordinates $[x, y]$ denotes that the corresponding tool solved y benchmarks that contained at most x MUSes. For a better clarity, we compare only our best wrapper, Wno5, with AMUSIC, MARCO, and UNIMUS. Note that whereas AMUSIC scales to instances

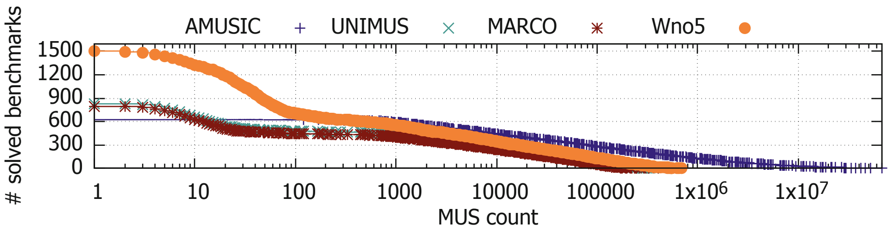


Fig. 3. The number of solved w.r.t. the MUS count.

with 10^8 MUSes, the remaining three tools scale only to instances with at most a million of MUSes. In fact, note that even though AMUSIC solved in overall *just* 623 benchmarks, there are 319 benchmarks that were solved only by AMUSIC. Based on a closer examination of the results, we identified that AMUSIC scales much better than the other tools w.r.t. the MUS count, however, it does not scale so well w.r.t. the number of clauses in the input formula F. This is not surprising since AMUSIC is *just* an approximate counter and as such, it needs to explicitly identify only logarithmically many MUSes w.r.t. $|F|$ even though there can be up to $\mathcal{O}(2^{|F|})$ many MUSes. On the other hand, AMUSIC relies on repeated calls to a 3-QBF solver whose efficiency highly depends on $|F|$.

5.3 Accuracy of Wrappers

Recall that a wrapper $\mathcal{W}$ *over-approximates* the set $\texttt{MUS}_F$ of all MUSes of F, i.e., $\mathcal{W} \supseteq \texttt{MUS}_F$ (Definition 5), and hence we are interested in measuring the *accuracy* of the over-approximations. In particular, given a wrapper $\mathcal{W}$ and its remainder $\mathcal{R}$ constructed over a formula F, we measure the ratio $\frac{|\mathcal{R}|}{|\mathcal{W}|}$. The range of the ratio is $[0, 1)$; the closer to 0 the more accurate the wrapper is, and especially when $\frac{|\mathcal{R}|}{|\mathcal{W}|} = 0$, the wrapper *exactly* captures the set $\texttt{MUS}_F$ (i.e., $\mathcal{W} = \texttt{MUS}_F$).

We illustrate the ratio $\frac{|\mathcal{R}|}{|\mathcal{W}|}$ achieved by individual wrappers in Fig. 4. A point with coordinates $[x, y]$ expresses that for x percent of benchmarks completed by the corresponding tool, the ratio $\frac{|\mathcal{R}|}{|\mathcal{W}|}$ was at most y. As expected, the ratio achieved by the most basic wrapper W1 ($\mathcal{W}_1$) is very high for all the benchmarks, i.e., the wrapper captures $\texttt{MUS}_F$ very inaccurately. On the other hand, the other wrappers achieved for a vast majority of benchmarks a very low ratio, i.e., they over-approximate $\texttt{MUS}_F$ very tightly. In fact, for 87% of benchmarks, the wrappers Wno23, Wno4, Wno5, Wno6, and Wall, achieved the ratio 0, i.e., the wrappers exactly captured the set $\texttt{MUS}_F$. In contrast, the wrappers Wno7 and Wno8910 achieved the ratio 0 for *only* 68 and 80% of benchmarks, which suggest that the use of the corresponding wrappers, $\mathcal{W}_7$, $\mathcal{W}_8$, $\mathcal{W}_9$, and $\mathcal{W}_{10}$, is vital for an accurate description of $\texttt{MUS}_F$. Moreover, note that the accuracy of the wrappers highly correlate with the number of solved benchmarks (Table 1), since Wno7 and Wno8910 (and W1) were the least efficient wrappers.

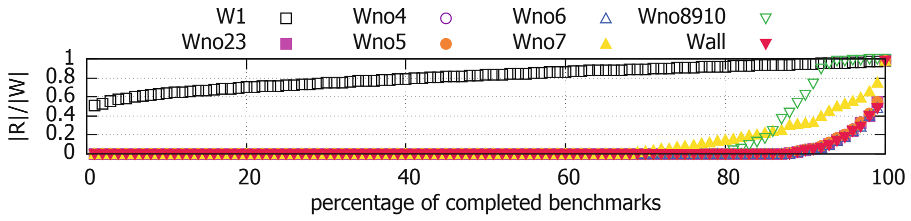


Fig. 4. The ratio $\frac{|\mathcal{R}|}{|\mathcal{W}|}$ expressing the inaccuracy of wrappers.

6 Future Possible Applications of Wrappers and Remainders

Recall that a wrapper $\mathcal{W}$ *over-approximates* the set $\texttt{MUS}_F$ of all MUSes of F, i.e., $\mathcal{W} \supseteq \texttt{MUS}_F$ (Definition 5). Moreover, in Sect. 5, we empirically witnessed that the best of our wrappers usually over-approximate $\texttt{MUS}_F$ very tightly or they even capture it exactly. Consequently, the propositional encodings $\mathbb{W}$ and $\mathbb{R}$ of a wrapper $\mathcal{W}$ and its remainder $\mathcal{R}$, respectively, can very precisely capture the set $\texttt{MUS}_F$. We strongly believe that such an accurate propositional description of $\texttt{MUS}_F$ paves the way (and will be thoroughly examined in our future work) to efficiently solve many other MUS related problems including, e.g., the following:

Approximate MUS Counting. Recall that $|\texttt{MUS}_F| = |\mathcal{W}| - |\mathcal{R}|$. Assuming that $|\mathcal{R}|$ is much smaller than $|\mathcal{W}|$ and observing that $\mathcal{R} \subseteq \mathcal{W}$, computing $|M_{\mathbb{R}\downarrow\mathcal{A}}| = |\mathcal{R}|$ should be much faster than computing $|M_{\mathbb{W}\downarrow\mathcal{A}}| = |\mathcal{W}|$. Hence, one could first relatively quickly *exactly* compute the value $|M_{\mathbb{R}\downarrow\mathcal{A}}|$, and then use an *approximate* model counter to find an *estimate* w' of $|M_{\mathbb{W}\downarrow\mathcal{A}}|$. The MUS count $|\texttt{MUS}_F|$ can be then approximated as $w' - |\mathcal{R}|$. The *accuracy* of the approximation depends on the approximation guarantees of the model counter (e.g. using ApproxMC4 [18,60], we get the (ϵ, δ)-guarantees provided by AMUSIC).

MUS Enumeration. Assume a valuation π of the activation variables $\mathcal{A}$ and the corresponding *activated* subset $\pi_{\mathcal{A},F} = \{f_i \in F \mid \pi(a_i) = 1\}$ of F. As shown in Sect. 4, $\pi_{\mathcal{A},F}$ is an MUS iff $\pi \in M_{\mathbb{W}\downarrow\mathcal{A}}$ and $\pi \notin M_{\mathbb{R}\downarrow\mathcal{A}}$. Hence, one can enumerate MUSes by enumerating projected models of $\mathbb{W}$ and discarding those that are also projected models of $\mathbb{R}$.

MUS Sampling. To sample an MUS of F, one can iteratively sample an element π of $M_{\mathbb{W}\downarrow\mathcal{A}}$ until it identifies π such that $\pi \notin M_{\mathbb{R}\downarrow\mathcal{A}}$, i.e., $\pi_{\mathcal{A},F}$ is an MUS. Note that while the past decade has witnessed significant progress in the development of projected model sampling approaches [16,22,55] (with various distribution guarantees), we are not aware of any existing MUS sampling technique (with reasonable distribution guarantees).

Minimum and Maximum MUS Cardinality. As discussed in Sect. 4.6 ($\mathcal{W}_5$), one can over-approximate the maximum MUS cardinality by finding a model $\pi \in M_{\mathbb{W}\downarrow\mathcal{A}}$ that maximizes the number of variables assigned 1. Similarly, one can under-approximate the minimum MUS cardinality by finding a model $\pi \in M_{\mathbb{W}\downarrow\mathcal{A}}$ that minimizes the number of variables assigned 1. Intuitively, the smaller $|\mathcal{R}|$ is, the more precise approximations can be expected. Moreover, by checking if $\pi \in M_{\mathbb{R}\downarrow\mathcal{A}}$, one can actually verify if $\pi_{\mathcal{A},F}$ is an MUS.

MUS Membership. The MUS membership problem is to decide if a clause $f_i \in F$ belongs to an MUS of F and it is known to be Σ_2^P-complete [31,35,37]. Contemporary techniques for deciding the problem are mainly based on solving 2-QBF or 3-QBF encodings [13,31]. Our wrapper-based framework allows for an alternative approach: to decide if a clause f_i belongs to an MUS of F, one can check if there exists a valuation π of $\mathcal{A}$ such that $\pi(a_i) = 1$, $\pi \in M_{\mathbb{W}\downarrow\mathcal{A}}$, and $\pi \notin M_{\mathbb{R}\downarrow\mathcal{A}}$. Note that when $|\mathcal{R}| = 0$ or when $|\mathcal{R}|$ can be bounded by a constant, this check boils down to a single call of a SAT solver.

7 Conclusion and Future Work

In this paper, we focused on the problem of MUS counting and proposed the first exact MUS counter, called CountMUST, that does not rely on explicit MUS enumeration. The base idea is to reduce the problem of MUS counting to (two queries of) projected model counting via the framework of wrappers and remainders. The availability of scalable projected model counter, GANAK, allowed CountMUST to scale much better and solve significantly more instances than other existing approaches. Moreover, as discussed in Sect. 6, the tightness of wrappers and remainders opens up new potential applications ranging from approximating counting, enumeration, membership, and the like.

We also revisit the complementary nature of CountMUST and AMUSIC with respect to the size of instances and the MUS count. The complementary performance opens up opportunities for a portfolio approach that can achieve the best of both of the worlds. Finally, let us note that we are fighting here the *chicken and egg* nature of the existence of practical applications and scalable algorithmic techniques for problems in automated reasoning. Often the lack of scalable techniques leads to a lack of incentives for end-users to design reductions to practical applications, and vice versa. Even though MUS counting has already many applications in the diagnosis domain [25,29,48–50,65], we hope that the availability of CountMUST will break this chicken and egg loop in other areas and enable further investigations into MUS counting applications.

Acknowledgements. This work was supported in part by the National Research Foundation Singapore under its NRF Fellowship Programme [NRF-NRFFAI1-2019-0004] and the AI Singapore Programme [AISG-RP-2018-005], NUS ODPRT Grant [R-252-000-685-13].

References

1. Andraus, Z.S., Liffiton, M.H., Sakallah, K.A.: CEGAR-based formal hardware verification: a case study. Technical report, University of Michigan, CSE-TR-531-07 (2007)

2. Arif, M.F., Mencía, C., Ignatiev, A., Manthey, N., Peñaloza, R., Marques-Silva, J.: BEACON: an efficient SAT-based tool for debugging $\mathcal{EL}^+$ ontologies. In: Creignou, N., Le Berre, D. (eds.) SAT 2016. LNCS, vol. 9710, pp. 521–530. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40970-2_32
3. Bacchus, F., Katsirelos, G.: Using minimal correction sets to more efficiently compute minimal unsatisfiable sets. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9207, pp. 70–86. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21668-3_5
4. Bacchus, F., Katsirelos, G.: Finding a collection of MUSes incrementally. In: Quimper, C.-G. (ed.) CPAIOR 2016. LNCS, vol. 9676, pp. 35–44. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33954-2_3
5. Bailey, J., Stuckey, P.J.: Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. In: Hermenegildo, M.V., Cabeza, D. (eds.) PADL 2005. LNCS, vol. 3350, pp. 174–186. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30557-6_14
6. Belov, A., Heule, M.J.H., Marques-Silva, J.: MUS extraction using clausal proofs. In: Sinz, C., Egly, U. (eds.) SAT 2014. LNCS, vol. 8561, pp. 48–57. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09284-3_5
7. Belov, A., Marques-Silva, J.: MUSer2: an efficient MUS extractor. JSAT **8**, 123–128 (2012)
8. Bendík, J., Beneš, N., Černá, I., Barnat, J.: Tunable online MUS/MSS enumeration. In: FSTTCS. LIPIcs, vol. 65, pp. 50:1–50:13. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2016)
9. Bendík, J., Černá, I.: MUST: minimal unsatisfiable subsets enumeration tool. In: TACAS 2020. LNCS, vol. 12078, pp. 135–152. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45190-5_8
10. Bendík, J., Černá, I.: Replication-guided enumeration of minimal unsatisfiable subsets. In: Simonis, H. (ed.) CP 2020. LNCS, vol. 12333, pp. 37–54. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-58475-7_3
11. Bendík, J., Černá, I.: Rotation based MSS/MCS enumeration. In: LPAR. EPiC Series in Computing, vol. 73, pp. 120–137. EasyChair (2020)
12. Bendík, J., Černá, I., Beneš, N.: Recursive online enumeration of all minimal unsatisfiable subsets. In: Lahiri, S.K., Wang, C. (eds.) ATVA 2018. LNCS, vol. 11138, pp. 143–159. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01090-4_9
13. Bendík, J., Meel, K.S.: Approximate counting of minimal unsatisfiable subsets. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12224, pp. 439–462. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53288-8_21
14. Bendík, J., Meel, K.S.: Counting maximal satisfiable subsets. In: AAAI (2021, to appear)
15. Chakraborty, S., Meel, K.S., Vardi, M.Y.: A scalable approximate model counter. In: CP, pp. 200–216 (2013)
16. Chakraborty, S., Meel, K.S., Vardi, M.Y.: Balancing scalability and uniformity in SAT witness generator. In: DAC, pp. 60:1–60:6. ACM (2014)
17. Chakraborty, S., Meel, K.S., Vardi, M.Y.: Algorithmic improvements in approximate counting for probabilistic inference: From linear to logarithmic SAT calls. In: IJCAI, pp. 3569–3576. IJCAI/AAAI Press (2016)
18. Chakraborty, S., Meel, K.S., Vardi, M.Y.: Algorithmic improvements in approximate counting for probabilistic inference: from linear to logarithmic sat calls. In: IJCAI (2016)
19. Chen, Z., Toda, S.: The complexity of selecting maximal solutions. Inf. Comput. **119**(2), 231–239 (1995)

20. Cohen, O., Gordon, M., Lifshits, M., Nadel, A., Ryvchin, V.: Designers work less with quality formal equivalence checking. In: Design and Verification Conference (DVCon). Citeseer (2010)
21. Durand, A., Hermann, M., Kolaitis, P.G.: Subtractive reductions and complete problems for counting complexity classes. Theor. Comput. Sci. **340**(3), 496–513 (2005)
22. Dutra, R., Laeufer, K., Bachrach, J., Sen, K.: Efficient sampling of SAT solutions for testing. In: ICSE, pp. 549–559. ACM (2018)
23. Fichte, J.K., Hecher, M., Hamiti, F.: The model counting competition 2020. arXiv preprint arXiv:2012.01323 (2020)
24. Han, B., Lee, S.: Deriving minimal conflict sets by CS-trees with mark set in diagnosis from first principles. IEEE Trans. Syst. Man Cybernet. Part B **29**(2), 281–286 (1999)
25. Hunter, A., Konieczny, S.: Measuring inconsistency through minimal inconsistent sets. In: KR. pp. 358–366. AAAI Press (2008)
26. Ignatiev, A., Janota, M., Marques-Silva, J.: Quantified maximum satisfiability. Constraints Int. J. **21**(2), 277–302 (2016)
27. Ignatiev, A., Previti, A., Liffiton, M., Marques-Silva, J.: Smallest MUS extraction with minimal hitting set dualization. In: Pesant, G. (ed.) CP 2015. LNCS, vol. 9255, pp. 173–182. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23219-5_13
28. Ivrii, A., Malik, S., Meel, K.S., Vardi, M.Y.: On computing minimal independent support and its applications to sampling and counting. Constraints **21**(1) (2016)
29. Jabbour, S., Raddaoui, B., Sais, L.: Inconsistency-based ranking of knowledge bases. In: ICAART (2), pp. 414–419. SciTePress (2015)
30. Jannach, D., Schmitz, T.: Model-based diagnosis of spreadsheet programs: a constraint-based debugging approach. Autom. Softw. Eng. **23**(1), 105–144 (2016)
31. Janota, M., Marques-Silva, J.: On deciding MUS membership with QBF. In: Lee, J. (ed.) CP 2011. LNCS, vol. 6876, pp. 414–428. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23786-7_32
32. de Kleer, J., Williams, B.C.: Diagnosing multiple faults. Artif. Intell. **32**(1), 97–130 (1987)
33. Kleine Büning, H., Kullmann, O.: Minimal unsatisfiability and autarkies. In: Handbook of Satisfiability, FAIA, vol. 185, pp. 339–401. IOS Press (2009)
34. Kullmann, O.: Investigations on autark assignments. Discrete Appl. Math. **107**(1–3), 99–137 (2000)
35. Kullmann, O.: Constraint satisfaction problems in clausal form: Autarkies and minimal unsatisfiability. Electronic Colloquium on Computational Complexity (ECCC) 14(055) (2007)
36. Kullmann, O., Marques-Silva, J.: Computing maximal autarkies with few and simple oracle queries. In: Heule, M., Weaver, S. (eds.) SAT 2015. LNCS, vol. 9340, pp. 138–155. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24318-4_11
37. Liberatore, P.: Redundancy in logic I: CNF propositional formulae. Artif. Intell. **163**(2), 203–232 (2005)
38. Liffiton, M.H., Mneimneh, M.N., Lynce, I., Andraus, Z.S., Marques-Silva, J., Sakallah, K.A.: A branch and bound algorithm for extracting smallest minimal unsatisfiable subformulas. Constraints An Int. J. **14**(4), 415–442 (2009)
39. Liffiton, M.H., Previti, A., Malik, A., Marques-Silva, J.: Fast, flexible MUS enumeration. Constraints **21**(2), 223–250 (2016)
40. Liffiton, M.H., Sakallah, K.A.: Algorithms for computing minimal unsatisfiable subsets of constraints. JAR **40**(1), 1–33 (2008)

41. Luo, J., Liu, S.: Accelerating MUS enumeration by inconsistency graph partitioning. Sci. China Inf. Sci. **62**(11), 212104 (2019)
42. Marques-Silva, J., Heras, F., Janota, M., Previti, A., Belov, A.: On computing minimal correction subsets. In: IJCAI, pp. 615–622. IJCAI/AAAI (2013)
43. Marques-Silva, J., Ignatiev, A., Morgado, A., Manquinho, V.M., Lynce, I.: Efficient autarkies. In: ECAI. FAIA, vol. 263, pp. 603–608. IOS Press (2014)
44. Marques-Silva, J., Janota, M.: On the query complexity of selecting few minimal sets. Electronic Colloquium on Computational Complexity (ECCC), vol. 21, 31 (2014)
45. Mencía, C., Kullmann, O., Ignatiev, A., Marques-Silva, J.: On computing the union of MUSes. In: Janota, M., Lynce, I. (eds.) SAT 2019. LNCS, vol. 11628, pp. 211–221. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-24258-9_15
46. Monien, B., Speckenmeyer, E.: Solving satisfiability in less than 2^{n} steps. Discrete Appl. Math. **10**(3), 287–295 (1985)
47. Mu, K.: Formulas free from inconsistency: An atom-centric characterization in priest's minimally inconsistent LP. J. Artif. Intell. Res. **66**, 279–296 (2019)
48. Mu, K.: Formulas free from inconsistency: an atom-centric characterization in priest's minimally inconsistent LP (extended abstract). In: IJCAI, pp. 5090–5094. ijcai.org (2020)
49. Mu, K., Liu, W., Jin, Z.: A general framework for measuring inconsistency through minimal inconsistent sets. Knowl. Inf. Syst. **27**(1), 85–114 (2011)
50. Mu, K., Liu, W., Jin, Z.: Measuring the blame of each formula for inconsistent prioritized knowledge bases. J. Log. Comput. **22**(3), 481–516 (2012)
51. Nadel, A., Ryvchin, V., Strichman, O.: Accelerated deletion-based extraction of minimal unsatisfiable cores. JSAT **9**, 27–51 (2014)
52. Narodytska, N., Bjørner, N., Marinescu, M., Sagiv, M.: Core-guided minimal correction set and core enumeration. In: IJCAI, pp. 1353–1361. ijcai.org (2018)
53. Oh, Y., Mneimneh, M.N., Andraus, Z.S., Sakallah, K.A., Markov, I.L.: AMUSE: a minimally-unsatisfiable subformula extractor. In: DAC, pp. 518–523. ACM (2004)
54. Piotrow, M.: Uwrmaxsat: Efficient solver for maxsat and pseudo-Boolean problems. In: 2020 IEEE 32nd International Conference on Tools with Artificial Intelligence (ICTAI), Los Alamitos, CA, USA, pp. 132–136. IEEE Computer Society, November 2020
55. Plazar, Q., Acher, M., Perrouin, G., Devroey, X., Cordy, M.: Uniform sampling of SAT solutions for configurable systems: are we there yet? In: ICST, pp. 240–251. IEEE (2019)
56. Reiter, R.: A theory of diagnosis from first principles. Artif. Intell. **32**(1), 57–95 (1987)
57. Sang, T., Bacchus, F., Beame, P., Kautz, H.A., Pitassi, T.: Combining component caching and clause learning for effective model counting. In: SAT (2004)
58. Sang, T., Beame, P., Kautz, H.A.: Performing Bayesian inference by weighted model counting. In: AAAI, pp. 475–482. AAAI Press/The MIT Press (2005)
59. Sharma, S., Roy, S., Soos, M., Meel, K.S.: GANAK: a scalable probabilistic exact model counter. In: IJCAI, pp. 1169–1176. ijcai.org (2019)
60. Soos, M., Meel, K.S.: BIRD: engineering an efficient CNF-XOR SAT solver and its applications to approximate model counting. In: AAAI, pp. 1592–1599. AAAI Press (2019)
61. Soos, M., Nohl, K., Castelluccia, C.: Extending SAT solvers to cryptographic problems. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 244–257. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02777-2_24

62. Sperner, E.: Ein satz über untermengen einer endlichen menge. Mathematische Zeitschrift **27**(1), 544–548 (1928)
63. Stockmeyer, L.J.: The complexity of approximate counting (preliminary version). In: STOC, pp. 118–126. ACM (1983)
64. Stuckey, P.J., Sulzmann, M., Wazny, J.: Interactive type debugging in haskell. In: Haskell, pp. 72–83. ACM (2003)
65. Thimm, M.: On the evaluation of inconsistency measures. Measuring Inconsistency in Information **73** (2018)
66. Thurley, M.: sharpSAT – counting models with advanced component caching and implicit BCP. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 424–429. Springer, Heidelberg (2006). https://doi.org/10.1007/11814948_38
67. Valiant, L.G.: The complexity of enumeration and reliability problems. SIAM J. Comput. **8**(3), 410–421 (1979)
68. Wieringa, S.: Understanding, improving and parallelizing MUS finding using model rotation. In: Milano, M. (ed.) CP 2012. LNCS, pp. 672–687. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33558-7_49
69. Xiao, G., Ma, Y.: Inconsistency measurement based on variables in minimal unsatisfiable subsets. In: ECAI. Frontiers in Artificial Intelligence and Applications, vol. 242, pp. 864–869. IOS Press (2012)

Sound Verification Procedures for Temporal Properties of Infinite-State Systems

Quentin Peyras[1], Jean-Paul Bodeveix[2], Julien Brunel[1](✉), and David Chemouil[1]

[1] ONERA DTIS, Université de Toulouse, Toulouse, France
{quentin.peyras,julien.brunel, david.chemouil}@onera.fr

[2] IRIT CNRS UPS, Université de Toulouse, Toulouse, France
jean-paul.bodeveix@irit.fr

Abstract. First-Order Linear Temporal Logic (FOLTL) is particularly convenient to specify distributed systems, in particular because of the unbounded aspect of their state space. We have recently exhibited novel decidable fragments of FOLTL which pave the way for tractable verification. However, these fragments are not expressive enough for realistic specifications. In this paper, we propose three transformations to translate a typical FOLTL specification into two of its decidable fragments. All three transformations are proved sound (the associated propositions are proved in Coq) and have a high degree of automation. To put these techniques into practice, we propose a specification language relying on FOLTL, as well as a prototype which performs the verification, relying on existing model checkers. This approach allows us to successfully verify safety and liveness properties for various specifications of distributed systems from the literature.

1 Introduction

Verifying properties of distributed protocols is a demanding endeavor. Several approaches have been proposed, ranging from verification frameworks, like IronFleet [12] or Verdi [27] to tool-supported languages like TLA$^+$ [17], Event-B [1] or Ivy [20,21]. However, when systems of *arbitrary size* are considered, verifying properties usually requires some remarkable effort: inductive invariants must be sought and exhibited (possibly with tool support), and some manual proof effort may still be necessary. Worse, when *liveness* properties are checked, this effort becomes very substantial and tool support is still quite limited.

A natural setting for specification, in particular for safety and liveness properties of infinite-state systems, is (mono- and many-sorted) first-order linear temporal logic (FOLTL). However, it is highly undecidable [13,14]. In recent work [23,24], some of the present authors devised the "Geneva" fragments of FOLTL, which were shown to be decidable. More precisely, these fragments

A. Silva and K. R. M. Leino (Eds.): CAV 2021, LNCS 12760, pp. 337–360, 2021.
https://doi.org/10.1007/978-3-030-81688-9_16

enjoy a "bounded domain property" (BDP), a form of computable finite model property over the first-order domains. Decidability is obtained by expanding first-order quantifiers over the domains (using the computed bounds) and then relying on (decidable) propositional-LTL satisfiability checking.

The Geneva fragments are rather expressive but still have limitations that thwart their use for the specification of systems. In particular, most forms of fairness assumptions, as well as frame conditions (which specify what does not change when a transition happens in a system), do not fit in the fragments. Furthermore, topological properties of systems (such as ring topologies) are hard or even impossible to specify.

In this article, we mitigate this deficiency by exhibiting three transformations that allow to map an *undecidable*, expressive fragment of $\mathrm{FOLTL}^*_=$ (FOLTL with equality and reflexive-transitive closure, to characterize topological properties) into decidable fragments (akin to the Geneva ones), thus allowing the automatic verification of safety and liveness properties of infinite-state systems. Then we apply these techniques to the verification of properties of various protocols.

Notice that none of the proposed transformations is complete. It is actually impossible to devise complete transformations, even assuming a procedure that would be fed additional user input. This is because FOLTL is not even semi-decidable.[1]

In more detail, we make the following contributions (cf. Fig. 1):

- we define an undecidable, expressive specification language, called Cervino, the semantics of which is expressed in terms of $\mathrm{FOLTL}^*_=$;
- we exhibit two fragments of many-sorted FOLTL that enjoy the BDP;
- we devise three abstraction transformations that map (the semantics of) Cervino into one of the said two fragments:
  - the first of these transformations (called TEA) is fully automatic while the other two (TTC and TFC) must be passed additional data (in the shape of peculiar formulas);
  - these three transformations, as well as other minor ones, are implemented as *tactics* in a prototype tool [22];
  - the associated theorems and lemmas are also formalized and proved correct, using Coq [22];
- we demonstrate our approach on several case studies that are often used as benchmarks in the literature.

This article is organized as follows: in Sect. 2, we illustrate our approach using an example (a leader election protocol). Section 3 introduces definitions as well as the two fragments used in the rest of the paper. In Sect. 4, we present basic techniques, which are used in some of our transformations. Then, in Sect. 5, we formalize the automatic TEA transformation. Section 6 and 7 present, respectively, the TFC and TTC transformations. In Sect. 8, we evaluate our approach on various protocols. Finally, we compare our results with related work in Sect. 9.

[1] Indeed, having such a transformation would give a procedure for semi-decidability by testing all possible inputs on this transformation.

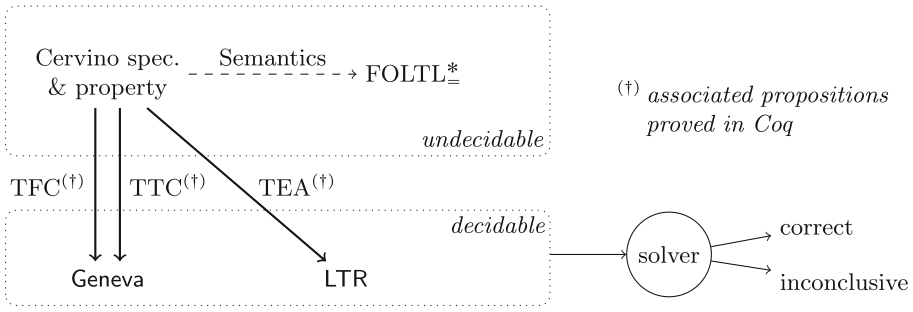


Fig. 1. Summary of the contributions of this article

2 The Cervino Language

In this section, we present the Cervino modeling language informally. Its semantics, given in terms of many-sorted FOLTL$^*_=$ (FOLTL with equality and reflexive-transitive closure), is formally introduced in Sect. 3.3. This language is suitable for specifying infinite-state systems. It is undecidable but we enforce some syntactic constraints anyway, in order to ease the further application of transformations mapping into decidable fragments of logic.

Cervino is illustrated in Fig. 2 using the example of a leader election protocol [6] in a ring of unbounded size. Nodes sit in a directed ring and each node has a unique ID. There is a total order on IDs. The goal of the protocol is to elect a leader (in practice, the one with the greatest ID). A node can send to its successor in the ring the IDs it knows about, the receiver keeping those that are greater than its own ID. A node is elected if it receives its own ID.

2.1 Sorts, Relations and Axioms

A Cervino specification may define sorts, (first-order) sorted relations and sorted constants. An interpretation structure for such a specification is a set of *infinite* traces of states. Classically, a state maps a sort to a non-empty set, a constant to an element of such a set and a relation to a set of tuples, all respecting the obvious sorting and arity constraints. The interpretation of sets and constants is *rigid* while that of relations is *flexible*.

In the example, nodes and their IDs are conflated into a single sort Node; and: an *elected* relation represents the set of elected nodes; a *succ* relation represents if two nodes are successive in the ring topology; a *toSend* relation represents the mailbox for each node; an *lte* relation defines a total ordering on nodes; an *lmax* constant represents the highest maximal identifier among nodes.

States can be constrained by `axioms`, *i.e.* sets of formulas. The latter belong to FOLTL$^*_=$, that is they can mix first-order logic (with equality) with the "always" (**G**), "eventually" (**F**) and "next" (written as a prime symbol and only applied to atoms), as well as a reflexive-transitive closure connective (written *). However, we enforce a syntactic constraint on axioms: after converting them to *negation*

```
sort Node // nodes are conflated with their identifiers
relation succ in Node * Node // successor in the ring
  using btw // btw[succ] is enabled, succ is a function
relation lte in Node * Node // "less than or equal" on nodes
relation toSend in Node * Node // toSend(x,id): id is in x's mailbox
relation elected in Node // set of elected nodes
constant lmax in Node // node with maximal identifier
axiom connected { G (∀ x, y: Node · succ*(x, y) ) }
axiom order { G { ∀ id: Node · lte(id, lmax)
                  ... } } // + classic total ordering axioms
axiom is_elected { G (∀ x: Node · elected'(x) ⇔ (elected(x) ∨ toSend'(x, x))) }
axiom init { // in the initial state...
  ∃ y: Node · succ(lmax, y) // the largest has a successor
  ∀ x, id: Node · !toSend(x, id) // empty mailboxes
  ∀ x: Node · !elected(x) } // no one is elected
event send [src: Node]
  modifies toSend at { (dst,id) · (toSend(src,id) ∨ id = src) ∧ succ(src,dst) },
    elected {
  ∀ dst: Node, id: Node · (succ(src,dst) ∧ (toSend(src,id) ∨ id = src)) ⇒
    (toSend'(dst, id) ⇔
      (toSend(dst, id) ∨ (lte(dst, id) ∧ (id = src ∨ toSend(src, id))))) }
check Safety { G (∀ x: Node · elected(x) ⇒ x = lmax ) }
  using TFC ... // + hidden parameters (see Sect. 6)
check Liveness { F (∃ y: Node · elected(y)) }
  assuming {
    ∀ src: Node · G F {
      ∀ dst: Node, id: Node · (succ(src,dst) ∧ (toSend(src,id) ∨ id = src)) ⇒
        (toSend'(dst, id) ⇔
          (toSend(dst, id) ∨ (lte(dst, id) ∧ (id = src ∨ toSend(src, id))))) } }
  using TTC ... // + hidden parameters (see Sect. 7)
```

Fig. 2. Specification of the leader election protocol (prettified syntax)

normal form (NNF), *an existential quantifier cannot appear in the scope of a universal quantifier or of a* **G** *connective* (no $\forall \ldots \exists \ldots$, no $\mathbf{G} \ldots \exists \ldots$).

A binary relation r can by "tagged" (written **using btw**) to force r to be a function[2] and enable a special ternary relation **btw**[r]. Then, **btw**[r](x,y,z) means that there is an acyclic path between x and z passing through y. The semantics of **btw**[r] is given through axioms (see Definition 14) and is related to r^* through the following equivalence: $r^*(x, y) \Leftrightarrow \mathbf{btw}[r](\mathbf{x}, \mathbf{y}, \mathbf{y})$.

2.2 Events

Events specify how the system may evolve from one state to another. Events (more precisely: event schemas) are declared with a name and a list of arguments that are the only variables that can appear free in the body of the event.

[2] $\forall$ x,y,z: s · r(x,y) $\wedge$ r(x,z) $\Rightarrow$ y = z.

The declaration of an event also features a `modifies` section describing which tuples of which relations may be modified by the event. Other relations or parts of relations are necessarily left unchanged. The body of an event is specified in *primed FO* (FO augmented with primed relation symbols representing the value of these relations in the next state) with the additional constraint that *no existential quantifier may appear positively in the body.*

The semantics for events is standard and comparable to the one used in TLA$^+$ or Electrum: in every state, at least one event is fired. In other words, there is a valuation for arguments of at least one event such that the body of the said event evaluates to true. More formally (and ignoring sorting constraints for the sake of readability), given event bodies $\phi_1, \ldots, \phi_n$ and arguments $y_1, \ldots, y_{m_i}$ appearing as free variables in ϕ_i, the semantics of event is given by the formula: $\mathbf{G}(\bigvee_{i=1}^{n} \exists y_1, \ldots y_{m_i} \cdot \phi_i)$. We insist that this formula is only implicit: it cannot be input by the specifier as it is the purpose of transformations to massage it. Finally, if needed, fairness constraints must be added by the specifier .

In the example, the *send* event represents the fact that a node updates its successor's mailbox by adding all IDs that are larger than the successor's ID. This way, the largest ID is passed along the ring. Notice we use *universal* quantification: we could have defined *dst* and *id* as parameters of *send*, but the implicit existential quantification, although theoretically acceptable, can be costly performance-wise (as *succ* is a function, this is significant for the *id* argument only). We also specify that the event `modifies` the *toSend* relation for specific pairs of a node and an identifier, only if these satisfy a condition saying that the ID is in the sender's mailbox (or corresponds to the sender's ID) and if the node is the sender's successor (the body of the event says what happens in that case).

2.3 Commands

A `check` declares a command to verify whether a property holds. To do so, a command uses a certain tactic (`TEA`, `TFC`, `TTC`), as well as additional parameters in the case of `TFC` and `TTC` (these are presented in Sect. 5 and 6, respectively). The purpose of this article is precisely to present these transformations. We notice that a command may also be associated with additional, specific axioms in an `assuming` section (in the example, this section contains a fairness property, necessary to prove the liveness property).

3 Background on FOLTL

3.1 Syntax and Semantics of FOLTL

The basic vocabulary of MSFOLTL (that we simply call FOLTL in the following) is defined out of a signature $\Sigma = (\mathcal{S}, Const, \mathcal{R})$ where $\mathcal{S}$ is a set of sorts, *Const* is the set of (sorted) constant symbols and $\mathcal{R} = (\mathcal{R}_{\vec{s}})_{\vec{s} \in \mathcal{S}^\star})$ is a family of sets of *relation symbols*, with $\mathcal{R}_{\vec{s}}$ the set of relation symbols over tuples of sort $\vec{s}$.

Definition 1 (Formulas). *Given a signature $\Sigma = (\mathcal{S}, Const, \mathcal{R})$ and a set of variables $\mathcal{V}$,* FOLTL$_=$ formulas *over Σ and $\mathcal{V}$ are defined inductively by the following grammar:*

$$\psi ::= \ r(t_1, \ldots, t_n) \mid t_1 = t_2 \mid \neg\psi \mid \psi \vee \psi \mid \mathbf{X}\psi \mid \mathbf{F}\psi \mid \forall x : s \cdot \psi \mid \exists x : s \cdot \psi$$

where $x \in \mathcal{V}_s$, $r \in \mathcal{R}_{s_1,\ldots s_n}$ and $t_i \in \mathcal{V}_{s_i} \cup Const_{s_i}$ for each i, with $\mathcal{V}_s$ (resp. $Const_s$) the set of variables (resp. constants) of sort s.

X and **F** stand for the "next" and "eventually" connectives. Usually FOLTL includes the **U** connectives, however it is not required in this paper. We also define "always" as $\mathbf{G}\psi = \neg\mathbf{F}(\neg\psi)$. Similarly, classical propositional connectives $\wedge$, $\Rightarrow$ and $\Leftrightarrow$ are defined in the natural way. Additionally:

- We write $\psi[x]$ for a formula ψ having x as a free variable.
- We write $\mathrm{FV}(\phi)$ for the set of *free variables* of a formula, defined in the obvious way. A formula ϕ is said to be *closed* if $\mathrm{FV}(\phi) = \varnothing$.
- Classically, a formula is in *negation normal form* (NNF) if negations only appear in front of relation symbols.
- If **C** is a subset of $\{\mathbf{X}, \mathbf{F}, \mathbf{G}\}$ then we denote by FOLTL$_=(C)$ (resp. FOLTL(C)) the set of FOLTL$_=$ formulas (resp. FOLTL formulas without equality) in NNF containing only temporal operators from **C**.
- A formula l is called literal if $l = r(t_1, \ldots, t_n)$ or $l = \neg r(t_1, \ldots, t_n)$ where $x \in \mathcal{V}$, $r \in \mathcal{R}_n$ and $t_i \in Const \cup \mathcal{V}$ for each i.

We now introduce the semantics of FOLTL$_=$. In the interpretation structures defined below, the interpretation of relations *varies* over time while that of function symbols *does not*.

Definition 2 (Interpretation Structure). *Given a signature $\Sigma = (\mathcal{S}, Const, \mathcal{R})$, an* (interpretation) structure $\mathcal{M}$ (over Σ) *is a triple $((D_s)_{s \in \mathcal{S}}, \sigma, \rho)$ where:*

- *$D = (D_s)_{s \in \mathcal{S}}$ is a family of pairwise-disjoint nonempty sets and each D_s is the domain of the sort s.*
- *σ maps each constant $c \in Const_s$ to an domain element $\sigma(c) \in D_s$.*
- *ρ maps any pair $(i, r) \in \mathbb{N} \times \mathcal{R}_{s_1 \ldots s_n}$ of instant and relation to the set $\rho(i, r) \subseteq D_{s_1} \times \ldots \times D_{s_n}$ of tuples satisfying r at instant i.*

Definition 3 (Assignment). *An* assignment $\mathcal{C}$ *in domains $(D_s)_{s \in \mathcal{S}}$ for variables in $\mathcal{V}$ is a map $\mathcal{V} \to D$. We write $\mathcal{C}[x \mapsto d]$ the assignment defined as $\mathcal{C}[x \mapsto d](x) = d$ and $\mathcal{C}[x \mapsto d](y) = \mathcal{C}(y)$ if $y \neq x$. The* extension *of $\mathcal{C}$ to* terms, *also written $\mathcal{C}$, is defined in the obvious way.*

Definition 4 (Satisfaction). *Given a structure $\mathcal{M} = (D, \sigma, \rho)$ and an assignment $\mathcal{C}$, the* satisfaction *relation $\vDash$ is defined by induction on formulas, for any $i \in \mathbb{N}$, as follows:*

- *$\mathcal{M}, i, \mathcal{C} \vDash t_1 = t_2$ iff $\mathcal{C}(t_1) = \mathcal{C}(t_2)$;*

- $\mathcal{M}, i, \mathcal{C} \models r(t_1, \ldots, t_n)$ *iff* $(\mathcal{C}(t_1), \ldots, \mathcal{C}(t_n)) \in \rho_i(r)$*;*
- $\mathcal{M}, i, \mathcal{C} \models \neg\phi$ *iff* $\mathcal{M}, i, \mathcal{C} \not\models \phi$*;*
- $\mathcal{M}, i, \mathcal{C} \models \phi_1 \vee \phi_2$ *iff* $\mathcal{M}, i, \mathcal{C} \models \phi_1$ *or* $\mathcal{M}, i, \mathcal{C} \models \phi_2$*;*
- $\mathcal{M}, i, \mathcal{C} \models \mathbf{X}\phi$ *iff* $\mathcal{M}, i+1, \mathcal{C} \models \phi$*;*
- $\mathcal{M}, i, \mathcal{C} \models \mathbf{F}\phi$ *iff there exists* $k \in \mathbb{N}$ *s.t.* $\mathcal{M}, i+k, \mathcal{C} \models \phi$*;*
- $\mathcal{M}, i, \mathcal{C} \models \exists y : s \cdot \phi$ *iff there exists* $d \in D_s$ *s.t.* $\mathcal{M}, i, \mathcal{C}[y \mapsto d] \models \phi$*;*
- $\mathcal{M}, i, \mathcal{C} \models \forall x : s \cdot \phi$ *iff for every* $d \in D_s$*, we have* $\mathcal{M}, i, \mathcal{C}[x \mapsto d] \models \phi$*.*

Given a closed formula ϕ*, we write* $\mathcal{M}, k \models \phi$ *if* $\mathcal{M}, k, [] \models \phi$*, where* $[]$ *is the empty assignment. Then* $\mathcal{M}od(\phi)$ *denotes the set of structures* $\mathcal{M}$ *such that* $\mathcal{M}, 0 \models \phi$*.*

Definition 5 (Reflexive-Transitive Closure[3]). *We write* $\text{FOLTL}^*_=$ *for the enrichment of* $\text{FOLTL}_=$ *with a reflexive-transitive closure connective. Then for any sort* $s \in \mathcal{S}$ *and any binary relation symbol* $r \in \mathcal{R}_{s,s}$*, the language of* $\text{FOLTL}^*_=$ *is augmented with a fresh binary relation symbol :* $r^* \in \mathcal{R}_{s,s}$*, and we have:*

$\mathcal{M}, i, \mathcal{C} \models r^*(t_1, t_2)$ *iff* $\mathcal{M}, i, \mathcal{C} \models t_1 = t_2$ *or there exists* $n \in \mathbb{N}$ *s.t.* $\mathcal{M}, i, \mathcal{C} \models \exists x_0, \ldots, x_n \cdot t_1 = x_0 \wedge t_2 = x_n \wedge (\bigwedge_{0 \leqslant i \leqslant n-1} r(x_i, x_{i+1}))$*.*

Let ϕ, ϕ' be two $\text{FOLTL}^*_=$ formulas. If for any structure $\mathcal{M}$ and any assignment $\mathcal{C}$, we have $\mathcal{M}, 0, \mathcal{C} \models \phi$ iff $\mathcal{M}, 0, \mathcal{C} \models \phi'$ then we say that ϕ and ϕ' are logically equivalent, written $\phi \equiv \phi'$.

3.2 Bounded Domain Property

In this section we introduce the Bounded Domain Property (BDP) and present two fragments of FOLTL that enjoy the BDP. These fragments play an important role in the verification procedures presented in this article.

Definition 6 (Bounded Domain Property). *A fragment Frag of* FOLTL *enjoys the* bounded domain property *(BDP) if given* $\phi \in$ *Frag,* ϕ *is not satisfiable, or there is a domain-finite structure* $\mathcal{M}$ *s.t.* $\mathcal{M}, 0 \models \phi$ *whose the domain size is computable from* ϕ*. Additionally, BDP implies decidability.*

We now present the two fragments that are used in this paper. Both fragments are included in a larger fragment for which the BDP is established in [24].

Definition 7 (*LTR* fragment). *A formula* ϕ *of* $\text{FOLTL}_=$ *is said to belong to the (multisorted) Linear-Temporal Reasoning (LTR) fragment if* ϕ *is in NNF and existential quantifiers only appear in the head of* ϕ*.*

Theorem 1 ([16,24]). *Any formula* $\phi \in$ *LTR (even with equality) enjoys the BDP. The bound of verification for each sort is the sum of the numbers of existential quantifiers and constant symbols over this sort.*

[3] It is possible to fully axiomatize the transitive closure in pure FOLTL, however since it does not fit into the scope of this paper such an axiomatization is not presented here and we simply extends FOLTL with the classical definition of transitive closure.

Definition 8. *An* FOLTL *formula ψ is in* $\mathrm{FOLTL}(\exists\uparrow, \forall\downarrow)$ *if $\psi = \exists y_1 : s_1 \ldots y_n : s_n \cdot \theta[y_1, \ldots, y_n]$, where θ has the following syntax: $\theta ::= \ell \mid \alpha \mid \theta \vee \theta \mid \theta \wedge \theta \mid \mathbf{X}\theta \mid \mathbf{G}\theta \mid \mathbf{F}\theta$, where α is an FO formula in NNF without any existential quantifier and ℓ is a literal.*

Definition 9. $\mathrm{FOLTL}(\mathbf{X}, \mathbf{F}, \forall\downarrow)$ *is defined by the following grammar: $\phi ::= \ell \mid \alpha \mid \phi \vee \phi \mid \phi \wedge \phi \mid \mathbf{X}\phi \mid \mathbf{F}\phi \mid \exists y : s \cdot \phi$, with α an FO formula in NNF without any existential quantifier, ℓ a literal and $y \in \mathcal{V}$.*

Definition 10 (*Geneva* fragment). *The Geneva fragment of* FOLTL *consists of formulas $\psi \wedge \mathbf{G}(\phi)$ s.t. ϕ is a closed formula of* $\mathrm{FOLTL}(\mathbf{X}, \mathbf{F}, \forall\downarrow)$ *and ψ is a closed formula of* $\mathrm{FOLTL}(\exists\uparrow, \forall\downarrow)$.

Definition 11. *Given a formula $\phi \in$* $\mathrm{FOLTL}(\mathbf{X}, \mathbf{F})$ *in NNF, we define its* stride *K_ϕ as the maximal number of nested* $\mathbf{X}$ *connectives. Formally :*

$$K_\ell = K_{\mathbf{F}\phi} = 0 \ (\textit{if } \ell \textit{ is a literal}) \qquad K_{\mathbf{X}\phi} = K_\phi + 1$$
$$K_{\forall x \cdot \phi} = K_{\exists x \cdot \phi} = K_\phi \qquad K_{\phi_1 \wedge \phi_2} = K_{\phi_1 \vee \phi_2} = \max(K_{\phi_1}, K_{\phi_2})$$

Theorem 2 ([24]). *The Geneva fragment enjoys the FDP. If $\psi \wedge \mathbf{G}(\phi)$ is a satisfiable formula in this fragment, for each sort s the (exact) bound on the domain size is: $|Const_s| + (K_\phi + 1) \times |\mathcal{V}_s|$.*

3.3 Semantics of Cervino

In this section, we define the semantics of a Cervino machine as an $\mathrm{FOLTL}^*_=$ formula. Notice first that, in Cervino, the next instant is referred using the prime symbol, applied to relations only: this translates to an FOLTL sub-formula using the $\mathbf{X}$ connective, after application of the semantics.

Now, a frame condition is defined as a formula that specifies that a certain relation will not change (between the instant before and after the event occuring) for tuples satisfying some constraints.

Definition 12 (Frame condition). *We define a frame condition as a formula expressing that, under some hypotheses, a certain relation does not change along a transition. Given the the tuple $(r, \vec{x}, \psi)$ where $r \in \mathcal{R}_{\vec{s}}$, $\vec{x} \in \mathcal{V}^{|\vec{s}|}$, ψ is a Boolean formula, where variables in $\vec{x}$ may appear free, we define the frame condition* $\mathit{unchanged}[r, \vec{x}, \psi]$ *as the formula $\forall \vec{x} : \vec{s} \cdot \psi \Rightarrow (r(\vec{x}) \Leftrightarrow \mathbf{X} r(\vec{x}))$.*

Definition 13 (Semantics of an event). *Let ev be an event of a Cervino machine declared as follows:* ***event*** *ev[$\vec{y}$: $\vec{s}$] modif {τ}, with modif =* ***modifies*** *q_1* ***at*** *$\{(\vec{x}_1) \cdot \psi_1\}, \ldots, q_j$* ***at*** *$\{(\vec{x}_j) \cdot \psi_j\}$, where the free variables in each ψ_k are included in $\vec{x_k}, \vec{y}$. Its semantics is defined as $[\![ev]\!] = \exists \vec{y} : \vec{s} \cdot (\tau \wedge [\![modif]\!])$, where*

$$[\![modif]\!] = \Big(\bigwedge_{r \in \mathcal{R} \setminus \{q_1, \ldots, q_j\}} \mathit{unchanged}[r, \vec{x}, \top]\Big) \wedge \Big(\bigwedge_{1 \le k \le j} \mathit{unchanged}[q_k, \vec{x_k}, \neg\psi_k]\Big)$$

where each list $\vec{x}$ of variables have sorts corresponding to the profile of r.

For any binary relation r that enables ***btw***[r], the ternary relation ***btw***[r] stating that there exists an acyclic path between two elements passing through a third element is axiomatized in FO following [18].

Definition 14 (Semantics of between). *Given a binary relation symbol r, the semantics of **btw**[r] is given by adding axioms of transitivity, antisymmetry, partial totality, partial reflexivity, cycle maximality, transitivity of reachability, path consistency, taken from [18] in addition to the following axiom:*

$$\forall x, y : s \cdot \Big[r(x,y) \Leftrightarrow \Big(\boldsymbol{btw}[r](x,y,y) \wedge (\forall z : s \cdot \boldsymbol{btw}[r](x,z,z) \Rightarrow \boldsymbol{btw}[r](x,y,z))\Big)\Big] \tag{S}$$

*The property (TC) relating **btw**[r] and r^* can be deduced from the axioms provided that the domain of s is finite.*

$$\forall x, y : s \cdot \big[\boldsymbol{btw}[r](x,y,y) \Leftrightarrow r^*(x,y)\big] \tag{TC}$$

Then, calling BTW *the conjunction of all between axioms,* $[\![\boldsymbol{btw}[r]]\!] = \mathbf{G}\textsc{btw}$.

Definition 15 (Semantics of Cervino). *Let Mch be a Cervino machine with axioms $\psi_1, \dots, \psi_n$, events $ev_1, \dots, ev_m$ and such that the relations enabling **btw** are $r_1, \dots, r_l$. Then its semantics is given by the following* FOLTL* *formula:*

$$[\![Mch]\!] = \phi_0 \wedge (\mathbf{G}\phi_{tr}) \wedge \phi_{\boldsymbol{btw}}$$

where $\phi_0 = \bigwedge_{i=1}^{n} \psi_i$, $\phi_{tr} = \bigvee_{i=1}^{m} [\![ev_i]\!]$ *and* $\phi_{\boldsymbol{btw}} = \bigwedge_{1 \leqslant i \leqslant l} [\![\boldsymbol{btw}[r_i]]\!]$

The semantics of a Cervino machine is then an $\mathrm{FOLTL}^*_=$ formula describing the set of its traces. But, since we aim at verifying systems, we are not only interested in the set of traces but also in the set of counterexamples of a property. This set is also described by an $\mathrm{FOLTL}^*_=$ formula which is the conjunction of the semantics of the machine and the negation of the property we aim to check.

Definition 16 (Counterexamples). *If Mch is a Cervino machine and ϕ is an* $\mathrm{FOLTL}^*_=$ *formula. Then we define* $[\![Mch]\!]_\phi = [\![Mch]\!] \wedge [\![\neg\phi]\!]$

4 Basic Transformations

In this section, we present basic transformations used to build the more complex TFC and TTC tactics (respectively presented in Sect. 6 and 7). These transformations are used to map (the semantics of) a system specification into a more general Geneva formula.

4.1 Transforming Equality

Equality is replaced[4] by a dynamic congruence relation $\equiv_s$, for every sort s. The signature is therefore extended with these fresh $\equiv_s$ relations.

[4] In practice, we ensure that the semantics of the `modifies` section, which uses equality, is also affected by this transformation.

Definition 17 (Equality transformation). *Given a fresh binary relation* $\equiv_s$ *for every sort* s *of a formula, the transformation of equality is defined recursively:*

- $Abs_=(t_1 = t_2) = t_1 \equiv_s t_2$ *if the sort of* t_1 *(and necessarily of* t_2*) is* s
- $Abs_=(\ell) = \ell$
- *(the rest is just a recursive walk on formulas)*

Furthermore, the following set $\boldsymbol{Eq}_\equiv$ *of axioms is added to the whole specification:*

- *for any sort* s*:* $\mathbf{G}\forall x : s \cdot x \equiv_s x$
- *for any sort* s*:* $\mathbf{G}\forall x : s, y : s \cdot x \equiv_s y \Rightarrow y \equiv_s x$
- *for any sort* s*:* $\mathbf{G}\forall x : s, y : s, z : s \cdot x \equiv_s y \wedge y \equiv_s z \Rightarrow x \equiv_s z$
- *for any relation* r *and adequate sorts* $\vec{s}$ *conforming to the profile of* r*:* $\mathbf{G}\,\forall \vec{x} : \vec{s}, \vec{y} : \vec{s} \cdot (x_1 \equiv_{s_1} y_1 \wedge \ldots \wedge x_n \equiv_{s_n} y_n) \Rightarrow (r(\vec{x}) \Leftrightarrow r(\vec{y}))$

Lemma 1. *Given an* FOLTL$_=$ *formula* ϕ*, if* ϕ *is satisfiable then* $Abs_=(\phi)$ *is satisfiable (and does contain* $=$ *anymore).*

Proof. Proof validated in Coq. It is easy to see that equality is a particular case of the equivalence relation introduced by this transformation. □

4.2 Restricted Skolemization

The following transformation corresponds to a form of Skolemization meant to create only new constants symbols. Its main purpose is to introduce constants that can then be used by instantiation (Sect. 4.3). Existentially-quantified variables can be substituted by fresh constants, except when under a **G** connective.

Definition 18 (Skolemization). *Skolemization is defined by the following operation (all fresh constant symbols are added to the signature):*

- $Abs_\exists(t_1 = t_2) = t_1 = t_2$ *and* $Abs_\exists(\ell) = \ell$
- $Abs_\exists(\mathbf{G}\phi) = \mathbf{G}\phi$
- $Abs_\exists(\forall x : s \cdot \phi) = \forall x : s \cdot \phi$
- $Abs_\exists(\exists y : s \cdot \phi) = Abs_\exists(\phi[y \mapsto c])$ *where* c *is a fresh constant symbol*
- *(the rest is just a recursive walk on formulas)*

Lemma 2. *Given an* FOLTL$_=$ *formula* ϕ*, then* $Abs_\exists(\phi)$ *and* ϕ *are equisatisfiable.*

Proof. Proof validated in Coq. Corresponds to a usual Skolemization procedure. □

4.3 Instantiation

One of the main limitations of the Geneva fragment is the prohibition of temporal operators under universal quantifiers. The solution we propose to this problem is to *finitely* instantiate such universal quantifiers. The following transformation formalizes this idea: all universal quantifiers over temporal formulas are replaced by a conjunction over the set of constants and existentially-bound variables.

Definition 19 (Forall instantiation). *Given a set $\mathcal{I}$ of constant and variable symbols, we define the transformation of universal quantifiers as follows:*

- $Abs_{\forall,\mathcal{I}}(t_1 = t_2) = t_1 = t_2$ *and* $Abs_{\forall,\mathcal{I}}(\ell) = \ell$
- $Abs_{\forall,\mathcal{I}}(\exists y : s \cdot \phi) = \exists y : s \cdot Abs_{\forall,\mathcal{I}\cup\{y\}}(\phi)$
- *if* $\phi \in \mathrm{FO}$ *(ϕ does not contain temporal connectives) then* $Abs_{\forall,\mathcal{I}}(\forall x : s \cdot \phi) = \forall x : s \cdot \phi$*, otherwise* $Abs_{\forall,\mathcal{I}}(\forall x : s \cdot \phi) = \bigwedge_{c \in \mathcal{I}_s} Abs_{\forall,\mathcal{I}}(\phi[x \mapsto c])$ *(where $\mathcal{I}_s$ is the set of terms in $\mathcal{I}$ of sort s)*
- *(the rest is just a recursive walk on formulas)*

Remark 1. There is no need to transform a universal quantifier if all temporal operators in its scope permute with it, for instance: $\forall x \cdot \mathbf{G}P$ is equivalent to $\mathbf{G}(\forall x \cdot P)$ and $\forall x \cdot (\mathbf{X}P) \Rightarrow (\mathbf{X}Q)$ is equivalent to $\mathbf{X}(\forall x \cdot P \Rightarrow Q)$.

Lemma 3. *Given an* $\mathrm{FOLTL}_=$ *formula ϕ, if ϕ is satisfiable and $\mathcal{I} \subseteq Const$ then $Abs_{\forall,\mathcal{I}}(\phi)$ is satisfiable.*

Proof. Proof validated in Coq. This operation consists in instantiating universal operators, thus preserving satisfiability. □

4.4 Addressing Transitive Closure and the Between Relation

Since we target fragments of FOLTL (without transitive closure), we define the transformation $Abs_*()$, which leaves a formula unchanged except it *uninterprets* the operator *, *i.e.*, $Abs_*(\phi)$ returns ϕ where every occurrence of r^* is considered as a new relation symbol, unrelated with r.

Besides, the between relation axioms does not fit into Geneva or LTR, so we define their abstract semantics as follows.

Definition 20 (Transformation of between axioms). *Given a binary relation symbol r, we define* $(\!| \boldsymbol{btw[r]} |\!) = \mathbf{G}\,\mathrm{BTW}$ *where* BTW *is the conjunction of the axioms from Definition 14, except that*

- *the axiom S is replaced by the axiom (AS) (in order to prevent existential quantifier in the scope of a universal one)*
- *and the property (TC) relating r^* and **btw[r]** is now considered as an axiom (since r^* has no semantics in the targeted* FOLTL *fragments)*

$$\forall x, y : s \cdot \Big[r(x, y) \Rightarrow \Big(\boldsymbol{btw[r]}(x, y, y) \wedge (\forall z : s \cdot \boldsymbol{btw[r]}(x, z, z) \Rightarrow \boldsymbol{btw[r]}(x, y, z))\Big)\Big] \quad \text{(AS)}$$

$$\forall x, y : s \cdot \big[\boldsymbol{btw[r]}(x, y, y) \Leftrightarrow r^*(x, y)\big] \quad \text{(TC)}$$

4.5 Geneva Transformation

The basic transformations introduced above are mainly used together, in a specific order.

Definition 21 (*Geneva* Transformation). *We define:*

$$Abs_{Gen}(\phi) = Abs_{*}(Abs_{\forall, Const}(Abs_{\exists}(\boldsymbol{Eq}_{\equiv} \wedge Abs_{=}(\phi))))$$

Theorem 3. *Given* $\psi \in \text{FOLTL}_{\vec{y_1}}(\forall)$ *and* $\phi \in \text{FOLTL}_{\vec{y_1} \cup \vec{y_2}}(\mathbf{X}, \mathbf{F}, \forall)$ *then* $Abs_{Gen}(\exists \vec{y_1} : \vec{s_1} \cdot (\psi \wedge \mathbf{G}(\exists \vec{y_2} : \vec{s_2} \cdot \phi)))$ *belongs to the* Geneva *fragment.*

Proof. Recall the conditions to belong to Geneva: (1) no **G** operator in the scope of an existential quantifier that is itself under an **G** connective; (2) no existential quantifier in the scope of a universal quantifier; (3) no equality; (4) no temporal quantifier in the scope of universal quantifiers; and (5) no transitive closure. Given ψ, ϕ satisfying the given hypotheses, let us write $\alpha = \exists \vec{y_1} : \vec{s_1} \cdot (\psi \wedge \mathbf{G}(\exists \vec{y_2} : \vec{s_2} \cdot \phi))$. Then, in α, existential quantifiers appear either at the head of the formula or under an **G** operator over the ϕ formula. Since ϕ contains no other temporal connectives than **X** and **F**, condition (1) is met. Condition (2) is met as all existential quantifiers appear before universal quantifiers. $Abs_{=}(.)$ ensures that equality is not used in the final formula, thus ensuring condition (3). $Abs_{\forall, Const}(.)$ instantiates all universal quantifiers that contain temporal connectives in their scope (we assume that if such operator could have been swapped with an universal quantifier, it has been done beforehand), which ensures condition (4). Finally $Abs_{*}(.)$ erases the reflexive transitive closure, ensuring condition (5). Since it is obvious that none of the transformations can introduce formulas breaking any of the conditions, we conclude that $Abs_{Gen}(\alpha)$ belongs to Geneva. □

5 TEA: Transforming Existential Quantifiers

We now present the fully-automatic TEA transformation. It starts with the observation that the formula specifying events (see Definition 13) is of the shape $\mathbf{G}\exists \vec{x} \cdot \bigvee_i ev_i(\vec{x})$, that is, in every state, at least an event is fired. The gist of the TEA transformation is then twofold: (1) we replace these existential quantifiers by *universal* ones; (2) for every such existential quantifier, we add a fresh relation $\mathbb{E}$, which holds only for the constant semantically associated to this quantifier.

The whole resulting abstract specification lies in the LTR fragment, which enjoys the BDP (Theorem 1). The formula specifying events is however more general than the original one, because it allows more transitions to happen. The abstract system may thus violate a property holding on the original specification. But it is now decidable to check whether the property holds in the abstract system and, if so, this entails that it also holds in the original system.

Before presenting the transformation, notice that, in the following, we consider *event formulas*, that is *primed* $\text{FO}_{=}$ formulas of the shape $\phi = \exists y_1 : s_{y_1}, \ldots, y_n : s_{y_n} \cdot \forall x_1 : s_{x_1}, \ldots, x_m : s_{x_m} \cdot \psi$, where ψ is in NNF and does not contain any first-order quantifiers. These formulas naturally arise when putting the semantics of events in prenex normal form. We also suppose we have a supply of fresh relation symbols, written $\mathbb{E}_i$ (one for every y_i, $1 \leqslant i \leqslant n$).

To devise the transformation and prove its soundness, we first introduce a formula specifying that the $\mathbb{E}$ relations are functional. This schema appears in the final abstract specification.

Definition 22 (Functional $\mathbb{E}$ relations). *Given an event formula $\phi = \exists y_1 : s_{y_1}, \ldots, y_n : s_{y_n} \cdot \forall x_1 : s_{x_1}, \ldots, x_m : s_{x_m} \cdot \psi$, we define the* functional formula *based on ϕ as:* $\mathrm{Ax}^{\mathbb{E}}(\phi) = \mathbf{G}\Big(\bigwedge_{i=1}^{n} \forall z_1, z_2 : s_{y_i} \cdot (\mathbb{E}_i(z_1) \wedge \mathbb{E}_i(z_2)) \Rightarrow z_1 = z_2\Big)$ *where $\mathbb{E}_1, \ldots, \mathbb{E}_n$ are fresh unary relation symbols.*

As we introduce these $\mathbb{E}$ relations, we also define an enrichment of the event formula accounting for the extended signature. This new formula appears as a link between the two lemmas entailing soundness.

Definition 23 (Enriched event formula). *Given an event formula $\phi = \exists y_1 : s_{y_1}, \ldots, y_n : s_{y_n} \cdot \forall x_1 : s_{x_1}, \ldots, x_m : s_{x_m} \cdot \psi$, we define the* enriched event formula *based on ϕ as:*

$$\overline{\phi} = \mathrm{Ax}^{\mathbb{E}}(\phi) \wedge \Big[\exists y_1 : s_{y_1}, \ldots, y_n : s_{y_n} \cdot \Big(\bigwedge_{i=1}^{n} \mathbb{E}_i(y_i) \wedge \forall x_1 : s_{x_1}, \ldots, x_m : s_{x_m} \cdot \psi\Big)\Big]$$

where $\mathbb{E}_1, \ldots, \mathbb{E}_n$ are fresh unary relation symbols.

We now present the essential part of the transformation, transforming an event formula ϕ into a purely universal one $\mathbb{U}(\!|\phi|\!)$, more general than $\overline{\phi}$. In other words, $\mathbb{U}(\!|\phi|\!)$ allows more transitions than ϕ if we ignore the specification of $\mathbb{E}_1, \ldots \mathbb{E}_n$. To do that, for any variable y whose corresponding fresh relation is $\mathbb{E}$, we proceed with the following steps. First: equality between y and another variable is replaced with the relation $\mathbb{E}$ applied to the latter; and any other literal ℓ containing y is replaced by $\mathbb{E}(y) \Rightarrow \ell$. Once these transformation are done, it is possible to replace existential quantification over y by a universal quantification.

Definition 24 (Transformation). *Given an event formula ϕ of shape $\exists y_1 : s_{y_1}, \ldots, y_n : s_{y_n} \cdot \forall x_1 : s_{x_1}, \ldots, x_m : s_{x_m} \cdot \psi$, we define the* (TEA) transformation function *on ϕ as:*

$$\mathbb{U}(\!|\phi|\!) = \forall y_1 : s_{y_1}, \ldots, y_n : s_{y_n} \cdot \mathbb{U}_{\vec{y}}(\!|\forall x_1 : s_{x_1}, \ldots, x_m : s_{x_m} \cdot \psi|\!)$$

where $\vec{y} = \{y_1, \ldots, y_n\}$ and where $\mathbb{E}_1, \ldots, \mathbb{E}_n$ are fresh relation symbols (one for every $y \in \vec{y}$); with $\mathbb{U}_{\vec{y}}(\!|\psi|\!)$ defined recursively as follows:

- $\mathbb{U}_{\vec{y}}(\!|y_i = y_j|\!) = (\mathbb{E}_i(y_i) \Rightarrow \mathbb{E}_j(y_i)) \wedge (\mathbb{E}_j(y_j) \Rightarrow \mathbb{E}_i(y_j)) = (\neg\mathbb{E}_i(y_i) \vee \mathbb{E}_j(y_i)) \wedge (\neg\mathbb{E}_j(y_j) \vee \mathbb{E}_i(y_j))$
- $\mathbb{U}_{\vec{y}}(\!|y_i \neq y_j|\!) = (\mathbb{E}_i(y_i) \Rightarrow \neg\mathbb{E}_j(y_i)) \wedge (\mathbb{E}_j(y_j) \Rightarrow \neg\mathbb{E}_i(y_j)) = (\neg\mathbb{E}_i(y_i) \vee \neg\mathbb{E}_j(y_i)) \wedge (\neg\mathbb{E}_j(y_j) \vee \neg\mathbb{E}_i(y_j))$
- $\mathbb{U}_{\vec{y}}(\!|y_i = d|\!) = \mathbb{U}_{\vec{y}}(\!|d = y_i|\!) = \mathbb{E}_i(d)$ *where $d \notin \vec{y}$ (d is either a constant or a variable in $\vec{x}$)*
- $\mathbb{U}_{\vec{y}}(\!|y_i \neq d|\!) = \mathbb{U}_{\vec{y}}(\!|d \neq y_i|\!) = \neg\mathbb{E}_i(d)$ *where $d \notin \vec{y}$ (d is either a constant or a variable in $\vec{x}$)*

– $\mathbb{U}_{\vec{y}}(\!|\ell|\!) = (\bigwedge_{k=1}^{i} \mathbb{E}_{a_k}(y_{a_k})) \Rightarrow \ell = (\bigvee_{k=1}^{i} \neg\mathbb{E}_{a_k}(y_{a_k})) \vee \ell$ *where* ℓ *is a (possibly primed) literal and* $\{y_{a_1}, \ldots y_{a_i}\} = \mathrm{FV}(\ell) \cap \vec{y}$
– *(the rest is just a recursive walk on formulas)*

Example 1. Consider the following event formula, stating that there is an event making R true in the next state for a variable y (other variables remain unchanged w.r.t. R): $\phi = \exists y : A \cdot R'(y) \wedge (\forall x : A \cdot x \neq y \Rightarrow (R(x) \Leftrightarrow R'(x)))$ that is, in prenex form: $\exists y : A \cdot \forall x : A \cdot R'(y) \wedge (x = y \vee (\neg R(x) \wedge \neg R'(x)) \vee (R(x) \wedge R'(x)))$. Then there is only one fresh $\mathbb{E}$ relation, and $\mathbb{U}(\!|\phi|\!)$ is:

$$\forall y, x : A \cdot (\neg\mathbb{E}(y) \vee R'(y)) \wedge \big(\mathbb{E}(x) \vee (\neg R(x) \wedge \neg R'(x)) \vee (R(x) \wedge R'(x))\big)$$

Now, the following lemma states that every model of the enriched event formula is also a model for the transformed event formula.

Lemma 4. *Given an event formula* $\phi = \exists y_1 : s_{y_1}, \ldots, y_n : s_{y_n} \cdot \forall x_1 : s_{x_1}, \ldots, x_m : s_{x_m} \cdot \psi$, *we have* $\overline{\phi} \vDash \mathbb{U}_{\vec{y}}(\!|\phi|\!)$.

Proof. Proof validated in Coq.

Lemma 5 applies to a formula representing a whole specification: if such a specification is satisfiable, then a certain transformed version of it is satisfiable too.

Lemma 5. *Let* θ *be an* FOLTL$_=$ *formula, and* ϕ *be an event formula on the same signature. Then if* $\theta \wedge \mathbf{G}\phi$ *is satisfiable,* $\theta \wedge \mathbf{G}(\mathbb{U}_{\vec{y}}(\!|\phi|\!)) \wedge \mathrm{Ax}^{\mathbb{E}}(\phi)$ *is also satisfiable.*

Proof. Proof validated in Coq.

Definition 25 (Abstract semantics). *Given a Cervino machine Mch such that the relations enabling* ***btw*** *are* $r_1, \ldots, r_l$, *we define* $\mathbb{U}(\!|Mch|\!) = \phi_0 \wedge \mathbf{G}\mathbb{U}(\!|\phi_{tr}|\!) \wedge \phi_{\boldsymbol{btw}}$, *where* ϕ_0 *and* ϕ_{tr} *are defined as in Definition 15 and* $\phi_{\boldsymbol{btw}} = \bigwedge_{1 \leqslant i \leqslant l} (\!|\boldsymbol{btw}[r_l]|\!)$. *Also, given an* FOLTL$_=$ *formula* ϕ, *we define* $\mathbb{U}_\phi(\!|Mch|\!) = Abs_*(\mathbb{U}(\!|Mch|\!) \wedge \neg\phi)$.

Theorem 4 (Soundness). *If* $[\![Mch]\!]_\phi$ *is satisfiable, then* $\mathbb{U}_\phi(\!|Mch|\!)$ *is also satisfiable.*

Proof. This is a direct application of Lemma 5.

Theorem 5. *Given a Cervino machine Mch such that* ϕ_0 *and* ϕ_{tr} *are defined as in Definition 15, if* $\phi_0, \phi \in \mathit{LTR}$ *then* $\mathbb{U}_\phi(\!|Mch|\!) \in \mathit{LTR}$.

Proof. Directly follows from the definition of $\mathbb{U}(\!|.|\!)$.

6 TFC: Transforming Frame Conditions

The TEA transformation has the advantage of being fully automatic but it can be inconclusive in a number of cases. For instance, the verification of a distributed system involving strong interactions between its components, which induces events with two or more parameters, is likely to be inconclusive using TEA. This is because the universal quantifiers that are introduced by TEA are abstracting these interactions (which are naturally expressed with existential quantifiers) in a too drastic way.

In this section, we present another transformation, called TFC, which overcomes these limitations but requires some intervention from the specifier.

Instead of targeting the LTR fragment, we now target the Geneva one, which allows for existential quantifiers in the scope of $\mathbf{G}$, but forbids temporal formulas in the scope of a universal quantifier. As a consequence, frame conditions, which are typically of shape $\forall x : s \cdot \varphi_{cond} \Rightarrow (r(x) \Leftrightarrow \mathbf{X}r(x))$, are not expressible in Geneva. In order to fit into it, such universal quantifiers are instantiated over constants (see $Abs_{\forall,Const}(\cdot)$ defined in Sect. 4.3). But then a large part of the information included in the frame conditions is lost. Therefore, we associate some particular kind of invariant properties, called stability axioms, with each event, as a finer transformation of frame conditions. Intuitively, a stability axiom is a pure FO formula that is preserved by an event. Since it is expressed in pure FO, the preservation of a stability axiom is then expressible in Geneva.

Definition 26 (Stability Axiom). *Given a set of frame conditions $\mathcal{C}$, an* FO *formula ϕ is a* stability axiom *for $\mathcal{C}$ if $\mathcal{C} \vDash \phi \Rightarrow \mathbf{X}\phi$.*

$\mathrm{St}_{\mathcal{C}}$ *denotes the set of stability axioms for $\mathcal{C}$.*

The specification of stability axioms is a creative step, but it can be eased with the help of a syntactic condition, which is sufficient to be a stability axiom. The idea is that a formula of the following shape is necessarily a stability axiom: $\varphi_{\mathrm{hyp}} \Rightarrow \varphi$, were φ_{hyp} corresponds to the guard of a frame condition that leaves a relation r unchanged, and φ only refers to the relation r.

Example 2. In order to illustrate the use of stability axioms, let us consider the leader election distributed system, introduced in Sect. 2. Since TEA does not succeed in proving the safety property, we can try TFC with the following stability axiom for event send:

$$\forall\ \mathrm{x,y:\ Node} \cdot \mathrm{!succ(src,x)} \Rightarrow (\mathrm{!toSend(x,\ y)} \vee\ (\mathrm{x} \neq \mathrm{lmax} \wedge \mathbf{btw}[\mathrm{succ}](\mathrm{x,\ lmax,\ y})))$$

This axiom expresses that if a node x different from the successor of src has an ID y in its mailbox, then the node with the greatest ID is located between x and y (recall that a node and its ID are conflated). This means that outside the scope of the event, an ID cannot jump over the node with the greatest identifier.

Exhibiting this stability axiom requires some work. It would also be possible to proceed using an inductive invariant but, since the property to check is not inductive, doing so would also require some effort.

Example 3. In order to illustrate the difference between stability axioms and inductive invariants, we take a toy token protocol as an example. For the sake of simplicity, we consider a property to check that is already inductive. The protocol features one token passing from nodes to nodes with only one send event send(x,y), with body: token(x) $\wedge$!token'(x) $\wedge$ token'(y) $\wedge$ frame, where frame := $\forall$ z $\cdot$ (z $\neq$ x $\wedge$ z $\neq$ y) $\Rightarrow$ (token(z) $\Leftrightarrow$ token'(z)).

The (inductive) property to check is that there is always at most one node holding the token. To prove this property without relying on its inductiveness, we can use the following stability axiom: stab := $\forall$ z $\cdot$ (z $\neq$ x $\wedge$ z $\neq$ y) $\Rightarrow$!token(z). Contrary to the inductive invariant, the stability axiom has free variables matching the parameters of the event (which are implicitly quantified existentially). Also the preservation of the stability axiom follows from the frame condition as frame $\models$ stab $\Rightarrow$ **X**stab, while the preservation of the inductive invariant follows from the whole transition.

Remark 2. Notice that this property is also true for the nodes that are in the scope of the event, *i.e.*, src and its successor. So in this case, the stability axiom is very close to an invariant property. But this is not the case in general. A distinguishing aspect is that TFC with this stability axiom succeeds in proving the safety property, whereas it would not be possible to deduce it from the "invariant" version of this stability axiom.

The TFC transformation is performed in two phases:

1. Stability axioms, which are provided by the specifier, are added to the body of each event. At this step, the semantics of Cervino is strengthened by the transformation. The obtained formula is not in the Geneva fragment, in particular because of the frame conditions.
2. The Geneva transformation, which is presented in Sect. 4, is applied. In particular, the frame conditions are abstracted by equality transformation and instantiation, but the stability axioms are left unchanged.

Definition 27 (Event enrichment with a stability axiom). *Let ev be an event of a Cervino machine declared as:* ***event*** *ev[$\vec{y}$: $\vec{s}$] modif{τ} and $\mathcal{C}$ be the frame condition of ev, $\mathcal{C} = [\![modif]\!]$. Given a stability axiom $\mathcal{I}$ for $\mathcal{C}$, we define the enrichment $\rho(\!|ev, \mathcal{I}|\!)$ of ev with $\mathcal{I}$ as: $\rho(\!|ev, \mathcal{I}|\!) = \exists \vec{y} : \vec{s} \cdot \tau \wedge \mathcal{C} \wedge (\mathcal{I} \Rightarrow \mathbf{X}\mathcal{I})$.*

Definition 28 (Cervino machine enrichment with stability axioms). *Let Mch be a Cervino machine with axioms $\psi_1, \ldots, \psi_n$, events $ev_1, \ldots, ev_m$ declared as* ***event*** *ev_i [$\vec{y}_1$: $\vec{s}_1$] modif {τ_i} for each $i \in 1..m$ and such that the relations enabling* ***btw*** *are $r_1, \ldots, r_l$. Let* sta *be a function mapping each event to a stability axiom for the according frame condition. Using the same notation as Definition 27, we define the stability axiom enrichment $\rho(\!|Mch, \mathsf{sta}|\!)$ of Mch as $\rho(\!|Mch, \mathsf{sta}|\!) = \phi_0 \wedge \mathbf{G}\phi_{tr} \wedge \phi_{\boldsymbol{btw}}$*

where $\phi_0 = \bigwedge\limits_{i=1}^{n} \psi_i$, $\phi_{tr} = \bigvee\limits_{i=1}^{m} \rho(\!|ev_i, \mathsf{sta}(ev_i)|\!)$ and $\phi_{\boldsymbol{btw}} = \bigwedge\limits_{1 \leqslant i \leqslant l} (\!|\boldsymbol{btw}[r_l]|\!)$.

Definition 29 (Abstract semantics). *Given a Cervino machine Mch and a function* sta*, mapping each event to a stability axiom, we define the stability axiom semantics as* $\mathbb{F}(\!|Mch|\!)_\phi = Abs_{Gen}(\rho(\!|Mch, \mathsf{sta}|\!) \wedge \neg\phi)$

Theorem 6 (Soundness). *If* $[\![Mch]\!]_\phi$ *is satisfiable then* $\mathbb{F}(\!|Mch|\!)_\phi$ *is satisfiable.*

Proof. Follows from Lemmas 1, 2 and 3.

Theorem 7. *If* $\neg\phi \in \mathit{LTR}$ *then* $\mathbb{F}(\!|Mch|\!)_\phi \in$ *Geneva.*

Proof. Follows from Theorem 3.

7 TTC: Transforming Reflexive-Transitive Closure

We now present a simple, effective transformation technique to approximate reflexive-transitive closure (which is present in Cervino and its FOLTL$^*_=$ semantics). This technique has shown to be useful to prove some liveness properties.

As is well known, transitive closure cannot be fully specified in pure FO. On the other hand, it *can* be specified in pure FOLTL, but the axiomatization we are aware of does not fit in the fragments considered here. However, it is possible to define an interesting *approximation* that does fit in the Geneva fragment.

Informally, the crux of our technique relies on the following observation: *any property propagating along a binary relation will eventually propagate to the reflexive-transitive closure thereof.* This is proved (see Theorem 8 below) by following the definitions of the transitive closure and of the eventually connective.

Definition 30 (Propagation schema). *Given binary relations* $\boldsymbol{r}$ *and* $\boldsymbol{t}$ *on a sort* s*, given a formula* P *with* $k+1$ *free variables (*$k \geqslant 0$*), the first of which (of sort* s*) is distinguished in the following. Given* k *variables* $\vec{x}$ *of appropriate typing, we define the* propagation *and* closure schemas *as follows:*

$$\begin{aligned} \boldsymbol{Propagates}[\boldsymbol{r}, P, \vec{x}] &= \forall u, v : s \cdot \boldsymbol{r}(u,v) \Rightarrow \mathbf{G}\,(P[u,\vec{x}] \Rightarrow \mathbf{F} P[v,\vec{x}]) \\ \boldsymbol{Closure}[\boldsymbol{r}, \boldsymbol{t}, P, \vec{x}] &= \boldsymbol{Propagates}[\boldsymbol{r}, P, \vec{x}] \Rightarrow \boldsymbol{Propagates}[\boldsymbol{t}, P, \vec{x}] \enspace . \end{aligned}$$

Theorem 8 (Propagation). *Given a binary relations* $\boldsymbol{r}$ *on a sort* s*, the following property over its reflexive-transitive* $\boldsymbol{r}^\star$ *closure is valid:* $\boldsymbol{Closure}[\boldsymbol{r}, \boldsymbol{r}^\star, P, \vec{x}]$*.*

Proof. Proof validated in Coq.

The proof sketch is the following : we consider the set of element to which the property eventually propagates. Then we use the hypothesis that the property propagates along a binary relation r to show that this set is closed under the relation r. Then as the transitive closure from some element is the smallest set closed under the relation r, we know that the property propagates to any element in the transitive closure.

We prove that under the **Propagates**$[\mathbf{r}, P, \vec{x}]$ hypothesis, for any u, the set of v's satisfying $\mathbf{G}\,(P[u,\vec{x}] \Rightarrow \mathbf{F} P[v,\vec{x}])$ is included in the set of v's that are reachable from u along $\mathbf{r}$. Let $\mathcal{M}$ be a structure and $\mathcal{C}$ an assignment s.t. $\mathcal{M}, \mathcal{C} \models$

Propagates$[\mathbf{r}, P, \vec{x}]$. We assume that there is an instant i such that $P[u, \vec{x}]$ holds (otherwise the satisfaction of the axiom is trivial). Then $\mathcal{M}, i, \mathcal{C} \vDash \mathbf{F}P[u, \vec{x}]$. Also, given v s.t. $\mathcal{M}, i, \mathcal{C} \vDash \mathbf{F}P[v, \vec{x}]$, there exists $k \geqslant i$ s.t. $\mathcal{M}, k, \mathcal{C} \vDash P[v, \vec{x}]$. For any v' s.t. $\mathcal{M}, 0, \mathcal{C} \vDash \mathbf{r}(v, v')$ **Propagates**$[\mathbf{r}, P, \vec{x}]$ implies $\mathcal{M}, k, \mathcal{C} \vDash \mathbf{F}P[v', \vec{x}]$. Thus $\mathcal{M}, i, \mathcal{C} \vDash P[v', \vec{x}]$. Then $\mathcal{M}, 0, \mathcal{C} \vDash \mathbf{r}^\star(u, v)$ implies $\mathcal{M}, i, \mathcal{C} \vDash \mathbf{F}P[v, \vec{x}]$. Hence **Closure**$[\mathbf{r}, \mathbf{r}^\star, P, \vec{x}]$ is valid. □

Given this theorem, the technique we propose consists in replacing the reflexive-transitive closure of a relation (which fits in Cervino and FOLTL$^*_=$) by an uninterpreted relation satisfying the closure schema shown above, for some property P that depends on the sort of the considered binary relation as well as, possibly, other arguments. Remark that finding such a property P requires creativity: the specifier must come up with a relevant propagating property.

Example 4. In the case of the leader election example, we use `TTC` to check that a leader will be elected at some point. The property we use is propagation along *succ* of having a given ID in one's mailbox (**Propagates**$[\mathit{succ}, \mathit{toSend}, \mathit{id}]$).

Definition 31 (Abstract semantics). *Let Mch be a Cervino machine, such that $r_{j_1}, \dots, r_{j_l}$ are binary relations enabling* ***btw****, and $r_{k_1}, \dots, r_{k_m}$ are binary relations whose reflexive-transitive closure is used in Mch. Now, given formulas $P_1, \dots, P_\ell$, where for every $1 \leqslant i \leqslant m$, $\mathrm{FV}(P_i) = \{x, x_1, \dots, x_{n_i}\}$ (with x the distinguished free variable), we define the transitive closure transformation as:*

$$\mathbb{T}(\!|\mathit{Mch}|\!) = \phi_0 \wedge \mathbf{G}\phi_{tr} \wedge \phi_{\boldsymbol{btw}} \\ \wedge \Big(\bigwedge_{1 \leqslant i \leqslant m} \bigwedge_{(c_1, \dots c_{n_i}) \in \mathit{Const}^{n_i}} \boldsymbol{Closure}[r_{k_i}, r^*_{k_i}, P_i, (c_1, \dots c_{n_i})]\Big)$$

where ϕ_0 and ϕ_{tr} are defined as in Definition 15 and $\phi_{\boldsymbol{btw}} = \bigwedge_{1 \leqslant i \leqslant l} (\!|\boldsymbol{btw}[r_{j_i}]|\!)$.

We also define $\mathbb{T}(\!|\mathit{Mch}|\!)_\phi = \mathit{Abs}_{\mathit{Gen}}(\mathbb{T}(\!|\mathit{Mch}|\!) \wedge \neg\phi)$ (notice that, due to the application of the Geneva *transformation, the r^*_i relations become uninterpreted).*

Theorem 9 (Soundness). *If $[\![\mathit{Mch}]\!]_\phi$ is satisfiable then $\mathbb{T}(\!|\mathit{Mch}|\!)_\phi$ is satisfiable.*

Proof. Follows directly from Theorem 8 and Lemmas 1, 2 and 3.

Theorem 10. *If $\neg\phi \in$ LTR then $\mathbb{T}(\!|\mathit{Mch}|\!)_\phi \in$ Geneva.*

Proof. Follows from Theorem 3.

8 Evaluation

To evaluate the relevance of our three tactics, we applied them to several models of distributed protocols. Our research questions were (1) to check that our methods were applicable to real models; (2) to check whether our approach was efficient enough; and (3) to assess the effort for the specifier to come up with

Specification	Type	Technique	Bound	Effort
TLB shootdown	Safety	TEA	2	–
Dining philosophers	Safety	TEA	2	–
Lock server	Safety	TEA	2	–
Gset (CRDT)	Liveness	TEA	2	–
2Pset (CRDT)	Liveness	TEA	2	–
Leader election	Safety	TFC	7	4
	Liveness	TTC	6	1
Token ring	Safety	TFC	7	7
	Liveness	TTC	6	1
FIFO	Liveness	TTC	6	1

Fig. 3. All verifications take less than 20 s ("effort": estimation of user effort with the number of atoms (literals and equality tests) used in the TTC or TFC parameters).

parameters for TFC and TTC. Our strategy was always first to apply the TEA tactic. If TEA failed, then in the case of safety properties, we devised stability axioms in order to apply TFC. Otherwise, for liveness properties and for systems relying on transitive closure, we relied on TTC.

The Cervino prototype takes a Cervino specification as input and generates Electrum models which are then fed to the Electrum Analyzer [4], which itself calls a complete procedure in nuXmv [5]. On a general note, efficiency can be compromised in the case of the TTC and TFC tactics due to larger inferred bounds than for TEA. Furthermore, the size of LTL formulas generated by Electrum for nuXmv grows quickly as the tool merely unfolds quantifiers into conjunction and disjunctions, depending on the bounds. For this reason, we leveraged some properties of the Geneva fragment to end up with smaller models: (1) the size of each domain is an *exact* bound rather than just an upper one; (2) all constants are distinct; (3) existential quantifiers can be unfolded on a limited part of the domain. This is the case because the proof of the BDP for the Geneva fragment [24] shows that, if there is a model of a Geneva formula, there is a model satisfying these properties. The specifications we evaluated are of moderate complexity but are not just toy models:

TLB shootdown The TLB Shootdown algorithm [3] is part of the Mach operating system. Processors keep a cache of page tables in a Translation Look-aside Buffer (TLB). The safety property we prove is that whenever the protocol ensures that the page table is updated, the corresponding update will be flushed by either the initiator or the responder.

Dining philosophers This classic protocol features an unbounded number of philosophers sharing forks. We prove a mutual exclusion property, that is that a fork cannot be simultaneously held by two different philosophers.

Lock server We present a simple lock server protocol studied in Verdi [27] and Ivy [21]. The protocol features a single server and an unbounded number of clients willing to hold a lock. The safety property we verify is that two different clients cannot simultaneously hold the lock.

Gset and 2Pset Conflict-free Replicated Data Types (CRDTs) are a family of concurrent protocols where a data structure is replicated in a network and where the replicas can be independently and concurrently updated. We model the Grow-only Set (G-Set) [26] and the 2-Phase Set [26] CRDTs. In both cases, we prove that any update is eventually delivered to all replicas.

Leader election The leader election protocol, presented in Sect. 2, is inspired by [6]. We notice however that a node sends all the contents of its mailbox at once, which is a strong simplification.

Token ring Token Ring is a classic protocol where a token is passed through nodes with mailboxes in a ring. We prove a safety and a liveness property. In the first case, we use the TFC tactic with 2 stability axioms, one for each event, which basically state that if there is no token apart from the one transferred, then no token can appear on unmodified nodes during the event. The TTC parameter says that the property of holding the token is the one that should propagate, under strong fairness.

FIFO This protocol is a simple mutual exclusion protocol based on a FIFO strategy. We prove a liveness property using TTC, stating that for any integer i, being in the i-th position of the list is a propagating property.

Our conclusion to these case studies is the following (Fig. 3). First the TEA tactic is only efficient for models involving few interactions, which can be attributed to the loss of precision when using universal quantifiers. Regarding TFC, the effort required to find stability axioms seems to be similar to finding an inductive invariant. For TTC, all propagating properties were very simple. Finally, we noticed that, for more complex systems, TTC and TFC can lead to problems that are too large for the model-checker to answer in time (*e.g.* 1 h.)

9 Related Work

The usual way to check a safety property is to exhibit an inductive invariant for the system. The TEA tactic is completely automatic and can handle safety properties but remains quite limited. In our experiments, the TFC tactic showed to be as flexible as an invariant to prove safety properties. Finding stability axioms or an inductive invariant appear similar in difficulty. However, once found, checking an inductive invariant is quicker in computation time than checking the abstract system obtained with stability axioms. On the other hand, stability axioms allow to check complex temporal properties.

Regarding liveness properties, important approaches are based on exhibiting a variant or using the Liveness-to-Safety reduction method proposed in [19]. For the simple examples done with TEA, such methods would allow to prove the properties with little efforts, if done right, but are not fully automatic contrary to the TEA tactic. In both case the computation time is really low.

With the TTC approach, we do not need to exhibit any sort of invariant and the propagating property to exhibit has always been straightforward. To our knowledge there is no easiest method to prove some of the examples we presented. For example, the liveness property of the leader election protocol requires to exhibit a variant and an invariant and both are harder to exhibit than the propagating property. The Liveness-to-Safety reduction method also applies here, but it requires to find an invariant on the system obtained by reduction, as well as finding an axiomatization of the reflexive-transitive closure preserving the Liveness property (while this axiomatization is embedded in TTC tactic). However, despite being more immediate in our examples, the TTC tactic is less flexible than these two alternatives since it applies for liveness properties based on the reflexive-transitive closure.

Our approach can also be compared with the specification of parameterized systems. Cubicle [7–10] is an SMT-based model-checker for the verification of safety properties on parameterized systems. Cubicle is efficient for challenging systems but, contrary to our techniques, it enforces strict syntactic constraints on guards and on the checked property. Others techniques based on labelled proof systems have also been proposed [2]. In [15], the safety of the TLB Shootdown algorithm is proved using such a technique. The user must exhibit the correct invariant for the proof system to conclude; while the TEA tactic is automatic. Also, some methods, such as invisible invariants [25], rely on finding automatically a candidate for being an inductive invariant and then checking if this is the case without needing any input from the user. Such an approach is automatic and efficient but only applies to Bounded-Data Parameterized Systems while our methods applies to a wider context. While most work on parameterized systems focuses on safety properties, [11] addresses liveness properties, but remains essentially theoretical. We remark that the techniques mainly used for parameterized systems are mostly orthogonal to those presented in this paper, and a combination of both could be fruitful.

10 Conclusion

We devised three original, sound (but incomplete) transformations, that allow to check that a state machine specification, expressed in a rather expressive fragment of $\mathrm{FOLTL}^*_=$, enjoys a *temporal* property, expressed in the same setting, whatever the bounds on domains (associated with sorts) are. The transformations were proved correct in Coq. We evaluated our approach on several case studies and found that the transformations were effective and, for the semi-automatic ones, demanded an effort comparable to other approaches. A drawback is that the computed bounds can sometimes grow too much for model-checking to be feasible with the back-end tools we used. Notice that our approach is orthogonal to the main other approaches (for instance, inference of invariants) and could certainly be combined with some of them. Once a universally quantified inductive invariant Inv is found, such a combination would be possible by adding an axiom of the form **G** Inv to our abtract specification. This refines the abstraction while fitting in both LTR and Geneva. This is left for future work.

Acknowledgements. We thank the article and artifact anonymous reviewers for their remarks that helped improve this work, as well as Nuno Macedo for his technical assistance.

References

1. Abrial, J.R.: Modeling in Event-B: System and Software Engineering, 1st edn. Cambridge University Press, Cambridge (2010). https://doi.org/10.1017/cbo9781139195881
2. Arons, T., Pnueli, A., Ruah, S., Xu, Y., Zuck, L.: Parameterized verification with automatically computed inductive assertions? In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 221–234. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44585-4_19
3. Black, D.L., Rashid, R.F., Golub, D.B., Hill, C.R.: Translation lookaside buffer consistency: a software approach. ACM SIGARCH Comput. Archit. News **17**(2), 113–122 (1989). https://doi.org/10.1145/68182.68193
4. Brunel, J., Chemouil, D., Cunha, A., Macedo, N.: The electrum analyzer: model checking relational first-order temporal specifications. In: 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018). Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ACM Press, Montpellier, France, September 2018. https://doi.org/10.1145/3238147.3240475
5. Cavada, R., et al.: The nuXmv symbolic model checker. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 334–342. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_22
6. Chang, E., Roberts, R.: An improved algorithm for decentralized extrema-finding in circular configurations of processes. Commun. ACM **22**(5), 281–283 (1979). https://doi.org/10.1145/359104.359108
7. Conchon, S., Declerck, D., Zaïdi, F.: Cubicle-$\mathcal{W}$: Parameterized model checking on weak memory. In: International Joint Conference on Automated Reasoning, pp. 152–160. Springer (2018). https://doi.org/10.1007/978-3-319-94205-6_11
8. Conchon, S., Declerck, D., Zaïdi, F.: Parameterized model checking on the TSO weak memory model. J. Autom. Reason. **64**(7), 1307–1330 (2020). https://doi.org/10.1007/s10817-020-09565-w
9. Conchon, S., Goel, A., Krstić, S., Mebsout, A., Zaïdi, F.: Cubicle: a parallel SMT-based model checker for parameterized systems. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 718–724. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31424-7_55
10. Conchon, S., Mebsout, A., Zaïdi, F.: Certificates for parameterized model checking. In: Bjørner, N., de Boer, F. (eds.) FM 2015. LNCS, vol. 9109, pp. 126–142. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19249-9_9
11. Farzan, A., Kincaid, Z., Podelski, A.: Proving liveness of parameterized programs. In: 2016 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), pp. 1–12 (2016). https://doi.org/10.1145/2933575.2935310
12. Hawblitzel, C., et al.: Ironfleet: proving practical distributed systems correct. In: Proceedings of the 25th Symposium on Operating Systems Principles, pp. 1–17 (2015). https://doi.org/10.1145/2815400.2815428
13. Hodkinson, I., Wolter, F., Zakharyaschev, M.: Decidable fragments of first-order temporal logics. Ann. Pure Appl. Logic **106**(1–3), 85–134 (2000). https://doi.org/10.1016/s0168-0072(00)00018-x

14. Hodkinson, I., Wolter, F., Zakharyaschev, M.: Monodic fragments of first-order temporal logics: 2000–2001 A.D. In: Nieuwenhuis, R., Voronkov, A. (eds.) LPAR 2001. LNCS (LNAI), vol. 2250, pp. 1–23. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45653-8_1
15. Hoenicke, J., Majumdar, R., Podelski, A.: Thread modularity at many levels: a pearl in compositional verification. ACM SIGPLAN Not. **52**(1), 473–485 (2017). https://doi.org/10.1145/3009837.3009893
16. Kuperberg, D., Brunel, J., Chemouil, D.: On finite domains in first-order linear temporal logic. In: Artho, C., Legay, A., Peled, D. (eds.) ATVA 2016. LNCS, vol. 9938, pp. 211–226. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46520-3_14
17. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley Professional (2002)
18. Padon, O.: Deductive Verification of Distributed Protocols in First-Order Logic. Ph.D. thesis, Ph.D. Dissertation. Tel Aviv University (2018)
19. Padon, O., Hoenicke, J., Losa, G., Podelski, A., Sagiv, M., Shoham, S.: Reducing liveness to safety in first-order logic. In: Proceedings of the ACM Conference on Principles of Programming Languages (POPL) 2, 26 (2017). https://doi.org/10.1145/3158114
20. Padon, O., Losa, G., Sagiv, M., Shoham, S.: Paxos made epr: decidable reasoning about distributed protocols. Proc. ACM on Program. Lang. **1**(OOPSLA), 108 (2017). https://doi.org/10.1145/3140568D
21. Padon, O., McMillan, K.L., Panda, A., Sagiv, M., Shoham, S.: Ivy: safety verification by interactive generalization. ACM SIGPLAN Not. **51**(6), 614–630 (2016). https://doi.org/10.1145/2980983.2908118
22. Peyras, Q., Bodeveix, J.P., Brunel, J., Chemouil, D.: Cervino prototype, Coq formalization and Benchmarks (CAV 2021 artifact) (Apr 2021). https://doi.org/10.5281/zenodo.4725675
23. Peyras, Q., Brunel, J., Chemouil, D.: A bounded domain property for an expressive fragment of first-order linear temporal logic. In: Gamper, J., Pinchinat, S., Sciavicco, G. (eds.) 26th International Symposium on Temporal Representation and Reasoning, TIME 2019, October 16–19, 2019, Málaga, Spain. LIPIcs, vol. 147, pp. 15:1–15:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019). https://doi.org/10.4230/LIPIcs.TIME.2019.15
24. Peyras, Q., Brunel, J., Chemouil, D.: A decidable and expressive fragment of many-sorted first-order linear temporal logic. Information and Computation p. 104641 (2020). https://doi.org/10.1016/j.ic.2020.104641
25. Pnueli, A., Ruah, S., Zuck, L.: Automatic deductive verification with invisible invariants. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 82–97. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45319-9_7
26. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: A comprehensive study of convergent and commutative replicated data types. Ph.D. thesis, Inria-Centre Paris-Rocquencourt; INRIA (2011)
27. Wilcox, J.R., Woos, D., Panchekha, P., Tatlock, Z., Wang, X., Ernst, M.D., Anderson, T.: Verdi: a framework for implementing and formally verifying distributed systems. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 357–368 (2015). https://doi.org/10.1145/2737924.2737958

Hardware and Model Checking

Progress in Certifying Hardware Model Checking Results

Emily Yu[1](✉), Armin Biere[1], and Keijo Heljanko[2,3]

[1] Johannes Kepler University, Linz, Austria
[2] University of Helsinki, Helsinki, Finland
[3] Helsinki Institute for Information Technology, Helsinki, Finland

Abstract. We present a formal framework to certify k-induction-based model checking results. The key idea is the notion of a k-witness circuit which simulates the given circuit and has a simple inductive invariant serving as proof certificate. Our approach allows to check proofs with an independent proof checker by reducing the certification problem to pure SAT checks and checking a simple QBF with one quantifier alternation. We also present CERTIFAIGER, the resulting certification toolkit, and evaluate it on instances from the hardware model checking competition. Our experiments show the practical use of our certification method.

1 Introduction

In many verification applications, k-induction [34] (also known as temporal induction) is used as a powerful technique that reduces model checking to a series of SAT problems. It has been extensively investigated as an effective approach for unbounded model checking [18,22]. As a generalisation of simple induction, for a given safety property, the k-induction method concerns a base case and an inductive case: the base case is a bounded model checking problem with a depth of k; the inductive case assumes the property holds for k consecutive steps, then checks it also holds for $k+1$ steps. The safety property is said to be k-inductive if both conditions are satisfied. The nature of the k-induction algorithm allows it to be integrated with modern SAT/SMT solvers. For example, reduction techniques such as preprocessing have been investigated with k-induction in an incremental setting [17]. The present state-of-the-art also concerns combining k-induction with existing SAT-based model checking (SMC) techniques including interpolation and property directed reachability [23,27]. Furthermore, k-induction has also been extended to the context of infinite-state systems [13,19,26,32], as well as software verification [16]. Another variant of this line of research is the use of k-induction in sequential equivalence checking [31].

Model checking has been an effective technique for the verification of safety-critical systems. In particular, applications deployed in industrial settings such as nuclear facilities, increasingly utilise model checking to gain trust in the correctness of their designs [20,30,36]. In such ultra safety-critical applications the certification that the model checking results are in fact correct is crucial. We argue that in model checking generic machine checkable certification is still in its infancy in contrast to related fields. For instance in SAT competitions [2,24], certifiable proofs are mandatory. This has helped to improve the trust we have in SAT solving results as well as the quality of SAT solvers tremendously.

Even though counterexample validation is commonly used in model checking to certify negative verification results through simulation, producing a generic machine checkable proof on success is less straight-forward. To mitigate this problem, certification of model checking has been suggested earlier in [14,21,23,29,33,36,37], but the methods presented in these works are either not directly applicable to k-induction (in its vanilla form), produce k-induction specific certificates (fail to provide an inductive invariant), or are considered to have exponential certificates. This apparently made it hard to, e.g., require all model checkers to produce proofs in the hardware model checking competitions.

As symbolic model checking of bit-level properties for hardware circuits is PSPACE-complete, we introduce in this paper a novel certification framework for k-induction-based model checking. Our proposed approach generates a fixed number of SAT problems together with a one-alternation only QBF, which are verified by an independent certifier, thereby enabling the certification of k-induction proofs at lower complexity. Our method efficiently extends the given model checking problem to finding a simple inductive invariant of a larger circuit as a proof of k-induction of the original circuit. In particular, the certificate size (as a circuit) is shown to be linear in size of the given model, and the inductive depth. We present CERTIFAIGER, which works as a complete tool suite for certification, independent of any model checker. Experimental results show that our technique works efficiently and can be adapted for practical use.

The rest of the paper is organised as follows: In Sect. 2 we introduce the notion of combinational simulation in the context of circuits. In Sect. 3, we study the formal property of combinational simulation and define k-induction-based model checking with an example. In Sect. 4, we present our proposed certification approach followed by theoretical results in terms of k-induction. We describe the implementation of our tool suite in Sect. 5, and report on experimental results in Sect. 6. Finally, we conclude in Sect. 7.

2 Circuits

In this section, we present a slightly non-standard notation to formalize systems. It allows us to represent systems and particularly circuits symbolically in a compact way and is crucial to reduce notational clutter in the following.

Let $\mathbb{B}(V)$ be the set of Boolean expressions (propositional formulas) over the Boolean variables V. We also write $\mathbb{B}(I, L)$ to denote the set of Boolean expressions over $I \cup L$, where I and L are two sets of Boolean variables. Given two Boolean expressions $f(V)$, $g(V) \in \mathbb{B}(V)$ we call them *equivalent*, written $f(V) \equiv g(V)$, if they have the same models. This notation is also applied to Boolean expressions over different sets of variables by simply interpreting them over the union of their variables. We use "$\simeq$" for syntactic equivalence [15],

"$\rightarrow$" for syntactic implication, and "$\Rightarrow$" for semantic implication. To define semantical concepts or abbreviations we stick to equality "$=$".

In the context of this paper, models are expressed in the form of finite logical circuits, where states can be seen as truth assignments to latches and inputs. Initial states are defined by the reset values of latches, in our case, represented by their reset functions. For each latch l in L, there is a reset function $r_l(L)$ which is a formula (Boolean expression) over a set of latches L, thus allowing cyclic definitions. Note that a cyclic definition can lead to unsatisfiable reset formulas, in which case there are simply no initial states. Additionally, for some $L'' \subseteq L$, we define $R(L'') = \bigwedge_{l \in L''} l \simeq r_l(L)$ to allow us to analyse reset functions of individual subsets of latches. The transition relation is expressed as a "next state" formula associated with each latch, whereas non-determinism comes from inputs (which act as the environment). The successor value of each latch is defined by applying its transition function on the current values of latches and inputs. Intuitively, a safety property specifies that the system must not violate certain behaviours, i.e., only "good states" are reachable. In this paper we focus on such simple safety properties and leave liveness properties (see e.g., [29]) etc. for future work.

Definition 1 (Circuit). *A circuit $C = (I, L, R, F, P)$ is defined as follows:*

1. *I: the set of Boolean* input *variables.*
2. *L: the set of Boolean* latch *variables.*
3. *$R = \{r_l(L) \mid l \in L\}$ is a set of* reset function *formulas.*
4. *$F = \{f_l(I, L) \mid l \in L\}$ is a set of* transition function *formulas, such that for every latch $l \in L$, there is a transition function formula $f_l(I, L) \in \mathbb{B}(I, L)$.*
5. *$P(I, L) \in \mathbb{B}(I, L)$ is a formula encoding the* (good states) property.

The reset functions characterise the initialisation of the circuit. Such definition of reset abstracts the way how circuits are reset. As a short-hand we use $L' \simeq F(I, L)$ to denote a conjunction of the corresponding equivalences, i.e., it is interpreted as $\bigwedge_{l \in L'} l' \simeq f_l(I, L)$. For clarity, we use **subscripts** as in L_i to denote a copy of the latch variables L in the **temporal direction** at some timestamp i, where L_0 is the set of latches at timestamp 0 when the circuit is supposed to be initialised. Note that, using such transition *functions* to describe transition relations implies that there will always be a successor state. The temporal evolution of a system is expressed using the notion of *unrolling*, which has a specific length and follows the transition relation at each step.

Definition 2 (Unrolling). *For an unrolling depth $m \in \mathbb{N}$, the* unrolling *of a circuit C of length m is defined as the formula $U_m = \bigwedge_{i \in [0,m)} (L_{i+1} \simeq F(I_i, L_i))$.*

Note that in this definition, we use I_i and L_i as sets of variables, whereas U_m is a formula. For $m = 0$, the conjunction is empty thus the formula is trivial.

Definition 3 (Initialised unrolling). *An* initialised unrolling *of a circuit C, with $C = (I, L, R, F, P)$, is defined as $U_m \wedge R(L_0)$, where U_m is an unrolling.*

We say an unrolling is *safe* if and only if the property holds at every timestamp along the whole length of the unrolling.

Definition 4 (Safe unrolling). *Unrolling U_m of a circuit $C = (I, L, R, F, P)$ is said to be* safe *if*

$$U_m \Rightarrow \bigwedge_{i \in [0,m]} P(I_i, L_i).$$

Definition 5 (Safe initialised unrolling). *An initialised unrolling $U_m \wedge R(L_0)$ of a circuit $C = (I, L, R, F, P)$ is said to be* safe *if*

$$U_m \wedge R(L_0) \Rightarrow \bigwedge_{i \in [0,m]} P(I_i, L_i).$$

We are now ready to introduce the notion of a *combinational extension* between two circuits. It is purely syntactic based on sharing inputs and latches.

Definition 6 (Combinational extension). *Given circuits $C = (I, L, R, F, P)$ and $C' = (I', L', R', F', P')$, C'* combinationally extends *C if $I = I'$ and $L \subseteq L'$.*

As noticed above, this definition allows us to interpret the inputs and latches of a circuit as being part of another circuit. In practice for instance we simply assume that the first $|L|$ latches of the circuit C' are mapped to those of C assuming some ordering of the latches, as it is for instance the case in the AIGER format [7] used in the Hardware Model Checking Competition (HWMCC) [5].

To tackle the problem of generating a proof certificate for k-induction of the safety of a circuit C, as is the main goal of this paper, we extend it to a larger circuit C' with additional "book-keeping" behaviours [1] for which we can show the same property by using standard induction. To ensure that the resulting extended circuit C' preserves the original property, we provide a formalization through a *combinational simulation* relation between two circuits, which needs to be formally verified by a certifier. One important aspect of our design principles is to keep the complexity of the required certification procedure low, in other words, to be done via pure SAT solver checks or by solving a QBF with at most one quantifier alternation. This leads to a more complicated non-standard design of the certification approach, the details of which will be described in Sect. 4.

From a practical perspective, under *combinational simulation* defined below in Definition 7, we require that the transition functions on the "common" parts of the two circuits are equivalent. For the new latches, the transition functions are always satisfiable (as they are functions), and thus we need no constraints on them. As second condition we require that if the safety property P' holds in the extended circuit, then the property P holds in the original circuit. The last condition we need to check is that all the new latches of the extended circuit can be initialised with some values whenever the original circuit can be initialised and using the same values for initialising the common latches. In other words, for all initialisations of the original circuit there is at least one initialisation of the extended circuit with the same values for common latches.

Under these conditions Theorem 1 in Sect. 3 shows that if the extended circuit (in this sense) combinationally simulates the original one and the extended circuit is safe then the original circuit is safe as well.

With some abuse of notation, we use $\exists L$ in a Quantified Boolean Formula (QBF) to denote existential quantification over variables in L. As usual, free variables are (implicitly) assumed to be quantified universally.

Definition 7 (Combinational simulation). *Given circuits $C = (I, L, R, F, P)$ and $C' = (I', L', R', F', P')$ where C' combinationally extends C, we say that C'* combinationally simulates *C, if the following holds:*

1. *$f_l(I, L) \equiv f'_l(I, L')$ for $l \in L$,* *"transition"*
2. *$P'(I, L') \Rightarrow P(I, L)$, and* *"property"*
3. *$R(L) \Rightarrow \exists(L' \backslash L) R'(L')$.* *"reset"*

In later context when verifying the combinational simulation relation between two circuits, we refer to Definition 7.1 as the *transition check*, Definition 7.2 as the *property check*, and Definition 7.3 as the *reset check*.

3 Model Checking

In this section, we consider model checking via k-induction. The model checking problem for safety properties concerns determining whether, given a circuit with a property P, it is the case that P holds in all reachable states, *i.e.,* the initialised unrolling of a circuit of any arbitrary length is safe.

Definition 8 (Safe circuit). *Let U_m be the unrolling of circuit C, C is safe iff $U_m \wedge R(L_0) \Rightarrow \bigwedge_{i \in [0,m]} P(I_i, L_i)$ holds for all $m \in \mathbb{N}$.*

Based on the above definition, we say the property P "holds" in C if the circuit is safe with respect to P.

Theorem 1. *Assume that the circuit C' combinationally simulates the circuit C. If C' is safe, then C is safe.*

Proof. We do a proof by contradiction. Let $m \in \mathbb{N}$ be a bound for which the claim does not hold. Thus the unrolling of length m of C' is safe w.r.t. P, and therefore $U'_m \wedge R'(L'_0) \Rightarrow \bigwedge_{i \in [0,m]} P'(I'_i, L'_i)$ holds. To obtain the contradiction we assume there is a satisfying assignment s of $U_m \wedge R(L_0) \wedge \neg \bigwedge_{i \in [0,m]} P(I_i, L_i)$, which would make C not to be safe. Thus $R(L_0)$ needs to be satisfiable. Now the reset check of Definition 7.3 implies that $R'(L'_0) \wedge R(L_0)$ is guaranteed to be satisfiable with L_0 being a subset of L'_0. Moreover, by Definition 7.1, the unrolling U'_m of C' is also satisfiable with the transition function F applied on the projected ("common") component on both circuits. Also for the new latches the fact that we use a transition function for them, they are also satisfiable (transition functions guarantee that there is always a successor state for all states). Therefore the initialised unrolling $R'(L'_0) \wedge U'_m$ is satisfiable. Furthermore, by our assumption, $\bigwedge_{i\in[0,m]} P'(I'_i, L'_i)$ holds. By Definition 7.1 and Definition 7.3, the projected latches of C' stay the same as L_i for all $i \in [0, m]$, and thus by Definition 7.2 we have that $\bigwedge_{i\in[0,m]} P(I_i, L_i)$ holds. □

As usual, we call a formula ϕ to be an *inductive invariant* ϕ of a circuit C if ϕ satisfies the following conditions: (1) $R(L) \Rightarrow \phi(I, L)$, (2) $\phi(I, L) \Rightarrow P(I, L)$, and (3) $U_1 \wedge \phi(I_0, L_0) \Rightarrow \phi(I_1, L_1)$. As a generalisation, k-induction looks at k steps of evolution rather than 1 step by assuming the property holds in k consecutive timestamps at the induction step.

Definition 9 (k-inductive). *Given a circuit C with a property P, define the formula $S_k = \bigwedge_{i\in[0,k)} P(I_i, L_i)$. Then P is called k-*inductive *in C if and only if the following two conditions hold:*

1. $U_{k-1} \wedge R(L_0) \Rightarrow S_k$, and *"initiation"*
2. $U_k \wedge S_k \Rightarrow P(I_k, L_k)$. *"consecution"*

The first condition Definition 9.1 in this definition is called *initiation check*, also *bounded model checking check* or simply BMC check on the initialised unrolling of length $k-1$, whereas the second condition Definition 9.2 is referred to as the *consecution check* for the unrolling of k. Note that a 1-inductive invariant is equivalent to an inductive invariant when $\phi(I, L) \equiv P(I, L)$.

```
MODULE main
VAR
    r : boolean;
    c0 : boolean;
    c1 : boolean;
    c2 : boolean;
DEFINE
    a0 := TRUE;
    a1 := c0 & a0;
    a2 := c1 & a1;
    m0 := (c0 = FALSE);
    m1 := (c1 = FALSE) & m0;
    m2 := (c2 = TRUE) & m1;
    b0 := (c0 = FALSE);
    b1 := (c1 = TRUE) & b0;
    b2 := (c2 = TRUE) & b1;
ASSIGN
    init(c1) := FALSE;
    init(c0) := FALSE;
    init(c1) := FALSE;
    next(c0) := !r & !m2 & (c0 != a0);
    next(c1) := !r & !m2 & (c1 != a1);
    next(c2) := !r & !m2 & (c2 != a2);
SPEC
    AG !b2
```

Fig. 1. The SMV code for the *Counter* example.

Example 1. We consider a simple example of an N-bit counter, where the counter counts up to a *modulo* bound m, then it resets to zero. There is also a *reset* signal which works as an enabler, such that when the signal is set to 1, the counter is forced to reset. The property checks whether the counter value reaches b.

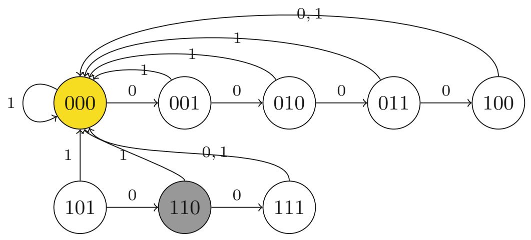


Fig. 2. The transition diagram of the *Counter* example. The initial state is "000" (colored yellow). In the (gray) "bad" State "110" the property does not hold. (Color figure online)

Here the exact modulo check makes the model checking problem k-inductive ($k = b - m + 1$). More precisely, for $N = 3$, the formal description of a 3-bit counter is given in the SMV language in Fig. 1, where $m = 5, b = 6$. (Note that our example can be easily extended to integers too.) The state diagram of this system is shown in Fig. 2. The input values are specified with the transition relations. This model is 2-inductive.

4 Certification

In our suggested approach, certifying model checking results concerns finding and checking an inductive invariant which implies the original specification, which in our case, is the safety property P. To tackle the problem of certifying k-induction-based model checking for any given circuit, in this section, we redirect the problem to generating a simple inductive invariant from a *k-witness circuit*, in which the original circuit is combinationally simulated.

We start by defining the formalism of a k-witness circuit. The main idea is to record the previous $k-1$ states and inputs of the circuit observed during the execution, "flattening" the k-induction procedure back to normal induction of a larger circuit. As a result, the size of the circuit increases by a factor of k, where k is the constant used in the k-induction scheme. The k-witness circuit has k local components of inputs and latches. Each component can be seen as representing a state in the original circuit. Whenever a new state is saved, the oldest one is discarded.

One of the key technical challenges is the proper initialisation of the k-witness circuit. We use an additional k initialisation bits for indicating which components of the circuit have been initialised. This helps accomplishing the combinational simulation relation later. We say a component is initialised if its initialisation bit is $\top$. At initialisation, the k-witness circuit can be either *fully* or *partially* initialised. Figure 3 displays three ways of initialising the components. In the case of full initialisation, the circuit pre-computes k steps of the original circuit as the initial state of the k-witness circuit. Thus intuitively in the full initialisation case the initial state of the k-witness circuit encodes the states reachable in the

k-step initialised BMC unrolling of the original circuit. In partial initialisation scenarios the circuit instead pre-computes an initialised BMC unrolling for fewer steps, where some components are left uninitialised. In the final case where there are no pre-computed steps, the circuit simply runs from an original initial state, leaving all the other components fully uninitialised.

In the definitions below, we use the **superscript** of i in L^i to denote a copy of latches L in the **spacial direction**, such that we introduce a set of new latch variables for every L^i, where $l^i \in L^i$ is the corresponding copy of $l \in L$, and similarly for inputs. We refer to l^i as some latch in L^i, where i is the index of a latch set L^i. The formal definition of k-witness circuit is given below. We continue to use **subscripts** for the **temporal direction**.

Definition 10 (k-witness circuit). *Given a circuit $C = (I, L, R, F, P)$, and $k \in \mathbb{N}^+$, the k-witness circuit* $\boxed{C'=(I', L', R', F', P')}$ *of C is defined as follows:*

1. $\boxed{I'} = I$*. For simplicity we also refer to I' as X^{k-1}.*
2. $\boxed{L'} = X^0 \cup \cdots \cup X^{k-2} \cup L^0 \cup \cdots \cup L^{k-1} \cup B$*, such that,*
   (a) X^i is a copy of the original inputs, for all $i \in [0, k-2]$.
   (b) L^i is a copy of the original latches, for all $i \in [0, k-1]$.
   (c) $B = \{b^0, \ldots, b^{k-1}\}$ is the set of initialisation bits.
3. *The reset function* $\boxed{R'} = \{r'_l(L') \mid l \in L'\}$ *is defined as follows:*
   (a) For $x \in X^0 \cup \cdots \cup X^{k-2}, r'_x = x$.
   (b) For $i \in [1, k-1)$, $u^i = R(L^i) \vee u^{i+1}$, and $u^{k-1} = R(L^{k-1})$.
   (c) For $l \in L^0$, $r'_l = \mathsf{ite}(u^1, l, r_l(L^0))$.
   (d) For $i \in [1, k)$, $r'_{l^i} = \mathsf{ite}(u^i, l^i, f_{l^i}(X^{i-1}, L^{i-1}))$.
   (e) $r'_{b^{k-1}} = \top$.
   (f) $r'_{b^0} = \neg u^1$.
   (g) For $i \in [1, k-1)$, $r'_{b^i} = b^{i-1} \vee (R(L^i) \wedge \neg u^{i+1})$.
4. $\boxed{F'} = \{f'_l(I', L') \mid l \in L'\}$ *is defined as follows:*
   (a) For $i \in [0, k-1)$, $f'_{x^i}(I', L') = x^{i+1}$.
   (b) For $l \in L^{k-1}$, $f'_l(I', L') = f_l(X^{k-1}, L^{k-1})$.
   (c) For $i \in [0, k-1)$, $f'_{l^i}(I', L') = l^{i+1}$.
   (d) For $i \in [0, k-1)$, $f'_{b^i}(I', L') = b^{i+1}$, and $f'_{b^{k-1}}(I', L') = b^{k-1}$.
5. *The property* $\boxed{P'}$ *is defined as* $P'(I', L') = \bigwedge_{i \in [0,4]} p_i(I', L')$ *such that:*
   (a) For $i \in [0, k-1)$, $h^i = (L^{i+1} \simeq F(X^i, L^i))$.
   (b) $p_0(I', L') = \bigwedge_{i \in [0,k-1)} (b^i \rightarrow b^{i+1})$.
   (c) $p_1(I', L') = \bigwedge_{i \in [0,k-1)} (b^i \rightarrow h^i)$.
   (d) $p_2(I', L') = \bigwedge_{i \in [0,k)} (b^i \rightarrow P(X^i, L^i))$.
   (e) $p_3(I', L') = \bigwedge_{i \in [1,k)} ((\neg b^{i-1} \wedge b^i) \rightarrow R(L^i))$.
   (f) $p_4(I', L') = b^{k-1}$.

In Definition 10 we list five parts of the k-witness circuit. For clarity, we explain each part in more details in the following text:

1. The set of **inputs** is identical to that of the original circuit.
2. The set of **latches** consists of the original latches, k initialisation bits, and an additional $k-1$ copies of inputs and latches which are introduced to save observations of previous states.
3. The **reset** function is defined to allow non-deterministic initialisation (see Fig. 3), where we use helper variables u^i for a more compact encoding. The formula u^i is satisfied whenever a component younger than the ith has the same reset value as the original circuit. The reset functions of the X^i latches (for $i < k-1$) ensure they are initialised in a non-deterministic fashion. As for the initialisation bits B, their reset values are deterministic, depending on the initialisation status of the components.
4. The **transition** function of the $(k-1)^{\text{th}}$ copy of latches is identical to the original transition function, while every older component simply saves the value of its one timestep younger component.
5. Finally, the **property** is composed of five sub-properties, where h^i is satisfied whenever the two adjacent components follow the original transition relation.

Figure 4 illustrates a comparison of variable structures of the original circuit and its k-witness (this also suggests their combinational extension relation). The area marked yellow (left box and top right box on the right) consists of the same set of variables. We consider each pair (X^i, L^i) as a *component* in the circuit and refer to (X^{k-1}, L^{k-1}) as the most recent component (youngest copy), and (X^0, L^0) as the oldest component (copy). Additionally we also refer to the inputs I' as X^{k-1} for convenience.

The property P' is comprised of five sub-properties. The *monotonicity* property p_0 expresses the monotonic nature of the initialisation bits. Intuitively, if a component is initialised, all components younger than it should also be initialised. The *transition* property p_1 expresses the property that every initialised component has to follow the transition relation in the original circuit. Of particular interest is the k-safety property p_2, which says the original property P needs to be satisfied in every initialised component. The *reset* property p_3 expresses the property that in the case of partial initialisation, the oldest initialised component needs to satisfy the original reset function. Finally, p_4 expresses that at least the youngest component should have the initialisation bit set.

We now show the combinational simulation relation between the original circuit and its k-witness circuit.

Theorem 2. *The circuit C is combinationally simulated by its k-witness circuit.*

Proof. By the construction in Definition 10, the inputs stay the same in the k-witness circuit C', and the new latches are a superset of the original ones (the youngest component in C'). Thus by Definition 6, C' combinationally extends C. Based on Definition 10.4, the transition function of L^{k-1} is identical to the original one, which satisfies Definition 7.1. In the new property, p_4 and p_2 together

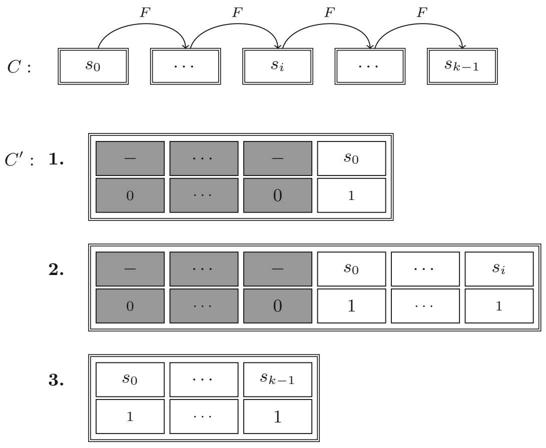


Fig. 3. The diagram shows three possible initial states of C'. Here (1) illustrates 1-initialisation, (2) is i-initialisation, and (3) full initialisation. The grey area are the uninitialised components (the “don't care”s).

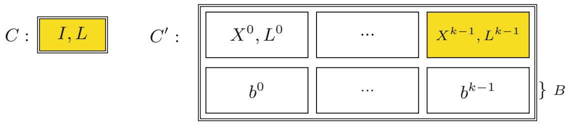


Fig. 4. The structure of input and latch variables in C and C'. (Color figure online)

imply $P(X^{k-1}, L^{k-1})$. In other words, the original property holds in the most recent component. This then satisfies Definition 7.2. By Definition 10, for every satisfiable assignment of $R(L)$, the same assignment satisfies $R'(L)$ on the common latches (the youngest component). For all the new latches we observe the following. Because the reset of the newest component is satisfiable with the same assignment as in the original circuit, we can see that u^{k-1} is true in the k-witness circuit and therefore all other u^i are also true. Therefore all the ite-statements of the reset definition become trivially satisfiable. To complete the argument, by Definition 10.3, all the initialisation bits can be now set to $\bot$ except b^{k-1} which can be set to $\top$. A satisfying assignment of $R'(L')$ can thus be directly constructed (deterministically in polynomial time) from any satisfying assignment of $R(L)$. This implies the reset condition of Definition 7.3 holds. (Sidenote: This implies that the QBF check needed in the combinational simulation relation could potentially be solved easily in practice for these k-witness circuits.) Therefore C' combinationally simulates C. □

In the following, we present the main result of this paper on the relationship between a circuit C and its k-witness circuit C' in terms of k-induction.

Theorem 3. *Given a circuit C, a fixed $k \in \mathbb{N}^+$, and its k-witness circuit C', P is k-inductive in C iff P' is 1-inductive in C'.*

Proof. We consider the two k-inductive checks in Definition 9 for both directions. In Theorem 4 we show that the BMC check (of the initialised unrolling of length $k-1$) in C passes, if and only if the same check (of the initialised unrolling of length 0) in C' also passes. In Theorem 5 we prove that if the consecution check of C' passes, then the consecution check also passes in C. Lastly, Theorem 6 shows that if P is k-inductive in C, then the consecution check of P' using the unrolling of length 1 passes in C'. By combining them together, we conclude P is k-inductive in C iff P' is 1-inductive in C'. □

For the BMC check in the two circuits, we need to analyse three separate cases as shown in Fig. 3, which correspond to Lemmas 2, 3, and 4, respectively. But before this we need a technical Lemma 1 on the initialisation bits. In the following context, we consider a given circuit C, and its k-witness circuit C' with a fixed k.

Lemma 1. *For the initialised unrolling of length* 0 *of the k-witness circuit C', the reset values of the initialisation bits B_0 are deterministic and depend only on the component with the largest index $i \in [0,k)$ for which $R(L_0^i)$ is satisfied.*

Proof. Firstly, we define $S = \{i \mid R(L_0^i)\}$, based on which we consider two cases.

(1). By Definition 10.3(c), if $\neg u_0^1$, then $0 \in S$. In this case, $b_0^0 = \top$ by Definition 10.3(f), and by Definition 10.3(e)(g), $b_0^1, ..., b_0^{k-1}$ are all set to $\top$.

(2). Otherwise we consider u_0^1, where S contains at least some $i \in [1,k)$. Let m be the maximum index in S, and $m \neq 0$. Since $R(L_0^m)$, u_0^m is satisfied, so are $u_0^{m-1}, ..., u_0^1$, while $u_0^{m+1}, ..., u_0^{k-1}$ are not. In Definition 10.3(g), for all $i \in S$, $R(L_0^i) \wedge \neg u^{i+1}$ is only satisfied when $i = m$, thus $b_0^m = \top$. Therefore $b_0^i = \top$ for all $i \in [m+1,k)$. By Definition 10.3(f), $b_0^0 = \bot$, therefore for all $i \in [1,m), b_0^i = \bot$. □

Initialisation bits are indicators for the initialisation status of the k-witness circuit. We observe that the sub-properties $p_0, ..., p_3$ of the k-witness circuit trivially hold for uninitialised components (*i.e.,* those for which the initialisation bit is 0), while p_4 solely depends on b^{k-1}.

Lemma 2. *If the initialised unrolling of length $k-1$ of the original circuit C is safe, the initialised unrolling of length* 0 *of the k-witness circuit C' is also safe, in the case of* 1*-initialisation.*

Proof. Assume $U_{k-1} \wedge R(L_0) \Rightarrow \bigwedge_{i \in [0,k)} P(I_i, L_i)$ such that the initialised unrolling of C is safe. In the case of 1-initialisation, we consider $R'(L_0') \wedge R(L_0^{k-1})$ as the initialised unrolling of C', as U_0' is trivial. By Lemma 1 and Definition 10.3, for the initialisation bits, only b_0^{k-1} is set to $\top$ and the rest remain $\bot$. The values of B_0 then satisfy $p_0(I_0', L_0'), p_1(I_0', L_0'), p_4(I_0', L_0')$ trivially. Every satisfying assignment of $R'(L_0') \wedge R(L_0^{k-1})$ satisfies $R(L_0)$ with $L_0 = L_0^{k-1}, I_0 = X_0^{k-1}$. Similar to our argument in Theorem 1, $U_{k-1} \wedge R(L_0)$ is then also satisfiable. By our assumption, $P(X_0^{k-1}, L_0^{k-1})$ is thus satisfied. The premise of $p_2(I_0', L_0')$ is only satisfied for b_0^{k-1}, and with the same assignment satisfying $P(X_0^{k-1}, L_0^{k-1})$, $p_2(I_0', L_0')$ is also satisfied. Lastly, the premise of $p_3(I_0', L_0')$ is only satisfied for $\neg b_0^{k-2} \wedge b_0^{k-1}$, and since $R(L_0^{k-1}), p_3(I_0', L_0')$ is satisfied. Therefore we have $P'(I_0', L_0')$. □

Lemma 3. *If the initialised unrolling of length $k-1$ of the original circuit C is safe, the initialised unrolling of length 0 of the k-witness circuit C' is also safe, in the case of i-initialisation.*

Proof. Firstly, we assume $U_{k-1} \wedge R(L) \Rightarrow \bigwedge_{i\in[0,k)} P(I_i, L_i)$. In the case of i-initialisation, we consider $R'(L'_0) \wedge R(L_0^m) \wedge \neg u^{m-1}$ as the initialised unrolling of C', where $m \in [1, k-1)$ is the largest index for which $R(L_0^m)$ is satisfied. As we showed in Lemma 1, $b_0^m, ..., b_0^{k-1}$ are set to $\top$ while $b_0^0, ...b_0^{m-1}$ are $\bot$. Following Definition 10.3, $L_0^i \simeq F(X_0^{i-1}, L_0^{i-1})$ for all $i \in (m, k)$, while all components older than m are uninitialised. Every satisfying assignment of $R'(L'_0) \wedge R(L_0^m) \wedge \neg u^{m-1}$ also satisfies $\bigwedge_{i\in[0,k-m-1)} (L_{i+1} \simeq F(I_i, L_i)) \wedge R(L_0)$ with $I_{i-m} = X_0^i, L_{i-m} = L_0^i$ for all $i \in [m, k)$. In the rest of the proof, we fix the assignment satisfying $R'(L'_0) \wedge R(L_0^m) \wedge \neg u^{m-1}$. Similar to our argument in Theorem 1, $U_{k-1} \wedge R(L_0)$ is satisfiable with our fixed assignment. By our assumption, $\bigwedge_{i\in[m,k)} P(X_0^i, L_0^i)$ is then satisfied. We now consider $P'(I'_0, L'_0)$. As the premise of $p_2(I'_0, L'_0)$ is only satisfied for $b_0^m, ..., b_0^{k-1}$, $p_2(I'_0, L'_0)$ is satisfied. Similarly for the transition property, with $L_0^i \simeq F(X_0^{i-1}, L_0^{i-1})$ for all $i \in (m, k)$, $p_1(I'_0, L'_0)$ is satisfied. Given the values of B_0, the monotonicity property is satisfied. In addition, $p_4(I'_0, L'_0)$ is also satisfied as $b_0^{k-1} = \top$. Finally, the premise of $p_3(I'_0, L'_0)$ is only satisfied for $\neg b_0^{m-1} \wedge b_0^m$, and as we already have $R(L_0^m)$, p_3 is satisfied. □

Lemma 4. *If the initialised unrolling of length $k-1$ of the original circuit C is safe, the initialised unrolling of length 0 of the k-witness circuit C' is also safe, in the case of full initialisation.*

Proof. We assume $U_{k-1} \wedge R(L) \Rightarrow \bigwedge_{i\in[0,k)} P(I_i, L_i)$ for the original circuit. Since we consider full initialisation, $R'(L'_0) \wedge R(L_0^0) \wedge \neg u_0^1$ is the initialised unrolling of C'. Following Definition 10.3, $L_0^i \simeq F(X_0^{i-1}, L_0^{i-1})$ for all $i \in [1, k)$. Every satisfying assignment of $R'(L'_0) \wedge R(L_0^0) \wedge \neg u_0^1$ satisfies $U_{k-1} \wedge R(L_0)$ with $I_i = X_0^i, L_i = L_0^i$ for all $i \in [0, k)$. The rest of the proof follows the same logic as in Lemma 3. □

Lemma 5. *If the BMC check for the unrolling of length $k-1$ of the original circuit C passes, then the BMC check for the unrolling of length* 0 *of the k-witness circuit C' also passes.*

Proof. Based on Definition 10.3, we consider the BMC check for all possible initial states. Lemma 2, 3 and 4 cover the case-split over all initial states of C' based on whether each component satisfies the original reset function $R(L_0^i)$ or not. We show that the BMC check of C' passes under the same assumption for three initialisation cases respectively. In particular, our construction in Definition 10.3 does not allow all components to be uninitialised, in which case $R'(L'_0)$ becomes unsatisfiable (more specifically, $R'(L_0^0)$ is unsatisfiable). We conclude the BMC check of the initialised unrolling of length 0 passes in C'. □

We proceed to prove the opposite direction of the BMC check for C and C' by considering the reset status in the k-witness circuit.

Lemma 6. *If the BMC check for the unrolling of length 0 of the k-witness circuit C' passes, then the BMC check for the unrolling of length $k-1$ of the original circuit C also passes.*

Proof. We assume the BMC check passes in the k-witness circuit, $R'(L'_0) \Rightarrow P'(I'_0, L'_0)$. We do a proof by contradiction by assuming the BMC check of length $k-1$ fails for the original circuit. Thus there exists a satisfying assignment s of $U_{k-1} \wedge R(L_0) \wedge \neg \bigwedge_{i \in 0,k)} P(I_i, L_i)$. We can construct a satisfying assignment of $R'(L'_0)$ as follows. Let $a \in [0,k)$ be some index for which $\neg P(I_a, L_a)$ is satisfied. Let $m \in [0,a]$ be the index for which $R(L_m) \wedge \neg \bigvee_{i \in (m,a]} R(L_i)$ is satisfied. Let $X_0^{k-1-i} = I_{a-i}, L_0^{k-1-i} = L_{a-i}, b_0^{k-1-i} = \top$ for all $i \in [0, a-m]$. The rest of initialisation bits of B_0 are set to $\bot$. By Definition 2, we have $L_{i+1} \simeq F(I_i, L_i)$ for all $i \in [m, a)$, which satisfies Definition 10.3(d). As our construction satisfies $R'(L'_0)$, by our assumption, $P'(I'_0, L'_0)$ is satisfied. By Theorem 2, $P(I_a, L_a)$ is satisfied. Since we assume s satisfies $\neg P(I_a, L_a)$, we have reached a contradiction. $\square$

As an immediate consequence of Lemma 5 and 6, the BMC check of C passes iff the same check passes in C'. We record the result in the following Theorem.

Theorem 4. *The BMC check for the unrolling of length 0 of the k-witness circuit C' passes, if and only if the BMC check for the unrolling of length $k-1$ of the original circuit C passes.*

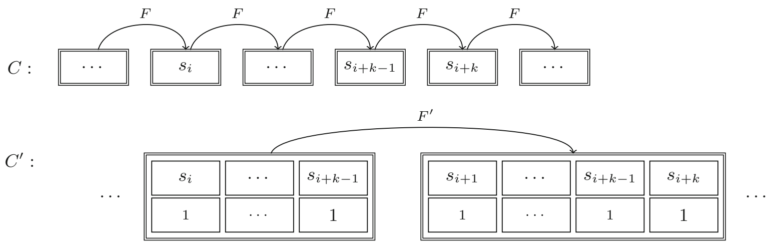


Fig. 5. The diagram shows the consecution check in C and C'.

We show in Fig. 5 an illustration of the consecution check in both circuits.

Theorem 5. *If the consecution check for the unrolling of length 1 of the k-witness circuit C' passes, then the consecution check for the unrolling of length k of the original circuit C passes too.*

Proof. We assume $U_1' \wedge P(I_0', L_0') \Rightarrow P(I_1', L_1')$ holds. We then do a proof by contradiction by assuming that the consecution check for the original circuit fails. Thus there is a satisfying assignment s of the formula $U_k \wedge \bigwedge_{i \in [0,k)} P(I_i, L_i) \wedge \neg P(I_k, L_k)$. Based on s, we have a satisfying assignment for $U_1' \wedge P'(I_0', L_0')$ as follows. Let $X_0^i = I_i, L_0^i = L_i$, and $b_0^i = \top$ for all $i \in [0,k)$. Let $X_1^{i-1} = I_i, L_1^{i-1} = L_i, b_1^{i-1} = \top$ for all $i \in [1,k]$. We now show this satisfies $L_1' \simeq F'(I_0', L_0')$. Since $X_1^{i-1} = I_i = X_0^i$ and $L_1^{i-1} = L_i = L_0^i$ for all $i \in [1,k)$, Definition 10.4(a) and Definition 10.4(c) are satisfied. Since s satisfies U_k, by Definition 2, it satisfies $L_k \simeq F(I_{k-1}, L_{k-1})$. With $X_0^{k-1} = I_{k-1}, L_0^{k-1} = L_{k-1}$, and $L_1^{k-1} = L_k$, we have $L_1^{k-1} \simeq F(X_0^{k-1}, L_0^{k-1})$, and thus Definition 10.4(b). As for the initialisation bits, since all of them are set to $\top$ in both B_0 and B_1, Definition 10.4(d) is satisfied. As a result, U_1' is satisfied, and we continue to show the same assignment satisfies $P'(I_0', L_0')$. Similar to our proof in Lemma 3, the values of B_0 satisfy $p_0(I_0', L_0')$ and $p_4(I_0', L_0')$ immediately. As the premiss of $p_3(I_0', L_0')$ is unsatisfiable, $p_3(I_0', L_0')$ trivially holds. Since U_k is satisfied, by Definition 2, we have $L_{i+1} \simeq F(I_i, L_i)$ which satisfies h_0^i for all $i \in [0, k-1)$, thus also $p_1(I_0', L_0')$. Lastly, since $P(I_i, L_i)$ is satisfied for all $i \in [0,k)$, the original property is satisfied in every component $P(X_0^i, L_0^i)$, resulting in the satisfaction of $p_2(I_0', L_0')$. By our initial assumption, $P'(I_1', I_1')$ is satisfied. By Theorem 2, we have $P(X_1^{k-1}, L_1^{k-1})$, thus $P(I_k, L_k)$. We reach a contradiction here. We can therefore conclude the consecution check of the original circuit passes. $\square$

Lemma 7. *If the safety property P is k-inductive in the original circuit C, the consecution check of the unrolling of length 1 passes in the k-witness circuit C', given that L_0' is partially initialised.*

Proof. Assume P is k-inductive in C. Let U_1' be the unrolling of C', and $m \in [1,k)$ is some index such that $b_0^0, ..., b_0^{m-1}$ are set to $\bot$, while $b_0^m, ..., b_0^{k-1}$ are set to $\top$ (as we consider partial initialisation here). We do a proof by contradiction, and assume there is a satisfying assignment s of the negation of the consecution check formula $U_1' \wedge P'(I_0', L_0') \wedge \neg P'(I_1', L_1')$. Since we assume $P'(I_0', L_0')$, it implies $R(L_0^m)$, based on $p_3(I_0', L_0')$. We also have $L_0^{i+1} \simeq F(X_0^i, L_0^i)$ for $i \in [m, k-1)$, based on $p_1(I_0', L_0')$. Furthermore, U_1' implies $L_1' \simeq F'(I_0', L_0')$, and by Definition 10.4, $L_1^{k-1} \simeq F(X_0^{k-1}, L_0^{k-1})$. Therefore the same assignment satisfies $U_{k-1} \wedge R(L_0)$ where $I_{i-m} = X_0^i, L_{i-m} = L_0^i$ for all $i \in [m,k)$, and $I_{k-m} = I_1', L_{k-m} = L_1^{k-1}$. By our assumption that the BMC check passes in C, we have $P(X_0^i, L_0^i)$ for all $i \in [m,k)$ and $P(I_1', L_1^{k-1})$.

We can then proceed to prove $P'(I_1', L_1')$ is indeed satisfied. Similar to our proof in Theorem 5, based on Definition 10.4, $b_1^i = \top$ for all $i \in [m,k)$ while $b_1^i = \bot$ for all $i \in [0,m)$. Additionally, $X_1^i = X_0^{i+1}, L_1^i = L_0^{i+1}$ for $i \in [0, m-1)$. The rest of the proof follows the same logic as Theorem 5 for showing $P'(I_1', L_1')$ is satisfied. We then reach a contradiction here, and thus conclude the consecution check for C' passes in this case. $\square$

Lemma 8. *If the consecution check for the unrolling of length k passes in the original circuit C, the consecution check for the unrolling of length 1 passes in the k-witness circuit C', given that L_0' is fully initialised.*

Proof. Let U_1' be the unrolling of C' with $b_0^0, ..., b_0^{k-1}$ all set to $\top$. Similar to Lemma 7, we do a proof by contradiction, and assume there is a satisfying assignment s of $U_1' \wedge P'(I_0', L_0') \wedge \neg P'(I_1', L_1')$. By the transition property $p_1(I_0', L_0')$, the components follow the transition function F, such that $L_1^{i+1} \simeq F(X_0^i, L_0^i)$ for all $i \in [0, k-1)$. Similar to our argument in Lemma 7, U_1' implies $L_1^{k-1} \simeq F(I_0', L_0^{k-1})$. We also have $\bigwedge\limits_{i\in[0,k)} P(X_0^i, L_0^i)$ based on $p_2(I_0', L_0')$ and the values of B_0. The same assignment thus satisfies $U_k \wedge \bigwedge\limits_{i\in[0,k)} P(L_i, L_i)$ where $L_i = L_0^i \wedge I_i = X_0^i$ for all $i \in [0, k)$ and $I_k = I_1', L_k = L_1^{k-1}$. Based on our assumption that the consecution of C passes, we have $P(I_1', L_1^{k-1})$. Following the same reasoning in Lemma 7, after one transition, $b_1^i = \top$ for all $i \in [0, k)$, and $X_1^i = X_0^{i+1}, L_1^i = L_0^{i+1}$ for $i \in [0, k-1)$.

We can now show $P'(I_1', L_1')$ is satisfied. The k-safety property $p_2(I_1', L_1')$ is satisfied as we have proved $p(X_1^i, L_1^i)$ for all $i \in [0, k)$. The transition property $p_1(I_0', L_0')$ is preserved, as U_k is satisfied which implies $L_1^{i+1} \simeq F(X_1^i, L_1^i)$. Based on the values of B_1, $p_0(I_1', L_1'), p_3(I_1', L_1'), p_4(I_1', L_1')$ are satisfied immediately. We conclude the $P'(I_1', L_1')$ is satisfied thus we reach a contradiction. Therefore the consecution check for C' passes in this case. □

Theorem 6. *If both k-induction checks pass in the original circuit C, then the consecution check of the unrolling of length 1 in the k-witness circuit C' passes.*

Proof. First of all, we assume both checks pass in C. We then do a proof by contradiction by assuming there is a satisfying assignment s for the negation of the consecution check $U_1' \wedge P'(I_0', L_0') \wedge \neg P'(I_1', L_1')$. Since s satisfies $U_1' \wedge P'(I_0', L_0')$, we consider two separate cases where the property $P'(I_0', L_0')$ is satisfied: full initialisation or partial initialisation. Note when all $b_0^0, ..., b_0^{k-1}$ are set to $\bot$, $P'(I_0', L_0')$ is not satisfied. Therefore applying Lemma 8 and Lemma 7 together, we conclude if both k-induction checks pass in C, the consecution check of the unrolling of length 1 in the k-witness circuit also passes. □

We briefly discuss why the k-witness circuit is linear in the size of the original circuit, and the value k. If we consider the circuit size in terms of gate numbers, the number of latches and inputs increase by a factor of approximately k. The transition functions are copied $k-1$ times, i.e., $k-2$ times for reset in Definition 10.3(d), and once more in 10.4(b), while the $k-2$ copies in the property part 10.5(a) have the same arguments and can be shared. For the reset predicates, defining $R(L^i)$ is linear in the number of the latches, while u^i is linear in k. We apply the same logic when defining the property, therefore we conclude our construction is linear in the size of the circuit and k.

5 Implementation

Based on our new construction we implemented CERTIFAIGER [12], which works as a tool suite comprised of multiple components as shown in Fig. 6. The tool takes as inputs a circuit which contains a safety property given in AIGER format [7] and a value k provided by a k-induction-based model checker which outputs a positive model checking result. Upon invocation, internally the inputs are passed on to the k-witness generator that parses the AIGER file and generates a k-witness circuit as defined in Definition 10. The new safety property is a simple inductive invariant (to be verified) for the k-witness circuit. We extended the reset logic definition of the existing AIGER format defined by the authors of [7] to enable reset functions, whereas all previous AIGER versions only allow reset values to be 0, 1, or uninitialised. The k-witness circuit from the k-witness generator is given in this extended AIGER format.

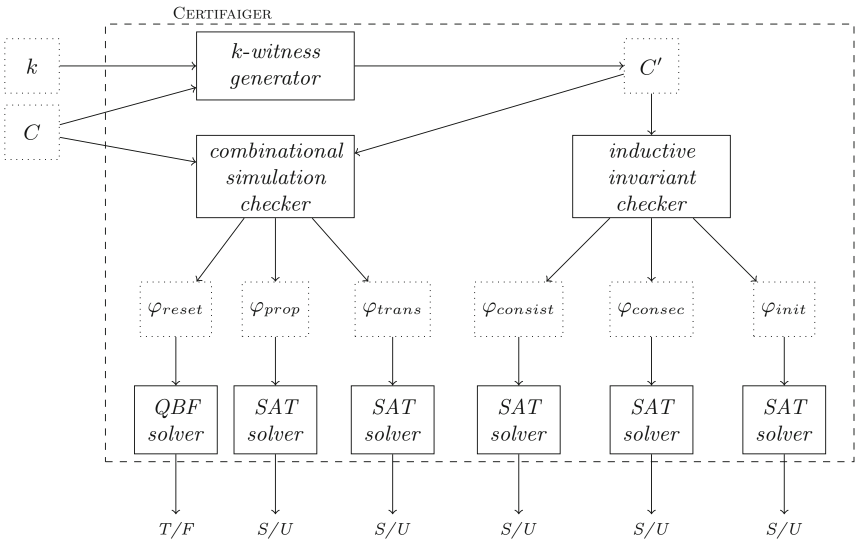


Fig. 6. The architecture of Certifaiger. C is the input circuit in AIGER format and k is the value given by a k-induction-based model checker. The final outputs of the SAT solvers are given in the form of S/U, for satisfiable or unsatisfiable. The QBF solver outputs true or false (T/F) as the result.

To verify the inductive invariant $\phi(I, L)$, as discussed in Sect. 3, our certifier generates three conditions. (Note that here we are only looking at extended circuits, therefore we use L instead of L'.)

Condition	Formula	The inductive invariant ...
"*initiation*"	$R(L) \Rightarrow \phi(I, L)$	... must hold at all initial states
"*consistency*"	$\phi(I, L) \Rightarrow P(I, L)$	... must hold at all good states
"*consecution*"	$U_1 \wedge \phi(I_0, L_0) \Rightarrow \phi(I_1, L_1)$	... is preserved during the transition

In our implementation, the latch variables used in the inductive invariant are updated with their next state literals after each transition. The consistency condition is rather trivial here, as the inductive invariant is exactly the property in the k-witness circuit, although this is only specific to our case.

Our certifier generates for each of the three conditions a (combinational) AIGER circuit which is then checked by a SAT solver. In our implementation, we used the SAT solver Kissat [6] for checking validity of the formulas after they have been converted to CNF by invoking AIGTOCNF from the AIGER library.

Furthermore, we implemented the combinational simulation checker for verifying the combinational simulation relation described in Definition 7. The checker takes as inputs the original circuit and the k-witness circuit. It generates two AIGER files for the transition check and the property check, as well as a QAIGER file for the reset check, as defined in Definition 7. Similar to the inductive invariant checker, the AIGER files are then converted to CNFs and verified by Kissat. QAIGER is a standard format used in QBF Competitions. In our experiments the formula is verified with the QBF solver QuAbS [35].

The tool CERTIFAIGER returns "SUCCESS" as a result if all six formulas hold, meaning that the circuit C' combinationally simulates C and C' is safe by the 1-induction proof. Thus by Theorem 1 the original circuit C is also safe. Note that this result holds regardless of how C' is constructed.

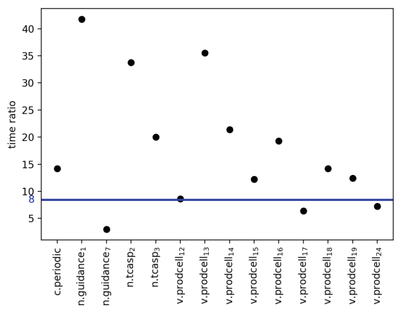
Given a scenario where we would want to place trust on the correctness of the extended circuit mapping inside the k-witness generator (to trust that the k-witness circuit construction of Definition 10 is correct and the program implementing it is also provably correct), all three combinational simulation checks (one QBF and two SAT checks) could be skipped in the certification procedure.

Intuitively, given a faulty generation of the k-witness circuit C', the error would either be caught by the combinational simulation check (due to an erroneous under-approximation of the set of reachable states) or the inductive invariant check (due to an erroneous over-approximation of the set of reachable states). Furthermore, we have also done a sanity check of certification on failure, where the model checking results are falsified by CERTIFAIGER. An incorrect value of k is detected by a negative result of φ_{consec}, whereas φ_{init} does not hold in cases where an initial state is a bad state.

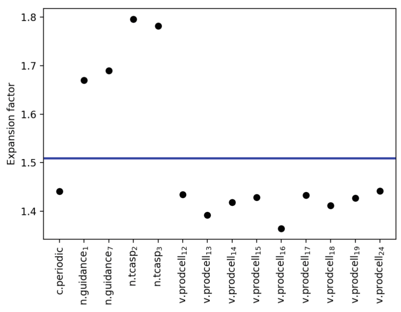
6 Experiments

As described in previous sections, the complexity of extending the original circuit into k-witness is linear in the size of the circuit, and the inductive depth. To evaluate the practicality of our tool, we now report the experimental results obtained by evaluating CERTIFAIGER against a number of widely used benchmarks. The benchmarks were first run on the open source k-induction-based model checker McAiger [3], which was modified to give the values of k explicitly. All experiments were carried out on an Intel® Core™ i9-9900 CPU 3.60 GHz computer with 32 GB RAM running Manjaro with kernel version 5.4.72-1.

We start with the TIP suite benchmarks which were originally used in [18]. The benchmarks were converted from .SMV to AIGER by invoking SMVTOAIG from the AIGER library. Table 1 reports the certification results obtained, where the file names are associated by the origin of the problems explained in [18]. The table displays the following information in each column:



(a)



(b)

Fig. 7. The time (a) and file size (b) comparison results for the TIP suite. The benchmark names are shown on the x-axis. Average values are shown as the blue horizontal line in each plot. The y-axis of (a) displays the time ratio of total certification time and model checking time. The y-axis of (b) shows the expansion factor indicating the comparison of circuit sizes (k-witness circuit v.s the original). (Color figure online)

1. the name of the AIGER file,
2. the verification time on McAiger,
3. the size of the original circuit, in terms of the number of gates (thousands),
4. the k-inductive value k given by the model checker,
5. the size of the k-witness circuit,
6. the time taken on the k-witness generator, and
7. the size and solving time (seconds) of each condition.

Note here we selected benchmarks that gave a positive model checking result, only in which case the original property is k-inductive. Moreover, three instances that require simple paths constraints (also called *loopFree* constraints in [34]) were ruled out. Handling these constraints is an interesting area for future study. We retrieved the inductive depths k from the model checker McAiger, and compared with the results in [18] to ensure the values are identical. As shown in Table 1, the values of k vary between 4 and 96. The SAT solver was able to handle the proof checking without experiencing time-outs. We observe that the k-witness circuit generation time is rather small, compared with the model checking time as well as the proof checking time. In the proof checking stage, Table 1 suggests that the SAT-solving time for φ_{consec} is much higher than the rest of the formulas. This is as expected, as the formula φ_{consec} is in general more complicated than the rest, and appears to be the most difficult formula to solve. In addition, QBF solving times are also worth-noting: in a few cases QBF solving time is longer than for other formulas, however, in most cases, it is rather small. To compare certification time with model checking time, we plotted the results in Fig. 7a, where the y-axis shows the ratio of certification and model checking.

Table 1. Experimental results for the TIP suite.

						φ_{init}		$\varphi_{consist}$		φ_{consec}		φ_{trans}		φ_{prop}		φ_{reset}	
Name	t	$\#C$	k	$\#C'$	t'	#	t	#	t	#	t	#	t	#	t	#	t
c.periodic	6.01	1.56	96	215.79	0.06	242.91	5.62	215.79	0.06	424.80	57.54	217.42	0.15	217.28	0.06	216.06	85.25
n.guidance$_1$	0.08	1.91	10	31.89	0.01	38.39	0.21	31.89	0.01	62.16	3.34	33.97	0.12	33.63	0.01	32.58	1.24
n.guidance$_7$	8.57	2.00	27	91.22	0.03	109.35	3.58	91.216	0.02	177.90	18.24	93.39	0.12	93.04	0.02	91.90	25.61
n.tcasp$_2$	0.08	3.02	6	32.54	0.01	39.76	0.16	32.542	0.01	63.28	2.70	35.92	0.26	35.23	0.02	33.92	1.84
n.tcasp$_3$	0.09	2.98	5	24.23	0.01	32.47	0.13	26.56	0.01	51.63	1.80	29.90	0.26	29.21	0.02	27.94	1.04
v.prodcell$_{12}$	7.60	2.91	29	121.07	0.04	133.59	2.66	121.07	0.03	239.01	65.76	124.19	0.12	123.88	0.03	121.69	8.61
v.prodcell$_{13}$	0.10	2.91	8	32.41	0.01	35.78	0.20	32.41	0.01	63.97	3.55	35.52	0.12	35.21	0.01	33.03	0.21
v.prodcell$_{14}$	0.81	2.91	16	66.03	0.02	72.88	0.73	66.03	0.02	130.34	17.30	69.14	0.12	68.83	0.02	66.65	1.48
v.prodcell$_{15}$	2.94	2.91	23	95.60	0.03	105.51	2.05	95.60	0.03	188.74	35.87	98.72	0.12	98.41	0.02	96.23	4.34
v.prodcell$_{16}$	0.07	2.91	5	19.85	0.01	21.91	0.04	19.85	0.01	39.18	1.35	22.97	0.12	22.66	0.01	20.47	0.06
v.prodcell$_{17}$	7.04	2.91	27	112.57	0.03	124220	2.46	112.57	0.03	222.22	44.88	115.68	0.12	115.37	0.03	113.19	6.95
v.prodcell$_{18}$	0.62	2.91	13	53.40	0.02	58.95	0.54	53.40	0.02	105.41	8.79	56.51	0.12	56.20	0.02	54.02	0.81
v.prodcell$_{19}$	2.67	2.91	22	91.37	0.03	100.84	1.99	91.37	0.03	180.37	33.09	94.48	0.12	94.17	0.03	91.99	3.81
v.prodcell$_{24}$	16.38	2.91	37	155.23	0.05	171.24	3.45	155.23	0.04	306.46	118.50	158.35	0.12	158.04	0.04	155.85	17.78

Here certification time is the sum of time taken on each component, assuming the six conditions are computed in parallel. As shown in the diagram, the average time ratio is around 8, which is quite promising. Furthermore, Fig. 7b shows a comparison of circuit sizes, where the *expansion factor* ε is computed by $\frac{\#C'}{\#C \times k}$(alternatively, $\#C' = \varepsilon \cdot \#C \cdot k$). The average value observed here is around 1.5. This is consistent with Definition 10, as we expected the size of the k-witness circuit to grow linearly with respect to the original circuit and the value of k.

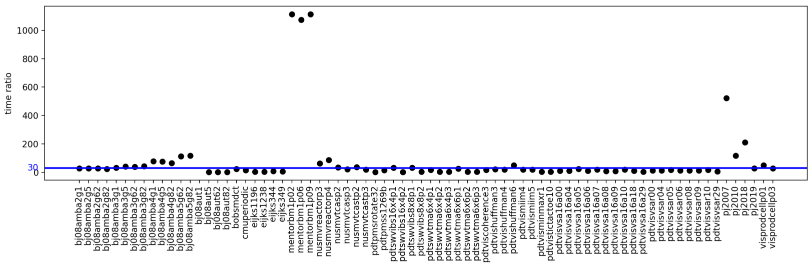


Fig. 8. Certification time *vs.* model checking time obtained by running HWMCC'10 benchmarks.

We also used benchmarks from the Hardware Model Checking Competition (HWMCC) 2010 [4]. The benchmarks were pre-filtered by running on McAiger with a time-out of 15 min. A total of 513 instances were solved by McAiger, from which we selected from the 216 *unsat* instances with a meaningful k (*i.e.*, $k \geq 2$). We also observed only 7 out of the 216 instances require simple path constraints. The results in Fig. 8 are sorted by the benchmark names, which enables us to compare individual benchmarks from the same family. In most cases, similar to our previous observation from the TIP suite, the SAT solving

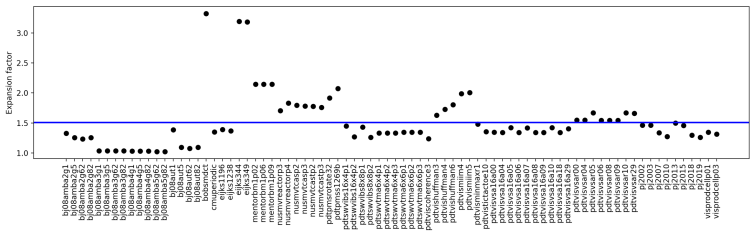


Fig. 9. The k-witness circuit size *vs.* the original circuit size.

time of φ_{consec} takes much longer than the rest, while in very few cases it is less than the QBF solving time for φ_{reset}. The average time ratio is 30, where we excluded 4 outliers in the plot coming from the $pj20$ family, that give a worse result (total certification time $\geq$15 min). We observe that this was due to the high format conversion time from QAIGER to QCIR [25] before the QBF solving handled by QuAbS, while the actual QBF solving time was significantly smaller and more feasible. We believe this can be overcome by generating an alternative format directly in practice. Finally, similar to our previous TIP results, Fig. 9 shows the values of the expansion factor with an average of 1.5.

In the final experiments, to further inspect the expansion factor, we generalise the *Counter* example in Example 1, where we scale the number of bits to 500 with a modulo value 32. To clarify the complexity of our construction for the k-witness circuit, we ran experiments with different values of k. The results are shown in Fig. 10, where the x-axis shows the values of b up to 431, meaning the value of k was scaled up to 400. The expansion factor gradually converges to a constant as we increase the value of b, as we expected.

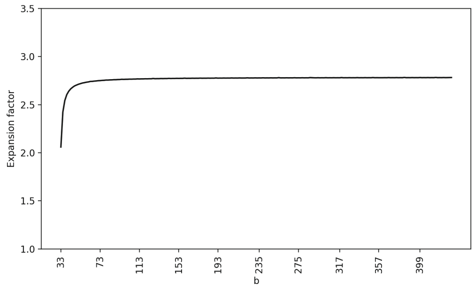


Fig. 10. The experimental results of the *Counter* example. The values of b are shown on the x-axis.

As noticed above, overall our approach works efficiently in the certification stage, in particular, in our implementation we adopted the linear construction of k-witness circuit in Definition 10, thus the size of the resulting AIGER circuit is linear in the size of the original circuit, and the value of k. Each component in the tool suite works independently from each other when performing verification, which increases trust in the verification results.

7 Conclusion

We propose an approach to certify k-induction-based model checking results, by extending the model to produce an inductive invariant. The resulting tool, Certifaiger, was evaluated experimentally on multiple sets of widely used benchmarks. The analysis showed our approach can be adapted to use in practice.

Our certificates are linear in size of the original problem and k. Validation requires several SAT checks and solving a simple QBF. In related work [8,23] the worst case is considered to be exponential. It is an interesting open question whether our notion of combinational simulation requiring a QBF check for the reset condition can be changed to use only SAT checks.

Further, we only considered k-induction without simple paths constraints, even though such constraints on executions of the original model can in principle be handled by adding unique state constraints to our k-witness circuit. For simplicity we stick to models without such constraints, a restriction also made for instance in the hardware model checking competition. Thus certifying k-induction with simple path constraints is left to future work as well as handling different types of properties such as liveness properties.

We also want to extend our approach to common preprocessing techniques including temporal decomposition [11] or retiming [28] with the goal to obtain a single certificate (witness circuit). This goal is particularly challenging for complex multi-engine model checkers [9,10]. Furthermore, we believe our approach can be extended to infinite-state systems, where k-induction is commonly used.

Acknowledgement. This work is supported by the Austrian Science Fund (FWF) under the project W1255-N23, the LIT AI Lab funded by the State of Upper Austria, and Academy of Finland under the project 336092.

References

1. Abadi, M., Lamport, L.: The existence of refinement mappings. In: LICS, pp. 165–175. IEEE Computer Society (1988)
2. Balyo, T., Heule, M.J.H., Järvisalo, M.: SAT competition 2016: recent developments. In: Singh, S.P., Markovitch, S. (eds.) Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, pp. 5061–5063. AAAI Press (2017)
3. Biere, A., Brummayer, R.: Consistency checking of all different constraints over bit-vectors within a SAT solver. In: FMCAD, pp. 1–4. IEEE (2008)
4. Biere, A., Claessen, K.: Hardware model checking competition 2010 (2010). http://fmv.jku.at/hwmcc10/

5. Biere, A., van Dijk, T., Heljanko, K.: Hardware model checking competition 2017. In: Stewart, D., Weissenbacher, G. (eds.) Formal Methods in Computer-Aided Design, FMCAD, p. 9. IEEE (2017)
6. Biere, A., Fazekas, K., Fleury, M., Heisinger, M.: CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT competition 2020. In: Balyo, T., Froleyks, N., Heule, M., Iser, M., Järvisalo, M., Suda, M. (eds.) Proc. of SAT Competition 2020 - Solver and Benchmark Descriptions. Department of Computer Science Report Series B, vol. B-2020-1, pp. 51–53. University of Helsinki (2020)
7. Biere, A., Heljanko, K., Wieringa, S.: AIGER 1.9 and beyond. Tech. rep. 11/2, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria (2011)
8. Bjørner, N., Gurfinkel, A., McMillan, K., Rybalchenko, A.: Horn clause solvers for program verification. In: Beklemishev, L.D., Blass, A., Dershowitz, N., Finkbeiner, B., Schulte, W. (eds.) Fields of Logic and Computation II. LNCS, vol. 9300, pp. 24–51. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23534-9_2
9. Brayton, R., Mishchenko, A.: ABC: an academic industrial-strength verification tool. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 24–40. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_5
10. Cabodi, G., Nocco, S., Quer, S.: Thread-based multi-engine model checking for multicore platforms. ACM Trans. Des. Autom. Electr. Syst. **18**(3), 36:1–36:28 (2013)
11. Case, M.L., Mony, H., Baumgartner, J., Kanzelman, R.: Enhanced verification by temporal decomposition. In: FMCAD, pp. 17–24. IEEE (2009)
12. Certifaiger: Certifaiger (2021). http://fmv.jku.at/certifaiger
13. Champion, A., Mebsout, A., Sticksel, C., Tinelli, C.: The Kind 2 model checker. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 510–517. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_29
14. Conchon, S., Mebsout, A., Zaïdi, F.: Certificates for parameterized model checking. In: Bjørner, N., de Boer, F. (eds.) FM 2015. LNCS, vol. 9109, pp. 126–142. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19249-9_9
15. Degtyarev, A., Voronkov, A.: Equality reasoning in sequent-based calculi. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning (in 2 volumes), pp. 611–706. Elsevier and MIT Press (2001)
16. Donaldson, A.F., Haller, L., Kroening, D., Rümmer, P.: Software verification using k-induction. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 351–368. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23702-7_26
17. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 61–75. Springer, Heidelberg (2005). https://doi.org/10.1007/11499107_5
18. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electron. Notes Theor. Comput. Sci. **89**(4), 543–560 (2003)
19. Gacek, A., Backes, J., Whalen, M., Wagner, L., Ghassabani, E.: The JKind model checker. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10982, pp. 20–27. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96142-2_3
20. Ge, N., Jenn, E., Breton, N., Fonteneau, Y.: Integrated formal verification of safety-critical software. Int. J. Softw. Tools Technol. Transf. **20**(4), 423–440 (2018)
21. Griggio, A., Roveri, M., Tonetta, S.: Certifying proofs for LTL model checking. In: FMCAD, pp. 1–9. IEEE (2018)
22. Große, D., Le, H.M., Drechsler, R.: Induction-based formal verification of SystemC TLM designs. In: MTV, pp. 101–106. IEEE Computer Society (2009)

23. Gurfinkel, A., Ivrii, A.: K-induction without unrolling. In: FMCAD, pp. 148–155. IEEE (2017)
24. Heule, M.J.H., Järvisalo, M., Suda, M.: SAT competition 2018. J. Satisf. Boolean Model. Comput. **11**(1), 133–154 (2019)
25. Jordan, C., Klieber, W., Seidl, M.: Non-cnf QBF solving with QCIR. In: AAAI Workshop: Beyond NP. AAAI Workshops, vol. WS-16-05. AAAI Press (2016)
26. Jovanovic, D., Dutertre, B.: Property-directed k-induction. In: FMCAD, pp. 85–92. IEEE (2016)
27. Vediramana Krishnan, H.G., Vizel, Y., Ganesh, V., Gurfinkel, A.: Interpolating strong induction. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11562, pp. 367–385. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25543-5_21
28. Kuehlmann, A., Baumgartner, J.: Transformation-based verification using generalized retiming. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 104–117. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44585-4_10
29. Kuismin, T., Heljanko, K.: Increasing confidence in liveness model checking results with proofs. In: Bertacco, V., Legay, A. (eds.) HVC 2013. LNCS, vol. 8244, pp. 32–43. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-03077-7_3
30. Lahtinen, J., Valkonen, J., Björkman, K., Frits, J., Niemelä, I., Heljanko, K.: Model checking of safety-critical software in the nuclear engineering domain. Reliab. Eng. Syst. Saf. **105**, 104–113 (2012)
31. Mishchenko, A., Brayton, R.K.: Recording synthesis history for sequential verification. In: FMCAD, pp. 1–8. IEEE (2008)
32. de Moura, L., et al.: SAL 2. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 496–500. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27813-9_45
33. Namjoshi, K.S.: Certifying model checkers. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 2–13. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44585-4_2
34. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: Hunt, W.A., Johnson, S.D. (eds.) FMCAD 2000. LNCS, vol. 1954, pp. 127–144. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-40922-X_8
35. Tentrup, L.: Non-prenex QBF solving using abstraction. In: Creignou, N., Le Berre, D. (eds.) SAT 2016. LNCS, vol. 9710, pp. 393–401. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40970-2_24
36. Wagner, L., Mebsout, A., Tinelli, C., Cofer, D., Slind, K.: Qualification of a model checker for avionics software verification. In: Barrett, C., Davies, M., Kahsai, T. (eds.) NFM 2017. LNCS, vol. 10227, pp. 404–419. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57288-8_29
37. Yu, Z., Biere, A., Heljanko, K.: Certifying hardware model checking results. In: Ait-Ameur, Y., Qin, S. (eds.) ICFEM 2019. LNCS, vol. 11852, pp. 498–502. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32409-4_32

Model-Checking Structured Context-Free Languages

Michele Chiari[1(✉)], Dino Mandrioli[1], and Matteo Pradella[1,2]

[1] DEIB, Politecnico di Milano, Milan, Italy
{michele.chiari,dino.mandrioli,
matteo.pradella}@polimi.it
[2] IEIIT, Consiglio Nazionale delle Ricerche, Milan, Italy

Abstract. The problem of model checking procedural programs has fostered much research towards the definition of temporal logics for reasoning on context-free structures. The most notable of such results are temporal logics on Nested Words, such as CaRet and NWTL. Recently, the logic OPTL was introduced, based on the class of Operator Precedence Languages (OPL), more powerful than Nested Words. We define the new OPL-based logic POTL, and provide a model checking procedure for it. POTL improves on NWTL by enabling the formulation of requirements involving pre/post-conditions, stack inspection, and others in the presence of exception-like constructs. It improves on OPTL by being FO-complete, and by expressing more easily stack inspection and function-local properties. We developed a model checking tool for POTL, which we experimentally evaluate on some interesting use-cases.

Keywords: Linear temporal logic · Operator precedence languages · Model Checking · Visibly pushdown languages · Input-driven languages

1 Introduction

Model checking is one of the most successful techniques for the verification of software programs. It consists in the exhaustive verification of the mathematical model of a program against a specification of its desired behavior. The kind of properties that can be proved in this way depends both on the formalism employed to model the program, and on the one used to express the specification. The initial and most classical frameworks consist in the use of operational formalisms, such as Transition Systems and Finite State Automata (generally Büchi automata) for the model, and temporal logics such as Linear-time Temporal Logic (LTL), Computation-Tree Logic (CTL) and CTL* for the specification [24]. The success of such logics is due to their ease in reasoning about linear or branching sequences of events over time, by expressing liveness and safety properties, their conciseness with respect to automata, and the complexity of their model checking.

In this paper we consider linear-time temporal domains. LTL limits its set of expressible properties to the First-Order Logic (FOL) definable fragment of regular languages. This is quite restrictive when compared with the most popular abstract models of procedural programs, such as Pushdown Systems, Boolean Programs [10], and Recursive State Machines [3]. All such stack-based formalisms show behaviors that are expressible by means of Context-Free Languages (CFL), rather than regular ones. State and configuration reachability, fair computation problems, and model checking of *regular specifications* have been thoroughly studied for such formalisms [3,4,13,17,28,30,32,40,51,55]. To expand the expressive power of specification languages too, [12,14] augmented LTL with Presburger arithmetic constraints on the occurrences of states, obtaining a logic capable of even some context-sensitive specifications, but with only restricted decidable fragments. [41] introduced model checking of pushdown tree automata specifications on regular systems, and Dynamic Logic was extended to some limited classes of CFL [34]. Decision procedures for different kinds of regular constraints on stack contents have been given in [18,29,37].

A coherent approach came with the introduction of temporal logics based on Visibly Pushdown Languages (VPL) [7], a.k.a. Input-Driven Languages [47]. Such logics, namely CaRet [6] and its FO-complete successor NWTL [2], model the execution trace of a procedural program as a Nested Word [8], consisting in a linear ordering augmented with a one-to-one matching relation between function calls and returns. They are the first ones featuring temporal modalities that explicitly refer to the nesting structure of CFL [4]. This enables requirement specifications to include Hoare-style pre/post-conditions, stack-inspection properties, and more. A μ-calculus based on VPL extends model checking to branching-time semantics in [5], while [16] introduces a temporal logic capturing the whole class of VPL. Timed extensions of CaRet are given in [15].

VPL too have their limitations. They are more general than Parenthesis Languages [46], but their *matching relation* is essentially constrained to be one-to-one [43]. This hinders their suitability to model processes in which a single event must be put in relation with multiple ones. Unfortunately, computer programs often present such behaviors: exceptions and continuations are single events that cause the termination (or re-instantiation) of multiple functions on the stack.

To reason about such behaviors, temporal logics based on Operator Precedence Languages (OPL) have been proposed [22]. OPL were initially introduced with the purpose of efficient parsing [31], a field in which they continue to offer useful applications [11]. They are capable of capturing the syntax of arithmetic expressions, and other constructs whose context-free structure is not immediately visible. The generality of the structure of their syntax trees is much greater than that of VPL, which are strictly included in OPL [25]. Nevertheless, they retain the same closure properties that make regular languages and VPL suitable for automata-theoretic model checking: OPL are closed under Boolean operations, concatenation, Kleene *, and language emptiness and inclusion are decidable [42]. They have been characterized by means of push-down automata, Monadic Second-Order Logic and, recently, by an extension of Regular Expressions [42,44].

OPTL [22] is the first linear-time temporal logic for which a model checking procedure has been given on both finite and ω-words of OPL. It enables reasoning on procedural programs with exceptions, expressing properties about whether a function can be terminated by an exception, or throw one, and also pre/post-conditions. NWTL can be translated into OPTL in linear time, thus the latter is capable of expressing all properties that can be formalized in CaRet and NWTL, and many more. [22] does not explore OPTL's expressiveness further, and does not investigate the practical applicability of their model checking construction.

In this article, we introduce Precedence Oriented Temporal Logic (POTL), which redefines the syntax and semantics of OPTL to be much closer to the context-free structure of words. With POTL, it is much easier to navigate a word's syntax tree, expressing requirements that are aware of its structure. From a more theoretical point of view, POTL is FO-complete whereas OPTL is not, so that CaRet, NWTL, OPTL and POTL constitute a strict hierarchy in terms of expressive power. Such a theoretical elaboration, however, is technically involved; thus, for length reasons, it is documented in a technical report [23].

In this paper, instead, we focus on the model-checking application of POTL. We provide a tableaux-construction procedure for model checking POTL, which yields nondeterministic automata of size at most singly exponential in the formula's length, and is thus not asymptotically greater than that of LTL, NWTL and OPTL. We implemented such a procedure in a tool called POMC, which we evaluate on several interesting case studies. POMC's performance is promising: almost all case studies are verified in seconds and with a reasonable memory consumption, with very few outliers. Such outliers are inevitable, due to the exponential complexity of the task.

The related work on tools is not as rich as the theoretical one. Tools and libraries such as VPAlib [48], VPAchecker [54], OpenNWA [27] and SymbolicAutomata [26] only implement operations such as union, intersection, universality/inclusion/emptiness check for Visibly Pushdown or Nested Word Automata, but have no model checking capabilities. PAL [19] uses nested-word based monitors to express program specifications, and a tool based on BLAST [36] implements its runtime monitoring and model checking. PAL follows the paradigm of program monitors, and is not—strictly speaking—a temporal logic. PTCaRet [52] is a past version of CaRet, and its runtime monitoring has been implemented in JavaMOP [20]. [49,50] describe a tool for model checking programs against CaRet specifications. Since its purpose is malware detection, it targets program binaries directly by modeling them as Pushdown Systems. Unfortunately, this tool does not seem to be available online. To the best of our knowledge, POMC is the only publicly-available[1] tool for model-checking temporal logics capable of expressing context-free properties.

The paper is organized as follows: we give some background on OPL in Sect. 2, we introduce POTL in Sect. 3 and its model checking in Sect. 4, and we evaluate our prototype model checker in Sect. 5. Due to space constraints, we leave all formal proofs to a technical report [21].

[1] https://github.com/michiari/POMC.

2 Operator Precedence Languages

We assume some familiarity with classical formal language theory concepts such as context-free grammar, parsing, shift-reduce algorithm, syntax tree (ST) [33,35]. Operator Precedence Languages (OPL) are usually defined through their generating grammars [31]; in this paper, however, we characterize them through their accepting automata [42] which are the natural way for stating equivalence properties with logic characterization, and for model checking. Readers not familiar with OPL may refer to [43] for more explanations on the following basic concepts; an explanatory example is also given at the end of this section.

Let Σ be a finite alphabet, and ε the empty string. We use a special symbol $\# \notin \Sigma$ to mark the beginning and the end of any string. An *operator precedence matrix* (OPM) M over Σ is a partial function $(\Sigma \cup \{\#\})^2 \to \{\lessdot, \doteq, \gtrdot\}$, that, for each ordered pair (a, b), defines the *precedence relation* (PR) $M(a, b)$ holding between a and b. If the function is total we say that M is *complete*. We call the pair (Σ, M) an *operator precedence alphabet*. Relations $\lessdot, \doteq, \gtrdot$, are respectively named *yields precedence, equal in precedence*, and *takes precedence*. By convention, the initial # yields precedence, and other symbols take precedence on the ending #. If $M(a, b) = \pi$, where $\pi \in \{\lessdot, \doteq, \gtrdot\}$, we write $a \; \pi \; b$. For $u, v \in \Sigma^+$ we write $u \; \pi \; v$ if $u = xa$ and $v = by$ with $a \; \pi \; b$. The role of PR is to give structure to words: they can be seen as special and more concise parentheses, where e.g. one "closing" $\gtrdot$ can match more than one "opening" $\lessdot$. Despite their graphical appearance, PR are not ordering relations.

Definition 1. *An* operator precedence automaton (OPA) *is a tuple* $\mathcal{A} = (\Sigma, M, Q, I, F, \delta)$ *where:* (Σ, M) *is an operator precedence alphabet,* Q *is a finite set of states (disjoint from* Σ*),* $I \subseteq Q$ *is the set of initial states,* $F \subseteq Q$ *is the set of final states,* $\delta \subseteq Q \times (\Sigma \cup Q) \times Q$ *is the transition relation, which is the union of the three disjoint relations* $\delta_{shift} \subseteq Q \times \Sigma \times Q$*,* $\delta_{push} \subseteq Q \times \Sigma \times Q$*, and* $\delta_{pop} \subseteq Q \times Q \times Q$*. An OPA is deterministic iff* I *is a singleton, and all three components of* δ *are—possibly partial—functions.*

To define the semantics of OPA, we need some new notations. Letters $p, q, p_i, q_i, \ldots$ denote states in Q. We use $q_0 \xrightarrow{a} q_1$ for $(q_0, a, q_1) \in \delta_{push}$, $q_0 \overset{a}{\dashrightarrow} q_1$ for $(q_0, a, q_1) \in \delta_{shift}$, $q_0 \overset{q_2}{\Longrightarrow} q_1$ for $(q_0, q_2, q_1) \in \delta_{pop}$, and $q_0 \overset{w}{\rightsquigarrow} q_1$, if the automaton can read $w \in \Sigma^*$ going from q_0 to q_1. Let $\Gamma = \Sigma \times Q$ and $\Gamma' = \Gamma \cup \{\bot\}$ be the *stack alphabet*; we denote symbols in Γ' as $[a,\ q]$ or $\bot$. We set $smb([a,\ q]) = a$, $smb(\bot) = \#$, and $st([a,\ q]) = q$. For a stack content $\gamma = \gamma_n \ldots \gamma_1 \bot$, with $\gamma_i \in \Gamma$, $n \geq 0$, we set $smb(\gamma) = smb(\gamma_n)$ if $n \geq 1$, $smb(\gamma) = \#$ if $n = 0$.

A *configuration* of an OPA is a triple $c = \langle w,\ q,\ \gamma \rangle$, where $w \in \Sigma^* \#$, $q \in Q$, and $\gamma \in \Gamma^* \bot$. A *computation* or *run* is a finite sequence $c_0 \vdash c_1 \vdash \ldots \vdash c_n$ of *moves* or *transitions* $c_i \vdash c_{i+1}$. There are three kinds of moves, depending on the PR between the symbol on top of the stack and the next input symbol:
Push move: if $smb(\gamma) \lessdot \ a$ then $\langle ax,\ p,\ \gamma \rangle \vdash \langle x,\ q,\ [a,\ p]\gamma \rangle$, with $(p, a, q) \in \delta_{push}$;
Shift move: if $a \doteq b$ then $\langle bx,\ q,\ [a,\ p]\gamma \rangle \vdash \langle x,\ r,\ [b,\ p]\gamma \rangle$, with $(q, b, r) \in \delta_{shift}$;
Pop move: if $a \gtrdot b$ then $\langle bx,\ q,\ [a,\ p]\gamma \rangle \vdash \langle bx,\ r,\ \gamma \rangle$, with $(q, p, r) \in \delta_{pop}$.

	call	**ret**	**han**	**exc**
call	$\lessdot$	$\doteq$	$\lessdot$	$\gtrdot$
ret	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
han	$\lessdot$	$\gtrdot$	$\lessdot$	$\doteq$
exc	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$

$\#[\mathbf{call}[[[\mathbf{han}[\mathbf{call}[\mathbf{call}[\mathbf{call}]]]\mathbf{exc}]\mathbf{call\ ret}]\mathbf{call\ ret}]\mathbf{ret}]\#$

Fig. 1. OPM $M_{\mathbf{call}}$ (left) and a string with chains shown by brackets (right).

Shift and pop moves are not performed when the stack contains only $\bot$. Push moves put a new element on top of the stack consisting of the input symbol together with the current state of the OPA. Shift moves update the top element of the stack by *changing its input symbol only*. Pop moves remove the element on top of the stack, and update the state of the OPA according to δ_{pop} on the basis of the current state of the OPA and the state of the removed stack symbol. They do not consume the input symbol, which is used only to establish the $\gtrdot$ relation, remaining available for the next move. The OPA accepts the language $L(\mathcal{A}) = \{x \in \Sigma^* \mid \langle x\#,\ q_I,\ \bot\rangle \vdash^* \langle \#,\ q_F,\ \bot\rangle, q_I \in I, q_F \in F\}$.

We now introduce the concept of *chain*, which makes the connection between OP relations and context-free structure explicit, through brackets.

Definition 2. *A* simple chain $^{c_0}[c_1 c_2 \dots c_\ell]^{c_{\ell+1}}$ *is a string* $c_0 c_1 c_2 \dots c_\ell c_{\ell+1}$*, such that:* $c_0, c_{\ell+1} \in \Sigma \cup \{\#\}$*,* $c_i \in \Sigma$ *for every* $i = 1, 2, \dots \ell$ *(*$\ell \geq 1$*), and* $c_0 \lessdot c_1 \doteq c_2 \dots c_{\ell-1} \doteq c_\ell \gtrdot c_{\ell+1}$*. A* composed chain *is a string* $c_0 s_0 c_1 s_1 c_2 \dots c_\ell s_\ell c_{\ell+1}$*, where* $^{c_0}[c_1 c_2 \dots c_\ell]^{c_{\ell+1}}$ *is a simple chain, and* $s_i \in \Sigma^*$ *is the empty string or is such that* $^{c_i}[s_i]^{c_{i+1}}$ *is a chain (simple or composed), for every* $i = 0, 1, \dots, \ell$ *(*$\ell \geq 1$*). Such a composed chain will be written as* $^{c_0}[s_0 c_1 s_1 c_2 \dots c_\ell s_\ell]^{c_{\ell+1}}$*.* c_0 *(resp.* $c_{\ell+1}$*) is called its* left *(resp.* right*)* context*; all symbols between them form its* body*.*

A finite word w over Σ is *compatible* with an OPM M iff for each pair of letters c, d, consecutive in w, $M(c, d)$ is defined and, for each substring x of $\#w\#$ that is a chain of the form $^a[y]^b$, $M(a, b)$ is defined.

Chains can be identified through the traditional operator precedence parsing algorithm. We apply it to the sample word w_{ex} = **call han call call exc call ret ret**, which is compatible with $M_{\mathbf{call}}$ (for a more complete treatment, cf. [33,43]). First, write all precedence relations between consecutive characters, according to $M_{\mathbf{call}}$. Then, recognize all innermost patterns of the form $a \lessdot c \doteq \dots \doteq c \gtrdot b$ as simple chains, and remove their bodies. Then, write the precedence relations between the left and right contexts of the removed body, a and b, and iterate this process until only $\#\#$ remains. This procedure is applied to w_{ex} as follows:

$$\begin{array}{r|l}
1 & \# \lessdot \mathbf{call} \lessdot \mathbf{han} \lessdot \mathbf{call} \lessdot \underline{\mathbf{call}} \gtrdot \mathbf{exc} \gtrdot \mathbf{call} \doteq \mathbf{ret} \gtrdot \mathbf{ret} \gtrdot \# \\
2 & \# \lessdot \mathbf{call} \lessdot \mathbf{han} \lessdot \underline{\mathbf{call}} \gtrdot \mathbf{exc} \gtrdot \mathbf{call} \doteq \mathbf{ret} \gtrdot \mathbf{ret} \gtrdot \# \\
3 & \# \lessdot \mathbf{call} \lessdot \underline{\mathbf{han}} \doteq \underline{\mathbf{exc}} \gtrdot \mathbf{call} \doteq \mathbf{ret} \gtrdot \mathbf{ret} \gtrdot \# \\
4 & \# \lessdot \mathbf{call} \lessdot \underline{\mathbf{call}} \doteq \underline{\mathbf{ret}} \gtrdot \mathbf{ret} \gtrdot \# \\
5 & \# \lessdot \underline{\mathbf{call}} \doteq \underline{\mathbf{ret}} \gtrdot \# \\
6 & \# \doteq \#
\end{array}$$

The chain body removed in each step is underlined. In step 1, $^{\mathbf{call}}[\underline{\mathbf{call}}]^{\mathbf{exc}}$ is a simple chain, so its body $\underline{\mathbf{call}}$ is removed. Then, in step 2 we recognize the simple chain $^{\mathbf{han}}[\underline{\mathbf{call}}]^{\mathbf{exc}}$, which means $^{\mathbf{han}}[\mathbf{call}[\mathbf{call}]]^{\mathbf{exc}}$, where $[\mathbf{call}]$ is the chain body removed in step 1, is a composed chain. This way, we recognize, e.g., $^{\mathbf{han}}[\mathbf{call}]^{\mathbf{exc}}$, $^{\mathbf{call}}[\mathbf{han}\,\mathbf{exc}]^{\mathbf{call}}$ as simple chains, and $^{\mathbf{han}}[\mathbf{call}[\mathbf{call}]]^{\mathbf{exc}}$ and $^{\mathbf{call}}[\mathbf{han}[\mathbf{call}[\mathbf{call}]]\mathbf{exc}]^{\mathbf{call}}$ as composed chains (with inner chain bodies enclosed in brackets). Figure 1 shows the structure of a longer version of w_{ex}, which is an isomorphic representation of its ST as depicted in Fig. 4. Each chain corresponds to an internal node, and the fringe of the subtree rooted at it is the chain's body.

Let $\mathcal{A}$ be an OPA. We call a *support* for the simple chain $^{c_0}[c_1 c_2 \dots c_\ell]^{c_{\ell+1}}$ any path in $\mathcal{A}$ of the form $q_0 \xrightarrow{c_1} q_1 \dashrightarrow \dots \dashrightarrow q_{\ell-1} \overset{c_\ell}{\dashrightarrow} q_\ell \overset{q_0}{\Longrightarrow} q_{\ell+1}$. The label of the last (and only) pop is exactly q_0, i.e. the first state of the path; this pop is executed because of relation $c_\ell \gtrdot c_{\ell+1}$. We call a *support for the composed chain* $^{c_0}[s_0 c_1 s_1 c_2 \dots c_\ell s_\ell]^{c_{\ell+1}}$ any path in $\mathcal{A}$ of the form $q_0 \overset{s_0}{\rightsquigarrow} q'_0 \xrightarrow{c_1} q_1 \overset{s_1}{\rightsquigarrow} q'_1 \overset{c_2}{\dashrightarrow} \dots \overset{c_\ell}{\dashrightarrow} q_\ell \overset{s_\ell}{\rightsquigarrow} q'_\ell \overset{q'_0}{\Longrightarrow} q_{\ell+1}$ where, for every $i = 0, 1, \dots, \ell$: if $s_i \neq \epsilon$, then $q_i \overset{s_i}{\rightsquigarrow} q'_i$ is a support for the chain $^{c_i}[s_i]^{c_{i+1}}$, else $q'_i = q_i$.

Chains fully determine the parsing structure of any OPA over (Σ, M). If the OPA performs the computation $\langle sb, q_i, [a, q_j]\gamma\rangle \vdash^* \langle b, q_k, \gamma\rangle$, then $^a[s]^b$ is necessarily a chain over (Σ, M), and there exists a support like the one above with $s = s_0 c_1 \dots c_\ell s_\ell$ and $q_{\ell+1} = q_k$. This corresponds to the parsing of the string $s_0 c_1 \dots c_\ell s_\ell$ within the contexts a,b, which contains all information needed to build the subtree whose frontier is that string.

Consider the OPA $\mathcal{A}(\Sigma, M) = (\Sigma, M, \{q\}, \{q\}, \{q\}, \delta_{max})$ where $\delta_{max}(q, q) = q$, and $\delta_{max}(q, c) = q$, $\forall c \in \Sigma$. We call it the *OP Max-Automaton* over Σ, M. For a max-automaton, each chain has a support. Since there is a chain $^{\#}[s]^{\#}$ for any string s compatible with M, a string is accepted by $\mathcal{A}(\Sigma, M)$ iff it is compatible with M. If M is complete, each string is accepted by $\mathcal{A}(\Sigma, M)$, which defines the universal language Σ^* by assigning to any string the (unique) structure compatible with the OPM. With $M_{\mathbf{call}}$ of Fig. 1, if we take e.g. the string **ret call han**, it is accepted by the max-automaton with structure $\#[[\mathbf{ret}]\mathbf{call}[\mathbf{han}]]\#$.

In conclusion, given an OP alphabet, the OPM M assigns a unique structure to any compatible string in Σ^*; unlike VPL, such a structure is not visible in the string, and must be built by means of a non-trivial parsing algorithm. An OPA defined on the OP alphabet selects an appropriate subset within the "universe" of strings compatible with M. For a more complete description of the OPL family and of its relations with other CFL we refer the reader to [43].

2.1 Operator Precedence ω-Languages

All definitions regarding OPL are extended to infinite words in the usual way, but with a few distinctions. Given an OP alphabet (Σ, M), an ω-word $w \in \Sigma^\omega$ is compatible with M if every prefix of w is compatible with M. OP ω-words are not terminated by the delimiter $\#$. An ω-word may contain never-ending chains of the form $c_0 \lessdot c_1 \doteq c_2 \doteq \cdots$, where the $\lessdot$ relation between c_0 and c_1 is never closed by a corresponding $\gtrdot$. Such chains are called *open chains* and may be simple or composed. A composed open chain may contain both open and closed subchains. Of course, a closed chain cannot contain an open one. A terminal symbol $a \in \Sigma$ is *pending* if it is part of the body of an open chain and of no closed chains.

OPA classes accepting the whole class of ωOPL can be defined by augmenting Definition 1 with Büchi or Muller acceptance conditions [42]. In this paper, we only consider the former. The semantics of configurations, moves and infinite runs are defined as for finite OPA. For the acceptance condition, let ρ be a run on an ω-word w. Define

$$\mathrm{Inf}(\rho) = \{q \in Q \mid \text{there exist infinitely many positions } i \text{ s.t. } \langle \beta_i, q, x_i \rangle \in \rho\}$$

as the set of states that occur infinitely often in ρ. ρ is successful iff there exists a state $q_f \in F$ such that $q_f \in \mathrm{Inf}(\rho)$. An ωOPBA $\mathcal{A}$ accepts $w \in \Sigma^\omega$ iff there is a successful run of $\mathcal{A}$ on w. The ω-language recognized by $\mathcal{A}$ is $L(\mathcal{A}) = \{w \in \Sigma^\omega \mid \mathcal{A} \text{ accepts } w\}$. Unlike OPA, ωOPBA do not require the stack to be empty for word acceptance: when reading an open chain, the stack symbol pushed when the first character of the body of its underlying simple chain is read remains into the stack forever; it is at most updated by shift moves.

The most important closure properties of OPL are preserved by ωOPL, which form a Boolean algebra and are closed under concatenation of an OPL with an ωOPL [42]. The equivalence between deterministic and nondeterministic automata is lost in the infinite case, which is unsurprising, since it also happens for regular ω-languages and ωVPL.

2.2 Modeling Programs with OPA

For readers not familiar with OPL, we show how OPA can naturally model programming languages such as Java and C++. Given a set AP of atomic propositions describing events and states of the program, we use $(\mathcal{P}(AP), M_{AP})$ as the OP alphabet. For convenience, we consider a partitioning of AP into a set of standard propositional labels (in round font), and *structural labels* (SL, in bold). SL define the OP structure of the word: M_{AP} is only defined for subsets of AP containing exactly one SL, so that given two SL $\mathbf{l}_1, \mathbf{l}_2$, for any $a, a', b, b' \in \mathcal{P}(AP)$ s.t. $\mathbf{l}_1 \in a, a'$ and $\mathbf{l}_2 \in b, b'$ we have $M_{AP}(a, b) = M_{AP}(a', b')$. Hence, we define an OPM on the entire $\mathcal{P}(AP)$ by only giving the relations between SL, as we did for $M_{\mathbf{call}}$. Figure 2 shows how to model a procedural program with an OPA. The OPA simulates the program's behavior with respect to the stack, by expressing its execution traces with four event kinds: **call** (resp. **ret**) marks a procedure call (resp. return), **han** the installation of an exception handler by a `try` statement, and **exc** an exception being raised. OPM $M_{\mathbf{call}}$ defines the context-free structure of the word, which is strictly linked with the programming language semantics: the $\lessdot$ PR causes nesting (e.g., **call**s can be nested into other **call**s), and the $\doteq$ PR implies a one-to-one relation, e.g. between a **call** and the **ret** of the same function, and a **han** and the **exc** it catches. Each OPA state represents a line in the source code. First, procedure p_A is called by the program loader (M0),

```
    pA() {                     pB() {                  pC() {
A0:   try {               B0:    pC();            C0:    if (*) {
A1:     pB();             Br: }                   C1:      throw;
A2:   } catch {                                   C2:    } else {
A3:     pErr();                                   C3:      pC();
A4:     pErr();                                        }
      }                                           Cr: }
Ar: }
```

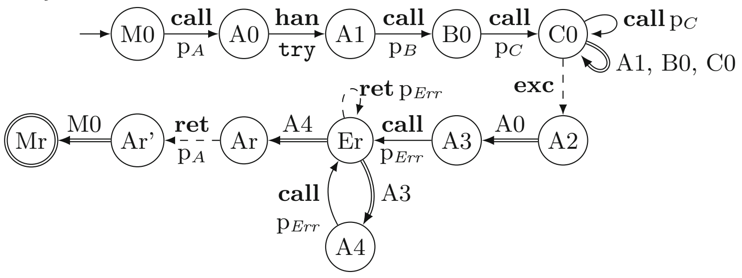


Fig. 2. Example procedural program (top) and the derived OPA (bottom). '`*`' implies a non-deterministic choice. Push, shift, pop moves are shown by, resp., solid, dashed and double arrows.

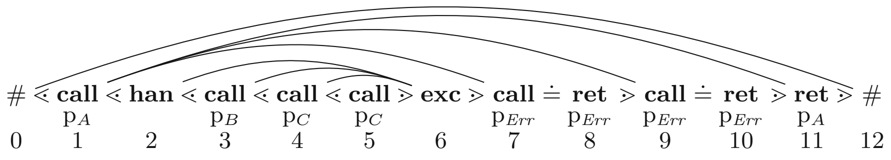
and $[\{\textbf{call}, \mathrm{p}_A\}, \mathrm{M0}]$ is pushed onto the stack, to track the program state before the **call**. Then, the `try` statement at line `A0` of p_A installs a handler. All subsequent calls to p_B and p_C push new stack symbols on top of the one pushed with **han**. p_C may only call itself recursively, or throw an exception, but never return normally. This is reflected by **exc** being the only transition leading from state C0 to the accepting state Mr, and p_B and p_C having no way to a normal **ret**. The OPA has a look-ahead of one input symbol, so when it encounters **exc**, it must pop all symbols in the stack, corresponding to active function frames, until it finds the one with **han** in it, which cannot be popped because $\textbf{han} \doteq \textbf{exc}$. Notice that such behavior cannot be modeled by Visibly Pushdown Automata or Nested Word Automata, because they need to read an input symbol for each pop move. Thus, **han** protects the parent function from the exception. Since the state contained in **han**'s stack symbol is A0, the execution resumes in the `catch` clause of p_A. p_A then calls twice the error-handling function p_{Err}, which ends regularly both times, and returns. The string of Fig. 1 is accepted by this OPA.

In this example, we only model the stack behavior for simplicity, but other statements, such as assignments, and other behaviors, such as continuations, could be modeled by a different choice of the OPA and OPM, and other aspects of the program's state by appropriate abstractions [38].

3 POTL: Syntax and Semantics

Given a finite set of atomic propositions AP, the syntax of POTL follows:

$$\begin{aligned} \varphi ::= \mathrm{a} \mid \neg\varphi \mid \varphi \vee \varphi \mid \bigcirc^t \varphi \mid \circleddash^t \varphi \mid \chi_F^t \varphi \mid \chi_P^t \varphi \mid \varphi \, \mathcal{U}_\chi^t \, \varphi \mid \varphi \, \mathcal{S}_\chi^t \, \varphi \\ \mid \bigcirc_H^t \varphi \mid \circleddash_H^t \varphi \mid \varphi \, \mathcal{U}_H^t \, \varphi \mid \varphi \, \mathcal{S}_H^t \, \varphi \end{aligned}$$



$\# \lessdot \mathbf{call} \lessdot \mathbf{han} \lessdot \mathbf{call} \lessdot \mathbf{call} \lessdot \mathbf{call} \gtrdot \mathbf{exc} \gtrdot \mathbf{call} \doteq \mathbf{ret} \gtrdot \mathbf{call} \doteq \mathbf{ret} \gtrdot \mathbf{ret} \gtrdot \#$
p_A p_B p_C p_C p_{Err} p_{Err} p_{Err} p_{Err} p_A
0 1 2 3 4 5 6 7 8 9 10 11 12

Fig. 3. The string of Fig. 1 as an OP word. Chains are shown by edges joining their contexts. Standard atomic propositions are shown below SL: p_l means a **call** or a **ret** is related to procedure p_l. First, procedure p_A is called (pos. 1), and it installs a handler in pos. 2. Then, three procedures are called, and one (p_C) throws an exception, which is caught by the handler. Two more functions are called and, finally, p_A returns.

where $\mathrm{a} \in AP$, and $t \in \{d, u\}$.

The semantics of POTL is based on the *word structure*—also called *OP word* for short—(U, M_{AP}, P), where $U = \{0, 1, \dots, n, n{+}1\}$, with $n \in \mathbb{N}$ is a set of word positions; $P \colon U \to \mathcal{P}(AP)$ is a function associating each position in U with the set of atomic propositions holding in that position, with $P(0) = P(n+1) = \{\#\}$. Given two positions i, j and a PR π, we write $i \;\pi\; j$ to say $P(i) \;\pi\; P(j)$.

We define the chain relation $\chi \subseteq U \times U$ so that $\chi(i,j)$ holds between two positions i, j iff $i < j - 1$, and i and j are resp. the left and right contexts of the same chain. For composed chains, χ may not be one-to-one, but also one-to-many or many-to-one. Given $i, j \in U$, relation χ has the following properties:

1. It never crosses itself: if $\chi(i,j)$ and $\chi(h,k)$, for any $h, k \in U$, then we have $i < h < j \implies k \leq j$ and $i < k < j \implies i \leq h$.
2. If $\chi(i,j)$, then $i \lessdot i+1$ and $j-1 \gtrdot j$.
3. There exists at most one single position h, called *leftmost context* of j, s.t. $\chi(h,j)$ and $h \lessdot j$ or $h \doteq j$; for any k s.t. $\chi(k,j)$ and $k \gtrdot j$ we have $k > h$.
4. There exists at most one single position h, called *rightmost context* of i, s.t. $\chi(i,h)$ and $i \gtrdot h$ or $i \doteq h$; for any k s.t. $\chi(i,k)$ and $i \lessdot k$ we have $k < h$.

Property 4 says that when the chain relation is one-to-many, the contexts of the outermost chains are in the $\doteq$ or $\gtrdot$ relation, while the inner ones are in the $\lessdot$ relation. Property 3 says that contexts of outermost many-to-one chains are in the $\doteq$ or $\lessdot$ relation, the inner ones being in the $\gtrdot$ relation. In the ST, the right context j of a chain is at the *same level* as the left one i when $i \doteq j$ (e.g., in Fig. 4, pos. 1 and 11), at a *lower level* when $i \lessdot j$ (e.g., pos. 1 with 7, and 9), at a *higher level* if $i \gtrdot j$ (e.g., pos. 3 and 4 with 6).

The truth of POTL formulas is defined w.r.t. a single word position. Let w be an OP word, and $\mathrm{a} \in AP$. Then, for any position $i \in U$ of w, we have $(w, i) \models \mathrm{a}$ if $\mathrm{a} \in P(i)$. Operators such as $\wedge$ and $\neg$ have the usual semantics from propositional logic. Next, while giving the formal semantics of POTL operators, we illustrate it by showing how it can be used to express properties on program execution traces, such as the one of Fig. 3.

a) Next/Back Operators. The *downward* next and back operators $\bigcirc^d$ and $\ominus^d$ are like their LTL counterparts, except they are true only if the next (resp. current) position is at a lower or equal ST level than the current (resp. preceding) one. The *upward* next and back, $\bigcirc^u$ and $\ominus^u$, are symmetric. Formally, $(w, i) \models \bigcirc^d\varphi$ iff $(w, i+1) \models \varphi$ and $i \lessdot (i+1)$ or $i \doteq (i+1)$, and $(w, i) \models \ominus^d\varphi$ iff $(w, i-1) \models \varphi$, and $(i-1) \lessdot i$ or $(i-1) \doteq i$. Substitute $\lessdot$ with $\gtrdot$ to obtain the semantics for $\bigcirc^u$ and $\ominus^u$. E.g., we can write $\bigcirc^d$**call** to say that the next position is an inner call (it holds in pos. 2, 3, 4 of Fig. 3), $\ominus^d$**call** to say that the previous position is a **call**, and the current is the first of the body of a function (pos. 2, 4, 5), or the **ret** of an empty one (pos. 8, 10), and $\ominus^u$**call** to say that the current position terminates an empty function frame (holds in 6, 8, 10). In pos. 2 $\bigcirc^d \mathrm{p}_B$ holds, but $\bigcirc^u \mathrm{p}_B$ does not.

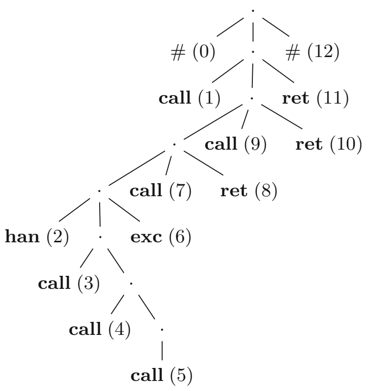


Fig. 4. The ST corresponding to the word of Fig. 3. Dots are internal nodes.

b) Chain Next/Back Operators. The *chain* next and back operators χ_F^t and χ_P^t, $t \in \{d, u\}$, evaluate their argument respectively on future and past positions in the chain relation with the current one. The *downward* (resp. *upward*) variant only considers chains whose right context goes down (resp. up) in the ST. E.g., in pos. 1 of Fig. 3, $\chi_F^d \mathrm{p}_{Err}$ holds because $\chi(1, 7)$ and $\chi(1, 9)$, meaning that p_A calls p_{Err} at least once. Formally, $(w, i) \models \chi_F^d\varphi$ iff there exists a position $j > i$ such that $\chi(i, j)$, $i \lessdot j$ or $i \doteq j$, and $(w, j) \models \varphi$. $(w, i) \models \chi_P^d\varphi$ iff there exists a position $j < i$ such that $\chi(j, i)$, $j \lessdot i$ or $j \doteq i$, and $(w, j) \models \varphi$. Replace $\lessdot$ with $\gtrdot$ for the upward versions. In Fig. 3, χ_F^u**exc** is true in **call** positions whose procedure is terminated by an exception thrown by an inner procedure (e.g. pos. 3 and 4). χ_P^u**call** is true in **exc** statements that terminate at least one procedure other than the one raising it, such as the one in pos. 6. χ_F^d**ret** and χ_F^u**ret** hold in **call**s to non-empty procedures that terminate normally, and not due to an uncaught exception (e.g., pos. 1).

c) Until/Since Operators. POTL has two kinds of until and since operators. They express properties on paths, which are sequences of positions obtained by iterating the different kinds of next or back operators. In general, a *path* of length $n \in \mathbb{N}$ between $i, j \in U$ is a sequence of positions $i = i_1 < i_2 < \cdots < i_n = j$. The *until* operator on a set of paths Γ is defined as follows: for any word w and position $i \in U$, and for any two POTL formulas φ and ψ, $(w, i) \models \varphi\, \mathcal{U}(\Gamma)\, \psi$ iff there exist a position $j \in U$, $j \geq i$, and a path $i_1 < i_2 < \cdots < i_n$ between i and j in Γ such that $(w, i_k) \models \varphi$ for any $1 \leq k < n$, and $(w, i_n) \models \psi$. *Since* operators are defined symmetrically. Note that, depending on Γ, a path from i to j may not exist. We define until/since operators by associating them with different sets of paths.

The *summary* until $\psi \mathcal{U}_\chi^t \theta$ (resp. since $\psi \mathcal{S}_\chi^t \theta$) operator is obtained by inductively applying the $\bigcirc^t$ and χ_F^t (resp. $\circleddash^t$ and χ_P^t) operators. It holds in a position in which either θ holds, or ψ holds together with $\bigcirc^t(\psi \mathcal{U}_\chi^t \theta)$ (resp. $\circleddash^t(\psi \mathcal{S}_\chi^t \theta)$) or $\chi_F^t(\psi \mathcal{U}_\chi^t \theta)$ (resp. $\chi_P^t(\psi \mathcal{S}_\chi^t \theta)$). It is an until operator on paths that can move not only between consecutive positions, but also between contexts of a chain, skipping its body. With the OPM of Fig. 1, this means skipping function bodies. The downward variants can move between positions at the same level in the ST (i.e., in the same simple chain body), or down in the nested chain structure. The upward ones remain at the same level, or move to higher levels of the ST.

Formula $\top \mathcal{U}_\chi^u$ **exc** is true in positions contained in the frame of a function that is terminated by an exception. It is true in pos. 3 of Fig. 3 because of path 3-6, and false in pos. 1, because no path can enter the chain whose contexts are pos. 1 and 11. Formula $\top \mathcal{U}_\chi^d$ **exc** is true in call positions whose function frame contains **exc**s, but that are not necessarily terminated by one of them, such as the one in pos. 1 (with path 1-2-6).

We define *Downward Summary Paths* (DSP) as follows. Given an OP word w, and two positions $i \le j$ in w, the DSP between i and j, if it exists, is a sequence of positions $i = i_1 < i_2 < \cdots < i_n = j$ such that, for each $1 \le p < n$,

$$i_{p+1} = \begin{cases} k & \text{if } k = \max\{h \mid h \le j \wedge \chi(i_p, h) \wedge (i_p \lessdot h \vee i_p \doteq h)\} \text{exists;} \\ i_p + 1 & \text{otherwise, if } i_p \lessdot (i_p+1) \text{ or } i_p \doteq (i_p+1). \end{cases}$$

The Downward Summary (DS) until and since operators $\mathcal{U}_\chi^d$ and $\mathcal{S}_\chi^d$ use as Γ the set of DSP starting in the position in which they are evaluated. The definition for the upward counterparts is, again, obtained by substituting $\lessdot$ with $\gtrdot$. In Fig. 3, **call** $\mathcal{U}_\chi^d$ (**ret** $\wedge$ p_{Err}) holds in pos. 1 because of path 1-7-8 and 1-9-10, (**call** $\vee$ **exc**) $\mathcal{S}_\chi^u$ p_B in pos. 7 because of path 3-6-7, and (**call** $\vee$ **exc**) $\mathcal{U}_\chi^u$ **ret** in 3 because of path 3-6-7-8.

d) Hierarchical Operators. A single position may be the left or right context of multiple chains. The operators seen so far cannot keep this fact into account, since they "forget" about a left context when they jump to the right one. Thus, we introduce the *hierarchical* next and back operators. The *upward* hierarchical next (resp. back), $\bigcirc_H^u \psi$ (resp. $\circleddash_H^u \psi$), is true iff the current position j is the right context of a chain whose left context is i, and ψ holds in the next (resp. previous) pos. j' that is the right context of i, with $i \lessdot j, j'$. So, $\bigcirc_H^u \mathrm{p}_{Err}$ holds in pos. 7 of Fig. 3 because p_{Err} holds in 9, and $\circleddash_H^u \mathrm{p}_{Err}$ in 9 because p_{Err} holds in 7. In the ST, $\bigcirc_H^u$ goes *up* between **call**s to p_{Err}, while $\circleddash_H^u$ goes down. Their *downward* counterparts behave symmetrically, and consider multiple inner chains sharing their right context. They are formally defined as:

- $(w, i) \models \bigcirc_H^u \varphi$ iff there exist a position $h < i$ s.t. $\chi(h, i)$ and $h \lessdot i$ and a position $j = \min\{k \mid i < k \wedge \chi(h, k) \wedge h \lessdot k\}$ and $(w, j) \models \varphi$;
- $(w, i) \models \circleddash_H^u \varphi$ iff there exist a position $h < i$ s.t. $\chi(h, i)$ and $h \lessdot i$ and a position $j = \max\{k \mid k < i \wedge \chi(h, k) \wedge h \lessdot k\}$ and $(w, j) \models \varphi$;
- $(w, i) \models \bigcirc_H^d \varphi$ iff there exist a position $h > i$ s.t. $\chi(i, h)$ and $i \gtrdot h$ and a position $j = \min\{k \mid i < k \wedge \chi(k, h) \wedge k \gtrdot h\}$ and $(w, j) \models \varphi$;

– $(w, i) \models \circleddash_H^d \varphi$ iff there exist a position $h > i$ s.t. $\chi(i, h)$ and $i \gtrdot h$ and a position $j = \max\{k \mid k < i \wedge \chi(k, h) \wedge k \gtrdot h\}$ and $(w, j) \models \varphi$.

In the ST of Fig. 4, $\bigcirc_H^d$ and $\circleddash_H^d$ go *down* and up among **call**s terminated by the same **exc**. For example, in pos. 3 $\bigcirc_H^d \mathrm{p}_C$ holds, because both pos. 3 and 4 are in the chain relation with 6. Similarly, in pos. 4 $\circleddash_H^d \mathrm{p}_B$ holds. Note that these operators do not consider leftmost/rightmost contexts, so $\bigcirc_H^u$**ret** is false in pos. 9, as **call** $\doteq$ **ret**, and pos. 11 is the rightmost context of pos. 1.

The hierarchical until and since operators are defined by iterating these next and back operators. The upward hierarchical path (UHP) between i and j is a sequence of positions $i = i_1 < i_2 < \cdots < i_n = j$ such that there exists a position $h < i$ such that for each $1 \leq p \leq n$ we have $\chi(h, i_p)$ and $h \lessdot i_p$, and for each $1 \leq q < n$ there exists no position k such that $i_q < k < i_{q+1}$ and $\chi(h, k)$. The until and since operators based on the set of UHP starting in the position in which they are evaluated are denoted as $\mathcal{U}_H^u$ and $\mathcal{S}_H^u$. E.g., **call**$\,\mathcal{U}_H^u\, \mathrm{p}_{Err}$ holds in pos. 7 because of the singleton path 7 and path 7-9, and **call**$\,\mathcal{S}_H^u\, \mathrm{p}_{Err}$ in pos. 9 because of paths 9 and 7-9.

The downward hierarchical path (DHP) between i and j is a sequence of positions $i = i_1 < i_2 < \cdots < i_n = j$ such that there exists a position $h > j$ such that for each $1 \leq p \leq n$ we have $\chi(i_p, h)$ and $i_p \gtrdot h$, and for each $1 \leq q < n$ there exists no position k such that $i_q < k < i_{q+1}$ and $\chi(k, h)$. The until and since operators based on the set of DHP starting in the position in which they are evaluated are denoted as $\mathcal{U}_H^d$ and $\mathcal{S}_H^d$. In Fig. 3, **call**$\,\mathcal{U}_H^d\, \mathrm{p}_C$ holds in pos. 3, and **call**$\,\mathcal{S}_H^d\, \mathrm{p}_B$ in pos. 4, both because of path 3-4.

The POTL until and since operators enjoy expansion laws similar to those of LTL. Here we give those for two until operators, those for their since and downward counterparts being symmetric.

$$\varphi\, \mathcal{U}_\chi^t\, \psi \equiv \psi \vee \Big(\varphi \wedge \big(\bigcirc^t (\varphi\, \mathcal{U}_\chi^t\, \psi) \vee \chi_F^t(\varphi\, \mathcal{U}_\chi^t\, \psi)\big)\Big)$$
$$\varphi\, \mathcal{U}_H^u\, \psi \equiv (\psi \wedge \chi_P^d \top \wedge \neg \chi_P^u \top) \vee \big(\varphi \wedge \bigcirc_H^u(\varphi\, \mathcal{U}_H^u\, \psi)\big)$$

3.1 Expressiveness of POTL

We first define some derived operators. For $t \in \{d, u\}$, we define the downward/upward summary *eventually* as $\Diamond^t \varphi := \top \mathcal{U}_\chi^t \varphi$, and the downward/upward summary *globally* as $\Box^t \varphi := \neg \Diamond^t (\neg \varphi)$. $\Diamond^u \varphi$ and $\Box^u \varphi$ resp. say that φ holds in one or all positions in the path from the current position to the root of the ST. $\Diamond^d \varphi$ says that φ holds in at least one position in the current subtree, and $\Box^d \varphi$ in all of them. E.g., if $\Box^d(\neg \mathrm{p}_A)$ holds in a **call**, it means that p_A never holds in its whole function body, which is the subtree rooted next to the **call**.

In the technical report, we prove

Theorem 1 ([23]). *POTL = FOL with one free variable on OP words.*

Equivalence to FOL on the relevant algebraic structure is a desirable feature of linear-time temporal logics, and it was proved for LTL [39] and NWTL [2]. It is in some sense a theoretical assurance of the sufficient expressive power of the logic. Moreover, NWTL $\subset$ OPTL was proved in [22], and OPTL $\subseteq$ POTL comes from Theorem 1 and the semantics of OPTL being expressible in FOL. In [23], we also prove that there exist POTL formulas not expressible in OPTL. Thus, we can claim CaRet [6] $\subseteq$ NWTL $\subset$ OPTL $\subset$ POTL. One of such formulas is $\Diamond^d \mathrm{p}_A$ which, evaluated e.g. on a **han** position with a matched **exc**, states that p_A holds in one of the positions in the same subtree.

More importantly, POTL can express many useful requirements of procedural programs. To emphasize the potential practical applications in automatic verification, we supply a few examples of typical program properties expressed as POTL formulas, not all of them being expressible in the other above languages.

The LTL *globally* can be written as $\Box\psi := \neg\Diamond^u(\Diamond^d\neg\psi)$. The two nested eventually operators enumerate all future positions by going up and then down in any direction in the syntax tree: when negated, this means $\neg\psi$ may never hold. POTL can express Hoare-style pre/postconditions with formulae such as $\Box(\mathbf{call} \wedge \rho \implies \chi_F^d(\mathbf{ret} \wedge \theta))$, where ρ is the precondition, and θ is the postcondition.

Unlike NWTL, POTL can easily express properties related to exception handling and interrupt management [43]. E.g., the shortcut $\mathit{CallThr}(\psi) := \bigcirc^u(\mathbf{exc} \wedge \psi) \vee \chi_F^u(\mathbf{exc} \wedge \psi)$, evaluated in a **call**, states that the procedure currently started is terminated by a **exc** in which ψ holds. So, $\Box(\mathbf{call} \wedge \rho \wedge \mathit{CallThr}(\top) \implies \mathit{CallThr}(\theta))$ means that if precondition ρ holds when a procedure is called, then postcondition θ must hold if that procedure is terminated by an exception. In object oriented programming languages, if $\rho \equiv \theta$ is a class invariant asserting that a class instance's state is valid, this formula expresses *weak exception safety* [1], and *strong exception safety* if ρ and θ express particular states of the class instance. The *no-throw guarantee* can be stated with $\Box(\mathbf{call} \wedge \mathrm{p}_A \implies \neg\mathit{CallThr}(\top))$, meaning procedure p_A is never interrupted by an exception.

Stack inspection [29,37], i.e. properties regarding the sequence of procedures active in the program's stack at a certain point of its execution, is an important class of requirements that can be expressed with shortcut $\mathit{Scall}(\varphi, \psi) := (\mathbf{call} \implies \varphi)\ \mathcal{S}_\chi^d\ (\mathbf{call} \wedge \psi)$, which subsumes the *call since* of CaRet, as it also works with exceptions. E.g., $\Box\big((\mathbf{call} \wedge \mathrm{p}_B \wedge \mathit{Scall}(\top, \mathrm{p}_A)) \implies \mathit{CallThr}(\top)\big)$ means that whenever p_B is executed and at least one instance of p_A is on the stack, p_B is terminated by an exception. The OPA of Fig. 2 satisfies this formula, because p_B is called by p_A, and p_C throws.

4 Model Checking

Given an OP alphabet $(\mathcal{P}(AP), M_{AP})$, where AP is a finite set of atomic propositions, and a POTL formula φ, we build an OPA $\mathcal{A}_\varphi = (\mathcal{P}(AP), M_{AP}, Q, I, F, \delta)$ that accepts models of φ. The construction of $\mathcal{A}_\varphi$ resembles the classical one for LTL and the ones for NWTL and OPTL, diverging from them significantly when dealing with temporal obligations that involve positions in the chain relation.

We first introduce $Cl(\varphi)$, the *closure* of φ, containing all subformulas of φ, and some auxiliary operators. The latter are needed to model-check chain next and back operators. For any PR $\pi \in \{\lessdot, \doteq, \gtrdot\}$, we define them as follows: $(w, i) \models \chi_F^\pi \varphi$ iff there exists $j > i$ such that $\chi(i, j)$, $i \; \pi \; j$, and $(w, j) \models \varphi$; $(w, i) \models \chi_P^\pi \varphi$ iff there exists $j < i$ such that $\chi(j, i)$, $j \; \pi \; i$, and $(w, j) \models \varphi$.

$Cl(\varphi)$ is the smallest set such that, for $t \in \{d, u\}$:

1. $\varphi \in Cl(\varphi)$,
2. $AP \subseteq Cl(\varphi)$,
3. if $\psi \in Cl(\varphi)$ and $\psi \neq \neg\theta$, then $\neg\psi \in Cl(\varphi)$ (we identify $\neg\neg\psi$ with ψ);
4. if $\neg\psi \in Cl(\varphi)$, then $\psi \in Cl(\varphi)$;
5. if any of $\psi \land \theta$ or $\psi \lor \theta$ is in $Cl(\varphi)$, then $\psi, \theta \in Cl(\varphi)$;
6. if any of $\bigcirc^t \psi$, $\circleddash^t \psi$, $\chi_F^t \psi$, or $\chi_P^t \psi$ is in $Cl(\varphi)$, then $\psi \in Cl(\varphi)$;
7. if $\chi_F^d \psi$ (resp. $\chi_F^u \psi$) is in $Cl(\varphi)$, then $\chi_F^{\lessdot} \psi$ (resp. $\chi_F^{\gtrdot} \psi$), $\chi_F^{\doteq} \psi$, χ_L are in it;
8. if $\chi_P^d \psi$ (resp. $\chi_P^u \psi$) is in $Cl(\varphi)$, then $\chi_P^{\lessdot} \psi$ (resp. $\chi_P^{\gtrdot} \psi$), $\chi_P^{\doteq} \psi$ are in it;
9. if any of $\psi \, \mathcal{U}_\chi^t \, \theta$, $\psi \, \mathcal{S}_\chi^t \, \theta$, $\psi \, \mathcal{U}_H^t \, \theta$, or $\psi \, \mathcal{S}_H^t \, \theta$ is in $Cl(\varphi)$, then $\psi, \theta \in Cl(\varphi)$;
10. if $\psi \, \mathcal{U}_\chi^t \, \theta \in Cl(\varphi)$, then $\bigcirc^t (\psi \, \mathcal{U}_\chi^t \, \theta), \chi_F^t (\psi \, \mathcal{U}_\chi^t \, \theta) \in Cl(\varphi)$ (since is symmetric).

The set $Atoms(\varphi)$ contains all consistent subsets of $Cl(\varphi)$, i.e. all $\Phi \subseteq Cl(\varphi)$ s.t.

– for every $\psi \in Cl(\varphi)$, $\psi \in \Phi$ iff $\neg\psi \notin \Phi$;
– $\psi \land \theta \in \Phi$, iff $\psi \in \Phi$ and $\theta \in \Phi$;
– $\psi \lor \theta \in \Phi$, iff $\psi \in \Phi$ or $\theta \in \Phi$, or both.

The consistency constraints on $Atoms(\varphi)$ will be augmented incrementally in the following, for each operator.

The set of states of $\mathcal{A}_\varphi$ is $Q = Atoms(\varphi)^2$, and its elements, which we denote with Greek capital letters, are of the form $\Phi = (\Phi_c, \Phi_p)$, where Φ_c is the set of formulas that hold in the current position, and Φ_p is the set of temporal obligations. The latter keep track of arguments of temporal operators that must be satisfied after a chain body, skipping it. The way they do so depends on the transition relation δ, which we also define incrementally. Each automaton state is associated to word positions. So, for $(\Phi, a, \Psi) \in \delta_{push/shift}$, with $\Phi \in Atoms(\varphi)^2$ and $a \in \mathcal{P}(AP)$, we have $\Phi_c \cap AP = a$ (by $\Phi_c \cap AP$ we mean the set of atomic propositions in Φ_c). *Pop* moves do not read input symbols, and the automaton remains at the same position when performing them: for any $(\Phi, \Theta, \Psi) \in \delta_{pop}$ we impose $\Phi_c = \Psi_c$. The initial set I contains states of the form (Φ_c, Φ_p), with $\varphi \in \Phi_c$, and the final set F states of the form (Ψ_c, Ψ_p), s.t. $\Psi_c \cap AP = \{\#\}$ and Ψ_c contains no future operators. We extend the construction to the most important operators, leaving the others and correctness proofs to [21].

Next/Back Operators. Let $(\Phi, a, \Psi) \in \delta_{shift} \cup \delta_{push}$, with $\Phi, \Psi \in Atoms(\varphi)^2$, $a \in \mathcal{P}(AP)$, and let $b = \Psi_c \cap AP$: we have $\bigcirc^d \psi \in \Phi_c$ iff $\psi \in \Psi_c$ and either $a \lessdot b$ or $a \doteq b$. The constraints introduced for the $\circleddash^d$ operator are symmetric, and for their upward counterparts it suffices to replace $\lessdot$ with $\gtrdot$.

If $\chi_F^d \psi \in Cl(\varphi)$, for each $\Phi \in Atoms(\varphi)^2$ we impose that $\chi_F^d \psi \in \Phi_c$ iff $\chi_F^{\lessdot} \psi \in \Phi_c$ or $\chi_F^{\doteq} \psi \in \Phi_c$. Analogous rules are defined for the upward and past chain operators. The auxiliary symbol χ_L forces the current position to be the first one of a chain body. Let the current state of the OPA be $\Phi \in Atoms(\varphi)^2$:

	input	state	stack	PR	move
1	**call han exc ret** #	$\Phi_c^0 = \{\mathbf{call}, \chi_F^d \mathbf{ret}, \chi_F^{\doteq} \mathbf{ret}\}$, $\Phi_p^0 = \{\chi_L\}$	$\bot$	$\# \lessdot \mathbf{call}$	push
2	**han exc ret** #	$\Phi^1 = (\{\mathbf{han}\}, \{\chi_F^{\doteq}\mathbf{ret}, \chi_L\})$	$[\mathbf{call}, \Phi^0]\bot$	$\mathbf{call} \lessdot \mathbf{han}$	push
3	**exc ret** #	$\Phi^2 = (\{\mathbf{exc}\}, \emptyset)$	$[\mathbf{han}, \Phi^1][\mathbf{call}, \Phi^0]\bot$	$\mathbf{han} \doteq \mathbf{exc}$	shift
4	**ret** #	$\Phi^3 = (\{\mathbf{ret}\}, \emptyset)$	$[\mathbf{exc}, \Phi^1][\mathbf{call}, \Phi^0]\bot$	$\mathbf{exc} \gtrdot \mathbf{ret}$	pop
5	**ret** #	$\Phi^4 = (\{\mathbf{ret}\}, \{\chi_F^{\doteq}\mathbf{ret}\})$	$[\mathbf{call}, \Phi^0]\bot$	$\mathbf{call} \doteq \mathbf{ret}$	shift
6	#	$\Phi^5 = (\{\#\}, \emptyset)$	$[\mathbf{ret}, \Phi^0]\bot$	$\mathbf{ret} \gtrdot \#$	pop
7	#	$\Phi^5 = (\{\#\}, \emptyset)$	$\bot$	–	–

Fig. 5. Example accepting run of the automaton for $\chi_F^d \mathbf{ret}$.

$\chi_L \in \Phi_p$ iff the next transition (i.e. the one reading the current position) is a push. Formally, if $(\Phi, a, \Psi) \in \delta_{shift}$ or $(\Phi, \Theta, \Psi) \in \delta_{pop}$, for any Φ, Θ, Ψ and a, then $\chi_L \notin \Phi_p$. If $(\Phi, a, \Psi) \in \delta_{push}$, then $\chi_L \in \Phi_p$. For any initial state $(\Phi_c, \Phi_p) \in I$, we have $\chi_L \in \Phi_p$ iff $\# \notin \Phi_c$.

If $\chi_F^{\doteq}\psi \in Cl(\varphi)$, its satisfaction is ensured by the following constraints on δ:

1. Let $(\Phi, a, \Psi) \in \delta_{push/shift}$: then $\chi_F^{\doteq}\psi \in \Phi_c$ iff $\chi_F^{\doteq}\psi, \chi_L \in \Psi_p$;
2. let $(\Phi, \Theta, \Psi) \in \delta_{pop}$: then $\chi_F^{\doteq}\psi \notin \Phi_p$, and $\chi_F^{\doteq}\psi \in \Theta_p$ iff $\chi_F^{\doteq}\psi \in \Psi_p$;
3. let $(\Phi, a, \Psi) \in \delta_{shift}$: then $\chi_F^{\doteq}\psi \in \Phi_p$ iff $\psi \in \Phi_c$.
   If $\chi_F^{\lessdot}\psi \in Cl(\varphi)$, $\chi_F^{\lessdot}\psi$ is allowed in the pending part of initial states, and we add the following constraints:
4. Let $(\Phi, a, \Psi) \in \delta_{push/shift}$: then $\chi_F^{\lessdot}\psi \in \Phi_c$ iff $\chi_F^{\lessdot}\psi, \chi_L \in \Psi_p$;
5. let $(\Phi, \Theta, \Psi) \in \delta_{pop}$: then $\chi_F^{\lessdot}\psi \in \Theta_p$ iff $\chi_L \in \Psi_p$, and either $\chi_F^{\lessdot}\psi \in \Psi_p$ or $\psi \in \Phi_c$.

We illustrate how the construction works for $\chi_F^{\doteq}$ with the example of Fig. 5. The OPA starts in state Φ^0, with $\chi_F^d \mathbf{ret} \in \Phi_c^0$, and guesses that χ_F^d will be fulfilled by $\chi_F^{\doteq}$, so $\chi_F^{\doteq}\mathbf{ret} \in \Phi_c^0$. **call** is read by a push move, resulting in state Φ^1. The OPA guesses the next move will be a push, so $\chi_L \in \Phi_p^1$. By rule 1, we have $\chi_F^{\doteq}\mathbf{ret} \in \Phi_p^1$. The last guess is immediately verified by the next push (step 2–3). Thus, the pending obligation for $\chi_F^{\doteq}\mathbf{ret}$ is stored onto the stack in Φ^1. The OPA, then, reads **exc** with a shift, and pops the stack symbol containing Φ^1 (step 4–5). By rule 2, the temporal obligation is resumed in the next state Φ^4, so $\chi_F^{\doteq}\mathbf{ret} \in \Phi_p^4$. Finally, **ret** is read by a shift which, by rule 3, may occur only if $\mathbf{ret} \in \Phi_c^4$. Rule 3 verifies the guess that $\chi_F^{\doteq}\mathbf{ret}$ holds in Φ_0, and fulfills the temporal obligation contained in Φ_p^4, by preventing computations in which $\mathbf{ret} \notin \Phi_c^4$ from continuing. Had the next transition been a pop (e.g. because there was no **ret** and $\mathbf{call} \gtrdot \#$), the run would have been blocked by rule 2, preventing the OPA from reaching an accepting state, and from emptying the stack.

Summary Until and Since. The construction for these operators is based on their expansion laws. For any $\Phi \in Atoms(\varphi)^2$, we have $\psi \, \mathcal{U}_\chi^t \, \theta \in \Phi_c$, with

$t \in \{d, u\}$ being a direction, iff either: 1. $\theta \in \Phi_c$, 2. $\bigcirc^t(\psi \, \mathcal{U}^t_\chi \, \theta), \psi \in \Phi_c$, or 3. $\chi^t_F(\psi \, \mathcal{U}^t_\chi \, \theta), \psi \in \Phi_c$. The rules for since are symmetric.

Hierarchical Operators. For the hierarchical operators, we do not give an explicit OPA construction, but we rely on a translation into other POTL operands. For each hierarchical operator η in φ, we add a propositional symbol $\mathrm{q}_{(\eta)}$. The upward hierarchical operators consider the right contexts of chains sharing the same left context. To distinguish such positions, we define formula $\gamma_{L,\eta} := \chi^{\lessdot}_P\big(\mathrm{q}_{(\eta)} \wedge \bigcirc(\square\neg\mathrm{q}_{(\eta)}) \wedge \circleddash(\boxminus\neg\mathrm{q}_{(\eta)})\big)$, where $\square$ and $\boxminus$ are as in Sect. 3.1. $\bigcirc$ and $\circleddash$ are the LTL next and back operators, for which model checking can be done as for $\bigcirc^d$ and $\circleddash^d$, but removing the restrictions on PR. $\gamma_{L,\eta}$, evaluated on a position i, asserts that $\mathrm{q}_{(\eta)}$ holds in the unique position h such that $\chi(h,i)$ and $h \lessdot i$. Thus, $\mathrm{q}_{(\eta)}$ can be used to distinguish other positions j such that $\chi(h,j)$ and $h \lessdot j$, as $\chi^{\lessdot}_P \mathrm{q}_{(\eta)}$ holds in them. The translations for future upward hierarchical operators follow, the others being analogous.

$$\bigcirc^u_H \psi := \gamma_{L,\bigcirc^u_H\psi} \wedge \bigcirc\big((\neg\chi^{\lessdot}_P\mathrm{q}_{(\bigcirc^u_H\psi)}) \, \mathcal{U}^u_\chi \, (\chi^{\lessdot}_P\mathrm{q}_{(\bigcirc^u_H\psi)} \wedge \psi)\big)$$
$$\psi \, \mathcal{U}^u_H \, \theta := \gamma_{L,\psi\mathcal{U}^u_H\theta} \wedge (\chi^{\lessdot}_P\mathrm{q}_{(\psi\mathcal{U}^u_H\theta)} \implies \psi) \, \mathcal{U}^u_\chi \, (\chi^{\lessdot}_P\mathrm{q}_{(\psi\mathcal{U}^u_H\theta)} \wedge \theta)$$

4.1 Model Checking for ω-Words

To perform model checking of a POTL formula φ on OP ω-words, we build a generalized ωOPBA $\mathcal{A}^\omega_\varphi = (\mathcal{P}(AP), M_{AP}, Q_\omega, I, \mathbf{F}, \delta)$, where $Q_\omega = \mathit{Atoms}(\varphi)^2 \times \mathcal{P}(\mathit{Cl}_{stack}(\varphi))$, which differs from the finite-word OPA only for the state set and the acceptance condition. As in [2], the generalized Büchi acceptance condition is a slight variation on the one shown in Sect. 2.1: $\mathbf{F}$ is the set of sets of Büchi final states, and an ω-word is accepted iff at least one state from each one of the sets contained in $\mathbf{F}$ is visited infinitely often during the computation.

In finite words, the stack is empty at the end of every accepting computation, which implies the satisfaction of all temporal constraints tracked by the pending part of stack symbols. In ωOPBAs, the stack may never be empty, and symbols with a non-empty pending part may remain in it indefinitely, never enforcing the satisfaction of the respective formulas. To overcome this issue, we use $\mathit{Atoms}(\varphi)^2 \times \mathcal{P}(\mathit{Cl}_{stack}(\varphi))$, with $\mathit{Cl}_{stack}(\varphi) \subseteq \mathit{Cl}(\varphi)$, as the state set of the ωOPBA. Such states have the form $\Phi = (\Phi_c, \Phi_p, \Phi_s)$, where Φ_c and Φ_p have the same role as in the finite-word case, and Φ_s is the *in-stack* part of Φ. All rules previously defined for Φ_c and Φ_p remain the same. Φ_s contains elements of $\mathit{Cl}_{stack}(\varphi)$ contained in any symbol currently on the stack. $\mathit{Cl}_{stack}(\varphi)$ contains formulas in $\mathit{Cl}(\varphi)$ that use the stack to ensure the satisfaction of future temporal requirements, namely all $\chi^\pi_F\psi \in \mathit{Cl}(\varphi)$, with $\pi \in \{\lessdot, \doteq, \gtrdot\}$. Thus, pending temporal obligations are moved from the stack to the ωOPBA state, and they can be considered by the Büchi acceptance condition.

Suppose we want to model check $\chi^{\doteq}_F\psi$. Formula $\chi^{\doteq}_F\psi$ must be inserted in the in-stack part of the current state whenever a stack symbol containing it in its pending part is pushed. It must be kept in the in-stack part of the current state until the last stack symbol containing it in its pending part is popped, marking

	input	state	stack	PR
1	$\mathbf{call}\,\mathbf{call}\,\mathbf{han}\,\mathbf{exc}\,\mathbf{ret}\,\mathbf{ret}\,(\mathbf{call})^\omega$	$\Phi^0 = (\{\mathbf{call}, \chi_F^d \mathbf{ret}, \chi_F^{\doteq} \mathbf{ret}\}, \{\chi_L\}, \emptyset)$	$\bot$	$\lessdot$
2	$\mathbf{call}\,\mathbf{han}\,\mathbf{exc}\,\mathbf{ret}\,\mathbf{ret}\,(\mathbf{call})^\omega$	$\Phi^1 = (\{\mathbf{call}, \chi_F^d \mathbf{ret}, \chi_F^{\doteq} \mathbf{ret}\}, \{\chi_L, \chi_F^{\doteq} \mathbf{ret}\}, \emptyset)$	$[\mathbf{call}, \Phi^0]\bot$	$\lessdot$
3	$\mathbf{han}\,\mathbf{exc}\,\mathbf{ret}\,\mathbf{ret}\,(\mathbf{call})^\omega$	$\Phi^2 = (\{\mathbf{han}\}, \{\chi_L, \chi_F^{\doteq} \mathbf{ret}\}, \{\chi_F^{\doteq} \mathbf{ret}\})$	$[\mathbf{call}, \Phi^1][\mathbf{call}, \Phi^0]\bot$	$\lessdot$
4	$\mathbf{exc}\,\mathbf{ret}\,\mathbf{ret}\,(\mathbf{call})^\omega$	$\Phi^3 = (\{\mathbf{exc}\}, \emptyset, \{\chi_F^{\doteq} \mathbf{ret}\})$	$[\mathbf{han}, \Phi^2][\mathbf{call}, \Phi^1][\mathbf{call}, \Phi^0]\bot$	$\doteq$
5	$\mathbf{ret}\,\mathbf{ret}\,(\mathbf{call})^\omega$	$\Phi^4 = (\{\mathbf{ret}\}, \emptyset, \{\chi_F^{\doteq} \mathbf{ret}\})$	$[\mathbf{exc}, \Phi^2][\mathbf{call}, \Phi^1][\mathbf{call}, \Phi^0]\bot$	$\gtrdot$
6	$\mathbf{ret}\,\mathbf{ret}\,(\mathbf{call})^\omega$	$\Phi^5 = (\{\mathbf{ret}\}, \{\chi_F^{\doteq} \mathbf{ret}\}, \{\chi_F^{\doteq} \mathbf{ret}\})$	$[\mathbf{call}, \Phi^1][\mathbf{call}, \Phi^0]\bot$	$\doteq$
7	$\mathbf{ret}\,(\mathbf{call})^\omega$	Φ^4	$[\mathbf{ret}, \Phi^1][\mathbf{call}, \Phi^0]\bot$	$\doteq$
8	$\mathbf{ret}\,(\mathbf{call})^\omega$	$\Phi^6 = (\{\mathbf{ret}\}, \{\chi_F^{\doteq} \mathbf{ret}\}, \emptyset)$	$[\mathbf{call}, \Phi^0]\bot$	$\doteq$
9	$(\mathbf{call})^\omega$	$\Phi^7 = (\{\mathbf{call}\}, \emptyset, \emptyset)$	$[\mathbf{ret}, \Phi^0]\bot$	$\gtrdot$

Fig. 6. Prefix of an accepting run of the automaton for $\chi_F^d \mathbf{ret}$.

the satisfaction of its temporal requirement. Then, it is possible to define an acceptance set $F_{\chi_F^{\doteq} \psi} \in \mathbf{F}$, as the set of states not containing $\chi_F^{\doteq} \psi$ in any part. Figure 6 shows an ωOPBA run of this kind. Notice that after step 7 $\chi_F^{\doteq} \psi$ does not appear in any state's in-stack part, so the run is accepting.

This construction is formalized as follows. Let $\psi \in Cl_{stack}(\varphi)$. We add a few constraints on the transition relations. For any $\Phi, \Theta, \Psi \in Q_\omega$ and $a \in \mathcal{P}(AP)$:

6. let $(\Phi, a, \Theta) \in \delta_{push}$: if $\psi \in \Phi_p$, then $\psi \in \Theta_s$;
7. let $(\Phi, a, \Theta) \in \delta_{push/shift}$: if $\psi \in \Phi_s$, then $\psi \in \Theta_s$;
8. let $(\Phi, \Theta, \Psi) \in \delta_{pop}$: if $\psi \in \Phi_s$ and $\psi \in \Theta_s$, then $\psi \in \Psi_s$.

An acceptance condition for summary until operators is also needed. For $\psi \, \mathcal{U}_\chi^d \, \theta \in Cl(\varphi)$, we add an acceptance set $\mathbf{F}_{\psi \mathcal{U}_\chi^d \theta}$ such that for any Φ in it we have $\chi_F^{\lessdot}(\psi \, \mathcal{U}_\chi^d \, \theta), \chi_F^{\doteq}(\psi \, \mathcal{U}_\chi^d \, \theta) \notin \Phi_s$, and either $\psi \, \mathcal{U}_\chi^d \, \theta \notin \Phi_c$ or $\theta \in \Phi_c$. The condition for $\psi \, \mathcal{U}_\chi^u \, \theta$ is symmetric.

4.2 Complexity

The set $Cl(\varphi)$ is linear in $|\varphi|$, the length of φ. *Atoms*(φ) has size at most $2^{|Cl(\varphi)|} = 2^{O(|\varphi|)}$, and the size of the set of states is the square of that in the finite case, and is bounded by its cube in the ω-case. Moreover, the use of the equivalences for the hierarchical operators causes only a linear increase in the length of φ. Therefore,

Theorem 2. *Given a POTL formula φ, it is possible to build an OPA or an ωOPBA $\mathcal{A}_\varphi$ accepting the language denoted by φ with at most $2^{O(|\varphi|)}$ states.*

$\mathcal{A}_\varphi$ can then be intersected [42] with an OPA/ωOPBA modeling a program (e.g. Fig. 2), and emptiness can be decided with *summarization* techniques [4].

Table 1. Results of the evaluation. '# states' refers to the OPA to be verified.

	Benchmark name	# states	Time (ms)	Memory (KiB)		Result
				Total	MC only	
1	Generic (Fig. 2)	12	867	70, 040	10, 166	True
2	Generic medium	24	673	70, 064	4, 043	False
3	Generic larger	30	1, 014	70, 063	14, 160	True
4	Jensen	42	305	70, 050	3, 154	True
5	Unsafe stack	63	1, 493	109, 610	43, 177	False
6	Safe stack	77	637	70, 089	7, 234	True
7	Unsafe stack neutrality	63	5, 286	383, 312	167, 654	True
8	Safe stack neutrality	77	840	70, 077	16, 773	True

5 Experimental Evaluation

We implemented the OPA construction of Sect. 4 in an explicit-state model checking tool called POMC. The tool is written in Haskell [45], a purely functional, statically typed programming language with lazy evaluation. POMC checks OPA for emptiness by checking the reachability of an accepting configuration, by means of a modified DFS of the transition relation. This algorithm, similar to the one in [9], exploits the fact that all transitions only consider the topmost stack symbol, so reachability is actually computed only for *semi-configurations* made of one stack symbol and one state. Each time a chain support is explored, its ending semi-configuration is saved and associated with the starting one, so the next time the latter is reached, the support does not have to be re-explored. This allows the algorithm to exploit the cyclicities of OPA to terminate after having explored the whole transition relation. Given a POTL specification φ and an OPA $\mathcal{A}$ to be checked, POMC executes the reachability algorithm, generating the product between $\mathcal{A}$ and the OPA for $\neg\varphi$ on-the-fly. The present prototype of POMC only supports finite-word model checking; its extension to deal with ω-languages is under development.

We checked with POMC several requirements on three case studies and we report the results in Table 1. Some additional formulas we checked are in Table 2. Such results can be reproduced through a publicly available artifact.[2] The experiments were executed on a laptop with a 2.2 GHz Intel processor and 15 GiB of RAM, running Ubuntu GNU/Linux 20.04. In the tables, by "Total" memory we mean the maximum resident memory including the Haskell runtime (which allocates 70 MiB by default), and by "MC only" the maximum memory used by model checking as reported by the runtime. Since model checking is polynomial in OPA size and exponential in formula length, we focus on checking a variety of requirements, rather than large OPA.

[2] https://doi.org/10.5281/zenodo.4723741.

Generic Procedural Program. We checked formula

$$\square\big((\mathbf{call} \land \mathrm{p}_B \land \mathit{Scall}(\top, \mathrm{p}_A)) \implies \mathit{CallThr}(\top)\big)$$

from Sect. 3.1 on the OPA of Fig. 2 (bench. 1), and also against two larger OPA (2, where the property does not hold, and 3, where it holds).

We also checked the largest of such OPA against a set of formulas devised with the purpose of testing all POTL operators. The results are reported in Table 2. All formulas are checked very quickly, with only one outlier that runs out of memory. We ran the same experiment on a machine with a 2.0 GHz AMD CPU and 512 GiB of RAM running Debian GNU/Linux 10, obtaining a time of 367 s with a memory occupancy of 16.3 GiB.

Stack Inspection. The security framework of the Java Development Kit (JDK) is based on stack inspection, i.e. the analysis of the contents of the program's stack during the execution. The JDK provides method `checkPermission(perm)` from class `AccessController`, which searches the stack for frames of functions that have not been granted permission `perm`. If any are found, an exception is thrown. Such permission checks prevent the execution of privileged code by unauthorized parts of the program, but they must be placed in sensitive points manually. Failure to place them appropriately may cause the unauthorized execution of privileged code. An automated tool to check that no code can escape such checks is thus desirable. Any such tool would need the ability to model exceptions, as they are used to avoid code execution in case of security violations.

[37] explains such needs by providing an example Java program for managing a bank account. It allows the user to check the account balance, and to withdraw money. To perform such tasks, the invoking program must have been granted permissions `CanPay` and `Debit`, respectively. We modeled such program as an OPA (4), and proved that the program enforces such security measures effectively by checking it against the formula

$$\square(\mathbf{call} \land \mathtt{read} \implies \neg(\top\ \mathcal{S}^d_\chi\ (\mathbf{call} \land \neg\mathtt{CanPay} \land \neg\mathtt{read})))$$

meaning that the account balance cannot be read if some function in the stack lacks the `CanPay` permission (a similar formula checks the `Debit` permission).

Exception Safety. [53] is a tutorial on how to make exception-safe generic containers in C++. It presents two implementations of a generic stack data structure, parametric on the element type `T`. The first one is not exception-safe: if the constructor of `T` throws an exception during a pop action, the topmost element is removed, but it is not returned, and it is lost. This violates the strong exception safety requirement that each operation is rolled back if an exception is thrown. The second version of the data structure instead satisfies such requirement.

While exception safety is, in general, undecidable, it is possible to prove the stronger requirement that each modification to the data structure is only committed once no more exceptions can be thrown. We modeled both versions as OPA, and checked such requirement with the following formula:

$$\square(\mathbf{exc} \implies \neg((\circleddash^u \mathtt{modified} \lor \chi^u_P \mathtt{modified}) \land \chi^u_P(\mathtt{Stack::push} \lor \mathtt{Stack::pop})))$$

POMC successfully found a counterexample for the first implementation (5), and proved the safety of the second one (6).

Additionally, we proved that both implementations are *exception neutral* (7, 8), i.e. `Stack` functions do not block exceptions thrown by the underlying type `T`. This was accomplished by checking the following formula:

$$\square(\mathbf{exc} \wedge \ominus^u \mathtt{T} \wedge \chi_P^d(\mathbf{han} \wedge \chi_P^d \mathtt{Stack}) \implies \chi_P^d \chi_P^d \chi_F^u \mathbf{exc}).$$

Table 2. Results of the additional experiments on OPA "generic larger".

Formula	Time (ms)	Memory (KiB) Tot.	Memory (KiB) MC	Result
$\chi_F^d \mathrm{p}_{Err}$	1.1	70,095	175	False
$\bigcirc^d(\bigcirc^d(\mathbf{call} \wedge \chi_F^u \mathbf{exc}))$	21.0	70,095	1,290	False
$\bigcirc^d(\mathbf{han} \wedge (\chi_F^d(\mathbf{exc} \wedge \chi_P^u \mathbf{call})))$	42.2	70,088	2,297	False
$\square(\mathbf{exc} \implies \chi_P^u \mathbf{call})$	10.7	70,099	839	True
$\top \, \mathcal{U}_\chi^d \, \mathbf{exc}$	2.2	70,093	121	False
$\bigcirc^d(\bigcirc^d(\top \, \mathcal{U}_\chi^d \, \mathbf{exc}))$	4.3	70,094	113	False
$\square((\mathbf{call} \wedge \mathrm{p}_A \wedge (\neg \mathbf{ret} \, \mathcal{U}_\chi^d \, \mathrm{WRx})) \implies \chi_F^u \mathbf{exc})$	3,257.7	238,833	102,582	True
$\bigcirc^d(\bigcirc^u \mathbf{call})$	0.7	70,094	139	False
$\bigcirc^d(\bigcirc^d(\bigcirc^d(\ominus^u \mathbf{call})))$	3.4	70,108	126	False
$\chi_F^d(\bigcirc^d(\ominus^u \mathbf{call}))$	1.3	70,096	137	False
$\square((\mathbf{call} \wedge \mathrm{p}_A \wedge \mathit{CallThr}(\top)) \implies \mathit{CallThr}(\mathrm{e}_B))$	7,793.7	402,420	173,639	False
$\Diamond(\bigcirc_H^d \mathrm{p}_B)$	2.1	70,097	114	False
$\Diamond(\ominus_H^d \mathrm{p}_B)$	2.8	70,097	114	False
$\Diamond(\mathrm{p}_A \wedge (\mathbf{call} \, \mathcal{U}_H^d \, \mathrm{p}_C))$	594.9	77,806	29,786	True
$\Diamond(\mathrm{p}_C \wedge (\mathbf{call} \, \mathcal{S}_H^d \, \mathrm{p}_A))$	676.6	96,296	37,949	True
$\square((\mathrm{p}_C \wedge \chi_F^u \mathbf{exc}) \implies (\neg \mathrm{p}_A \, \mathcal{S}_H^d \, \mathrm{p}_B))$	—	—	—	OOM
$\square(\mathbf{call} \wedge \mathrm{p}_B \implies \neg \mathrm{p}_C \, \mathcal{U}_H^u \, \mathrm{p}_{Err})$	198.2	70,088	10,606	True
$\Diamond(\bigcirc_H^u \mathrm{p}_{Err})$	1.1	70,093	114	False
$\Diamond(\ominus_H^u \mathrm{p}_{Err})$	1.2	70,089	114	False
$\Diamond(\mathrm{p}_A \wedge (\mathbf{call} \, \mathcal{U}_H^u \, \mathrm{p}_B))$	10.3	70,105	115	False
$\Diamond(\mathrm{p}_B \wedge (\mathbf{call} \, \mathcal{S}_H^u \, \mathrm{p}_A))$	10.8	70,095	115	False
$\square(\mathbf{call} \implies \chi_F^d \mathbf{ret})$	3.0	70,095	112	False
$\square(\mathbf{call} \implies \neg \bigcirc^u \, \mathbf{exc})$	1.9	70,106	113	False
$\square(\mathbf{call} \wedge \mathrm{p}_A \implies \neg \mathit{CallThr}(\top))$	110.7	70,094	4,937	False
$\square(\mathbf{exc} \implies \neg(\ominus^u(\mathbf{call} \wedge \mathrm{p}_A) \vee \chi_P^u(\mathbf{call} \wedge \mathrm{p}_A)))$	28.9	70,095	112	False
$\square((\mathbf{call} \wedge \mathrm{p}_B \wedge (\mathbf{call} \, \mathcal{S}_\chi^d \, (\mathbf{call} \wedge \mathrm{p}_A))) \implies \mathit{CallThr}(\top)$	926.1	70,104	13,310	True
$\square(\mathbf{han} \implies \chi_F^u \mathbf{ret})$	17.0	70,079	1,252	True
$\top \, \mathcal{U}_\chi^u \, \mathbf{exc}$	7.7	70,101	121	True
$\bigcirc^d(\bigcirc^d(\top \, \mathcal{U}_\chi^u \, \mathbf{exc}))$	44.6	70,104	2,376	True
$\bigcirc^d(\bigcirc^d(\bigcirc^d(\top \, \mathcal{U}_\chi^u \, \mathbf{exc})))$	123.7	70,090	5,261	False
$\square(\mathbf{call} \wedge \mathrm{p}_C \implies (\top \, \mathcal{U}_\chi^u \, \mathbf{exc} \wedge \chi_P^d \mathbf{han}))$	92.9	70,096	1,346	False
$\mathbf{call} \, \mathcal{U}_\chi^d \, (\mathbf{ret} \wedge \mathrm{p}_{Err})$	1.8	70,107	114	False
$\chi_F^d(\mathbf{call} \wedge ((\mathbf{call} \vee \mathbf{exc}) \, \mathcal{S}_\chi^u \, \mathrm{p}_B))$	10.8	70,086	117	False
$\bigcirc^d(\bigcirc^d((\mathbf{call} \vee \mathbf{exc}) \, \mathcal{U}_\chi^u \, \mathbf{ret}))$	5.3	70,094	114	False

6 Conclusions

We introduced the temporal logic POTL, gave an automata-theoretic model checking procedure, and implemented it in a prototype tool. The results obtained in its experimental evaluation are promising. Additionally, POTL is proved to be FO-complete in a technical report [23]. We argue that the strong gain in expressive power w.r.t. previous approaches to model checking CFL, which comes without an increase in computational complexity, is worth the technicalities needed to achieve the present—and future—results.

In the evaluation, we used models directly coded into OPAs. To ease user interaction with our tool, we additionally implemented a new input format based on a simple procedural language with exceptions and Boolean variables, which is automatically translated into OPA. Moreover, we are currently working on the implementation of the model checking for ω-words, described in Sect. 4.1.

As a future research step, we plan to develop user-friendly domain-specific languages for specification too, to prove that OP languages and logics are suitable in practice to program verification.

Acknowledgments. We are thankful to Davide Bergamaschi for developing an early POMC prototype, and to Francesco Pontiggia for implementing performance optimizations.

References

1. Abrahams, D.: Exception-Fsaety in generic components. In: Jazayeri, M., Loos, R.G.K., Musser, D.R. (eds.) Generic Programming. LNCS, vol. 1766, pp. 69–79. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-39953-4_6
2. Alur, R., Arenas, M., Barceló, P., Etessami, K., Immerman, N., Libkin, L.: First-order and temporal logics for nested words. LMCS **4**(4), 1–44 (2008)
3. Alur, R., Benedikt, M., Etessami, K., Godefroid, P., Reps, T., Yannakakis, M.: Analysis of recursive state machines. ACM Trans. Program. Lang. Syst. **27**(4), 786–818 (2005). https://doi.org/10.1145/1075382.1075387
4. Alur, R., Bouajjani, A., Esparza, J.: Model checking procedural programs. Handbook of Model Checking, pp. 541–572. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_17
5. Alur, R., Chaudhuri, S., Madhusudan, P.: Software model checking using languages of nested trees. ACM Trans. Program. Lang. Syst. **33**(5), 15:1–15:45 (2011)
6. Alur, R., Etessami, K., Madhusudan, P.: A temporal logic of nested calls and returns. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 467–481. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_35
7. Alur, R., Madhusudan, P.: Visibly pushdown languages. In: ACM STOC (2004)
8. Alur, R., Madhusudan, P.: Adding nesting structure to words. JACM **56**(3), 1–43 (2009)
9. Alur, R., Chaudhuri, S., Etessami, K., Madhusudan, P.: On-the-fly reachability and cycle detection for recursive state machines. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 61–76. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31980-1_5

10. Ball, T., Rajamani, S.K.: Bebop: a symbolic model checker for Boolean programs. In: Havelund, K., Penix, J., Visser, W. (eds.) SPIN 2000. LNCS, vol. 1885, pp. 113–130. Springer, Heidelberg (2000). https://doi.org/10.1007/10722468_7
11. Barenghi, A., Crespi Reghizzi, S., Mandrioli, D., Panella, F., Pradella, M.: Parallel parsing made practical. Sci. Comput. Program. **112**, 195–226 (2015). https://doi.org/10.1016/j.scico.2015.09.002
12. Bouajjani, A., Echahed, R., Habermehl, P.: On the verification problem of nonregular properties for nonregular processes. LICS **95**, 123–133 (1995)
13. Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: application to model-checking. In: Mazurkiewicz, A., Winkowski, J. (eds.) CONCUR 1997. LNCS, vol. 1243, pp. 135–150. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-63141-0_10
14. Bouajjani, A., Habermehl, P.: Constrained properties, semilinear systems, and Petri nets. In: Montanari, U., Sassone, V. (eds.) CONCUR 1996. LNCS, vol. 1119, pp. 481–497. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61604-7_71
15. Bozzelli, L., Murano, A., Peron, A.: Timed context-free temporal logics. In: GandALF 2018. EPTCS, vol. 277, pp. 235–249. Open Publishing Association (2018). https://doi.org/10.4204/EPTCS.277.17
16. Bozzelli, L., Sánchez, C.: Visibly linear temporal logic. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) IJCAR 2014. LNCS (LNAI), vol. 8562, pp. 418–433. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08587-6_33
17. Burkart, O., Steffen, B.: Model checking the full modal mu-calculus for infinite sequential processes. Theor. Comput. Sci. **221**(1–2), 251–270 (1999). https://doi.org/10.1016/S0304-3975(99)00034-1
18. Chatterjee, K., Ma, D., Majumdar, R., Zhao, T., Henzinger, T.A., Palsberg, J.: Stack size analysis for interrupt-driven programs. Inf. Comput. **194**(2), 144–174 (2004). https://doi.org/10.1016/j.ic.2004.06.001
19. Chaudhuri, S., Alur, R.: Instrumenting C programs with nested word monitors. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 279–283. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73370-6_20
20. Chen, F., Roşu, G.: Java-MOP: a monitoring oriented programming environment for Java. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 546–550. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31980-1_36
21. Chiari, M., Mandrioli, D., Pradella, M.: POTL: a first-order complete temporal logic for operator precedence languages. CoRR abs/1910.09327 (2019). http://arxiv.org/abs/1910.09327
22. Chiari, M., Mandrioli, D., Pradella, M.: Operator precedence temporal logic and model checking. Theor. Comput. Sci. **848**, 47–81 (2020). https://doi.org/10.1016/j.tcs.2020.08.034
23. Chiari, M., Mandrioli, D., Pradella, M.: A first-order complete temporal logic for structured context-free languages. CoRR abs/2105.10740 (2021). https://arxiv.org/abs/2105.10740
24. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.): Handbook of Model Checking. Springer, Heidelberg (2018). https://doi.org/10.1007/978-3-319-10575-8
25. Crespi Reghizzi, S., Mandrioli, D.: Operator precedence and the visibly pushdown property. JCSS **78**(6), 1837–1867 (2012). https://doi.org/10.1016/j.jcss.2011.12.006

26. D'Antoni, L.: A symbolic automata library. https://github.com/lorisdanto/symbolicautomata
27. Driscoll, E., Thakur, A., Reps, T.: OpenNWA: a nested-word automaton library. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 665–671. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31424-7_47
28. Esparza, J., Hansel, D., Rossmanith, P., Schwoon, S.: Efficient algorithms for model checking pushdown systems. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 232–247. Springer, Heidelberg (2000). https://doi.org/10.1007/10722167_20
29. Esparza, J., Kučera, A., Schwoon, S.: Model checking LTL with regular valuations for pushdown systems. Inf. Comput. **186**(2), 355–376 (2003)
30. Finkel, A., Willems, B., Wolper, P.: A direct symbolic approach to model checking pushdown systems. In: Infinity 1997. ENTCS, vol. 9, pp. 27–37. Elsevier (1997). https://doi.org/10.1016/S1571-0661(05)80426-8
31. Floyd, R.W.: Syntactic analysis and operator precedence. JACM **10**(3), 316–333 (1963). https://doi.org/10.1145/321172.321179
32. Godefroid, P., Yannakakis, M.: Analysis of Boolean programs. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 214–229. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_16
33. Grune, D., Jacobs, C.J.: Parsing Techniques: A Practical Guide. Springer, New York (2008). https://doi.org/10.1007/978-0-387-68954-8
34. Harel, D., Kozen, D., Tiuryn, J.: Dynamic logic. In: Gabbay, D.M., Guenthner, F. (eds.) Handbook of Philosophical Logic. Handbook of Philosophical Logic, vol. 4. Springer, Dordrecht (2001). https://doi.org/10.1007/978-94-017-0456-4_2
35. Harrison, M.A.: Introduction to Formal Language Theory. Addison Wesley, Boston (1978)
36. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Software verification with BLAST. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. Software verification with BLAST, vol. 2648, pp. 235–239. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-44829-2_17
37. Jensen, T., Le Metayer, D., Thorn, T.: Verification of control flow based security properties. In: Proceedings of 1999 IEEE Symposium on Security and Privacy, pp. 89–103 (1999). https://doi.org/10.1109/SECPRI.1999.766902
38. Jhala, R., Podelski, A., Rybalchenko, A.: Predicate abstraction for program verification. Handbook of Model Checking, pp. 447–491. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_15
39. Kamp, H.: Tense logic and the theory of linear order. Ph.D. thesis, University of California, Los Angeles (1968)
40. Kupferman, O., Piterman, N., Vardi, M.Y.: Model checking linear properties of prefix-recognizable systems. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 371–385. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45657-0_31
41. Kupferman, O., Piterman, N., Vardi, M.Y.: Pushdown specifications. In: Baaz, M., Voronkov, A. (eds.) LPAR 2002. LNCS (LNAI), vol. 2514, pp. 262–277. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-36078-6_18
42. Lonati, V., Mandrioli, D., Panella, F., Pradella, M.: Operator precedence languages: their automata-theoretic and logic characterization. SIAM J. Comput. **44**(4), 1026–1088 (2015). https://doi.org/10.1137/140978818
43. Mandrioli, D., Pradella, M.: Generalizing input-driven languages: theoretical and practical benefits. Comput. Sci. Rev. **27**, 61–87 (2018). https://doi.org/10.1016/j.cosrev.2017.12.001

44. Mandrioli, D., Pradella, M., Crespi Reghizzi, S.: Star-freeness, first-order definability and aperiodicity of structured context-free languages. In: Pun, V.K.I., Stolz, V., Simao, A. (eds.) ICTAC 2020. LNCS, vol. 12545, pp. 161–180. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-64276-1_9
45. Marlow, S.: Haskell 2010 language report (2010). https://www.haskell.org/onlinereport/haskell2010/
46. McNaughton, R.: Parenthesis grammars. JACM **14**(3), 490–500 (1967)
47. Mehlhorn, K.: Pebbling mountain ranges and its application to DCFL-recognition. In: de Bakker, J., van Leeuwen, J. (eds.) ICALP 1980. LNCS, vol. 85, pp. 422–435. Springer, Heidelberg (1980). https://doi.org/10.1007/3-540-10003-2_89
48. Nguyen, H.: Visibly pushdown automata library (2006). https://web.imt-atlantique.fr/x-info/hnguyen/vpa
49. Nguyen, H., Touili, T.: CARET model checking for malware detection. In: SPIN 2017, pp. 152–161. ACM (2017). https://doi.org/10.1145/3092282.3092301
50. Nguyen, H., Touili, T.: CARET model checking for pushdown systems. In: SAC 2017, pp. 1393–1400. ACM (2017). https://doi.org/10.1145/3019612.3019829
51. Piterman, N., Vardi, M.Y.: Global model-checking of infinite-state systems. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 387–400. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27813-9_30
52. Roşu, G., Chen, F., Ball, T.: Synthesizing monitors for safety properties: this time with calls and returns. In: Leucker, M. (ed.) RV 2008. LNCS, vol. 5289, pp. 51–68. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89247-2_4
53. Sutter, H.: Exception-safe generic containers. C++ Report (1997). https://ptgmedia.pearsoncmg.com/imprint_downloads/informit/aw/meyerscddemo/DEMO/MAGAZINE/SU_FRAME.HTM
54. Tang, N.V., Ohsaki, H.: Checking on-the-fly universality and inclusion problems of visibly pushdown automata. IEICE Trans. Fundam. Electron. Commun. Comput. Sci. **94-A**(12), 2794–2801 (2011). https://doi.org/10.1587/transfun.E94.A.2794
55. Walukiewicz, I.: Pushdown processes: games and model-checking. Inf. Comput. **164**(2), 234–263 (2001). https://doi.org/10.1006/inco.2000.2894

Model Checking ω-Regular Properties with Decoupled Search

Daniel Gnad[1(✉)], Jan Eisenhut[1], Alberto Lluch Lafuente[2], and Jörg Hoffmann[1]

[1] Saarland University, Saarland Informatics Campus, Saarbrücken, Germany
{gnad,hoffmann}@cs.uni-saarland.de,
s8jaeise@stud.uni-saarland.de

[2] Technical University of Denmark, Kongens Lyngby, Denmark
albl@dtu.dk

Abstract. Decoupled search is a state space search method originally introduced in AI Planning. Similar to partial-order reduction methods, decoupled search exploits the independence of components to tackle the state explosion problem. Similar to symbolic representations, it does not construct the explicit state space, but sets of states are represented in a compact manner, exploiting component independence. Given the success of both partial-order reduction and symbolic representations when model checking liveness properties, our goal is to add decoupled search to the toolset of liveness checking methods. Specifically, we show how decoupled search can be applied to liveness verification for composed Büchi automata by adapting, and showing correct, a standard algorithm for detecting lassos (i.e., infinite accepting runs), namely nested depth-first search. We evaluate our approach using a prototype implementation.

1 Introduction

Model checking is a well-known problem in formal verification. Given a formal description of a system $\mathcal{M}$, the model checking problem is to decide whether the system satisfies a property ϕ. In contrast to safety properties, which can only express whether there exists a finite run of the system that reaches a state with certain (bad) properties, liveness properties can express good behaviours of the system that should occur repeatedly, i.e., infinite runs in which something good happens infinitely often.

In this work, we consider a liveness verification problem that arises when composing a set $\mathcal{A}^1, \dots, \mathcal{A}^n$ of *non-deterministic Büchi automata* (NBA), each with its own acceptance condition. We recall that an accepting run for a single NBA is a *lasso* $\rho_p(\rho_c)^\omega$ with a prefix ρ_p and a cycle ρ_c that visits an accepting state. For the composition of a set of NBAs into an NBA we consider the following liveness property: a composed run is accepting if there is a cycle visiting a state that is accepting for *all* components. Such a general problem captures standard liveness verification problems related to ω-regular properties. An archetypal example is automata-based LTL checking, where system components are represented as NBAs and are composed with a property monitor, represented as a Büchi automaton (often the negation of an LTL property). In this case an accepting composed run witnesses a violation of a linear-time property.

The predominant approach to address such verification problems using explicit state space search is to use *nested depth-first search* (NDFS) algorithms [5,22,32],

A. Silva and K. R. M. Leino (Eds.): CAV 2021, LNCS 12760, pp. 411–434, 2021.
https://doi.org/10.1007/978-3-030-81688-9_19

also called *double depth-first search*, which perform on-the-fly checking of liveness properties while composing the NBAs. NDFS, like all state space search methods, suffers from the state explosion problem. Various methods, such as partial-order reduction [10,19,27,30,34], symbolic representations [2,28], symmetry reduction [7,23], or Petri-net unfolding [8,9] have been proposed to alleviate the state explosion problem. Here, we add *decoupled state space search* [14], shortly *decoupled search*, as a new method for model checking liveness properties, complementary to the existing approaches. Indeed, as Gnad and Hoffmann [14,15] have shown, decoupled search complements these techniques in the sense that there exist cases where it yields exponentially stronger reductions. It has also been shown that decoupled search can be fruitfully combined with partial-order reduction [16], symmetry reduction [18], and symbolic search [17].

Decoupled search has recently been introduced in AI planning [14], addressing goal reachability problems. Its applicability to model checking of safety properties has been shown in [12], where it was effectively introduced into the SPIN model checker [20]. However, the extension of decoupled search to cycle detection problems inherent to liveness model checking and NDFS algorithms has not yet been investigated. This paper addresses that investigation for the first time.

Decoupled search exploits the independence of system components, similar to partial-order reduction techniques, by not enumerating all interleavings of transitions across components. Similar to symbolic representations, decoupled search does not construct the explicit state space of the product. Instead, search nodes, called *decoupled states*, symbolically represent sets of states. Each decoupled state compactly represents many global states and their closure up to internal transitions of individual components. Similar to partial-order reduction or symbolic search, decoupled search can be exponentially more efficient than explicit search of the state space, as shown for reachability problems in the domains of AI planning [14] and model checking [12].

The main contribution of our paper is to extend the scope of decoupled search from safety properties, as done in [12], to liveness properties. In particular, we adapt a standard NDFS algorithm to the decoupled state representation. The resulting algorithms are able to solve the verification problem mentioned above, namely checking acceptance of composed NBAs. The main technical challenge for the correctness of our algorithms was to identify the conditions that imply existence of accepting runs in decoupled search and to show how such runs can be constructed efficiently.

We evaluate our decoupled NDFS algorithm using a prototype implementation on two showcase examples similar to the dining philosophers problem, and a set of randomly generated models. We compare to established tools, namely the SPIN model checker [20], and Petri-net unfolding with Cunf [30]. The results show that, like for safety properties, decoupled search can yield exponential advantages over state-of-the-art methods. In particular, its advantage grows with the degree to which components act independently of others, via internal transitions that do not affect other components.

The rest of the paper is structured as follows. We start in Sect. 2 by recalling the necessary background on NBAs, the verification problem we consider, and a standard NDFS algorithm typically used to solve the problem. Sections 3–5 present our contribution: Sect. 3 formalizes decoupled search in terms of composed NBAs, and shows its desired properties; Sect. 4 discusses some issues that would arise in a naïve attempt to (incorrectly) adapt it, and describes the (correct) adapted NDFS algorithm; Sect. 5 provides its correctness proof. In Sect. 6 we show our experimental evaluation, whose code and models are publicly available at [13]. Section 7 concludes the paper discussing related works and future research avenues.

2 Büchi Automata, Composition and Verification

This section recalls some basic notions of Büchi automata, their composition, the verification problem we consider in this paper for such composition, and its standard algorithmic resolution based on NDFS.

Büchi Automata and Accepting Runs. We start with the definition of non-deterministic Büchi automata (NBA).

Definition 1 (Non-deterministic Büchi Automaton). *A* non-determinitic Büchi automaton *$\mathcal{A}$ is a tuple $\langle S, \rightarrow, L, s_0, A\rangle$, where S is a finite set of* states*, L is a finite set of* transition labels*, $\rightarrow \subseteq S \times L \times S$ is a* transition relation*, $s_0 \in S$ is an* initial state*, and $A \in (S \rightarrow \mathbb{B})$ is an* acceptance function*.*

A run ρ of an NBA is an infinite sequence of states $s_0, s_1, s_2, \cdots \in S^\omega$ starting from the initial state. The i-th state of a run ρ is denoted by $\rho[i]$ and we will use the same notation for other lists and sequences. A run ρ is accepting if it traverses accepting states infinitely often. Formally, $\overset{\infty}{\exists} j \in \mathbb{N} : A(s[j])$. We define a *trace* π of a run $\rho = s_0, s_1, s_2, \cdots \in S^\omega$ as a sequence of labels $\pi = l_0, l_1, \cdots \in L^\omega$ such that $\forall_{i\in\mathbb{N}} : \langle s_i, l_i, s_{i+1}\rangle \in\rightarrow$. We will also consider *finite* runs $\rho \in S^n$ and *finite* traces $\pi \in L^n$.

As hinted in Sect. 1, the existence of accepting runs is interesting for several theoretical and practical reasons. On the theoretical side, the language of an NBA is the set of all traces σ in L^ω for which an accepting run exists such that $\rho[i] \xrightarrow{\sigma[i]} \rho[i+1]$ for all $i \in \mathbb{N}$. On the practical side, model checking ω-regular properties, including LTL properties, can be reduced to checking the existence of accepting runs. Such runs, indeed, provide witnesses or counterexamples for the properties of interest.

Composition of NBAs. From now on we assume that the set of labels L of an NBA is partitioned into a set L_I of *internal labels* and a set L_G of *global labels*. The notion of composition we use is based on (maximal) synchronisation on global labels, in words: in every transition involving a global label, each component having the global label in its set of labels must perform a local transition, while transitions with internal labels can be performed independently. When composing NBAs we assume w.l.o.g. that they do not share any internal label. Further, we assume that every global label is shared by at least two component NBAs. Otherwise, such labels can be made internal. We will use the following notation: for a set $\mathcal{A}^1, \dots, \mathcal{A}^n$ of NBAs, we use superscripting to denote the components of each $\mathcal{A}^i$, i.e., we assume $\mathcal{A}^i = \langle S^i, \rightarrow^i, L^i = L^i_I \cup L^i_G, s^i_0, A^i\rangle$.

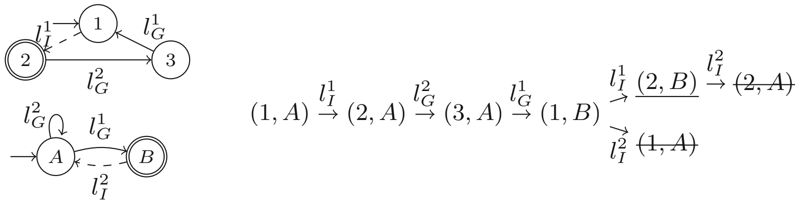
Definition 2 (Composition of NBAs). *The* composition *of n NBAs $\mathcal{A}^1, \dots, \mathcal{A}^n$, denoted by $\mathcal{A}^1 \parallel \dots \parallel \mathcal{A}^n$, is the NBA $\langle S, \rightarrow, L, \boldsymbol{s}_0, A\rangle$, where $S = S^1 \times \dots \times S^n$, $L = \bigcup_{i\in\{1,\dots,n\}} L^i$, $\boldsymbol{s}_0 = (s_0^1, \dots, s_0^n)$, $A = \{(s_1, \dots, s_n) \mapsto \wedge_{i=1,\dots,n} A^i(s_i)\}$ and $\rightarrow$ is the smallest set of transitions closed under the following rules for interleaving of local transitions (1) and maximal synchronization on global labels (2):*

$$(1)\ \frac{s_i \xrightarrow{l_I} s_i' \qquad l_I \in L_I^i}{(s_1, \dots, s_i, \dots, s_n) \xrightarrow{l_I} (s_1, \dots, s_i', \dots, s_n)}$$

$$(2)\ \frac{\exists_{i\in\{1,\dots,n\}} : l_G \in L_G^i \quad \forall_{j\in\{1,\dots,n | l_G \in L_G^j\}} : s_j \xrightarrow{l_G} s_j' \quad \forall_{j\in\{1,\dots,n | l_G \notin L_G^j\}} : s_j' = s_j}{(s_1, \dots, s_n) \xrightarrow{l_G} (s_1', \dots, s_n')}$$

As notation convention, we will denote component states simply by small case letters, e.g. s, and composed states $(s_1, \dots, s_n) \in S$ by $\boldsymbol{s}$, i.e., as a vector, and similarly for local runs ρ (resp. traces π) and composed runs $\boldsymbol{\rho}$ (composed traces $\boldsymbol{\pi}$).

In Fig. 1 we illustrate a small example of a composition of two NBAs $\mathcal{A}^1, \mathcal{A}^2$. In the top of the figure, we show the local state space of the two components ($\mathcal{A}^1$ left, $\mathcal{A}^2$ right), where the component states are $S^1 = \{1, 2, 3\}$, $S^2 = \{A, B\}$, and the labels are defined as $L_G^1 = L_G^2 = \{l_G^1, l_G^2\}$, $L_I^1 = \{l_I^1\}$, $L_I^2 = \{l_I^2\}$. A local state is accepting for $\mathcal{A}^1$, so $A^1(s) = \top$, iff $s = 2$, and similar $A^2(s) = \top$ iff $s = B$. The initial states are $s_0^1 = 1$ and $s_0^2 = A$. The transitions are as shown. In the bottom, we depict the part of the state space of the composition $\mathcal{A}^1 \parallel \mathcal{A}^2$ reachable from $\boldsymbol{s}_0 = (1, A)$ as it would be generated by a standard DFS. Here, transitions via global labels synchronize the components, internal transitions are executed independently. The states crossed out would be pruned by duplicate checking, the underlined state is accepting.



1
2
3
l_I^1
l_G^1
l_G^2
A
B
l_G^2
l_G^1
l_I^2
$(1, A) \xrightarrow{l_I^1} (2, A) \xrightarrow{l_G^2} (3, A) \xrightarrow{l_G^1} (1, B)$
$\xrightarrow{l_I^1} \underline{(2, B)} \xrightarrow{l_I^2} \cancel{(2, A)}$
$\xrightarrow{l_I^2} \cancel{(1, A)}$

Fig. 1. Example of two NBAs, $\mathcal{A}^1$ and $\mathcal{A}^2$, and the state space of their composition $\mathcal{A}^1 \parallel \mathcal{A}^2$.

Verification Problem and Its Resolution with NDFS. The verification problem we address in this paper is the existence of accepting runs in the composed NBA $\mathcal{A}^1 \parallel \dots \parallel \mathcal{A}^n$. In words, we look for runs in $\mathcal{A}^1 \parallel \dots \parallel \mathcal{A}^n$ that infinitely often traverse states in which *all* component NBAs are in an accepting state. We discuss alternative acceptance conditions in Sect. 7.

Determining the existence of accepting runs in an NBA can be boiled down to the existence of so-called *lassos*, i.e., finite sequences of states in the NBA of the form $\boldsymbol{\rho}_p\boldsymbol{\rho}_c$

```
**CheckEmptiness**($\mathcal{A}^1 \parallel \dots \parallel \mathcal{A}^n$):
    Stack $\leftarrow \langle \boldsymbol{s}_0 \rangle$
    $V \leftarrow \emptyset$
    $V' \leftarrow \emptyset$
    DFS($\boldsymbol{s}_0$)
    **return** empty

**NestedDFS**($\boldsymbol{s}$):
    **for all** $\boldsymbol{t}$ s.t. $\boldsymbol{s} \rightarrow \boldsymbol{t}$ **do**
        **if** $\boldsymbol{t} \in V'$ **then continue**
        **if** $\boldsymbol{t} \in$ Stack **then return** cycle
        $V' = V' \cup \{\boldsymbol{t}\}$
        NestedDFS($\boldsymbol{t}$)

**DFS**($\boldsymbol{s}$):
    $V = V \cup \{\boldsymbol{s}\}$
    **for all** $\boldsymbol{t}$ s.t. $\boldsymbol{s} \rightarrow \boldsymbol{t}$ **do**
        **if** $\boldsymbol{t} \in V$ **then continue**
        push(Stack, $\boldsymbol{t}$)
        DFS($\boldsymbol{t}$)
        pop(Stack)
    **if** $A(\boldsymbol{s})$ **then**
        NestedDFS($\boldsymbol{s}$)
        $V' = V' \cup \{\boldsymbol{s}\}$
```

Fig. 2. A standard NDFS algorithm for lasso search in composed NBAs.

where $\boldsymbol{\rho}_p$ is the prefix of the lasso and $\boldsymbol{\rho}_c$ is the cycle of the lasso, which contains at least one accepting state and closes the cycle (i.e., $\boldsymbol{\rho}_p[|\boldsymbol{\rho}_p| - 1] = \boldsymbol{\rho}_c[|\boldsymbol{\rho}_c| - 1]$. Such a finite sequence of states represents an accepting run $\boldsymbol{\rho}_p(\boldsymbol{\rho}_c)^\omega$.

Several algorithms can be used to check the existence of lassos. The predominant family of algorithms are the variants of NDFS, originally introduced in [5]. Figure 2 shows the pseudo-code for one such variant, based on NDFS as presented in [4]. The algorithm is based on an ordinary depth-first search algorithm (**DFS**) that works as usual: a set V is used to record already visited states, and recursion enforces the depth-first exploration order of the state space. Moreover, a stack $Stack$ is used to keep track of the states on the current initial trace being explored. The main difference w.r.t. ordinary DFS is that a second, nested, depth-first search algorithm (**NestedDFS**) is invoked from accepting states on backtracking, i.e., after the recursive call to **DFS**. The idea is that, if this second depth-first search finds a state that is on $Stack$, then it is guaranteed that a cycle has been found, which contains at least one accepting state. That is, one finds the (un)desired lasso. The algorithm is also complete: no accepting cycle is missed.

$$\begin{tikzcd}
(1,A) \arrow[r, "l_I^1"] & (2,A) \arrow[r, "l_G^2"] & (3,A) \arrow[r, "l_G^1"] & (1,B) \arrow[r, "l_I^1"] & \underline{(2,B)} \arrow[r, "l_I^2"] \arrow[d, squiggly] & \cancel{(2,A)} \\
 & & & & \underline{(2,B)} \arrow[r, "l_I^2"] & (2,A) \arrow[ullll, dashed, bend left]
\end{tikzcd}$$

Fig. 3. Example run of **CheckEmptiness**. The wavy arrow indicates the invocation of **NestedDFS**$((2, B))$; the dashed arrow indicates how the cycle is closed.

In Fig. 3, we illustrate an example run of the **CheckEmptiness** algorithm on our example. When **DFS** backtracks from $(2, B)$, **NestedDFS** is invoked, illustrated by the wavy arrow. **NestedDFS** generates the successor $(2, A)$, which is on *Stack*, so a cycle is reported. We can construct an accepting run $\boldsymbol{\rho}_p(\boldsymbol{\rho}_c)^\omega$ with prefix $\boldsymbol{\rho}_p$ induced by the trace l_I^1 and cycle $\boldsymbol{\rho}_c$ induced by the trace $l_G^2, l_G^1, l_I^1, l_I^2$.

3 The Decoupled State Space for Composed NBAs

As previously stated, decoupled state space search was recently developed in AI planning [14], and adapted to model checking of safety properties later on [12]. It is designed to tackle the state explosion problem inherent in search problems that result from compactly represented systems with exponentially large state spaces. In AI planning, where decoupled search was originally introduced, such systems are modelled through state variables and a set of transition rules (called "actions"). The adaptation of decoupled search to reachability checking in SPIN presented in [12] devised decoupled search for automata models, but informally only. Here, we introduce decoupled search formally for NBA models. We define the decoupled state space for composed NBAs, as the result from the composition of a set of NBAs.

3.1 Decoupled Composition of NBAs

In contrast to the explicit construction of the state space, where all reachable states are generated by searching over all traces of enabled transitions, decoupled search only searches over traces of global transitions, the ones that synchronize the component NBAs. In decoupled search, a *decoupled state* $s^{\mathcal{D}}$ compactly represents a set of states closed by internal steps. This is done in terms of the sequence of global labels used to reach these states, plus a set of reached states for each component. Definition 3 formalizes this through the operation *decoupled composition of NBAs*, which adapts the composition operation provided in Definition 2 to decoupled state space search.

Definition 3 (Decoupled composition of NBAs). *The* decoupled composition *of* n *NBAs* $\mathcal{A}^1, \dots, \mathcal{A}^n$*, denoted by* $\mathcal{A}^1 \parallel_{\mathcal{D}} \dots \parallel_{\mathcal{D}} \mathcal{A}^n$*, is a tuple* $\langle S^{\mathcal{D}}, \rightarrow_{\mathcal{D}}, L_G, s_0^{\mathcal{D}}, A^{\mathcal{D}} \rangle$ *defined as follows:*

– $S^{\mathcal{D}} = \mathcal{P}^+(S^1) \times \dots \times \mathcal{P}^+(S^n)$*, with* $\mathcal{P}^+(S) := 2^S \setminus \emptyset$.
– $s_0^{\mathcal{D}} = \langle \mathrm{iclose}(s_0^1), \dots, \mathrm{iclose}(s_0^n) \rangle$*, with* $\mathrm{iclose}(s)$ *being the set of states* s' *that are reachable from* s *in* $\mathcal{A}^i$ *using only* $\mathcal{A}^i$*'s internal transitions* L_I^i*:*
  $\mathrm{iclose}(s) = \{s' \mid s \xrightarrow{l_I \in L_I^i} {}^* s'\}$ *and* $\mathrm{iclose}(S) = \bigcup_{s \in S} \mathrm{iclose}(s)$.
– $A^{\mathcal{D}}(s^{\mathcal{D}}) = \forall_{i \in \{1,\dots,n\}} : \exists s^i \in S_i : A^i(s^i)$*, where* $s^{\mathcal{D}} = \langle S_1, \dots, S_n \rangle$.
– $\rightarrow_{\mathcal{D}}$ *is the smallest set of transitions closed under the following rule:*

$$\frac{l_G \in L_G \quad \forall_{i \in \{1,\dots n\}} : S_i' = \{s_i' \mid \boldsymbol{s} \in s^{\mathcal{D}} : \boldsymbol{s} \xrightarrow{l_G} (s_1', \dots, s_i', \dots, s_n')\} \quad S_i' \neq \emptyset}{s^{\mathcal{D}} \xrightarrow{l_G}_{\mathcal{D}} \langle \mathrm{iclose}(S_1'), \dots, \mathrm{iclose}(S_n') \rangle}$$

where, abusing notation, we write $\boldsymbol{s} \in s^{\mathcal{D}}$ *if* $s^{\mathcal{D}} = \langle S_1, \dots, S_n \rangle$ *and* $\boldsymbol{s} \in S_1 \times \dots \times S_n$.

In the decoupled composition $\mathcal{A}^1 \parallel_{\mathcal{D}} \dots \parallel_{\mathcal{D}} \mathcal{A}^n$ a decoupled state $s^{\mathcal{D}}$ is defined by a tuple $\langle s^{\mathcal{D}}[\mathcal{A}^1], \dots, s^{\mathcal{D}}[\mathcal{A}^n] \rangle$, consisting of a non-empty set of component states $s^{\mathcal{D}}[\mathcal{A}^i]$ for each $\mathcal{A}^i$. A decoupled state represents exponentially many *member states*, namely all composed states $\boldsymbol{s} = (s_1, \dots, s_n)$ such that $\boldsymbol{s} \in s^{\mathcal{D}}[\mathcal{A}^1] \times \dots \times s^{\mathcal{D}}[\mathcal{A}^n]$. We will always use a superscript $\mathcal{D}$ to denote decoupled states $s^{\mathcal{D}}$.

We overload the subset operation $\subseteq$ for decoupled states $s^{\mathcal{D}}$ by doing it component-wise on the sets of reached local states, namely $s^{\mathcal{D}} \subseteq t^{\mathcal{D}} \Leftrightarrow \forall \mathcal{A}^i : s^{\mathcal{D}}[\mathcal{A}^i] \subseteq t^{\mathcal{D}}[\mathcal{A}^i]$.

During a search in the decoupled composition we define the *global trace* of a decoupled state $s^{\mathcal{D}}$, denoted $\pi^G(s^{\mathcal{D}})$, as the sequence of global transitions on which $s^{\mathcal{D}}$ was reached from $s_0^{\mathcal{D}}$. For DFS, as considered in this work, this is well-defined.

In explicit state search, states that have been visited before – duplicates – are pruned to avoid repeating the search effort unnecessarily. The corresponding operation in decoupled search is *dominance pruning* [14]. A newly generated decoupled state $t^{\mathcal{D}}$ is pruned if there exists a previously seen decoupled state $s^{\mathcal{D}}$ that *dominates* $t^{\mathcal{D}}$, i.e., where $t^{\mathcal{D}} \subseteq s^{\mathcal{D}}$. With the correctness result given below, this is safe. One can make the representation of decoupled states, and thereby also the dominance checking, more efficient by representing the state sets $s^{\mathcal{D}}[\mathcal{A}^i]$ symbolically [17].

The initial decoupled state is obtained by closing each local state with internal steps (iclose), and decoupled transitions generate decoupled states whose local states are also closed under internal steps. This maximally preserves the decomposition afforded by the decoupled representation. Namely, as we will prove in what follows, a decoupled state $s^{\mathcal{D}}$ compactly represents all explicit states that are reachable via traces that extend the global trace $\pi^G(s^{\mathcal{D}}) = l_G^1, l_G^2, \ldots, l_G^k$ with local transition labels. That is, for every component $\mathcal{A}^i$, $s^{\mathcal{D}}$ contains the non-empty subset of its local states $s^{\mathcal{D}}[\mathcal{A}^i] \subseteq S^i$ that can be reached with traces $\pi_i = l_1, l_2, \ldots, l_n$ such that there exist indices $j_1 < j_2 < \cdots < j_k$ where $l_{j_t} = \pi^G(s^{\mathcal{D}})[j_t]$ for all $1 \leq t \leq k$. In words, after every global label on $\pi^G(s^{\mathcal{D}})$, arbitrary enabled sequences of internal transitions are allowed.

We remark that the decoupled composition of a set of NBAs is always deterministic. For every pair of decoupled state $s^{\mathcal{D}}$ and global label l_G, there is a unique successor $t^{\mathcal{D}}$. This is easy to see, since if there is a composed state $\boldsymbol{s}$ contained in $s^{\mathcal{D}}$ that has multiple outgoing transitions labelled with l_G, all of the composed successor states are contained in $t^{\mathcal{D}}$. This increases the possible state space reduction compared to standard search, which needs to branch over all these successors. Note that this is different from the determinization of NBA, which comes with a blow-up [31]. The determinism is a consequence of the compact representation where all possible outcome states of a non-deterministic transition are contained in the decoupled successor state.

3.2 Correctness of Decoupled Composition

In this section we show that decoupled search, as presented here, is sound and complete w.r.t. reachability properties. We adapt the corresponding result from AI planning [14].

We require some additional notation. For a trace $\boldsymbol{\pi}$, by $\pi^G(\boldsymbol{\pi})$ we denote the subsequence of $\boldsymbol{\pi}$ that is obtained by projecting onto the global labels L_G.

As previously stated, the decoupled state space captures reachability of the composed system exactly. The proof is an adaptation of previous results from AI Planning [14] to composed NBAs as considered here.

Theorem 1. *A state $\boldsymbol{t}$ of a composition of NBAs $\mathcal{A}^1 \parallel \ldots \parallel \mathcal{A}^n$ is reachable from a state $\boldsymbol{s}$ via a trace $\boldsymbol{\pi}$, iff there exist decoupled states $s^{\mathcal{D}}$, $t^{\mathcal{D}}$ in the decoupled composition $\mathcal{A}^1 \parallel_{\mathcal{D}} \ldots \parallel_{\mathcal{D}} \mathcal{A}^n$, such that $\boldsymbol{s} \in s^{\mathcal{D}}$, $\boldsymbol{t} \in t^{\mathcal{D}}$, and $t^{\mathcal{D}}$ is reachable from $s^{\mathcal{D}}$ via $\pi^G(\boldsymbol{\pi})$.*

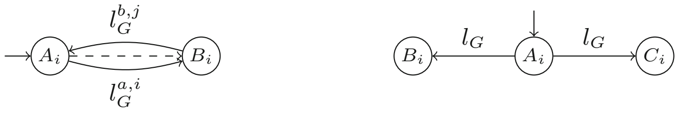


Fig. 4. Illustration of the exponential separations to ample sets (left) and unfolding (right).

Proof. Let $\pi^G(\boldsymbol{\pi}) = l_G^1, \dots, l_G^k$, and $s_i^{\mathcal{D}} \xrightarrow{l_G^{i+1}}_{\mathcal{D}} s_{i+1}^{\mathcal{D}}$ for all $1 \leq i < k$. We prove the claim by induction over the length of $\pi^G(\boldsymbol{\pi})$. For the base case $|\pi^G(\boldsymbol{\pi})| = 0$, the claim trivially holds, since, by the definition of iclose(), $s^{\mathcal{D}}$ contains all composed states $\boldsymbol{t}$ that are reachable from any $\boldsymbol{s} \in s^{\mathcal{D}}$ via only internal transitions.

Assume a decoupled state $s_i^{\mathcal{D}}$ is reachable from $s^{\mathcal{D}}$ via $l_G^1, \dots, l_G^i$. Then, by the definition of decoupled transitions and iclose(), the state $s_{i+1}^{\mathcal{D}}$ contains all composed states $\boldsymbol{s}_{i+1}$ that are reachable from a state $\boldsymbol{s}_i \in s_i^{\mathcal{D}}$ via a trace $\pi^{i \rightarrow i+1}$ that consists of only internal transitions and l_G^{i+1}. By hypothesis, we can extend the traces reaching every such $\boldsymbol{s}_i$ from a $\boldsymbol{s} \in s^{\mathcal{D}}$ by $\pi^{i \rightarrow i+1}$ and obtain a trace reaching $\boldsymbol{s}_{i+1}$ from $\boldsymbol{s}$ with global sub-trace $l_G^1, \dots, l_G^i, l_G^{i+1}$.

For the other direction, if a composed state $\boldsymbol{s}_i$ is reached in a decoupled state $s_i^{\mathcal{D}}$ and can reach a state $\boldsymbol{s}_{i+1}$ via a trace $\pi^{i \rightarrow i+1}$ that consists of internal labels and l_G^{i+1}, then there exists a decoupled transition $s_i^{\mathcal{D}} \xrightarrow{l_G^{i+1}}_{\mathcal{D}} s_{i+1}^{\mathcal{D}}$ and, again by the definition of decoupled transitions and iclose(), $s_{i+1}^{\mathcal{D}}$ contains $\boldsymbol{s}_{i+1}$. By hypothesis $s_i^{\mathcal{D}}$ is reachable from $s^{\mathcal{D}}$, where $\boldsymbol{s}_i$ is reachable from $\boldsymbol{s} \in s^{\mathcal{D}}$. Thus, $s_{i+1}^{\mathcal{D}}$ is reachable from $s^{\mathcal{D}}$ via $l_G^1, \dots, l_G^i, l_G^{i+1}$. □

3.3 Relation to Other State-Space Reduction Methods

Prior work has investigated the relation of decoupled search to other state-space reduction methods in the context of AI planning [14,15], in particular to strong stubborn sets [34], Petri-net unfolding [8,9], and symbolic representations using BDDs [2,28]. For all these techniques, there exist families of scaling examples where decoupled search is exponentially more efficient.

We capture this formally in terms of *exponential separations*. A search method X is *exponentially separated* from decoupled search if there exists a family of models $\{M^n{=}\mathcal{A}^1, \dots, \mathcal{A}^m \mid n \in \mathbb{N}\}$ of size polynomially related to n such that (1) the number of reachable decoupled states in $\mathcal{A}^1 \parallel_{\mathcal{D}} \dots \parallel_{\mathcal{D}} \mathcal{A}^m$ is bounded by a polynomial in n, and (2) the state space representation of $\mathcal{A}^1 \parallel \dots \parallel \mathcal{A}^m$ under X is exponential in n.

We next describe two scaling models showing that the ample sets variant of SPIN [21,29], as a representative for partial-order reduction in explicit-state search, and Petri-net unfolding are exponentially separated from decoupled search. For symbolic search with BDDs, the reduction achieved by both methods is in general incomparable.

For ample sets, a simple model family looks as follows: there are n components, each with the same state space: two local states A_i, B_i, initial state A_i, two global transitions $l_G^{a,i}$, $l_G^{b,j}$, one internal transition. A component and the transitions are depicted in the left of Fig. 4 (the dashed transition is internal). The global transitions synchronize components pairwise; our argument holds for every possible such synchronization.

Under ample set pruning, no reduction is achieved (no state is pruned) because there is a global transition enabled in every state. Thus, there exists no state where only safe (i.e. internal) transitions are enabled, and the search always branches over all enabled transitions of all components. The decoupled state space, in contrast, only has a single decoupled state, where both local states are reached in each component. All decoupled successor states are dominated and will be pruned.

Similar to decoupled search, Petri-net unfolding exploits component independence by a special representation. Instead of searching over composed states and pruning transitions, the states of individual components are maintained separately.[1]

A scaling model showing that unfolding is exponentially separated from decoupled search is illustrated in the right of Fig. 4. There are n components, each with the same state space with three local states A_i, B_i, C_i, a global label l_G, and transitions as shown in the figure. In a Petri net, this model is encoded with $3n$ places and 2^n transitions, one for every combination of one output place in each of the components. In the unfolding, this results in an event (the equivalent of a state) for every net transition. The decoupled state space has only two decoupled states: the initial state where $\{A_i\}$ is reached for all components, and its l_G-successor where $\{B_i, C_i\}$ is reached in every component.

4 NDFS for Decoupled Search

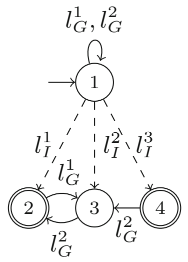
We now adapt NDFS to decoupled search. We start by discussing the deficiencies of a naïve adaptation. We will then introduce the key concepts in our fixed algorithm in Sect. 4.2, and present the algorithm itself in Sect. 4.3. We close this section by showing that the exponential separations to partial-order reduction and unfolding from Sect. 3.3 carry over to liveness checking by simple modifications of the models.

4.1 Issues with a Naïve Adaptation of NDFS

In a naïve adaptation of NDFS to decoupled search, the only thing that changes is the treatment of decoupled states, which represent *sets of composed states*, compared to single states in the standard variant. This leads to three mostly minor changes: (1) instead of duplicate checking we perform dominance pruning; (2) checking if a decoupled state is accepting boils down to checking if it contains an accepting member state; and (3) to see if a state $\boldsymbol{t}$ contained in a state $t^{\mathcal{D}}$ generated in **NestedDFS** is on the stack, we need to check if $t^{\mathcal{D}}$ has a non-empty intersection with a state on $Stack$.

As we will show next, it turns out that this naïve adaptation can *miss* cycles due to pruning. Revisiting a composed state in **NestedDFS** does actually not imply a cycle, because reaching $t^{\mathcal{D}}$ from $s^{\mathcal{D}}$ entails only that every member state of $t^{\mathcal{D}}$ can be reached from *at least one* member state of $s^{\mathcal{D}}$, not from all of them. The critical point is that pruning does not take into account *from where* states are reachable.

[1] A general difference between the methods is that checking reachability of a conjunctive property is linear in the number of decoupled states, but **NP**-complete for an unfolding prefix [27].



$$\begin{tikzcd}
& s_0^{\mathcal{D}}[\mathcal{A}^1] = \{1,2,3,4\} \arrow[dl, "l_G^1"'] \arrow[d, squiggly] \arrow[dr, "l_G^2"] & \\
\cancel{s_1^{\mathcal{D}}[\mathcal{A}^1] = \{1,2,3,4\}} \;\; s_1^{\mathcal{D}} = s_0^{\mathcal{D}} & s_{0,A}^{\mathcal{D}}[\mathcal{A}^1] = \{2,4\} \arrow[dl, "l_G^2"'] \arrow[dr, "l_G^1"] & \cancel{s_2^{\mathcal{D}}[\mathcal{A}^1] = \{1,2,3,4\}} \;\; s_2^{\mathcal{D}} = s_0^{\mathcal{D}} \\
s_{1,A}^{\mathcal{D}}[\mathcal{A}^1] = \{3\} \arrow[d, "l_G^2"] & & \cancel{s_{4,A}^{\mathcal{D}}[\mathcal{A}^1] = \{3\}} \;\; s_{4,A}^{\mathcal{D}} = s_{1,A}^{\mathcal{D}} \\
s_{2,A}^{\mathcal{D}}[\mathcal{A}^1] = \{2\} \arrow[d, "l_G^1"] & & \\
\cancel{s_{3,A}^{\mathcal{D}}[\mathcal{A}^1] = \{3\}} \;\; s_{3,A}^{\mathcal{D}} = s_{1,A}^{\mathcal{D}} & &
\end{tikzcd}$$

Fig. 5. Counterexample showing that a naïve adaptation of the NDFS algorithm is incomplete. The (only) component NBA $\mathcal{A}^1$ is depicted on the left. The search tree on the right shows the entire reachable decoupled state space, where pruned states are crossed out; the wavy arrow depicts the invocation of **NestedDFS** on the acceptance restriction $s_{0,A}^{\mathcal{D}}$ of $s_0^{\mathcal{D}}$.

Consider the example in Fig. 5. The left part of the figure shows the local state space of component NBA $\mathcal{A}^1$. For simplicity, we only show a single component, which is sufficient to illustrate the issue. Here, $\mathcal{A}^1$ is defined as follows: $S^1 = \{1,2,3,4\}$, $L_G^1 = \{l_G^1, l_G^2\}$, $L_I^1 = \{l_I^1, l_I^2, l_I^3\}$, $A^1(s) = \top$ iff $s \in \{2,4\}$, and $s_0^1 = 1$. The transitions are as shown in the left of the figure. The decoupled search space generated using NDFS is depicted in the right of the figure. Pruned states are crossed out.

NestedDFS is launched (indicated by the wavy arrow) on the accepting initial state $s_0^{\mathcal{D}}$. Before explaining the main issue, we remark that, to ensure that a cycle through an *accepting* member state of $s_0^{\mathcal{D}}$ is found, not a cycle through a non-accepting one, we need to restrict the set of reached local states to those that are accepting, and the states internally reachable from those via iclose(). Thus, **NestedDFS** starts in what we call the acceptance-restriction $s_{0,A}^{\mathcal{D}}$ of $s_0^{\mathcal{D}}$, where $s_{0,A}^{\mathcal{D}}[\mathcal{A}^1] = \{2,4\}$. Now, the issue results from the fact that $s_{0,A}^{\mathcal{D}}$ contains two accepting member states, only one of which, namely state 2, is on a cycle. Assuming that the decoupled states are generated in order of increasing subscripts, so $s_1^{\mathcal{D}}$ before $s_2^{\mathcal{D}}$ and so on, state 2 is first reached in **NestedDFS** as a member state of $s_{2,A}^{\mathcal{D}}$, but via the transition labelled with l_G^2 from state 3, so the cycle cannot be closed. When generating the l_G^1 successor $s_{4,A}^{\mathcal{D}}$ of $s_{0,A}^{\mathcal{D}}$, its only member state 3 has already been reached in $s_{1,A}^{\mathcal{D}}$, so $s_{4,A}^{\mathcal{D}}$ is pruned and the cycle of state 2 via l_G^1, l_G^2 is missed. In the next Section we show how to fix this, through an extended state representation that keeps track of reachability from a set of reference states.

Another minor issue are lassos $\boldsymbol{\rho}_p(\boldsymbol{\rho}_c)^\omega$ whose cycle $\boldsymbol{\rho}_c$ is induced by internal labels only. These will not be detected, because **NestedDFS** only considers traces via global labels. We fix this by checking for L_I-cycles in every accepting decoupled state generated during **DFS**, to see if there exists a component that can reach such a state.

4.2 Reference-State Splits

The problem underlying the issue described in the previous section is that pruning is done regardless of the accepting states in the root node of **NestedDFS**. We now introduce an operation on decoupled states splitting them with respect to the set of reached local accepting states for each component. In our algorithm, this will serve to distinguish the different accepting states, and thus force dominance pruning to distinguish reachability from these. Formally, we define the restriction to accepting local states as a new transition with a global label l_G^A that is a self-loop for all accepting states:

Definition 4 (Acceptance-Split Transition). *Let $\langle S^{\mathcal{D}}, \rightarrow_{\mathcal{D}}, L, s_0^{\mathcal{D}}, A^{\mathcal{D}}\rangle$ be the decoupled composition of $\mathcal{A}^1, \ldots, \mathcal{A}^n$. Let $s^{\mathcal{D}}$ be an accepting decoupled state, and for $1 \leq i \leq n$ let $\langle s_1^i, \ldots, s_{c_i}^i\rangle \subseteq s^{\mathcal{D}}[\mathcal{A}^i]$ be the list of reached accepting states of $\mathcal{A}^i$, where for all $1 \leq j \leq c_i : A^i(s_j^i) = \top$. Then the* acceptance-split transition *l_G^A in $s^{\mathcal{D}}$ is defined as follows:*

$$\frac{A^{\mathcal{D}}(s^{\mathcal{D}}) = \top \quad \forall i \in \{1, \ldots n\}, j \in \{1, \ldots, c_i\} : s_j^i \in s^{\mathcal{D}}[\mathcal{A}^i] \wedge A^i(s_j^i) = \top}{s^{\mathcal{D}} \xrightarrow{l_G^A}_{\mathcal{D}} \langle\langle \text{iclose}(s_1^1), \ldots, \text{iclose}(s_{c_1}^1)\rangle, \ldots, \langle \text{iclose}(s_1^n), \ldots, \text{iclose}(s_{c_n}^n)\rangle\rangle}$$

The outcome state $s_A^{\mathcal{D}}$ of an acceptance-split transition is a split decoupled state*. The set of* reference states *of $s_A^{\mathcal{D}}$ is $R(s_A^{\mathcal{D}}) := \{s \mid \exists \mathcal{A}_i : s \in s^{\mathcal{D}}[\mathcal{A}^i] \wedge A^i(s) = \top\}$.*

In words, the operation splits up the single set of reached component states $s^{\mathcal{D}}[\mathcal{A}^i]$ of $\mathcal{A}^i$ into a list of state sets, where each such set $s_A^{\mathcal{D}}[\mathcal{A}^i]_s$ contains the states that can be reached via internal transitions from the respective accepting state $s \in s^{\mathcal{D}}[\mathcal{A}^i]$.

Our search algorithm will use the acceptance-split transition to generate the root node $s_A^{\mathcal{D}}$ of **NestedDFS** from an accepting state $s^{\mathcal{D}}$ backtracked from in **DFS**. Hence **NestedDFS** will search in the space of split decoupled states. The transitions over these behind an $s_A^{\mathcal{D}}$ are defined as follows:

Definition 5 (Split Transitions). *Let $\langle S^{\mathcal{D}}, \rightarrow_{\mathcal{D}}, L, s_0^{\mathcal{D}}, A^{\mathcal{D}}\rangle$ be the decoupled composition of $\mathcal{A}^1, \ldots, \mathcal{A}^n$. Let $s^{\mathcal{D}}$ and $t^{\mathcal{D}}$ be decoupled states, with a transition $s^{\mathcal{D}} \xrightarrow{l_G}_{\mathcal{D}} t^{\mathcal{D}}$. Let $\langle s_1^i, \ldots, s_{c_i}^i\rangle \subseteq S^i$ be reference states for $\mathcal{A}^i$. Then the* split transition *$s_R^{\mathcal{D}} \xrightarrow{l_G}_{\mathcal{D}} t_R^{\mathcal{D}}$ is defined such that for every $\mathcal{A}^i$ and every $1 \leq j \leq c_i$ we have:*

$$t_R^{\mathcal{D}}[\mathcal{A}^i]_{s_j^i} = \begin{cases} \text{iclose}(\{s' \in t^{\mathcal{D}}[\mathcal{A}^i] \mid \exists s \in s_R^{\mathcal{D}}[\mathcal{A}^i]_{s_j^i} : s \xrightarrow{l_G}{}^i s'\}) & l_G \in L_G^i \\ s_R^{\mathcal{D}}[\mathcal{A}^i]_{s_j^i} & l_G \notin L_G^i \end{cases}$$

The list of reference states for an $\mathcal{A}_i$ does not change along a trace of split transitions. Let $s_A^{\mathcal{D}}$ be a decoupled state generated by an acceptance-split transition $s^{\mathcal{D}} \xrightarrow{l_G^A}_{\mathcal{D}} s_A^{\mathcal{D}}$, then for all successor states $t^{\mathcal{D}}$ of $s_A^{\mathcal{D}}$, the set of reference states is $R(t^{\mathcal{D}}) = R(s_A^{\mathcal{D}})$.

We extend set operations to the split representation as follows. A split decoupled state $s_R^{\mathcal{D}}$ *dominates* a split decoupled state $t_R^{\mathcal{D}}$, denoted $t_R^{\mathcal{D}} \subseteq_R s_R^{\mathcal{D}}$, if $R(t_R^{\mathcal{D}}) \subseteq R(s_R^{\mathcal{D}})$ and for all components $\mathcal{A}^i$ and reference states $s \in R(t_R^{\mathcal{D}}) \cap S^i$ we have $t_R^{\mathcal{D}}[\mathcal{A}^i]_s \subseteq s_R^{\mathcal{D}}[\mathcal{A}^i]_s$. In contrast, state membership is defined in a global manner, across reference states. Namely, the set of local states of an $\mathcal{A}^i$ reached in a split decoupled state $s_R^{\mathcal{D}}$ is defined as $s_R^{\mathcal{D}}[\mathcal{A}^i] := \bigcup_{s \in R(s_R^{\mathcal{D}}) \cap S^i} s_R^{\mathcal{D}}[\mathcal{A}^i]_s$. Composed state membership is defined relative to these $s_R^{\mathcal{D}}[\mathcal{A}^i]$ as before.

An important property of the splitting is that it preserves reachability of member states. Concretely, for a split-transition $s_R^{\mathcal{D}} \xrightarrow{l_G}_{\mathcal{D}} t_R^{\mathcal{D}}$ induced by a transition $s^{\mathcal{D}} \xrightarrow{l_G}_{\mathcal{D}} t^{\mathcal{D}}$ for all $\mathcal{A}^i$ it holds that if $s_R^{\mathcal{D}}[\mathcal{A}^i] = s^{\mathcal{D}}[\mathcal{A}^i]$, then $t_R^{\mathcal{D}}[\mathcal{A}^i] = t^{\mathcal{D}}[\mathcal{A}^i]$.

As a notation convention, we will always denote split states $s^{\mathcal{D}}_{R}$ by a subscript R, and the direct outcome of an acceptance-split transition by $s^{\mathcal{D}}_{A}$, with a subscript A.

$$\begin{tikzcd}
 & s^{\mathcal{D}}_{1,R}[\mathcal{A}^1] = \langle \{\}_2, \{3\}_4 \rangle \arrow[r, "l^2_G"] & s^{\mathcal{D}}_{2,R}[\mathcal{A}^1] = \langle \{\}_2, \{2\}_4 \rangle \arrow[r, "l^1_G"] & \cancel{s^{\mathcal{D}}_{3,R}[\mathcal{A}^1] = \langle \{\}_2, \{3\}_4 \rangle} \\
 & & & s^{\mathcal{D}}_{3,R} \subseteq_R s^{\mathcal{D}}_{1,R} \\
s^{\mathcal{D}}_{1,A}[\mathcal{A}^1] = \langle \{2\}_2, \{4\}_4 \rangle \arrow[uur, "l^2_G"] \arrow[r, "l^1_G"'] & s^{\mathcal{D}}_{4,R}[\mathcal{A}^1] = \langle \{3\}_2, \{\}_4 \rangle \arrow[r, "l^2_G"] & s^{\mathcal{D}}_{5,R}[\mathcal{A}^1] = \langle \{2\}_2, \{\}_4 \rangle &
\end{tikzcd}$$

Fig. 6. With acceptance-splitting, **NestedDFS** invoked on the l^A_G-successor $s^{\mathcal{D}}_{1,A}$ of $s^{\mathcal{D}}_0$ of the example in Fig. 5 correctly detects the cycle of state 2 induced by the trace l^1_G, l^2_G.

Considering our example again, Fig. 6 illustrates how, on split decoupled states, the cycle $2 \xrightarrow{l^1_G} 3 \xrightarrow{l^2_G} 2$ is not pruned. The state $s^{\mathcal{D}}_{3,R}$ is still pruned, as it contains only component states reached from state 4. In $s^{\mathcal{D}}_{4,R}$ and $s^{\mathcal{D}}_{5,R}$, the decoupled state keeps track of the traces from the origin state 2, so none of the two is pruned, since they are not dominated by any state $s^{\mathcal{D}}_{i,R}$ (the root node $s^{\mathcal{D}}_{1,A}$ of **NestedDFS** is not yet visited).

As indicated before, in our emptiness checking algorithm we will use split decoupled states only within **NestedDFS**. The seed state $s^{\mathcal{D}}_{A}$ of **NestedDFS** will always be the l^A_G-successor of an accepting state $s^{\mathcal{D}}$ backtracked from in **DFS**. Every member state of $s^{\mathcal{D}}_{A}$ is accepting, or can be reached with L_I-transitions from an accepting state.

4.3 Putting Things Together: Decoupled NDFS

We are now ready to describe our adaptation of the standard NDFS algorithm to decoupled compositions. The pseudo-code is shown in Fig. 7. The differences w.r.t the standard algorithm (Fig. 2) are highlighted in blue. The basic structure of the algorithm is preserved. It starts by putting the decoupled initial state $s^{\mathcal{D}}_0$ onto the Stack in **CheckEmptiness**, and launches the main **DFS** from it.

In **DFS**, the control flow does not change, decoupled states are generated in depth-first order by recursion, updating the stack accordingly. There are however three differences to the standard variant:

1. Before generating the successors, we call **CheckLocalAccept** on each accepting decoupled state $s^{\mathcal{D}}$. This detects cycles resulting from L_I-transitions, i.e., cycles that occur "within" a decoupled state. To this end, we check whether there exists a component $\mathcal{A}^i$ for which an accepting local state s^i_a is reached that can reach itself using only internal labels L^i_I (the set of such local states can be precomputed, so that the check becomes a lookup operation). We can then construct an accepting run for the composed system by appending the L^i_I-cycle to the sequence of states that reaches s^i_a in $s^{\mathcal{D}}$ for $\mathcal{A}^i$. Note that it suffices if a single component moves and all other components remain in a reached accepting state.
2. Instead of doing duplicate checking, the algorithm performs dominance pruning, pruning a new decoupled state $t^{\mathcal{D}}$ if all its member states have been reached in an already visited decoupled state $r^{\mathcal{D}}$.

```
**CheckEmptiness**($\mathcal{A}^1 \parallel_{\mathcal{D}} \dots \parallel_{\mathcal{D}} \mathcal{A}^n$):
    Stack ← $\langle s_0^{\mathcal{D}} \rangle$
    $V \leftarrow \emptyset$
    $V' \leftarrow \emptyset$
    DFS($s_0^{\mathcal{D}}$)
    **return** empty

**NestedDFS**($s_R^{\mathcal{D}}$):
    **for all** $t_R^{\mathcal{D}}$ s.t. $s_R^{\mathcal{D}} \rightarrow_{\mathcal{D}} t_R^{\mathcal{D}}$ **do**
        **if** $\exists r_R^{\mathcal{D}} \in V'$ s.t. $t_R^{\mathcal{D}} \subseteq_R r_R^{\mathcal{D}}$ **then**
            **continue**
        **if** $\forall \mathcal{A}^i : \exists s : s \in t_R^{\mathcal{D}}[\mathcal{A}^i]_s$ **then**
            **return** cycle
        **if** $\exists r^{\mathcal{D}} \in$ Stack s.t. $r^{\mathcal{D}} \subseteq t_R^{\mathcal{D}}$ **then**
            **return** cycle
        $V' = V' \cup \{t_R^{\mathcal{D}}\}$
        NestedDFS($t_R^{\mathcal{D}}$)

**DFS**($s^{\mathcal{D}}$):
    $V = V \cup \{s^{\mathcal{D}}\}$
    **if** $A^{\mathcal{D}}(s^{\mathcal{D}})$ **then**
        CheckLocalAccept($s^{\mathcal{D}}$)
    **for all** $t^{\mathcal{D}}$ s.t. $s^{\mathcal{D}} \rightarrow_{\mathcal{D}} t^{\mathcal{D}}$ **do**
        **if** $\exists r^{\mathcal{D}} \in V$ s.t. $t^{\mathcal{D}} \subseteq r^{\mathcal{D}}$ **then**
            **continue**
        push(Stack, $t^{\mathcal{D}}$)
        DFS($t^{\mathcal{D}}$)
        pop(Stack)
    **if** $A^{\mathcal{D}}(s^{\mathcal{D}})$ **then**
        Let $s_A^{\mathcal{D}}$ s.t. $s^{\mathcal{D}} \xrightarrow{l_G^A}_{\mathcal{D}} s_A^{\mathcal{D}}$.
        NestedDFS($s_A^{\mathcal{D}}$)
        $V' = V' \cup \{s_A^{\mathcal{D}}\}$

**CheckLocalAccept**($s^{\mathcal{D}}$):
    **if** $\exists \mathcal{A}^i, s \in s^{\mathcal{D}}[\mathcal{A}^i] : A^i(s) \wedge s \xrightarrow{l_I \in L_I^i}{}^{+} s$
        **then return** cycle
```

Fig. 7. Adaptation of a standard NestedDFS for lasso search in decoupled compositions of NBA.

3. As discussed in Sect. 4.2, when we launch **NestedDFS** at a decoupled state $s^{\mathcal{D}}$, we do so on the acceptance-split l_G^A-successor $s_A^{\mathcal{D}}$ of $s^{\mathcal{D}}$.

NestedDFS now starts in the acceptance-split $s_A^{\mathcal{D}}$, and traverses split transitions as per Definition 5. On generation of a new state $t_R^{\mathcal{D}}$, we perform dominance pruning against the decoupled states visited during all prior calls to **NestedDFS**. If in an $t_R^{\mathcal{D}}$ for every component $\mathcal{A}^i$ there exists a reference state $s \in S^i$ that is reachable from itself, so $s \in t^{\mathcal{D}}[\mathcal{A}^i]_s$, then we can construct a cycle. As we will show in Theorem 4, this test is guaranteed to find all cycles that start from an accepting state $\boldsymbol{s}_A \in s_A^{\mathcal{D}}$.

Note that we cannot check for a non-empty intersection with states $r^{\mathcal{D}}$ on $Stack$, since these are not split relative to the reference states of $s_A^{\mathcal{D}}$. Thus, since we do not know from which local state in $r^{\mathcal{D}}$ the state in the intersection was reached, such a non-empty intersection would *not* imply a cycle. What we can do, however, is check for dominance instead, as an algorithm optimization inspired by [22]. The pseudo-code in Fig. 7 does so by checking whether $t_R^{\mathcal{D}} \supseteq r^{\mathcal{D}}$, where the $\supseteq$ relation between a split vs. non-split state is simply evaluated based on the overall sets $t_R^{\mathcal{D}}[\mathcal{A}^i]$ vs. $r^{\mathcal{D}}[\mathcal{A}^i]$ of reached components states. If this domination relation holds true, then the reachability issue mentioned in the previous section is resolved because *all* $\boldsymbol{t} \in r^{\mathcal{D}}$ are then reachable from $s_A^{\mathcal{D}}$ – including those $\boldsymbol{t}$ from which an accepting state $\boldsymbol{s} \in s_A^{\mathcal{D}}$ is reachable. Lemma 1 in the next section will spell out this argument as part of our correctness proof.

Observe that splitting a decoupled state incurs an increase in the size of the state representation, as the same local state may be reached from several reference states. More importantly, as dominance pruning is weaker on split states (which after all is the purpose of the split operation) the size of the search space may increase. As shown by the example in Fig. 5, though, there is no easy way around the splitting, since the

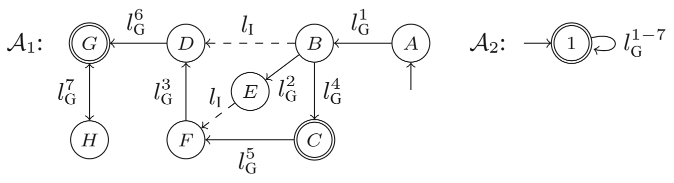


Fig. 8. Illustration of the component NBAs used in Example 1.

algorithm has to be able to know *from which* component state the successors states are reached. Assuming a component has M accepting states, then in the worst case all local successor states that are shared between these accepting states can be visited M times across all **NestedDFS** invocations. Unless some of the decoupled states revisiting the same member state are pruned by dominance pruning, it can actually happen that the revisits multiply across the components, so the size of the decoupled state space in **NestedDFS** can potentially be exponentially larger than the standard state space. As we shall see in our experimental evaluation, typically such blow-ups do not seem to occur.

In case we want to construct a lasso, we need to store a pointer to the predecessor of each decoupled state and the label of the generating transition. With this, we can, for each component $\mathcal{A}^i$ separately, reconstruct a trace $\boldsymbol{\pi}$ of a state $\boldsymbol{t} \in t^{\mathcal{D}}$ reached from a state $\boldsymbol{s} \in s^{\mathcal{D}}$ where $\pi^G(s^{\mathcal{D}}, t^{\mathcal{D}}) = \pi^G(\boldsymbol{\pi})$. Here, for a decoupled state $t^{\mathcal{D}}$ that was reached from another decoupled state $s^{\mathcal{D}}$, by $\pi^G(s^{\mathcal{D}}, t^{\mathcal{D}})$ we denote the global trace via which $t^{\mathcal{D}}$ was reached from $s^{\mathcal{D}}$. This can be done in time polynomial in the size of the component and linear in the length of $\pi^G(s^{\mathcal{D}}, t^{\mathcal{D}})$. Since the traces for all components are synchronized via $\pi^G(\boldsymbol{\pi})$, we add the required internal labels for each component in between every pair of global labels. We remark that, to decide if a lasso exists, we do not need to store any predecessor or generating label pointers.

We next show on an example how our algorithm works.

Example 1. The model has two component NBAs $\mathcal{A}_1, \mathcal{A}_2$ illustrated in Fig. 8. It is a variant of an example from [26]. The Figure should be self-explanatory, we remark that all global transitions $l_G^1, \dots l_G^7$ induce a self loop in the only state 1 of $\mathcal{A}_2$.

CheckEmptiness starts by putting $s_0^{\mathcal{D}}$ onto *Stack* and enters **DFS**($s_0^{\mathcal{D}}$). Let $s_1^{\mathcal{D}} = \langle\{B, D\}, \{1\}\rangle$, $s_2^{\mathcal{D}} = \langle\{E, F\}, \{1\}\rangle$, and $s_3^{\mathcal{D}} = \langle\{D\}, \{1\}\rangle$ be the successors generated along the trace l_G^1, l_G^2, l_G^3 in **DFS**. Since $s_3^{\mathcal{D}} \subseteq s_1^{\mathcal{D}} \in V$, $s_3^{\mathcal{D}}$ is pruned and the search backtracks to $s_1^{\mathcal{D}}$. Say **DFS** selects the transition via l_G^4 next, generating the state $s_4^{\mathcal{D}} = \langle\{C\}, \{1\}\rangle$ and its l_G^5-successor $s_5^{\mathcal{D}} = \langle\{F\}, \{1\}\rangle$. Then $s_5^{\mathcal{D}}$ is pruned because it is dominated by $s_2^{\mathcal{D}} \in V$, and the search backtracks from $s_4^{\mathcal{D}}$, which is accepting.

Thus, **NestedDFS**($s_{5,A}^{\mathcal{D}}$) is invoked, where $s_{5,A}^{\mathcal{D}} = \langle\langle\{C\}_C\rangle, \langle\{1\}_1\rangle\rangle$, because C and 1 are accepting local states that become the reference states of $s_{5,A}^{\mathcal{D}}$. **NestedDFS** will follow the trace $l_G^5, l_G^3, l_G^6, l_G^7, l_G^7$, which among others generates the state $s_{6,R}^{\mathcal{D}} = \langle\langle\{G\}_C\rangle, \langle\{1\}_1\rangle\rangle$ by l_G^6, and ends in $s_{7,R}^{\mathcal{D}} = \langle\langle\{G\}_C\rangle, \langle\{1\}_1\rangle\rangle$. The latter is pruned, because it is dominated by $s_{6,R}^{\mathcal{D}}$, which is contained in V'. No cycle is reported. This is correct, because the only member state $(C, 1)$ of $s_{5,A}^{\mathcal{D}}$ does not occur on a cycle.

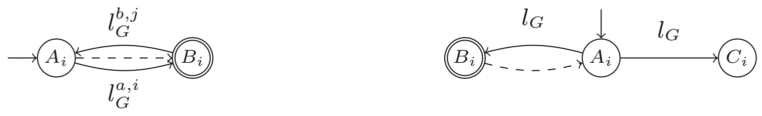


Fig. 9. Illustration of the exponential separations to ample sets (left) and unfolding (right).

DFS then backtracks to $s_1^{\mathcal{D}} = \langle\{B, D\}, \{1\}\rangle$ and generates its remaining successor $s_8^{\mathcal{D}} = \langle\{G\}, \{1\}\rangle$ via l_G^6. **DFS** further generates the l_G^7-successors of $s_8^{\mathcal{D}}$ and eventually backtracks from $s_8^{\mathcal{D}}$, invoking **NestedDFS**($s_{8,A}^{\mathcal{D}}$), where $s_{8,A}^{\mathcal{D}} = \langle\langle\{G\}_G\rangle, \langle\{1\}_1\rangle\rangle$.

After two transitions via l_G^7 the resulting state $s_{9,R}^{\mathcal{D}} = \langle\langle\{G\}_G\rangle, \langle\{1\}_1\rangle\rangle$ satisfies the condition that for all components $\mathcal{A}_i$ $\exists s : s \in s_{9,R}^{\mathcal{D}}[\mathcal{A}^i]_s$, namely G and 1. Thus, a cycle is reported. It is induced by the trace $l_G^1, l_I, l_G^6, l_G^7, l_G^7$.

Note that no decoupled state in the second **NestedDFS** is pruned, since none of them is dominated by a state in V' of the first **NestedDFS** invocation. In particular, $s_{8,A}^{\mathcal{D}} = \langle\{G\}_G, \{1\}_1\rangle$ is not dominated by $s_{6,R}^{\mathcal{D}} = \langle\{G\}_C, \{1\}_1\rangle$, because the reference states differ – G and 1 for $s_{8,R}^{\mathcal{D}}$ and C and 1 for $s_{6,R}^{\mathcal{D}}$.

4.4 Relation to Other State-Space Reduction Methods

The comparison to ample set pruning and Petri-net unfolding from Sect. 3.3 carries over directly to liveness checking via simple adaptations to the examples, see Fig. 9.

Theorem 2. ***CheckEmptiness*** *with explicit-state search and ample sets pruning is exponentially separated from* ***CheckEmptiness*** *with decoupled search.*

Proof (sketch). The argument from Sect. 3.3 remains valid. With the states B_i accepting (see Fig. 9, left), explicit-state search with ample sets pruning in the worst case has to exhaust the entire state space. It invokes **NestedDFS** on the accepting state $(B_i)^n$ and, worst-case, needs to exhaust the state space again to detect the cycle. Decoupled search invokes **NestedDFS** on the initial state restricted to the component states B_i. Every successor of that state closes the cycle via an arbitrary l_G^b transition. So there are only three decoupled states overall (including the acceptance-restricted initial state). □

Theorem 3. *Constructing a complete unfolding prefix is exponentially separated from* ***CheckEmptiness*** *with decoupled search.*

Proof (sketch). The component states B_i are made accepting and internal transitions $B_i \rightarrow A_i$ are added to the model (see Fig. 9, right). Unfolding constructs a complete prefix as described in Sect. 3.3, plus one event for each new internal transition.[2] Decoupled search generates the two states as described. The second state has $\{A_i, B_i, C_i\}$ reached for all components, its successor via l_G is pruned. **NestedDFS** is invoked on its restriction to B_i, in which all A_i get reached via the new internal transitions. The l_G-successor of this state closes the cycle, so there are only four decoupled states. □

[2] A weaker cut-off rule is required for liveness checking that can only increase the prefix size [8].

5 Decoupled NDFS Correctness

We now show the correctness of our approach. In Lemmas 1, 2, 3, we show that if our algorithm reports a cycle, then there exists an accepting run for $\mathcal{A}^1 \parallel \ldots \parallel \mathcal{A}^n$. In Theorem 4, we then show that decoupled NDFS does not miss an accepting run.

We first show that the optimization of checking dominance of states in **NestedDFS** against states on the stack is sound, i.e., that an accepting run exists.

Lemma 1. *Let $r^{\mathcal{D}}$ be a decoupled state on the current **DFS** Stack, and let $t^{\mathcal{D}}_R$ be a decoupled state generated by **NestedDFS**. If $t^{\mathcal{D}}_R \supseteq r^{\mathcal{D}}$, then there exists an accepting run for $\mathcal{A}^1 \parallel \ldots \parallel \mathcal{A}^n$.*

Proof. Let $s^{\mathcal{D}}$ be the accepting state that is backtracked from in **DFS**, i.e., the current **NestedDFS** was invoked on its l^A_G-successor $s^{\mathcal{D}}_A$.

From Theorem 1 we know that if $s^{\mathcal{D}}_2$ is reachable from $s^{\mathcal{D}}_1$, then for every state $\boldsymbol{s}_2 \in s^{\mathcal{D}}_2$ there exists a state $\boldsymbol{s}_1 \in s^{\mathcal{D}}_1$ such that $\boldsymbol{s}_1 \xrightarrow{\boldsymbol{\pi}} \boldsymbol{s}_2$, where $\pi^G(\boldsymbol{\pi}) = \pi^G(s^{\mathcal{D}}_1, s^{\mathcal{D}}_2)$.

This result also holds for decoupled states reached in **NestedDFS** from states in **DFS**. This is because the acceptance-split transition l^A_G only restricts the set of reached member states of $s^{\mathcal{D}}$ in $s^{\mathcal{D}}_A$, so in particular $s^{\mathcal{D}}_A \subseteq s^{\mathcal{D}}$. Furthermore, split transitions generating states behind $s^{\mathcal{D}}_A$ do not affect reachability of member states of these split-decoupled states compared to their non-split counterparts.

In particular, (1) for every state $\boldsymbol{s}_2 \in s^{\mathcal{D}}$ there exists a state $\boldsymbol{s}_1 \in s^{\mathcal{D}}_0$ that reaches $\boldsymbol{s}_2$ on a trace $\boldsymbol{\pi}$ where $\pi^G(\boldsymbol{\pi}) = \pi^G(s^{\mathcal{D}}_0, s^{\mathcal{D}})$, which, with $s^{\mathcal{D}}_A \subseteq s^{\mathcal{D}}$ also holds for all $\boldsymbol{s}_2 \in s^{\mathcal{D}}_A$; and (2) for every state $\boldsymbol{t} \in t^{\mathcal{D}}_R$ there exists an accepting state $\boldsymbol{s}_A \in s^{\mathcal{D}}_A$ that reaches $\boldsymbol{t}$ on a trace $\boldsymbol{\pi}$ where $\pi^G(\boldsymbol{\pi}) = \pi^G(s^{\mathcal{D}}_A, t^{\mathcal{D}}_R)$.

Since $r^{\mathcal{D}}$ is on *Stack*, it holds that every $\boldsymbol{s} \in s^{\mathcal{D}}_A$ is reachable from a $\boldsymbol{r} \in r^{\mathcal{D}}$, and, with $t^{\mathcal{D}}_R \supseteq r^{\mathcal{D}}$, that every $\boldsymbol{r} \in r^{\mathcal{D}}$ is reachable from an accepting state $\boldsymbol{s}_A \in s^{\mathcal{D}}_A$.

Let $pred(s^{\mathcal{D}}_1, s^{\mathcal{D}}_2, \boldsymbol{s}_2)$ be a function that, if $s^{\mathcal{D}}_2$ is reachable from $s^{\mathcal{D}}_1$ and $\boldsymbol{s}_2 \in s^{\mathcal{D}}_2$, outputs a state $\boldsymbol{s}_1 \in s^{\mathcal{D}}_1$ that reaches $\boldsymbol{s}_2$ via a trace $\boldsymbol{\pi}$ with $\pi^G(s^{\mathcal{D}}_1, s^{\mathcal{D}}_2) = \pi^G(\boldsymbol{\pi})$.

Let $\boldsymbol{s}_0$ be a state reached in both $t^{\mathcal{D}}_R$ and $r^{\mathcal{D}}$, and let $\boldsymbol{s}_1 = pred(r^{\mathcal{D}}, t^{\mathcal{D}}_R, \boldsymbol{s}_0)$ be its predecessor in $r^{\mathcal{D}}$. If $\boldsymbol{s}_1 = \boldsymbol{s}_0$, then we are done, because there exists a lasso $\boldsymbol{s}_0, \ldots, \boldsymbol{s}_0, \ldots, \boldsymbol{s}_A, \ldots, \boldsymbol{s}_0, \ldots, \boldsymbol{s}_A$, where $\boldsymbol{s}_A$ is an accepting state traversed in $s^{\mathcal{D}}_A$. Such an accepting state exists because all member states of a decoupled state in **NestedDFS** are reachable from an accepting state in $s^{\mathcal{D}}_A$.

If $\boldsymbol{s}_1 \neq \boldsymbol{s}_0$, then we iterate and set $\boldsymbol{s}_i = pred(r^{\mathcal{D}}, t^{\mathcal{D}}_R, \boldsymbol{s}_{i-1})$, where such $\boldsymbol{s}_i$ exist because $r^{\mathcal{D}} \subseteq t^{\mathcal{D}}_R$. Because there are only finitely many states in $r^{\mathcal{D}}$, eventually we get $\boldsymbol{s}_i = \boldsymbol{s}_j$ (where $j < i$) and there exists a lasso as follows:

First, there exists a cycle $\boldsymbol{s}_i, \ldots, \boldsymbol{s}_{i-1}, \ldots, \boldsymbol{s}_j = \boldsymbol{s}_i$, where between every pair of states $\boldsymbol{s}_k, \boldsymbol{s}_{k-1}$ an accepting state $\boldsymbol{s}_{k,A}$ in $s^{\mathcal{D}}_A$ is traversed, for the same reason as before. We can obviously shift and truncate the cycle to start right after and end in $\boldsymbol{s}_{i,A}$. The prefix of the lasso is $\boldsymbol{s}_0, \ldots, \boldsymbol{s}_{i,A}$. □

Lemmas 2 and 3 show the soundness of our main termination criterion, and of **CheckLocalAccept**.

Lemma 2. *Let $t^{\mathcal{D}}_R$ be a split decoupled state generated in **NestedDFS**. If for every component $\mathcal{A}^i$ there exists a component state s^i such that $s^i \in t^{\mathcal{D}}[\mathcal{A}^i]_{s^i}$, then there exists an accepting run for $\mathcal{A}^1 \parallel \ldots \parallel \mathcal{A}^n$.*

Proof. Let $s^{\mathcal{D}}_A$ be the acceptance-split decoupled state from which **NestedDFS** was started. If for every component $\mathcal{A}^i$ such an s^i exists, then the state $\boldsymbol{s} = (s^1, \dots, s^n)$ is reachable in both $s^{\mathcal{D}}_A$ and $t^{\mathcal{D}}_R$. By the construction of the reached state sets $t^{\mathcal{D}}_R[\mathcal{A}^i]_{s^i}$, $\boldsymbol{s}$ is reachable from itself and is accepting. Hence, there exists a lasso $\boldsymbol{s}_0, \dots, \boldsymbol{s}, \dots, \boldsymbol{s}$. □

Lemma 3. *Let $t^{\mathcal{D}}$ be an accepting decoupled state generated in **DFS** such that a cycle is reported by **CheckLocalAccept**$(t^{\mathcal{D}})$, then an accepting run for $\mathcal{A}^1 \parallel \dots \parallel \mathcal{A}^n$ exists.*

Proof. By prerequisite, there exists an accepting member state $\boldsymbol{s}$ of $t^{\mathcal{D}}$. If **CheckLocalAccept**$(t^{\mathcal{D}})$ reports a cycle, then there exists a component $\mathcal{A}^i$, where an accepting state $s^i \in t^{\mathcal{D}}[\mathcal{A}^i]$ is reached that lies on an cycle induced by transitions labelled with L^i_I. Thus, we can set the local state of $\mathcal{A}^i$ in $\boldsymbol{s}$ to s^i, and the lasso looks as follows: $\boldsymbol{s}_0, \dots, \boldsymbol{s}, \dots, \boldsymbol{s}$, where on the cycle only $\mathcal{A}^i$ moves. □

We are now ready to prove the correctness of our decoupled NDFS algorithm.

Theorem 4. *Let $\mathcal{A}^1 \parallel \dots \parallel \mathcal{A}^n$ be the composition of n NBA and let $\mathcal{A}^1 \parallel_{\mathcal{D}} \dots \parallel_{\mathcal{D}} \mathcal{A}^n$ be its decoupled composition. Then **CheckEmptiness**$(\mathcal{A}^1 \parallel_{\mathcal{D}} \dots \parallel_{\mathcal{D}} \mathcal{A}^n)$ reports a cycle if and only if an accepting run for $\mathcal{A}^1 \parallel \dots \parallel \mathcal{A}^n$ exists.*

Proof. If **CheckEmptiness** reports a cycle, then by Lemmas 1, 2, and 3, which cover exactly the cases where a cycle is reported, an accepting run for $\mathcal{A}^1 \parallel \dots \parallel \mathcal{A}^n$ exists.

For the other direction, assume that $\boldsymbol{\rho}$ is an accepting run for $\mathcal{A}^1 \parallel \dots \parallel \mathcal{A}^n$. Let $\boldsymbol{s}_a$, with $0 \leq a < k$, be the accepting state that starts the cycle of the lasso $\boldsymbol{\rho}_p = \boldsymbol{s}_0, \dots, \boldsymbol{s}_a$, $\boldsymbol{\rho}_c = \boldsymbol{s}_{a+1}, \dots, \boldsymbol{s}_k$, where $\boldsymbol{s}_a = \boldsymbol{s}_k$. Let $\boldsymbol{\pi} = l_1, \dots, l_k$ be the trace on which $\boldsymbol{s}_k$ is reached, i.e., for all $1 \leq i < k : \langle \boldsymbol{s}_i, l_{i+1}, \boldsymbol{s}_{i+1} \rangle \in \rightarrow$.

By Theorem 1, there exists a decoupled state $s^{\mathcal{D}}$ reached in **DFS** that contains $\boldsymbol{s}_a$.

If $\boldsymbol{\pi}$ is such that for all $a < i \leq k : l_i \in L_I$, i.e., the cycle $\boldsymbol{\rho}_c$ is induced only by internal labels, we next proof that **CheckLocalAccept**$(s^{\mathcal{D}})$ reports a cycle: As $\boldsymbol{s}_a$ is accepting, $s^{\mathcal{D}}$ is accepting, too, so unless a cycle is reported before, eventually **CheckLocalAccept**$(s^{\mathcal{D}})$ is called. If $\boldsymbol{\rho}_c$ is induced by only internal labels, then, because there cannot be any component interaction via L_I-transitions, there must exist a component $\mathcal{A}^i$ for which the local state s^i in $\boldsymbol{s}_a$ reaches itself with only L^i_I-transitions. We can pick any such $\mathcal{A}^i$ and ignore transitions from $\boldsymbol{\rho}_c$ that are labelled by an element of $L_I \setminus L^i_I$, since these are not required for an accepting cycle. Consequently, **CheckLocalAccept**$(s^{\mathcal{D}})$ reports a cycle.

We next show that, if $\boldsymbol{\pi}$ contains a global label on the cycle, i.e., there exists an $i \in \{a+1, \dots, k\}$ such that $l_i \in L_G$, then, unless a cycle is reported before, **NestedDFS**$(s^{\mathcal{D}}_A)$ reports a cycle, where $s^{\mathcal{D}}_A$ is the l^A_G-successor of $s^{\mathcal{D}}$.

Assume for contradiction that this is not the case, i.e., no cycle has been reported before, and **NestedDFS**$(s^{\mathcal{D}}_A)$ does not report a cycle. Let **NestedDFS**$(s^{\mathcal{D}}_A)$ be the first call to **NestedDFS** that misses a cycle, although an $\boldsymbol{s}_a \in s^{\mathcal{D}}_A$ that is on a cycle exists.

If $\boldsymbol{s}_a$ is on a cycle, then by Theorem 1 there exists a decoupled state $t^{\mathcal{D}}$ reachable from $s^{\mathcal{D}}_A$ that also contains $\boldsymbol{s}_a$. The result of Theorem 1 holds in this case because, by the definition of split transitions, the splitting does not affect reachability of member states. So there exists $t^{\mathcal{D}}_R$ reachable from $s^{\mathcal{D}}_A$ that contains $\boldsymbol{s}_a$.

Dining Philosophers										Ring Topology									
	SPIN			Cunf			DecNDFS				SPIN			Cunf			DecNDFS		
$\#\mathcal{A}$	Time	#States	Mem	Time	#E	M	Time	#States	M	$\#\mathcal{A}$	Time	#States	Mem	Time	#E	M	Time	#S	M
3	0.0	76	129	0.00	75	6	0.00	36	8	6	0.10	81.5K	133	0.00	342	6	0.00	8	8
4	0.0	348	129	0.00	162	6	0.00	97	8	7	0.95	560K	157	0.00	484	7	0.00	9	8
5	0.0	2000	129	0.00	293	6	0.00	272	8	8	8.35	3.7M	303	0.01	651	7	0.00	10	8
6	0.0	9416	131	0.01	482	7	0.01	783	8	9	73.6	24.6M	1367	0.01	843	8	0.00	11	8
7	0.2	45132	139	0.01	735	8	0.06	2290	8	10	-	-	-	0.01	1060	9	0.00	12	8
8	1.3	212K	175	0.02	1066	9	0.60	6761	8	15	-	-	-	0.03	2525	17	0.00	17	8
9	7.9	992K	333	0.02	1481	11	5.49	20.1K	9	20	-	-	-	0.10	4570	37	0.00	22	8
10	46.8	4.6M	993	0.04	1994	15	56.7	59.9K	14	25	-	-	-	0.22	7240	74	0.01	27	8
11	278.0	21.6M	3965	0.04	2386	18	558	179K	44	50	-	-	-	3.80	30K	917	0.06	52	8
12	-	-	-	0.06	2874	23	-	-	-	75	-	-	-	-	-	-	0.26	77	8

Fig. 10. Statistics on the two scaling models, where $\#\mathcal{A}$ is the number of philosophers, resp. the number of NBAs, Time is runtime in seconds, #States (#S) and #E are the number of visited states, resp. generated events, and Mem (M) is the memory usage in MiB.

Denote by $\boldsymbol{\pi}_c = l_{a+1}, \dots l_k$ the cycle part of $\boldsymbol{\pi}$. Because $\boldsymbol{s}_a$ is an accepting member state of $s^{\mathcal{D}}$, all its component states s^i_A become reference states in $s^{\mathcal{D}}_A$. Therefore, assuming that $\pi^G(s^{\mathcal{D}}_A, t^{\mathcal{D}}_R) = \pi^G(\boldsymbol{\pi}_c)$, for all components we have $s^i_A \in t^{\mathcal{D}}_R[\mathcal{A}^i]_{s^i_A}$ and a cycle is reported. If this is not the case, then either (1) $s^{\mathcal{D}}$ was not reached in **DFS**, or (2) $t^{\mathcal{D}}_R$ was not reached in **NestedDFS**$(s^{\mathcal{D}}_A)$.

In case (1), there must exist a state $s^{\mathcal{D}}_P \supseteq s^{\mathcal{D}}$ that prunes $s^{\mathcal{D}}$. But then, $s^{\mathcal{D}}_P$ contains $\boldsymbol{s}_a$, too, and **NestedDFS** was called on its l^A_G-successor $s^{\mathcal{D}}_{P,A}$ and the cycle of $\boldsymbol{s}_a$ was missed before, in contradiction.

For (2), either (a) there exists a state $t^{\mathcal{D}}_{P,R} \supseteq_R t^{\mathcal{D}}_R$ that was reached in a prior invocation of **NestedDFS** on an accepting state $s^{\mathcal{D}}_{P,A}$, or (b) a state $t^{\mathcal{D}}_{P,R} \supseteq t^{\mathcal{D}}_R$ was reached in **NestedDFS**$(s^{\mathcal{D}}_A)$ before $t^{\mathcal{D}}_R$. In both cases, $t^{\mathcal{D}}_R$ is pruned and the cycle through $\boldsymbol{s}_a$ is missed. Case (a) can only happen if $s^{\mathcal{D}}_{P,A}$ contains $\boldsymbol{s}_a$, too, because the reference states of $s^{\mathcal{D}}_A$ need to be a subset of the ones of $s^{\mathcal{D}}_{P,A}$. But then, the cycle of $\boldsymbol{s}_a$ was missed before, in contradiction. For (b), if $t^{\mathcal{D}}_R \subseteq_R t^{\mathcal{D}}_{P,R}$, then for all $\mathcal{A}^i$ we have $s^i_A \in t^{\mathcal{D}}_{P,R}[\mathcal{A}^i]_{s^i_A}$, so the cycle would have been reported before, in contradiction.

The reachability argument in (1,2a,2b) applies recursively to all predecessors of $s^{\mathcal{D}}$ in **DFS**, and of $t^{\mathcal{D}}_R$ in **NestedDFS**$(s^{\mathcal{D}}_A)$, so, unless a cycle is reported before, eventually a state $s^{\mathcal{D}}$ is reached in **DFS** that contains $\boldsymbol{s}_a$, and a state $t^{\mathcal{D}}_R$ with $s^i_A \in t^{\mathcal{D}}_R[\mathcal{A}^i]_{s^i_A}$ in **NestedDFS**$(s^{\mathcal{D}}_A)$. □

6 Experimental Evaluation

We implemented a prototype of the decoupled NDFS algorithm from Fig. 7. The input is specified in the Hanoi Omega-Automata format [1], describing a set of NBAs synchronized via global labels as in Definition 2. We compare our prototype to the SPIN model checker [20] (v6.5.1), and to the Cunf Petri-net unfolding tool [30] (v1.6.1). We also experimented with the symbolic model checkers NuSMV and PRISM [3,25], but both are significantly outperformed by the other methods. We conjecture that this is

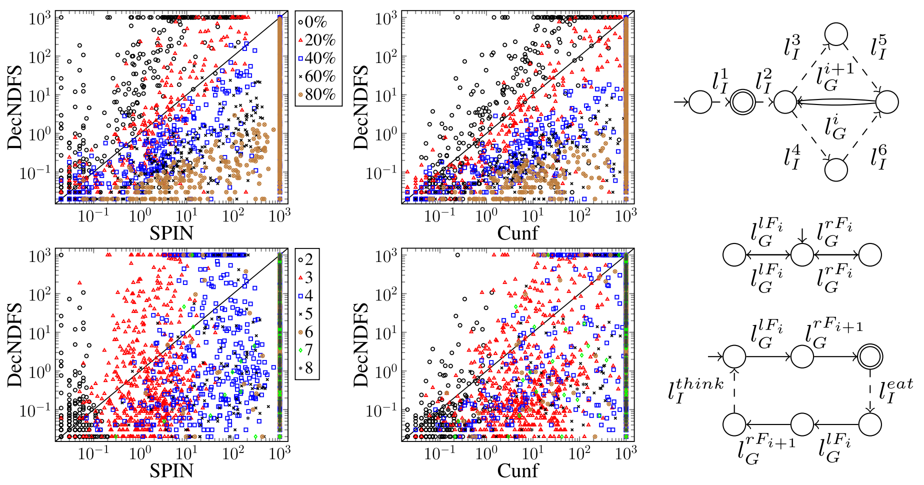


Fig. 11. Left part: scatterplots with the runtime of DecNDFS on the y-axis and the one of SPIN (left column) and Cunf (right column) on the x-axis, on randomly generated models. Each point represents one instance. In the top row, we highlight different ratios of local labels with different colors/shapes, in the bottom row we highlight different numbers of components. Right part: illustrations of the ring model (top) and the fork (middle) and philosopher (bottom) NBAs of the philosophers model. Initial (accepting) states are marked by an incoming arrow (double circle).

because both systems are not specifically designed for asynchronous execution of processes, or LTL model checking. For SPIN, we translate each NBA to a process where NBA states are represented by state labels, internal transitions by `goto` statements, and global transitions by rendezvous channel operations. For the latter, SPIN only supports synchronization of two processes at a time, so we restrict the models to global transitions with exactly two components. We model acceptance for SPIN explicitly using a monitor process that gets into an accepting state if all processes are in a local accepting state. The translation for Cunf encodes NBA states as net places and transitions as net transitions into a single Petri net, ignoring the individual components. In our prototype and in SPIN, when a lasso is reported or the algorithm proved that no lasso exists within the cut-off limits, we say that the instance was *solved*. For Cunf, we attempt to construct a complete unfolding prefix. We consider an instance solved if the construction terminates, i.e., we do not actually check the liveness property. The experiments were performed on a cluster of Intel E5-2660 machines running at 2.20 GHz, with time (memory) cut-offs of 15 min (4 GiB). Our code and models are publicly available [13].

We compare SPIN with standard options, i.e., with partial-order reduction enabled, Cunf with the cut-off rule of [10], and decoupled search (DecNDFS), using two kinds of benchmarks: (1) two scaling examples to showcase the behaviour on well-known models. One is an encoding of the dining philosophers problem, the other is a ring-shaped synchronisation topology. Both are illustrated in Fig. 11 (right). The philosophers model has $2N$ NBAs, N philosophers and N forks, synchronized by global transitions $l_G^{lF_i}$ and $l_G^{rF_i}$. After synchronizing with its left and right fork, a philosopher can perform an

Ratio	#	SPIN	Cunf	DecNDFS	$\#\mathcal{A}$	#	SPIN	Cunf	DecNDFS
					2	750	721	750	749
					3	750	696	745	712
0%	1050	373	369	319	4	750	411	243	541
20%	1050	385	384	462	5	750	130	114	372
40%	1050	397	384	573	6	750	49	53	266
60%	1050	422	394	723	7	750	24	34	180
80%	1050	468	431	888	8	750	14	23	145
$\sum$	5250	2045	1962	2965	$\sum$	5250	2045	1962	2965

Fig. 12. Number of solved instances on the random models as a function of the ratio of internal transitions (left) and the number of components $\#\mathcal{A}$ (right).

internal *eat* transition; after releasing the forks it can perform an internal *think* transition. In the ring-topology model, each component can enter a diamond-shaped region via internal transitions, followed by a synchronization with its left or right neighbor via l_G^i or l_G^{i+1}. No accepting run exists for either model. Moreover, (2) we use a set of random automata, where for each combination of a ratio of internal transitions in $\{0\%, 20\%, \dots, 80\%\}$, i.e., the number of transitions labelled with L_I divided by the total number of transitions, and a number of components in $\{2, \dots, 8\}$, we generated sets of 150 random graphs. Each component has 15 to 100 local states, out of which up to 3% are accepting (at least one). We ensure that none of the instances has an internal accepting cycle to focus on more interesting cases. One could easily implement a lookup similar to **CheckLocalAccept**, which is necessary for DecNDFS, for the other methods, too, which then essentially simplifies the problem to basic reachability.

In Fig. 10, we show detailed statistics for the scaling models, with increasing number of components $\#\mathcal{A}$ (Time in seconds, #States is the sum of states visited in both DFSs, #E is the number of events in the prefix, Memory in MiB). In dining philosophers, SPIN and DecNDFS show similar results. SPIN has a runtime advantage in the larger instances of roughly a factor of 2, but DecNDFS uses only a fraction of the memory. Cunf clearly outperforms both. This model is not very well suited to decoupled search. Only half of the NBAs have internal transitions, and only two each, and there are no non-deterministic transitions that DecNDFS could represent compactly. On the ring-topology model, SPIN manages to exhaust the search space for up to 9 components. Cunf and DecNDFS scale significantly higher, the number of decoupled states grows only linearly in the number of components. Cunf on the other hand does show a blow-up and runs out of memory between 50 and 75 components. This showcase example only serves to illustrate a near-to-optimal case for decoupled search reductions, which likely does not carry over in this extent to real-world models.

In Fig. 11 (left part), we show detailed runtime behaviour in terms of scatter plots with a per-instance comparison on the random models. Each point corresponds to one instance, where the x-value is the runtime of SPIN, resp. Cunf, and the y-value is the runtime of DecNDFS, so points below the diagonal indicate an advantage of DecNDFS. Different ratios of internal labels (top row) and numbers of components (bottom row) are depicted in different colors/shapes. We observe that, as expected, with a higher ratio of internal transitions, the advantage of DecNDFS increases significantly. For all ratios, DecNDFS clearly improves with a higher number of components.

In Fig. 12, for the same benchmark set we show the number of solved instances as a function of the ratio (left) and of the number of components (right). Here, we see that from around 20% internal transitions, DecNDFS consistently beats both SPIN and Cunf. SPIN and Cunf also benefit from the decrease in synchronizing statements, although not as much as DecNDFS. On the right, we see that starting with 4 component NBAs (#$\mathcal{A}$), DecNDFS consistently beats SPIN and Cunf. While SPIN and Cunf show a significant decline with more components, this effect is less pronounced for DecNDFS.

7 Concluding Remarks and Future Work

We have presented an approach to adapt decoupled search, an AI planning technique to mitigate the state-space explosion, to the verification of liveness properties of composed NBAs. Specifically, we have adapted a standard on-the-fly algorithm for checking ω-regular properties, nested depth-first search (NDFS), and proven its correctness. The necessary adaptations essentially pertain to the conditions that identify the existence of accepting runs, which must be handled differently given the different properties of decoupled states. Our approach extends the scope of decoupled search from safety properties, as done in [12], to liveness properties. Our experimental evaluation has shown that decoupled search can yield significant reductions in search effort across random models that consist of a set of synchronized NBAs, and simple scaling showcase examples.

We have focused on a verification problem for composed NBAs that is sufficiently general to cover significant cases like automata-based LTL model checking. We believe that our solution can be adapted to other verification problems for composed NBAs, including Büchi automata with multiple acceptance conditions such as *generalized Büchi automata*, and language intersection of the involved automata. Indeed, NDFS has successfully been used for emptiness checking of generalized NBA. We are confident that decoupled NDFS can be adapted to the compilation introduced by [33], where an additional "counter component" is added to keep track of the components that already have an accepting cyle during the nested DFS. Concretely, we believe that the verification problem of generalized NBA can be handled with adaptations by our approach: In the compilation by [33], the counter component increases its local state from 1 to n (assuming n components), one by one whenever component i has an accepting state. We can essentially apply the same compilation in decoupled NDFS, restricting the set of local states of $\mathcal{A}^i$ to the accepting ones when the counter is increased from i to $i+1$ by a separate acceptance-split transition $l_G^{A^i}$ for each $\mathcal{A}_i$. This ensures that a global cycle includes an accepting state for all components.

There are several interesting topics for future work, like the adaptation of optimizations proposed for basic NDFS (e.g. [22,32]), or the combination with orthogonal state space reduction methods, as previously done in the context of AI planning for partial-order reduction [16], symmetry reduction [18], and symbolic search [17]. Having focused on NDFS [5,22,32] in this work, we believe that the adaptation of SCC-based algorithms is a promising line of research [6,11], extending the scope of decoupled search further to model checking of CTL properties [24].

Acknowledgment. We thank Álvaro Torralba for helpful discussions about the state-splitting approach. Daniel Gnad was supported by the German Research Foundation (DFG), as part of project grant HO 2169/6-2, "Star-Topology Decoupled State Space Search". Jörg Hoffmann's research group has received support by DFG grant 389792660 as part of TRR 248 (see perspicuous-computing.science).

References

1. Babiak, T., et al.: The Hanoi omega-automata format. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 479–486. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_31
2. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. Comput. **35**(8), 677–691 (1986)
3. Cimatti, A., et al.: NuSMV 2: an OpenSource tool for symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 359–364. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45657-0_29
4. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (2001)
5. Courcoubetis, C., Vardi, M.Y., Wolper, P., Yannakakis, M.: Memory-efficient algorithms for the verification of temporal properties. Form. Methods Syst. Des. **1**(2/3), 275–288 (1992). https://doi.org/10.1007/BF00121128
6. Couvreur, J.-M.: On-the-fly verification of linear temporal logic. In: Wing, J.M., Woodcock, J., Davies, J. (eds.) FM 1999. LNCS, vol. 1708, pp. 253–271. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48119-2_16
7. Emerson, E.A., Sistla, A.P.: Symmetry and model-checking. Form. Methods Syst. Des. **9**(1/2), 105–131 (1996). https://doi.org/10.1007/BF00625970
8. Esparza, J., Heljanko, K.: A new unfolding approach to LTL model checking. In: Montanari, U., Rolim, J.D.P., Welzl, E. (eds.) ICALP 2000. LNCS, vol. 1853, pp. 475–486. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-45022-X_40
9. Esparza, J., Heljanko, K.: Implementing LTL model checking with net unfoldings. In: Dwyer, M. (ed.) SPIN 2001. LNCS, vol. 2057, pp. 37–56. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45139-0_4
10. Esparza, J., Römer, S., Vogler, W.: An improvement of McMillan's unfolding algorithm. Form. Methods Syst. Des. **20**(3), 285–310 (2002). https://doi.org/10.1023/A:1014746130920
11. Geldenhuys, J., Valmari, A.: Tarjan's algorithm makes on-the-fly LTL verification more efficient. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 205–219. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_18
12. Gnad, D., Dubbert, P., Lluch Lafuente, A., Hoffmann, J.: Star-topology decoupling in SPIN. In: Gallardo, M.M., Merino, P. (eds.) SPIN 2018. LNCS, vol. 10869, pp. 103–114. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94111-0_6
13. Gnad, D., Eisenhut, J., Lluch Lafuente, A., Hoffmann, J.: Code and benchmark models of the CAV'21 paper "Model Checking ω-Regular Properties with Decoupled Search", February 2021 https://doi.org/10.5281/zenodo.4501646
14. Gnad, D., Hoffmann, J.: Star-topology decoupled state space search. Artif. Intell. **257**, 24–60 (2018)
15. Gnad, D., Hoffmann, J.: On the relation between star-topology decoupling and petri net unfolding. In: Proceedings of the 29th International Conference on Automated Planning and Scheduling (ICAPS 2019), pp. 172–180. AAAI Press (2019)
16. Gnad, D., Hoffmann, J., Wehrle, M.: Strong stubborn set pruning for star-topology decoupled state space search. J. Artif. Intell. Res. **65**, 343–392 (2019)

17. Gnad, D., Torralba, Á., Hoffmann, J.: Symbolic leaf representation in decoupled search. In: Fukunaga, A., Kishimoto, A. (eds.) Proceedings of the 10th Annual Symposium on Combinatorial Search (SOCS 2017). AAAI Press (2017)
18. Gnad, D., Torralba, Á., Shleyfman, A., Hoffmann, J.: Symmetry breaking in star-topology decoupled search. In: Proceedings of the 27th International Conference on Automated Planning and Scheduling (ICAPS 2017), pp. 125–134. AAAI Press (2017)
19. Godefroid, P. (ed.): Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem. LNCS, vol. 1032. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-60761-7
20. Holzmann, G.: The Spin Model Checker - Primer and Reference Manual. Addison-Wesley, Boston (2004)
21. Holzmann, G.J., Peled, D.: An improvement in formal verification. In: Hogrefe, D., Leue, S. (eds.) Formal Description Techniques VII. IAICT, pp. 197–211. Springer, Boston, MA (1995). https://doi.org/10.1007/978-0-387-34878-0_13
22. Holzmann, G.J., Peled, D.A., Yannakakis, M.: On nested depth first search. In: Grégoire, J., Holzmann, G.J., Peled, D.A. (eds.) The Spin Verification System, Proceedings of a DIMACS Workshop. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, New Brunswick, New Jersey, USA, August 1996, vol. 32, pp. 23–31. DIMACS/AMS (1996)
23. Ip, C.N., Dill, D.L.: Better verification through symmetry. Form. Methods Syst. Des. **9**(1/2), 41–75 (1996). https://doi.org/10.1007/BF00625968
24. Kupferman, O., Vardi, M.Y., Wolper, P.: An automata-theoretic approach to branching-time model checking. J. ACM **47**(2), 312–360 (2000)
25. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_47
26. Laarman, A., Olesen, M.C., Dalsgaard, A.E., Larsen, K.G., van de Pol, J.: Multi-core emptiness checking of timed Büchi automata using inclusion abstraction. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 968–983. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_69
27. McMillan, K.L.: Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In: von Bochmann, G., Probst, D.K. (eds.) CAV 1992. LNCS, vol. 663, pp. 164–177. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-56496-9_14
28. McMillan, K.L.: Symbolic Model Checking. Kluwer Academic Publishers, Boston (1993)
29. Peled, D.A.: Combining partial order reductions with on-the-fly model-checking. Form. Methods Syst. Des. **8**(1), 39–64 (1996). https://doi.org/10.1007/BF00121262
30. Rodríguez, C., Schwoon, S.: Cunf: a tool for unfolding and verifying petri nets with read arcs. In: Van Hung, D., Ogawa, M. (eds.) ATVA 2013. LNCS, vol. 8172, pp. 492–495. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-02444-8_42
31. Roggenbach, M.: Determinization of Büchi-automata. In: Grädel, E., Thomas, W., Wilke, T. (eds.) Automata Logics, and Infinite Games. LNCS, vol. 2500, pp. 43–60. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-36387-4_3
32. Schwoon, S., Esparza, J.: A note on on-the-fly verification algorithms. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 174–190. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31980-1_12
33. Tauriainen, H.: Nested emptiness search for generalized büchi automata. Fund. Inform. **70**(1–2), 127–154 (2006)
34. Valmari, A.: A stubborn attack on state explosion. Form. Methods Syst. Des. **1**(4), 297–322 (1992). https://doi.org/10.1007/BF00709154

AIGEN: Random Generation of Symbolic Transition Systems

Swen Jacobs[1] and Mouhammad Sakr[1,2](✉)

[1] CISPA Helmholtz Center for Information Security, Saarbrücken, Germany
`jacobs@cispa.de`
[2] Saarland University, Saarbrücken, Germany
`sakr@react.uni-saarland.de`

Abstract. AIGEN is an open source tool for the generation of transition systems in a symbolic representation. To ensure diversity, it employs a uniform random sampling over the space of all Boolean functions with a given number of variables. AIGEN relies on reduced ordered binary decision diagrams (ROBDDs) and canonical disjunctive normal form (CDNF) as canonical representations that allow us to enumerate Boolean functions, in the former case with an encoding that is inspired by data structures used to implement ROBDDs. Several parameters allow the user to restrict generation to Boolean functions or transition systems with certain properties, which are then output in AIGER format. We report on the use of AIGEN to generate random benchmark problems for the reactive synthesis competition SYNTCOMP 2019, and present a comparison of the two encodings with respect to time and memory efficiency in practice.

1 Introduction

Verification and synthesis algorithms require benchmark problems that can be used for testing and evaluation. Unfortunately, a diverse set of benchmarks is very hard to obtain. This is a problem not only for tool developers, but also for organizers of competitions [3,4,8,11] that need to evaluate tools on a wide range of benchmarks, and to regularly search for new meaningful benchmarks.

If done properly, the generation of random benchmarks can be a solution to this problem by providing the best possible diversity and by generating new benchmarks whenever needed. On the other hand, random benchmarks come with a few caveats. First of all, completely random generation is usually not desired, since it could result in many benchmarks that, while drawn from a diverse set, are not interesting, e.g., they may be too easy or too difficult to solve for existing tools. Secondly, users may be interested in how their implementation handles benchmarks with specific properties, for instance those that require long chains of computations to reach a conclusion. Finally, if users know how *realistic* benchmarks for a certain type of verification or synthesis problem usually look like, they may want to restrict the random generation to such benchmarks, e.g., by forcing them to comply with certain conditions on their structure.

A. Silva and K. R. M. Leino (Eds.): CAV 2021, LNCS 12760, pp. 435–446, 2021.
https://doi.org/10.1007/978-3-030-81688-9_20

In this paper we present AIGEN, a tool for random generation of transition systems in a symbolic representation. We generated transition systems with partitioned transition relation, i.e., consisting of sets of Boolean functions. We ensure diversity at the level of individual Boolean functions by requiring a uniform random sampling over all Boolean functions with a given number of variables.

While for some application areas there exist tools that generate random Boolean functions in a specific form (e.g. randomly generated propositional formulas in CNF [9,16]), to the best of our knowledge none of these supports uniformly random distributions. The obvious benefit of this approach is that random samplings allow to make statements about the actual space of Boolean functions, instead of statements about a specific representation of the functions, and these benefits extend to the random generation of transition systems.

To ensure uniform random sampling, we rely on an enumeration of all Boolean functions with a given number of variables, based on their truth tables. From the truth tables one can generate in a straightforward way standard canonical representations of the functions, e.g., in canonical disjunctive normal form (CDNF) or canonical conjunctive normal form. As a more memory-efficient alternative, we developed an encoding that is inspired by data structures used for implementing reduced ordered binary decision diagrams (ROBDDs).

AIGEN implements our ROBDD-based algorithm and a CDNF-based algorithm. Development of AIGEN was motivated by the evaluation of reactive synthesis tools [13], and it was used to generate benchmarks for the reactive synthesis competition (SYNTCOMP) [11,12]. Since the existing benchmark library of SYNTCOMP consists mostly of benchmarks that were hand-crafted by tool developers, the diversity of benchmarks is limited, and their choice may be skewed towards problems or encodings that are well-suited for the existing tools. Hence, as an addition to the existing hand-crafted examples, random benchmarks are a valuable source of insight into the performance of synthesis algorithms.

Outline. We introduce BDDs and ROBDDs in Sect. 2. In Sect. 3 we present our basic idea for the random generation of symbolic transition systems, based on enumerating Boolean functions. In Sect. 4, we present a detailed description of the ROBDD-based algorithm, and in Sect. 5 the algorithm based on CDNF. Finally, in Sect. 6 we present a comparison between the ROBDD and the CDNF approaches, and we give details about our implementation and how to effectively use the tool to produce diverse benchmarks.

2 Canonical Representation of Boolean Functions

A *Binary Decision Diagram (BDD)* over a set of variables X is a directed acyclic graph $G = (V, E)$ with $V \subset \mathbb{N}$, exactly one root $v_r \in V$, and a labeling on nodes. Each terminal node $v \in V$ is labeled with a value $val(v) \in \{0, 1\}$. Each non-terminal node $v \in V$ is labeled with a variable $var(v) \in X$ and has exactly two outgoing edges, leading to nodes that are denoted by $high(v) \in V$ and $low(v) \in V$, respectively. Note that if $v \in V$ is a non-terminal node, then the directed acyclic graph rooted in v is also a BDD. It is called the *sub-BDD of G with root v*.

A BDD $G(V, E)$ over a set of variables X is *ordered* if on every path from the root to a terminal node, variables in node labels occur in the same order and each variable occurs at most once. A BDD is *reduced* if it does not contain any of the following:

- non-terminal nodes $v \neq w \in V$ with $var(v) = var(w)$, $low(v) = low(w)$ and $high(v) = high(w)$,
- terminal nodes $v \neq w \in V$ with $val(v) = val(w)$,
- a non-terminal node $v \in V$ with $low(v) = high(v)$.

Any ordered BDD can be transformed into a reduced BDD by using the isomorphism and Shannon reductions (cp. [10]). A BDD that is reduced and ordered is called a *Reduced Ordered Binary Decision Diagram (ROBDD)*.

Note that in an ROBDD, a triple $(x, high(v), low(v))$ of a node v, where $x = var(v)$, uniquely defines a sub-ROBDD. This implies that ROBDDs are a canonical representation of Boolean functions [10], i.e., for a fixed variable order there is a unique ROBDD representation for every Boolean function.

3 Enumerating Boolean Functions

Based on a canonical representation of Boolean functions, we define an enumeration, i.e., a bijective mapping from natural numbers to Boolean functions (or ROBDDs), such that any procedure that produces uniformly random natural numbers (in some range) can be used to produce uniformly random Boolean functions (in some range, see below for details).

To define our mapping, we first describe the data structure for ROBDDs that is used by various BDD packages. Then we will illustrate the data structure we use for ROBDDs and how it guarantees canonicity and uniform random distribution. In the following, we assume that $X = \{x_1, \ldots, x_m\}$ is a set of variables with a fixed order.

Unique Table. BDD packages use the so-called *unique table* as a data structure for storing ROBDD nodes. The unique table of a BDD $G = (V, E)$ over a set of variables X is a hash table that establishes a bijection between nodes $v \in V$ and triples $(x, h, l) \in X \times V \times V$ that uniquely identify them, where $x = val(v)$ if v is a terminal node, and $x = var(v)$ otherwise, $h = high(v)$ and $l = low(v)$.

Virtual ROBDD Table. We will use the ideas from the unique table that is used in BDD packages to define the virtual ROBDD table that enumerates all possible ROBDDs with respect to our variable order. This table can of course not be constructed explicitly, but the idea of this table can be used to define a (bijective) mapping from natural numbers to ROBDDs. We want to generate random Boolean functions that are based on a uniform distribution. For this reason the algorithm generates randomly a natural number $bddID \leq 2^{2^m}$ (since there are 2^{2^m} different Boolean functions of type $\mathbb{B}^m \to \mathbb{B}$), then computes a unique triple similar to the one above that corresponds to *bddID*, and then iteratively builds the complete ROBDD.

For the sake of illustrating how the algorithm computes the triple, assume that there exists a table, called *Virtual ROBDD Table* (or short: VRT), that maps natural numbers to ROBDDs, identified by a triple of variable index, and high and low children. In other words, every entry in the table maps uniquely a number $bddID \in \mathbb{N}$ (i.e. a BDD node) to a triple $(level, high, low)$ where **level** is a variable index, $high = high(bddID)$, and $low = low(bddID)$. Like the unique table, none of the entries (i.e., ROBDDs) appears twice. However, in contrast to the unique table, the VRT is based on the fixed variable order, and uses the variable index in this order instead of the variable itself. Table 1 depicts a sketch of the VRT.

Table 1. VRT: Entries in the table are in ascending order over *bddID*. Each row is annotated with a *level* and a *sublevel*. L_i denotes the i^{th} level, containing all triples with variable index i. The *sublevel* sl_{i_j} denotes the j^{th} sublevel of L_i which contains all triples of L_i in which j is the *high* or the *low* child, and the other child j' is a *bddID* that belongs to a level $L_{i'}$ with $i' < i$ such that $[j.j']$ has not appeared before in L_i. Each cell in a row annotated with L_i and sl_{i_j} is of the form $(bddID)[high.low]$ where *bddID* is the unique identifier of the triple $(i, high, low)$. Let $Y_1 = 2^{2^{i-1}}$ and $Y_2 = \sum_{m=1}^{j-1} 2(2^{2^{i-1}} - m)$.

L_0		(1)[0]	(2)[1]				
L_1	sl_{1_1}	(3)[1.2]	(4)[2.1]				
L_2	sl_{2_1}	(5)[1.2]	(6)[2.1]	(7)[1.3]	(8)[3.1]	(9)[1.4]	(10)[4.1]
	sl_{2_2}	(11)[2.3]	(12)[3.2]	(13)[2.4]	(14)[4.2]		
	sl_{2_3}	(15)[3.4]	(16)[4.3]				
$\vdots$	$\vdots$		$\vdots$				
L_i	sl_{i_1}	$(Y_1{+}1)[1.2]$	$(Y_1{+}2)[2.1]$	$\ldots$			$(Y_1{+}2(Y_1{-}1))[Y_1.1]$
	$\vdots$		$\vdots$				
	sl_{i_j}	$(Y_1{+}Y_2{+}1)[j.j{+}1]$		$\ldots$			$(Y_1{+}Y_2{+}2(Y_1{-}j))[Y_1.j]$
	$\vdots$		$\vdots$				
	$sl_{i_{Y_1-1}}$						
$\vdots$	$\vdots$						

Note that a *bddID* between 1 and 2^{2^m} corresponds to a Boolean function with at most m input variables, and a *bddID* between $2^{2^{m-1}}+1$ and 2^{2^m} corresponds to a function with exactly m input variables. Thus, to uniformly sample Boolean functions, we can use a random number generator that uniformly samples natural numbers in such a range.

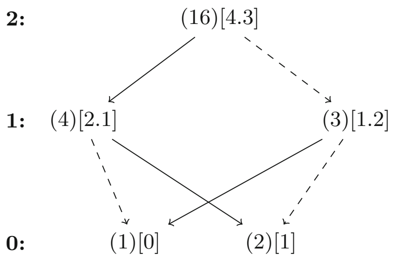


Fig. 1. BDD generated for number 16. Equivalent to boolean function: $x_2 x_1 + \bar{x_2}\bar{x_1}$. The numbers on the left of the BDD represent the *level* i.e. corresponding variable indices.

It is important to remember that the VRT is not constructed explicitly. Instead, given a number of variables m, and based on the predefined ordering of ROBDD in the VRT (2^{2^m} ROBDDs), the algorithm generates first a random number $bddID \leq 2^{2^m}$, then computes the triple $(level, high, low)$ to which $bddID$ maps. We note: $level$ (or i) is equal to $\lceil log_2(log_2(bddID)) \rceil$. Let $Y_1 = 2^{2^{i-1}}$, then we solve the following system of equations to compute x which is equivalent to the *sublevel*:

$Y_1 + 2(Y_1 - 1) + \ldots + 2(Y_1 - x) < bddID$

$Y_1 + 2(Y_1 - 1) + \ldots + 2(Y_1 - (x+1)) \geq bddID$

High and low are then computed according to what is given in the table, see Sect. 4 for more details. Figure 1 shows the BDD generated for $bddID = 16$ which is equivalent to: $x_2 x_1 + \bar{x_2}\bar{x_1}$.

4 Random Generation of (Controllable) Transition Systems

In this section we present our algorithm for generating random transition systems, represented as AIGER circuits [5]. We use a generalization of the usual notion of transition systems that allows some of the input signals to be declared as controllable. This is useful to define synthesis problems, i.e., a synthesis procedure can define how these inputs should behave depending on the state and uncontrollable inputs of the system.

A *controllable transition system* (or short: controllable system) TS is a 6-tuple $(L, X_u, X_c, \boldsymbol{F}, BAD, q_0)$, where L is a set of state variables (also called *latches*), X_u is a set of uncontrollable input variables, X_c is a set of controllable input variables, $\boldsymbol{F} = (f_1, ..., f_{|L|})$ with $f_i : \mathbb{B}^L \times \mathbb{B}^{X_u} \times \mathbb{B}^{X_c} \rightarrow \mathbb{B}$ is a vector of update functions for the latches, $BAD : \mathbb{B}^L \rightarrow \mathbb{B}$ is the set of unsafe states, and q_0 is the initial state where all latches are initialized to 0.

Then, the idea of our tool for random generation of transition systems can be summarized in the following way:

- The user input determines parameters of the system, such as the number of latches and controllable or uncontrollable inputs.

- For every latch, we generate a random Boolean function that determines how this latch is updated based on the current state and input of the system, represented as ROBDD as described in Sect. 3.
- Additionally, we generate a random Boolean function that determines the set of unsafe states of the system.
- The system composed of these functions is then encoded into an AIGER circuit.

4.1 Random Generation Algorithm

The procedure GenerateRandomAiger takes as input the number of latches l, uncontrollable inputs u, controllable inputs c, the bound o, optionally a list of seeds (i.e., natural numbers used to initialize a pseudorandom number generator). As output it produces a file in AIGER format.

Lines 3–6 generate for every latch a random ROBDD that represents an *update function* $\mathbb{B}^{l+c+u} \rightarrow \mathbb{B}$ for the latch, i.e., a function that takes all current values of inputs and latches as input, and returns a new value for the given latch. Line 4 generates a random integer with 2^{vars} random bits, i.e., a natural number between 1 and $2^{2^{vars}}$. All the seeds used for generating the random integers will be written in the comment section at the end of the generated file. These seeds can be fed to the algorithm in order to regenerate the same instance. Line 5 constructs the ROBDD that corresponds to the generated number. Line 6 converts the constructed ROBDD into an AIG (And-Inverter Graph) relying on the fact that a BDD can be seen as a network of multiplexers.

Lines 8–10 construct the ROBDD of the function $f_{BAD} : \mathbb{B}^o \rightarrow \mathbb{B}$ which uses $o \leq l$ latch variables. The set of unsafe states BAD is then defined as $f(x_{i_1}, \ldots, x_{i_o}) \wedge \bigwedge_{j \in \{1,\ldots,l\} \setminus \{i_1,\ldots,i_o\}} x_j$ where the indices $\{i_1, \ldots, i_o\}$ are also picked randomly. Line 11 creates the AIGER file that corresponds to the total number of variables and to the update functions that were randomly generated. Line 12 uses the *ABC* [7] tool to reduce the size of the generated AIGER file.

ConstructBDD is a recursive procedure for constructing all the nodes of the ROBDD that corresponds to the unique ID *bddID*. It starts with the root node and recursively proceeds to the child nodes until it reaches the nodes 0 or 1. Line 14 checks if the node was already created. If not, Line 15 computes the triple $(level, high, low)$ that uniquely represent the node and adds it to the table $robddTable$. Lines 18–17 construct the child nodes. Note that the $robddTable$ is initialized with the IDs 1 and 2 which correspond respectively to nodes 0 and 1.

Given an ID, procedure GetChildren computes the triple $(level, high, low)$. Line 20 computes the level. Lines 21–24 compute the sublevel. Note that, as depicted in Table 1, a sub-level s_{i_j} has size $2(2^{2^{i-1}} - j)$, where $2^{2^{i-1}}$ is the sum of the sizes of all levels that are smaller than i. To compute the sublevel, we have to compute the single solution of the system of inequations in Lines 22, 23, to see that check the VRT table. Line 25 computes the ID of the left-most bit in the sub-level. Lines 26–27 compute the ID of the second child node, and Lines 28–30 check which node is the low edge and which node is the high edge.

Algorithm 1. Generate Random Aiger

```
1: procedure GenerateRandomAiger($l, u, c, o$)
2:     $vars \leftarrow l + u + c, l' = l, robddTable = [(1, 0), (2, 1)]$
3:     while $l' > 0$ do
4:         $rand\_fct\_ID = random.getrandbits(2^{vars}) + 1$
5:         ConstructBDD($rand\_fct\_ID, robddTable$)
6:         $aigerTable[rand\_fct\_ID]$ =ConvertToAIG($rand\_fct\_ID, robddTable$)
7:         $l' \leftarrow l' - 1$
8:     $bad\_ID = random.getrandbits(2^o) + 1$
9:     ConstructBDD($bad\_ID, robddTable$)
10:    $aigerTable[bad\_ID]$ =ConvertToAIG($bad\_ID, robddTable$)
11:    $aigerFilePath$ ←CreateAiger($aigerTable$)
12:    abcMinimize($aigerFilePath$)
13: procedure ConstructBDD($bddID, robddTable$)
14:    if $bddID \notin robddTable$ then
15:        $(level, high, low) \leftarrow$ GetChildren($bddID$)
16:        $robddTable[bddID] \leftarrow (level, high, low)$
17:        ConstructBDD($high$)
18:        ConstructBDD($low$)
19: procedure GetChildren($bddID$)
20:    $level = \lceil log_2(log_2(bddID)) \rceil$
21:    $n \leftarrow 2^{2^{level-1}}$
22:    $sli$ ←ComputeASol($n + 2(n-1) + \ldots + 2(n-x) < bddID$,
23:                       $n + 2(n-1) + \ldots + 2(n-(x+1)) \geq bddID$)
24:    $child_1 \leftarrow sli + 1$
25:    $sl\_1\_ID \leftarrow n + 2(n-1) + \ldots + 2(n - sli)$
26:    $sle \leftarrow bddID - sl\_1\_ID$
27:    $child_2 \leftarrow child_1 + \lceil sle/2 \rceil$
28:    if $sle \mod 2 \neq 0$ then
29:        return $(level, child_1, child_2)$
30:    return $(level, child_2, child_1)$
```

5 CDNF-based Algorithm

An obvious alternative to our ROBDD approach is to make use of the canonical disjunctive or conjunctive normal forms to generate random Boolean functions. Algorithm 2 employs CDNF as it is easier to convert to And-Inverter graph. CDNF is usually constructed directly from a truth table by taking the OR of all satisfying assignments. To convert a Boolean formula $f_i = cl_1 \vee cl_2 \vee \ldots \vee cl_n$ in CDNF to AIG, we consider its equivalent $f'_i = \neg(\neg cl_1 \wedge \neg cl_2 \wedge \ldots \wedge \neg cl_n)$.

The procedure DNFGenerateRandomAiger takes as input the number of latches l, uncontrollable inputs u, controllable inputs c, the bound o, and produces a file in AIGER format as output. Lines 3–6 generate a random update function for every latch. Line 4 generates a random bit vector of size 2^{vars}.

Algorithm 2. Random Aiger generation using DNF approach

```
 1: procedure DNFGenerateRandomAiger($l, u, c, o$)
 2:     $vars \leftarrow l + u + c, l' \leftarrow 0$
 3:     while $l' < l$ do
 4:         $truthTable = random.getrandbits(2^{vars})$
 5:         $dnfFormula =$ ConstructDNF($truthTable, vars$)
 6:         $aigerTable[l'] =$ ConvertToAIG($dnfFormula$)
 7:         $l' \leftarrow l' + 1$
 8:     $badTruthTable = random.getrandbits(2^{o})$
 9:     $badDnfFormula =$ ConstructDNF($truthTable, o$)
10:     $aigerTable[l'] =$ ConvertToAIG($badDnfFormula$)
11:     $aigerFilePath \leftarrow$ CreateAiger($aigerTable$)
12:     ABCMinimize($aigerFilePath$)
13: procedure ConstructDNF($bitVec, vars$)
14:     $dnfFormula \leftarrow True, i \leftarrow 0$
15:     while $i < bitVec.size()$ do
16:         if $bitVec[i] = 1$ then
17:             $clauseBitvec \leftarrow$ToBinary($i, vars$)
18:             $dnfClause \leftarrow$ToClause($clauseBitvec$)
19:             $dnfFormula \leftarrow dnfFormula \wedge negate(dnfClause)$
20:     return $negate(dnfFormula)$
```

This bit vector represents the valuation of all the *minterms*[1] of the truth table that represents the random function f_i. For instance, if the left-most bit of the bit vector is equal to 1, then $x_{c_0} = 0, \ldots, x_{c_{|c|-1}} = 0, x_{u_0} = 0, \ldots, x_{u_{|u|-1}} = 0, x_{l_0} = 0, \ldots, x_{l_{|l|-1}} = 0$ is a satisfying assignment of f_i. Similarly, if the last element of the bit vector is equal to 1, then $x_{c_0} = 1, \ldots, x_{c_{|c|-1}} = 1, x_{u_0} = 1, \ldots, x_{u_{|u|-1}} = 1, x_{l_0} = 1, \ldots, x_{l_{|l|-1}} = 1$ is a satisfying assignment of f_i. Line 5 builds the random function that corresponds to the generated bit vector, and Line 6 converts it to AIG. Lines 8–10 generate the output random function, and Lines 11, 12 creates the AIGER file and call ABC to minimize it.

The procedure ConstructDNF takes as input a bit vector and the number of variables and generates the corresponding Boolean function. Line 14 initializes the DNF function to be created. For every element in the bit vector, if the ith element is equal to 1 (Line 15) then, in order to obtain the corresponding minterm, Line 17 converts the positive integer i to binary. For instance if $i = 3$ and vars $= 3$, then the minterm $x_c \wedge \neg x_u \wedge x_l$ is created. Line 18 creates the corresponding minterm. Line 19 negates the created clause and adds is to the DNF formula. Line 20 returns the negation of the constructed formula. As mentioned earlier, as the formula represented by the truth table is in DNF, we need to generate its equivalent that includes only AND and NOT logical gates. For instance giving a formula $f_i = cl_1 \vee cl_2 \vee \ldots \vee cl_n$ in CDNF, we construct its equivalent $f_i' = \neg(\neg cl_1 \wedge \neg cl_2 \wedge \ldots \wedge \neg cl_n)$.

[1] A minterm of n variables is a product (logical AND) of the variables in which each appears exactly once in uncomplemented or complemented form.

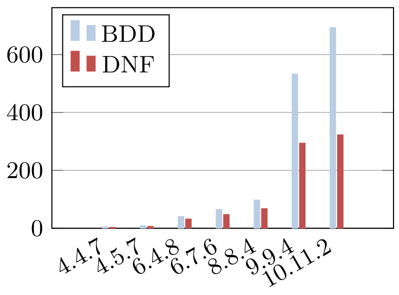


Fig. 2. Average number of AND gates.

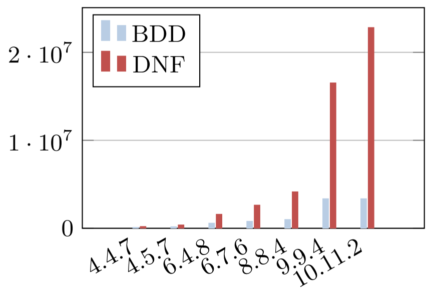


Fig. 3. Average running time in seconds including the time needed to minimize the generated Aiger circuit using ABC tool.

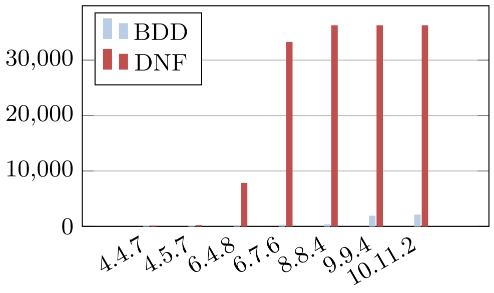


Fig. 4. Average running times.

6 Implementation and Evaluation

AIGEN is implemented in Python, and a virtual machine with the tool ready to run is available at https://doi.org/10.5281/zenodo.4721314 [14]. The source code of AIGEN is also publicly available at https://github.com/mhdsakr/AIGEN-Tool, allowing interested users to add functionality, e.g., in order to add further parameters to generate only Boolean functions or transition systems with certain properties. It uses the mpmath [15] library together with GMPY [1] to deal with large numbers. By default, mpmath uses Python integers, however if GMPY is also installed on the operating system, mpmath will automatically detect it and use gmpy integers intead. This makes mpmath perform much faster, particularly at high precision (approximately above 100 digits). Furthermore, AIGEN uses ABC [7], and the AIGER tool set [6] to post-process AIGER circuits.

AIGEN has been used to generate thousands of random transition systems. Figures 4, 2, 3 shows average times and sizes for generating systems where, for example, 4.3.7 denotes systems with 4 controllable inputs, 3 uncontrollable inputs, and 7 latches ($o = l = 7$). These times were measured on a laptop with quad-core i7-6600U CPU at 2.6 GHz and 20 GB RAM.

Figures 4 and 2 compare average running time and average number of AND-gates between the ROBDD and DNF approaches. These results are without the use of the ABC tool (i.e. the command "ABCMinimize(aigerFilePath)" was skipped). Figure 4 shows that the DNF approach was faster in all cases which was expected due to the fact that generating a random ROBDD is much more complex than generating a truth table. Figure 2 shows that the ROBDD approach is much better in all cases. Figure 3 compares average running time between the ROBDD and DNF approaches, including the time needed for the ABC tool to minimize the generated transition system. Benchmarks 8.8.4, 9.9.4, and 10.11.2 timed out for the DNF approach(we used 10 h as a time limit). Obviously the ABC tool needed a lot of time to process these benchmarks. After a thorough inspection, the reason was, in addition to the huge size of these circuits, the incredibly long chains of AND-gates for every generated Boolean function. This figure shows that the total running time of the tool was way better when used with the ROBDD approach.

The Effect of Parameters. Although the benchmarks are randomly generated, AIGEN allows the user to choose the input parameters to obtain benchmarks with certain properties that correspond to their needs, for example:

- The degree of the generated graph (i.e., the transition system) is equal to 2^{u+c}, therefore increasing the ratio $(u + c)/l$ will make the graph more congested and consequently more complex.
- The parameter o gives the user the ability to determine the size of the set of unsafe states, i.e., the number of unsafe states cannot exceed 2^o. Accordingly, increasing the ratio o/l will increase the probability that the error set is reachable, and decreasing this ratio will lower the probability.
- Increasing the ratio c/u will increase the probability that the benchmark is realizable, and decreasing it will serve the opposite goal. Moreover, if this ratio is close to 1 the realizability check will be harder, since the probability of realizability will be roughly equal to the probability of unrealizability.

To demonstrate the effect of these parameters, Table 2 shows the running time and results (realizable or unrealizable) of the synthesis tool *SimpleBDDSolver* on selected benchmarks, generated using the ROBDD-based approach, in SyntComp 2019. SimpleBDDSolver has won all previous iterations of the Syntcomp competition. A benchmark name contains the parameters that were used to generate the file, e.g., *random_n_19_1_3_15_14_1_abc* means that the benchmark has in total 19 variables with 1 controllable input, 3 uncontrollable inputs, 15 latches, and $o = 14$. The table shows that the example benchmarks with ratio $c/u = 1/3$ or $c/u = 1/5$ were unrealizable, the benchmarks with ratio $c/u = 2$ were realizable, while benchmarks with ratio $c/u = 1/2$ were difficult to solve for the tool, which timed out while trying to solve them. Note that a benchmark with $c/u = 1/5$ *can* still be realizable, and one with $c/u = 2$ *can* be unrealizable—it is just unlikely that this is the case for a randomly generated benchmark.

Table 2. Results of SimpleBDDSolver on selected random benchmarks generated by AIGEN in SyntComp 2019 [2]

Benchmark	Time (s)	Result
random_n_19_1_3_15_14_1_abc	3412.41	UNREALIZABLE
random_n_19_1_5_13_13_2_abc	1361.39	UNREALIZABLE
random_n_19_1_2_16_14_8_abc	Timeout	–
random_n_19_1_4_14_13_11_abc	Timeout	–
random_n_19_4_2_13_12_11_abc	43.68	REALIZABLE
random_n_19_4_2_13_12_12_abc	35.71	REALIZABLE
random_n_19_4_2_13_12_3_abc	240.61	REALIZABLE
random_n_19_4_2_13_12_62_abc	299.5	REALIZABLE
random_n_19_4_2_13_12_95_abc	258.92	REALIZABLE

7 Conclusion

We have presented AIGEN, a tool for the generation of random transition systems in a symbolic representation, using either ROBDDs or CDNF for representing Boolean functions. Although the ROBDD based approach generates much smaller symbolic transition systems, the CDNF approach is faster when ABC minimization procedure is disabled. In contrast to the ROBDD approach, to generate a random formula in CDNF, no complex computation is needed. However, when using minimization, the huge size of these formulas becomes a problem for ABC as it has to deal and inspect all the generated AND-gates.

In future work, instead of using a fixed variable order, we will also allow to use a random order. The drawback of a fixed order is that some Boolean functions only have a large ROBDD representation, even though smaller ones exist with different orderings, and vice versa. Going further, we plan to include variable reorder techniques to find an order that leads to small ROBDDs at runtime. Finally, we also plan to investigate the use of AIGEN for finding bugs in verification and synthesis tools.

References

1. Gmpy. https://pypi.python.org/pypi/gmpy2/
2. Online results of SYNTCOMP 2019. https://www.starexec.org/starexec/secure/details/job.jsp?id=35621
3. Barrett, C.W., de Moura, L.M., Stump, A.: Design and results of the first satisfiability modulo theories competition (SMT-COMP 2005). J. Autom. Reasoning **35**(4), 373–390 (2005). https://doi.org/10.1007/s10817-006-9026-1
4. Beyer, D.: Competition on software verification. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 504–524. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28756-5_38
5. Biere, A.: AIGER Format and Toolbox. http://fmv.jku.at/aiger/

6. Biere, A.: The AIGER And-Inverter Graph (AIG) format version 20071012. Technical report, FMV Reports Series, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria (2007)
7. Brayton, R., Mishchenko, A.: ABC: an academic industrial-strength verification tool. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 24–40. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_5
8. Cabodi, G., et al.: Hardware model checking competition 2014: An analysis and comparison of solvers and benchmarks. J. Satis. Boolean Model. Comput. **9**, 135–172 (2016)
9. Cheeseman, P.C., Kanefsky, B., Taylor, W.M.: Where the really hard problems are. In: IJCAI, pp. 331–340. Morgan Kaufmann (1991). http://ijcai.org/Proceedings/91-1/Papers/052.pdf
10. Drechsler, R., Becker, B.: Binary Decision Diagrams: Theory and Implementation. Springer, Heidelberg (2013)
11. Jacobs, S., et al.: The first reactive synthesis competition (SYNTCOMP 2014). Int. J. Softw. Tools Technol. Transfer **19**(3), 367–390 (2016). https://doi.org/10.1007/s10009-016-0416-3
12. Jacobs, S., et al.: The 5th reactive synthesis competition (SYNTCOMP 2018): Benchmarks, participants & results. CoRR abs/1904.07736 (2019). http://arxiv.org/abs/1904.07736
13. Jacobs, S., Sakr, M.: A symbolic algorithm for lazy synthesis of eager strategies. In: Lahiri, S.K., Wang, C. (eds.) ATVA 2018. LNCS, vol. 11138, pp. 211–227. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01090-4_13
14. Jacobs, S., Sakr, M.: AIGEN: Random generation of symbolic Boolean functions and transition systems (2021). https://doi.org/10.5281/zenodo.4721314
15. Johansson, F., et al.: mpmath: a Python library for arbitrary-precision floating-point arithmetic (version 0.18), December 2013. http://mpmath.org/
16. Mitchell, D.G., Selman, B., Levesque, H.J.: Hard and easy distributions of SAT problems. In: AAAI, pp. 459–465. AAAI Press/The MIT Press (1992). http://www.aaai.org/Library/AAAI/1992/aaai92-071.php

GPU Acceleration of Bounded Model Checking with ParaFROST

Muhammad Osama(✉) and Anton Wijs

Eindhoven University of Technology,
Eindhoven, The Netherlands
{o.m.m.muhammad,a.j.wijs}@tue.nl

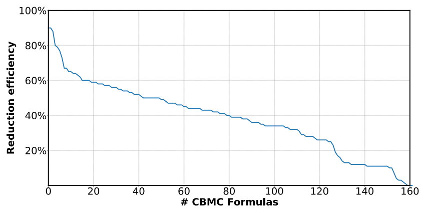
Abstract. The effective parallelisation of Bounded Model Checking is challenging, due to SAT and SMT solving being hard to parallelise. We present ParaFROST, which is the first tool to employ a graphics processor to accelerate BMC, in particular the simplification of SAT formulas before and repeatedly during the solving, known as pre- and inprocessing. The solving itself is performed by a single CPU thread. We explain the design of the tool, the data structures, and the memory management, the latter having been particularly designed to handle SAT formulas typically generated for BMC, i.e., that are large, with many redundant variables. Furthermore, the solver can make multiple decisions simultaneously. We discuss experimental results, having applied ParaFROST on programs from the Core C99 package of Amazon Web Services.

Keywords: Bounded model checking · SAT solving · GPU computing

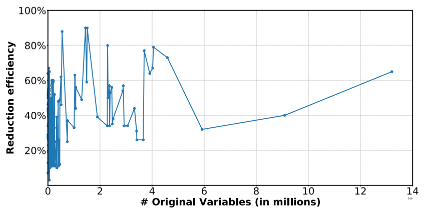
1 Introduction

Bounded Model Checking (BMC) [5] determines whether a model M satisfies a certain property φ expressed in temporal logic, by translating the model checking problem to a propositional satisfiability (SAT) problem or a Satisfiability Modulo Theories (SMT) problem. The term *bounded* refers to the fact that the BMC procedure searches for a counterexample to the property, i.e., an execution trace, which is bounded in length by an integer k. If no counterexample up to this length exists, k can be increased and BMC can be applied again. This process can continue until a counterexample has been found, a user-defined threshold has been reached, or it can be concluded (via k-induction [38]) that increasing k further will not result in finding a counterexample. CBMC [14] is an example of a successful BMC model checker that uses SAT solving. CBMC can check ANSI-C programs. The verification is performed by *unwinding* the loops in the program under verification a finite number of times, and checking whether the bounded

M. Osama—This work is part of the GEARS project with project number TOP2.16.044, which is (partly) financed by the Netherlands Organisation for Scientific Research (NWO).



(a) The amount of variable redundancy in CBMC formulas



(b) The amount of variable redundancy w.r.t. the number of variables

Fig. 1. Variable redundancy in CBMC SAT formulas

executions of the program satisfy a particular safety property [22]. These properties may address common program errors, such as null-pointer exceptions and array out-of-bound accesses, and user-provided assertions.

The performance of BMC heavily relies on the performance of the solver. Over the last decade, efficient SAT solvers [3,6,17,26] have been developed and applied for BMC [5,10–12,25]. Effectively *parallelising* BMC is hard. Parallel SAT solving often involves running several solvers, each solving the problem in its own way [18]. For BMC, multiple solvers can be used to solve the problem for different values of the bound k in parallel [1,21]. However, in these approaches, the individual solvers are still single-threaded.

Recently, Leiserson *et al.* [23] concluded that in the future, advances in computational performance will come from *many-threaded* algorithms that can employ hardware with a massive number of processors. Graphics processors (GPUs) are an example of such hardware. Multi-threaded BMC model checkers have been proposed, such as in [13,19,35], but these address tens of threads, not thousands.

In this paper, we propose the application of GPUs to accelerate SAT-based BMC. To the best of our knowledge, this is the first time this is being addressed. Recently, GPUs have been applied for explicit-state model checking and graph analysis [8,9,40,41]. In SAT solving, we used GPUs to accelerate test pattern generation [31], metaheuristic search [42], *preprocessing* [32,33] and *inprocessing* [34]. In these operations, a given SAT formula is simplified, i.e., it is rewritten to a formula with fewer variables and/or clauses, while preserving satisfiability, using various simplification rules. In preprocessing, this is only done once before the solving starts, while in inprocessing, this is done periodically during the solving. While the impact of accelerating these procedures has been demonstrated [34], its impact on BMC has not yet been addressed.

The structure of typical BMC SAT formulas suggests that GPU pre- and inprocessing will be effective. Figure 1a shows for a BMC benchmark set taken from the Core C99 package of Amazon Web Services (AWS)[1] [2], consisting of 168 problems of various data structures, that propositional formulas produced by CBMC tend to have a substantial amount of redundant variables that can

[1] We thank Daniel Kroening and Natasha Jebbo for pointing us to this package.

be removed using simplification procedures. For approximately 50% of the cases, 40% of the variables can be removed. Furthermore, Fig. 1b presents the amount of redundancy in relation to the total number of variables in the formula. It indicates that when a formula contains one million variables or more, at least 25% of those are redundant, and often many more. In the benchmark set, the maximum number of variables in one formula is 13 million (encoding the verification of the `priority-queue shift-down` routine), of which 65% is redundant. In contrast, the largest formula we encountered in the application track of the 2013–2020 SAT competitions that is not encoding a verification problem only has 0.2 million variables (it encodes a graph coloring problem [29]).

Contributions. We present the SAT solver ParaFROST that applies Conflict Driven Clause Learning (CDCL) [26] with GPU acceleration of pre- and inprocessing [32–34], tuned for BMC. It has been implemented in CUDA C++ v11 [28], is based on CaDiCaL [6], and interfaces with CBMC.

Having to deal on a GPU with large formulas with a lot of redundancy offers particular challenges. The elimination of variables typically leads to actually adding new clauses, and since the amount of memory on a GPU is limited, this cannot be done carelessly. Therefore, first of all, we have worked on compacting the data structure used to store formula clauses in ParaFROST as much as possible, while still allowing for the application of effective solving optimisations. Second of all, we introduce *memory-aware* variable elimination, to avoid running out of memory due to adding too many new clauses. In practice, we experienced this problem when applying the original procedure of [34] for BMC.

Additionally, to support BMC, ParaFROST must be an *incremental* solver, i.e., it must exploit that a number of very similar SAT problems are solved in sequence [16]. The procedure in [34] does not support this, so we extended it.

Finally, because of the many variables in BMC SAT formulas, ParaFROST supports *Multiple Decision Making* (MDM) in the solving procedure, as presented in [30]. With MDM, multiple decisions can be made at once, periodically during the solving. In case there are many variables, there is more potential to make many decisions simultaneously. We have generalised the original MDM decision procedure [30], making it easier to integrate MDM in solvers other than MiniSat and Glucose [3]. The effectiveness of MDM in BMC has never been investigated before, nor has been combined with GPU pre- and inprocessing.

2 Background

SAT Solving. We assume that SAT formulas are in conjunctive normal form (CNF). A CNF formula is a conjunction of m clauses $C_1 \wedge \cdots \wedge C_m$, and each clause C_i is a disjunction of n literals $\ell_1 \vee \cdots \vee \ell_n$. A literal is a Boolean variable x or its negation $\neg x$, also referred to as $\bar{x}$. The domain of all literals is $\mathbb{L}$. A clause can be interpreted as a set of literals, i.e., $\{\ell_1, \dots, \ell_n\}$ encodes $\ell_1 \vee \ldots \vee \ell_n$, and a SAT formula $\mathcal{S}$ as a set of clauses, i.e., $\{C_1, \dots, C_m\}$ encodes $C_1 \wedge \ldots \wedge C_m$. With $\mathit{Var}(C)$, we refer to the set of variables in C: $\mathit{Var}(C) = \{x \mid x \in C \vee \bar{x} \in C\}$. The set $\mathcal{S}_\ell$ consists of all clauses in $\mathcal{S}$ containing ℓ: $\mathcal{S}_\ell = \{C \in \mathcal{S} \mid \ell \in C\}$.

In CDCL, clauses are `LEARNT` or `ORIGINAL`. A `LEARNT` clause has been derived by the CDCL clause learning process during solving, and an `ORIGINAL` clause is part of the formula. We refer with $\mathcal{L}$ to the set of `LEARNT` clauses.

For a set of assignments Σ, consisting of all literals that have been assigned **true**, a formula $\mathcal{S}$ evaluates to **true** iff $\forall C \in \mathcal{S}. \exists \ell \in C. \ell \in \Sigma$. When a *decision* is made, a literal is picked and added to Σ. Each assignment is associated with a *decision level* (time stamp) to monitor the assignment order. We call a clause C *unit* iff a single literal in it is still unassigned, and the others are assigned **false**, i.e., $|\mathit{Var}(C) \setminus \mathit{Var}(\Sigma)| = 1$ and $C \cap \Sigma = \emptyset$.

Variable-Clause Elimination (VCE). Variables and clauses can be removed from formulas by applying *simplification rules* [15,20]. They rewrite a formula to an equi-satisfiable one with fewer variables and/or clauses. Applying them is referred to as pre- and inprocessing, before and during the solving, respectively.

Incremental Bounded Model Checking. Since 2001, incremental BMC has been applied to hardware and software verification [16,39]. It relies on incremental SAT solving [16]. In CDCL, clauses are learnt during the solving each time a wrong decision has been made, to avoid making those decisions again in the future. Incremental SAT solving builds on this: when multiple SAT formulas with similar characteristics are solved sequentially, then in each iteration, the clauses learnt in previous iterations are reused. An efficient approach to add and remove clauses is by using *assumptions* [16], which are initial assignments.

For BMC, the transition relation of a system design and the (negation of) the property to be verified are encoded in a SAT formula. A predicate $\mathcal{I}(s_0)$ identifies the initial states, $\delta(s_i, s_{i+1})$ encodes the transition relation at trace depth i, and $\mathcal{E}(i) = \bigvee_{0 \leq j \leq i} e(s_j)$ encodes the reachability of an error state up to trace depth i, where $e(s_j)$ is **true** iff state s_j is an error state. For incremental BMC, additional unit clauses σ_i are used. These predicates are combined to define the following series of SAT formulas $\mathcal{S}(i)$ that must be solved incrementally:

$$\begin{aligned} \mathcal{S}(0) &= \mathcal{I}(s_0) \wedge (\mathcal{E}(0) \vee \sigma_0), \text{ under assumption } \neg\sigma_0 \\ \mathcal{S}(i+1) &= \mathcal{S}(i) \wedge \delta(s_i, s_{i+1}) \wedge \sigma_i \wedge (\mathcal{E}(i+1) \vee \sigma_{i+1}), \text{ under assumption } \neg\sigma_{i+1} \end{aligned}$$

Formula $\mathcal{S}(i)$ is satisfiable iff an error state is reachable via a trace with a length up to i [16,39]. At iteration $i+1$, we know that $\mathcal{E}(i)$, included via $\mathcal{S}(i)$, cannot be satisfied (otherwise iteration $i+1$ would not have been started). This means that $\mathcal{E}(i)$ must be removed to avoid that $\mathcal{S}(i+1)$ is unsatisfiable. To effectively remove $\mathcal{E}(i)$, σ_i is assigned **true**, resulting in $\mathcal{E}(i) \vee \sigma_i$ being satisfied. In general, at iteration i, σ_i is assigned **false**, while in iterations $i' > i$, it is assigned **true**.

GPU Programming. CUDA [28] is NVIDIA's parallel computing platform that can be used to develop general purpose GPU programs. A GPU consists of multiple streaming multiprocessors (SMs), and each SM contains several streaming processors (SPs). A GPU program consists of a *host* part, executed on a CPU, and *device* functions, or *kernels*, executed on a GPU. Each time a kernel is launched, the number of threads that need to execute it is given. On the SPs, the threads are executed. Compared to a CPU thread, GPU threads perform

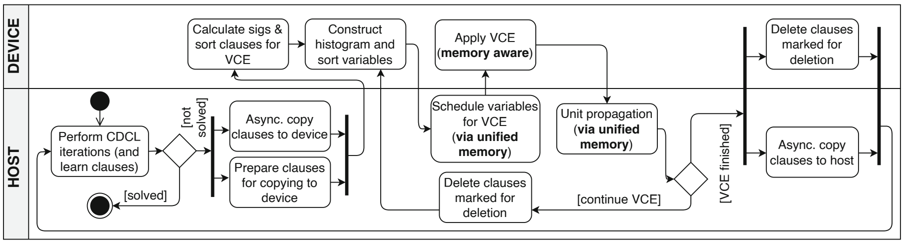

DEVICE
HOST
Calculate sigs & sort clauses for VCE
Construct histogram and sort variables
Apply VCE (**memory aware**)
Delete clauses marked for deletion
Perform CDCL iterations (and learn clauses)
[not solved]
[solved]
Async. copy clauses to device
Prepare clauses for copying to device
Schedule variables for VCE (**via unified memory**)
Unit propagation (**via unified memory**)
Delete clauses marked for deletion
[continue VCE]
[VCE finished]
Async. copy clauses to host

Fig. 2. An activity diagram for the workflow of ParaFROST.

a relatively simple task. In particular, they read some data, perform a computation, and write the result. This allows the SPs to switch contexts easily. In practice, one to two orders of magnitude more threads are typically launched than the number of SPs, which results in hiding the memory latency: whenever a thread is waiting for some data, the associated SP can switch to another thread.

A GPU has various types of memory. Relevant here are *registers* and *global* memory. Global memory is used to copy data between the host and the device. *Registers* are used for on-chip storage of thread-local data. Global memory has a much higher latency than registers. We use *unified memory* [28] to store clauses. Unified memory creates one virtual memory pool for host and device. In this way, the same memory addresses can be used by the host and the device, combining the main memory of the host side and the global memory of the device side.

3 GPU-Accelerated Bounded Model Checking

We implemented ParaFROST[2] with CUDA C++ v11. It is a hybrid CPU-GPU tool, with (sequential) solving done on the host side, and (parallel) VCE done on the device side. An interface with CBMC is implemented in C++. CBMC is patched to read a configuration file before ParaFROST is instantiated. This file contains all options supported by ParaFROST.

The Workflow. Figure 2 presents the general workflow of ParaFROST in the form of an activity diagram with host and device lanes. The diagram is focused on inprocessing; preprocessing works similarly on the device. First, the host performs a predetermined number of solving iterations. Once those have finished, and (un)satisfiability has not yet been proven, relevant clause data is copied to the global memory. To hide the latency of this operation as much as possible, clauses are copied asynchronously in batches. One batch is copied while the next is formatted for the GPU, as not all clause information on the host side is relevant for the device (see the next paragraph on data structures). On the device, signatures are computed for fast clause comparison, and the clauses are sorted for VCE (more on VCE later). Next, the device constructs a histogram, for fast lookup of clauses, and sorts the variables. The Thrust library is used

[2] The tool is available at https://gears.win.tue.nl/software/gpu4bmc.

for sorting.[3] After that, the host *schedules variables* for VCE, marking those variables in the global memory using unified memory. Next, the device applies VCE, marking clauses to be removed as `DELETED`. The host propagates units (literals in unit clauses are assigned **true**), which directly has an effect on the formula in the global memory. The VCE procedure is repeated until it has been performed a predetermined number of times. After each time, `DELETED` clauses are removed, and after the last iteration, this is done while the new clauses are copied to the host. Once this has been done, the overall procedure is repeated.

Data Structures and Memory Management. We have worked on making the storage of each clause in the GPU global memory as efficient as possible. However, we also wanted to annotate each clause with sufficient information for effective optimisations. In ParaFROST, the following information is stored for each clause:

- The `state` field (*2 bits*) stores if the state is `ORIGINAL`, `LEARNT` or `DELETED`.
- The `used` field (*2 bits*) keeps track of how many search iterations a `LEARNT` clause can still be used. `LEARNT` clauses are used at most twice [6].
- Two fields (*1 bit each*) are used for VCE bookmarking.
- The *literal block distance* (`lbd`) (*26 bits*) stores the number of decision levels contributing to a conflict, if there is one [3]. A maximum value of 2^{26} turns out to be sufficient. This field is updated when the clause is altered.
- The `size` (*32 bits*) of the clause, i.e., the number of literals.
- A signature `sig` (*32 bits*) is a clause hash, for fast clause comparison [15].

In addition, a list of literals is stored, each literal taking 32 bits (1 bit to indicate whether it is negated or not, and 31 bits to identify the variable). In total, a clause requires $12 + 4t$ bytes, with t the number of literals in the clause. For comparison, MiniSat only requires $4 + 4t$ bytes, but it does not involve the `used`, `lbd` and `sig` fields, thereby not supporting the associated optimisations. CaDiCaL [6] uses $28 + 4t$ bytes, since it applies solving and VCE on the same structures. In ParaFROST, the GPU is only used for VCE, in which information for *probing* [24] and *vivification* [36], for instance, is irrelevant. Finally, in [34], $20 + 4t$ bytes are used, storing the same information as ParaFROST.

To store a formula $\mathcal{S}$, a clause array is preallocated in the global memory, and filled with the clauses of $\mathcal{S}$. More space is allocated than the size of $\mathcal{S}$, to allow the addition of clauses that result from VCE. As the amount of allocated space is the limiting factor for the addition of new clauses, we have developed a memory-aware VCE mechanism, which we explain later in the current section.

Parallel VCE. ParaFROST supports the VCE rules *substitution* (i.e., gate equivalence reasoning), *resolution* (RES), *subsumption elimination* (SUB) and *eager redundancy elimination* (ERE) [15,20]. Substitution applies to patterns representing logical gates, and substitutes the involved variables with their gate definitions. ParaFROST supports ***`AND/OR`***, ***`Inverter`***, ***`If Then Else`*** and ***`XOR`***.

[3] https://docs.nvidia.com/cuda/thrust.

$$\begin{array}{rlll}
\text{RES:} & x \cup C_1, \bar{x} \cup C_2 & \Rightarrow C_1 \cup C_2 & (x \cup C_1 \notin \mathcal{L} \wedge \bar{x} \cup C_2 \notin \mathcal{L}) \\
\text{SUB1:} & x \cup C_1 \cup C_2, \bar{x} \cup C_2 & \Rightarrow C_1 \cup C_2, \bar{x} \cup C_2 & \\
\text{SUB2:} & C_1 \cup C_2, C_2 & \Rightarrow C_2 & (C_2 \in \mathcal{L} \implies \mathcal{L}' = \mathcal{L} \setminus \{C_2\}) \\
\text{ERE:} & x \cup C_1, \bar{x} \cup C_2, C_1 \cup C_2 & \Rightarrow x \cup C_1, \bar{x} \cup C_2 & (\{x \cup C_1, \bar{x} \cup C_2\} \cap \mathcal{L} \neq \emptyset \implies C_1 \cup C_2 \in \mathcal{L})
\end{array}$$

Fig. 3. VCE rules in ParaFROST. C_1 and C_2 are non-empty sets of literals.

In Fig. 3, we provide rewrite rules for SUB and RES. If clauses exist in $\mathcal{S}$ of the form expressed by the left hand side of a rule, then the rule is applicable, and the involved clauses are replaced by the clauses (called *resolvents*) on the right hand side. RES is applicable if there are two clauses of the form $x \cup C_1$ and $\bar{x} \cup C_2$, and applying it results in replacing those with a clause $C_1 \cup C_2$. SUB consists of two rules; the second is applied once the first is no longer applicable.

Conditions are given between parentheses. For RES, only `ORIGINAL` clauses are considered. Besides that, if $C_1 \cup C_2$ evaluates to **true**, it is actually not created. As `LEARNT` clauses are sometimes deleted during solving, SUB2 should only produce `ORIGINAL` clauses; if C_2 is `LEARNT` before applying the rule, it will become `ORIGINAL` ($\mathcal{L}'$ refers to the set of `LEARNT` clauses after application). For ERE, `LEARNT` clauses cannot cause the deletion of an `ORIGINAL` clause.

VCE is applied in parallel by ParaFROST by scheduling sets of *mutually-independent* variables for analysis. Two variables x and y are independent in $\mathcal{S}$ iff $\mathcal{S}$ does not contain a clause containing literals that refer to both variables, i.e., $\mathcal{S}_x \cup \mathcal{S}_{\bar{x}}$ and $\mathcal{S}_y \cup \mathcal{S}_{\bar{y}}$ are disjoint. This ensures that two threads focussing on x and y, respectively, does not lead to data races. In incremental solving, variables referred to by assumptions must be excluded from VCE. In each VCE iteration, a different set Ψ of variables is selected. This is achieved by using an upper-bound μ for the number of occurrences of a variable in $\mathcal{S}$. After each iteration, μ is increased, allowing the selection of more variables. ParaFROST supports configuring μ and the number of VCE iterations.

As already mentioned, clauses that can be removed are marked `DELETED` before they are removed. The removal of clauses is done once VCE has finished (see Fig. 2) to avoid data races. However, because of this, VCE may at first require more memory to store clauses. The clauses added during VCE must fit in the memory, otherwise the procedure fails. To ensure this, we have developed a memory-aware mechanism for VCE. Next, we explain this mechanism for the RES rule and substitution, as the application of those rules results in new clauses.

Algorithm 1 presents how RES and substitution are applied in ParaFROST. It requires $\mathcal{S}$, stored in a clause array `clauses`. As clauses are of varying sizes, we need an array `references` that provides a reference to each clause. In addition, arrays `varinfo`, `cindex` and `rindex` are given, which are filled in the first lines.

At line 1, the kernel VceScan is called in which a different thread is assigned to each variable $x \in \Psi$. Each thread checks the applicability of VCE rules on its variable and computes the number of clauses and literals that will be produced by the first applicable rule. A thread with ID i stores the type τ of the applicable rule (`NONE`, `RESOLVE`, or `SUBSTITUTE`) and the number of clauses β and

Algorithm 1: Parallel memory-aware application of RES and substitution

```
Input : global $\Psi$, clauses, references, varinfo, cindex, rindex
varinfo ← VceScan($\Psi, \mathcal{S}$)
cindex ← computeClauseIndices(varinfo, size(clauses))
rindex ← computeClauseRefIndices(varinfo, size(references))
VceApply($\Psi$, clauses, references, varinfo, cindex, rindex)
kernel VceApply($\Psi$, clauses, references, varinfo, cindex, rindex):
    for all $i \in [0, |\Psi|\rangle$ do in parallel
        register $cidx$ ← cindex[$i$], $ridx$ = rindex[$i$]
        register $\tau, \beta, \gamma$ ← varinfo[$i$]
        if $\tau = \mathit{RESOLVE} \wedge$ memorySafe($ridx, cidx, \beta, \gamma$) then
            ResApply(clauses, references, $x$, $ridx$, $cidx$)
        if $\tau = \mathit{SUBSTITUTE} \wedge$ memorySafe($ridx, cidx, \beta, \gamma$) then
            SubApply(clauses, references, $x$, $ridx$, $cidx$)
device function memorySafe($ridx$, $cidx$, $\beta$, $\gamma$):
    $reqSpace \leftarrow cidx + 12 \times \beta + (4 \cdot \gamma)$          // required number of bytes
    if $reqSpace >$ capacity(clauses) then return false
    $numRefs \leftarrow ridx + \beta$          // required number of clause references
    if $numRefs >$ capacity(references) then return false
    return true
```

literals γ produced by that rule in one integer at `varinfo`$[i]$. At lines 2–3, kernels computeClauseIndices and computeClauseRefIndices are called to add up the β's and γ's to obtain offsets into the arrays `references` and `clauses` (the method size(A) refers to the amount of data in array A). Both methods apply a parallel exclusive prefix sum [37], involving the β's and γ's. The result is that thread i, assigned to x, is instructed to start writing clause references at `references`[`rindex`$[i]$] and clauses at `clauses`[`cindex`$[i]$] when applying the next VCE rule for x. Whether the data actually fits is checked later.

Next, the kernel VceApply is called (lines 5–12). To each variable in Ψ, a thread is assigned. It retrieves the precomputed data (lines 7–8) and either applies the RES rule (lines 9–10), substitution (lines 11–12), or nothing, in case $\tau = \mathtt{NONE}$. However, a condition for applying a rule is that there is enough space, which is checked using the device function memorySafe (lines 13–18). The amount of allocated space for A is reflected by capacity(A), and memorySafe checks if there is enough space in `clauses`, starting at $cidx$ (lines 14–15). If there is, it is checked if the references can be stored in `references` (lines 16–17).

4 Multiple Decision Making in Incremental Solving

Given the fact that BMC SAT formulas often have many variables, a recently proposed extension of CDCL [30], in which periodically multiple decisions are made (MDM) at the same time, has much potential to speed up BMC. When the MDM method is called, it constructs a set $\mathcal{M} = \{\ell \in \mathbb{L} \mid \mathit{Var}(\{\ell\}) \cap \mathit{Var}(\Sigma) = \emptyset\}$ such that there does not exist a clause $C \in \mathcal{S}$ with $|\mathit{Var}(C) \setminus \mathit{Var}(\Sigma)| = 1$. In other words, the decisions $\mathcal{M}$ do not lead to logical follow-up assignments, i.e., implications. The reason for this restriction is that implications may lead to conflicts (clauses that cannot be satisfied). When a single decision is made, this decision needs to be rolled back when a conflict is caused, but when multiple

Algorithm 2: Decision making method DECIDE, with integrated MDM

```
   Input: $\Sigma$, $\mathbb{L}$, $decqueue$, $r$, $nConflicts$, $ConfFactor$, $prevMDsize$
1  $freevars \leftarrow Var(\mathbb{L}) \setminus Var(\Sigma)$
2  if $r > 0$ then
3      $\mathcal{M} \leftarrow$ MDM($freevars$, $decqueue$)
4      $r \leftarrow r - 1$, $prevMDsize \leftarrow |\mathcal{M}|$
5  else
6      $\mathcal{M} \leftarrow$ SINGLEDECISION($freevars$, $decqueue$)
7  if $r = 0 \wedge |freevars| \geq prevMDsize$ then
8      $r \leftarrow$ PERIODICFUSE($nConflicts$, $ConfFactor$)
9  return $\mathcal{M}$
10 function PERIODICFUSE($nConflicts$, $ConfFactor$):
11     if $nConflicts \geq ConfFactor$ then
12         UPDATEFACTOR($ConfFactor$)
13         return mdmrounds
14     else
15         return 0
```

decisions are made, detecting which decisions actually cause a conflict is more difficult. Note that MDM cannot always make multiple decisions; implications are needed to solve a formula, so single decisions still have to be made frequently.

In [30], MDM was integrated into MiniSat and Glucose, and since multiple decisions should be selected periodically, a mechanism was proposed that decides when to make multiple decisions based on the solver restart policy. However, since solvers can differ greatly in this policy, we wanted to create an alternative mechanism not depending on this. ParaFROST is based on CaDiCaL [6], which has a very different restart policy compared to MiniSat and Glucose.

Algorithm 2 presents ParaFROST's DECIDE method, which is called every time a decision must be made. Besides Σ and $\mathbb{L}$, it is given a queue *decqueue*, in which the variables are ordered based on a decision heuristic. In ParaFROST, the heuristics Variable State Independent Decaying Sum (VSIDS) [27] and Variable Move-To-Front (VMTF) [7] are alternatingly used. The latter was not used before in [30]. DECIDE also gets a variable r, initially set to the constant `mdmrounds`. These values are used to control the periodic call of MDM, in which a set of multiple decisions is made per round. Experiments have shown that `mdmrounds` = 3 is effective [30]. Finally, the number of conflicts so far (*nConflicts*), a variable *ConfFactor* used to switch MDM on and off, and a variable *prevMDsize*, storing the size of the most recent set of multiple decisions, are given.

To select new decisions, the set of unassigned variables is created at line 1. If we are calling MDM `mdmrounds` times (line 2), then MDM is called again and r is updated. The alternative is to make a single decision (line 6). If we have stopped calling MDM, and enough unassigned variables are present (line 7), method PERIODICFUSE is called, which either sets r back to `mdmrounds` or to 0, depending on *nConflicts* (lines 10–15). There are enough unassigned variables if there are more unassigned variables than variables in the most recent multiple decisions set. In PERIODICFUSE, *nConflicts* is compared to *ConfFactor*, which is initially set to a configurable value (default 2,000). *ConfFactor* is updated using a function UPDATEFACTOR. This makes *ConfFactor* grow linearly, to achieve a suitable balance between *ConfFactor* and *nConflicts* as the solving progresses.

5 Benchmarks

We conducted experiments with CBMC in combination with MiniSat (the default), Glucose, CaDiCaL, ParaFROST, ParaFROST with MDM, and a CPU-only version, referred to as ParaFROST (noGpu).[4] We used the AWS benchmarks in which the data structures `hash table`, `array list`, `array buff`, `linked list`, `priority queue`, `byte cursor` and `string` were analysed. The loop unwinding upper-bounds `8`, `16`, `64`, `128` and `1,000` were used, resulting in 168 different verification problems.

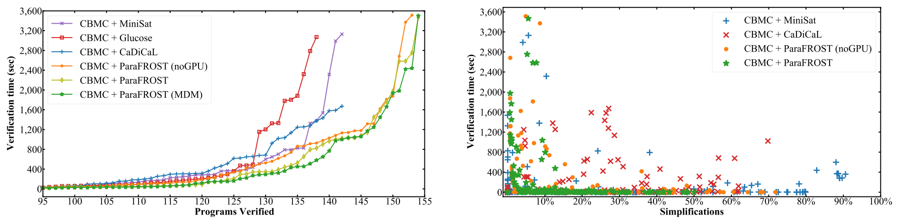
All experiments were executed on the DAS-5 cluster [4]. Each program was verified in isolation on a separate node, with a time-out of 3,600 s. Each node had an Intel Xeon E5-2630 CPU (2.4 GHz) with 64 GB of memory, and an NVIDIA RTX 2080 Ti, with 68 SMs (64 cores/SM) and 11 GB global memory.

Figure 4 presents the decision procedure runtime, and how much time was spent on VCE. ParaFROST outperforms all sequential solvers including CaDiCaL (plot 4a). Even though ParaFROST is based on CaDiCaL, its different data structures, simplification mechanism and parameters tuned for large formulas makes ParaFROST more effective in these experiments. MDM further improves ParaFROST. Plot 4b demonstrates that CBMC with MiniSat often spends most of the time on VCE. ParaFROST significantly reduces the time spent on VCE compared to other solvers.

In Table 1, the `Verified` column lists per solver the number of verified programs, and `PAR-2` gives the *penalized average runtime-2* metric. `PAR-2` score accumulates the running times of all solved instances with 2× the time-out of unsolved ones, divided by the total number of formulas. The solver with the lowest score is the winner. The triangles ▲ and ▼ mean significantly better and worse, respectively. The `MiniSat` column lists how many programs were verified faster with the other solvers compared to MiniSat. Between parentheses, it is given how many of those programs were not solved by MiniSat at all. The final four columns serve the same purpose for the other solvers. For example, ParaFROST-MDM verified 123 programs faster than CaDiCaL, in which 12 could not be verified by the latter. The last two rows provide a similar comparison. Clearly, ParaFROST-MDM verified the largest number of programs, with the lowest score.

Figure 5 presents the speedups of the ParaFROST configurations for the individual cases. Overall, SAT solving was accelerated effectively with ParaFROST and ParaFROST-MDM. Compared to ParaFROST (noGpu), ParaFROST (and ParaFROST-MDM), accelerated multiple instances by up to 18× (and 27×), and the geometric average speedup for all programs was 1.3× (and 1.6×).

[4] We also tried to use CBMC with Z3, but were not able to correctly configure this combination at the time of writing.



(a) Verification time (timeout: 3600 seconds)

(b) Percentage of verification time used for VCE

Fig. 4. CBMC runtimes for all solvers over the benchmark suite.

Table 1. CBMC performance analysis using the various solvers.

Configuration	Verified	PAR-2	MiniSat	Glucose	CaDiCaL	PFCPU	PFGPU
CBMC + MiniSat	143	1219	n/a	n/a	n/a	n/a	n/a
CBMC + Glucose	139	▼ 1388	▼ 49 (−4)	n/a	n/a	n/a	n/a
CBMC + CaDiCaL	143	1226	43	53 (+4)	n/a	n/a	n/a
CBMC + PFCPU	154	824	51 (+11)	62 (+15)	83 (+11)	n/a	n/a
CBMC + PFGPU	**155**	▲ **765**	▲ **66 (+12)**	▲ **83 (+16)**	▲ **96 (+12)**	**120 (+1)**	n/a
CBMC + PFGPU-MDM	**155**	▲ **743**	▲ **84 (+12)**	▲ **102 (+16)**	▲ **123 (+12)**	**133 (+1)**	**121**

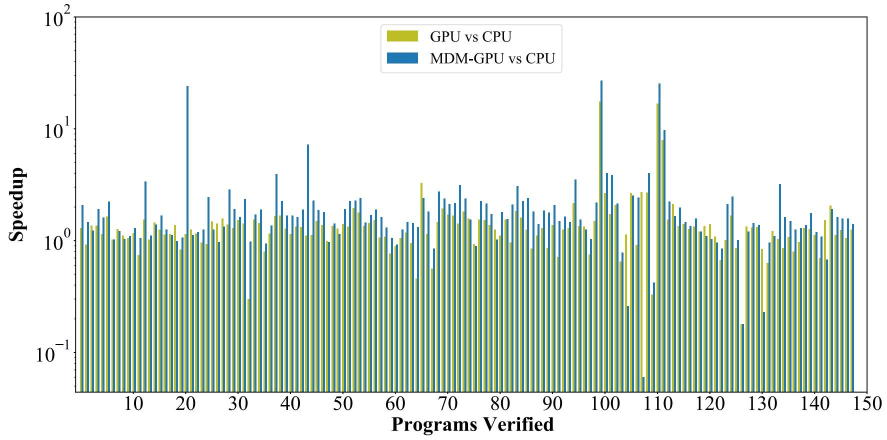


Fig. 5. Speedups of the individual cases.

6 Conclusion

We have presented ParaFROST, the first tool to accelerate BMC using GPUs. Given that BMC formulas tend to have much redundancy, ParaFROST effectively reduces solving times with GPU pre- and inprocessing, and by using MDM, which is particularly effective when many variables are present. In the future, we will combine our approach with (existing) multi-threaded BMC. We expect these techniques to strengthen each other.

References

1. Ábrahám, E., Schubert, T., Becker, B., Fränzle, M., Herde, C.: Parallel SAT solving in bounded model checking. J. Logic Comput. **21**(1), 5–21 (2009)
2. Amazon: The Amazon Web Services Core C99 Package Benchmark Set (2021). https://github.com/awslabs/aws-c-common/tree/main/verification/cbmc/proofs
3. Audemard, G., Simon, L.: Predicting Learnt Clauses Quality in Modern SAT Solvers. In: IJCAI, pp. 399–404. Morgan Kaufmann Publishers Inc. (2009)
4. Bal, H., et al.: A medium-scale distributed system for computer science research: infrastructure for the long term. IEEE Comput. **49**(5), 54–63 (2016)
5. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-49059-0_14
6. Biere, A., Fazekas, K., Fleury, M., Heisinger, M.: CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In: SAT Competition 2020, pp. 51–53 (2020)
7. Biere, A., Fröhlich, A.: Evaluating CDCL variable scoring schemes. In: Heule, M., Weaver, S. (eds.) SAT 2015. LNCS, vol. 9340, pp. 405–422. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24318-4_29
8. Bošnački, D., Edelkamp, S., Sulewski, D., Wijs, A.: GPU-PRISM: an extension of PRISM for general purpose graphics processing units. In: PDMC-HiBi, pp. 17–19. IEEE Computer Society (2010)
9. Bošnački, D., Odenbrett, M., Wijs, A., Ligtenberg, W., Hilbers, P.: Efficient reconstruction of biological networks via transitive reduction on general purpose graphics processors. BMC Bioinform. **13**(281) (2012)
10. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_7
11. Brown, C.E.: Reducing higher-order theorem proving to a sequence of SAT problems. J. Autom. Reason. **51**(1), 57–77 (2013)
12. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L.: Sequential circuit verification using symbolic model checking. In: DAC, pp. 46–51. IEEE (1990)
13. Chatterjee, P., Roy, S., Diep, B., Lal, A.: Distributed bounded model checking. In: FMCAD, pp. 47–56. TU Wien Academic Press (2020)
14. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_15
15. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 61–75. Springer, Heidelberg (2005). https://doi.org/10.1007/11499107_5
16. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electron. Notes Theor. Comput. Sci. **89**(4), 543–560 (2003)
17. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24605-3_37
18. Hamadi, Y., Jabbour, S., Sais, L.: ManySAT: a parallel SAT solver. J. Satisf. **6**, 245–262 (2009)
19. Inverso, O., Trubiani, C.: Parallel and distributed bounded model checking of multi-threaded programs. In: PPoPP, pp. 202–216. ACM (2020)

20. Järvisalo, M., Heule, M.J.H., Biere, A.: Inprocessing rules. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS (LNAI), vol. 7364, pp. 355–370. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31365-3_28
21. Kahsai, T., Tinelli, C.: PKind: a parallel k-induction based model checker. In: PDMC, EPTCS, vol. 72, pp. 55–62. Open Publishing Association (2011)
22. Kroening, D., Strichman, O.: Quantified formulas. In: Decision Procedures. TTC-SAES, pp. 199–227. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-50497-0_9
23. Leiserson, C.E., et al.: There's plenty of room at the top: what will drive computer performance after Moore's law? Science **368**(6495) (2020)
24. Lynce, I., Marques-Silva, J.: Probing-based preprocessing techniques for propositional satisfiability. In: ICTAI, pp. 105–110. IEEE (2003)
25. Marques-Silva, J., Glass, T.: Combinational equivalence checking using satisfiability and recursive learning. In: DATE, pp. 145–149. ACM, March 1999
26. Marques-Silva, J.P., Sakallah, K.A.: GRASP: a search algorithm for propositional satisfiability. IEEE Trans. Comput. **48**(5), 506–521 (1999)
27. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: DAC, pp. 530–535. ACM (2001)
28. NVIDIA: CUDA C Programming Guide (2020). https://docs.nvidia.com/cuda/cuda-c-programming-guide
29. Oostema, P., Martins, R., Heule, M.: Coloring unit-distance strips using SAT. In: LPAR23. EPiC Series in Computing, vol. 73, pp. 373–389. EasyChair (2020)
30. Osama, M., Wijs, A.: Multiple decision making in conflict-driven clause learning. In: ICTAI, pp. 161–169. IEEE (2020)
31. Osama, M., Gaber, L., Hussein, A.I., Mahmoud, H.: An efficient SAT-based test generation algorithm with GPU accelerator. J. Electron. Test. **34**(5), 511–527 (2018). https://doi.org/10.1007/s10836-018-5747-4
32. Osama, M., Wijs, A.: Parallel SAT simplification on GPU architectures. In: Vojnar, T., Zhang, L. (eds.) TACAS 2019. LNCS, vol. 11427, pp. 21–40. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17462-0_2
33. Osama, M., Wijs, A.: SIGmA: GPU accelerated simplification of SAT formulas. In: Ahrendt, W., Tapia Tarifa, S.L. (eds.) IFM 2019. LNCS, vol. 11918, pp. 514–522. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-34968-4_29
34. Osama, M., Wijs, A., Biere, A.: SAT solving with GPU accelerated inprocessing. In: TACAS 2021. LNCS, vol. 12651, pp. 133–151. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72016-2_8
35. Phan, Q.S., Malacaria, P., Pasareanu, C.: Concurrent bounded model checking. ACM SIGSOFT Softw. Eng. Notes **40**(1), 1–5 (2015)
36. Piette, C., Hamadi, Y., Saïs, L.: Vivifying propositional clausal formulae. In: ECAI, pp. 525–529. IOS Press, NLD (2008)
37. Sengupta, S., Harris, M., Garland, M., Owens, J.: Efficient parallel scan algorithms for manycore GPUs. In: SCMA, pp. 413–442. Taylor & Francis (2011)
38. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: Hunt, W.A., Johnson, S.D. (eds.) FMCAD 2000. LNCS, vol. 1954, pp. 127–144. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-40922-X_8
39. Shtrichman, O.: Pruning techniques for the SAT-based bounded model checking problem. In: Margaria, T., Melham, T. (eds.) CHARME 2001. LNCS, vol. 2144, pp. 58–70. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44798-9_4

40. Wijs, A.: BFS-based model checking of linear-time properties with an application on GPUs. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 472–493. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_26
41. Wijs, A., Neele, T., Bošnački, D.: GPUexplore 2.0: unleashing GPU explicit-state model checking. In: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (eds.) FM 2016. LNCS, vol. 9995, pp. 694–701. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48989-6_42
42. Youness, H., Ibraheim, A., Moness, M., Osama, M.: An efficient implementation of ant colony optimization on GPU for the satisfiability problem. In: PDP, pp. 230–235 (2015)

Pono: A Flexible and Extensible SMT-Based Model Checker

Makai Mann[1(✉)], Ahmed Irfan[1], Florian Lonsing[1], Yahan Yang[1,3], Hongce Zhang[2], Kristopher Brown[1], Aarti Gupta[2], and Clark Barrett[1]

[1] Stanford University, Stanford, USA
{makaim,irfan,lonsing,barrett}@cs.stanford.edu,
ksb@stanford.edu

[2] Princeton University, Princeton, USA
hongcez@princeton.edu, aartig@cs.princeton.edu

[3] University of Pennsylvania, Philadelphia, USA
yangy96@seas.upenn.edu

Abstract. Symbolic model checking is an important tool for finding bugs (or proving the absence of bugs) in modern system designs. Because of this, improving the ease of use, scalability, and performance of model checking tools and algorithms continues to be an important research direction. In service of this goal, we present Pono, an open-source SMT-based model checker. Pono is designed to be both a research platform for developing and improving model checking algorithms, as well as a performance-competitive tool that can be used for academic and industry verification applications. In addition to performance, Pono prioritizes transparency (developed as an open-source project on GitHub), flexibility (Pono can be adapted to a variety of tasks by exploiting its general SMT-based interface), and extensibility (it is easy to add new algorithms and new back-end solvers). In this paper, we describe the design of the tool with a focus on the flexible and extensible architecture, cover its current capabilities, and demonstrate that Pono is competitive with state-of-the-art tools.

1 Introduction

Model checking [39,61] is an influential verification capability in modern system design. Its greatest success has been with finite-state systems, where propositional methods such as binary decision diagrams (BDDs) [28] and Boolean satisfiability (SAT) solvers [69] are used as verification engines. At the same time, significant efforts have been made to lift model checking techniques from finite-state to infinite-state systems [24,30,31,35,46,63]. This requires more expressive verification engines, such as solvers for satisfiability modulo theories (SMT) [19]. Proponents of SMT-based techniques argue that such techniques can also benefit

"Pono" is the Hawaiian word for proper, correct, or goodness. Our goal is that Pono can be a useful tool for people to verify the correctness of systems.

A. Silva and K. R. M. Leino (Eds.): CAV 2021, LNCS 12760, pp. 461–474, 2021.
https://doi.org/10.1007/978-3-030-81688-9_22

finite-state systems, due to their ability to leverage word-level reasoning. Indeed, a word-level model checker won the most recent hardware model checking competition [22], giving credence to this claim. Despite these successes, there remain many directions for exploration in model checking. In this paper, we present `Pono`, an SMT-based model checking tool, with the goal of providing an open research platform for advancing these efforts.

`Pono` is designed with three use cases in mind: 1) *push-button verification*; 2) *expert verification*; and 3) *model checker development*. For 1, `Pono` provides competitive implementations of standard model checking algorithms. For 2, it exposes a flexible API, affording expert users fine-grained control over the tool. This can be useful in traditional model checking tasks (e.g., manually guiding the tool to an invariant, or adjusting the encoding for better performance), but it also enables the tool to be easily adapted for other tasks. In addition, `Pono` is designed using a completely generic SMT solver interface, making it trivial to experiment with different back-end solvers. For 3, `Pono` is open-source [7] and designed to be easily modifiable and extensible with a simple, modular, and hierarchical architecture. Taken together, these features make it relatively easy to do controlled experiments by comparing results obtained using `Pono`, while varying only the SMT solver or the model checking algorithm. `Pono` has already been used in a variety of research projects, both for model checking and other custom applications. It has also been used in two graduate level courses at Stanford University, where students used both the command-line interface and the API. With this promising start, we hope it will have a long and productive existence supporting research, education, and industry.

2 Design

`Pono` is designed around the manipulation and analysis of transition systems. A symbolic transition system is a tuple $\langle X, I, T \rangle$, where X is a set of (sorted) uninterpreted constants referred to as the current-state variables of the system and coupled with corresponding next-state variables X'; $I(X)$ is a formula constraining the initial states of the system; and $T(X, X')$ is a formula expressing the transition relation, which encodes the dynamics of the system. The transition system representation provides a clean and general interface, allowing `Pono` to target both hardware and software model checking. `Pono` is designed to fully leverage the expressivity and reasoning power of modern SMT solving. Its formulas use the language and semantics of the SMT-LIB standard [17], and its model checking algorithms use an SMT solving oracle. To streamline the interaction with SMT solvers, `Pono` uses `Smt-Switch` [59], an open-source C++ API for SMT solving. `Smt-Switch` provides a convenient, efficient, and generic interface for SMT solving. `Smt-Switch` supports a variety of SMT solver back-ends and can switch between them easily.

The diagram in Fig. 1 displays the overall architecture of `Pono`. The blocks with a dashed outline are globally available and used throughout the codebase. The `Pono` API provides access to all of the components shown, supporting the design goal of giving expert users control and flexibility.

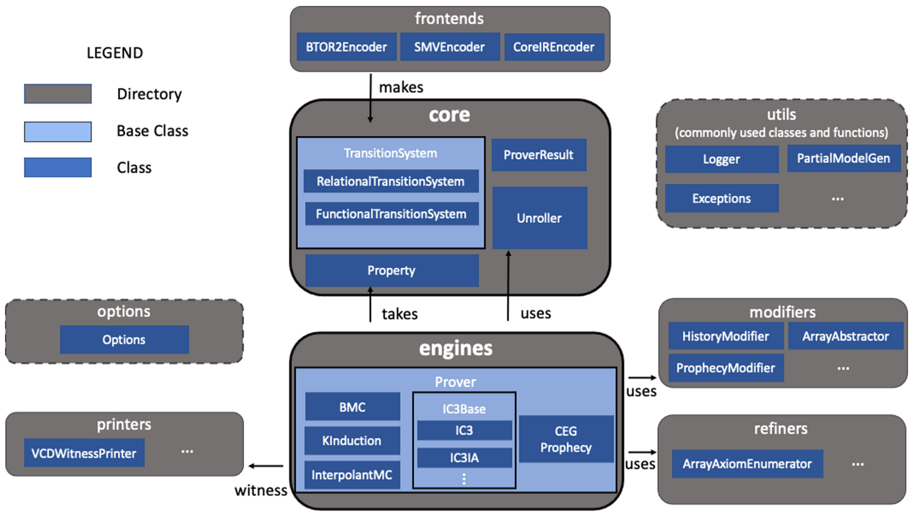


Fig. 1. Architecture diagram

Core. The TransitionSystem class in Pono represents symbolic transition systems as structured Smt-Switch terms. Key data structures include the following: i) inputvars: a vector of Smt-Switch symbolic constants representing primary inputs to the system (i.e., they are part of X, but their primed versions are not used and cannot appear in T); ii) statevars: a vector of Smt-Switch symbolic constants corresponding to the non-input state variables (the remaining variables in X); iii) next_map: a map from current (X) to next-state (X') variables; iv) init: an Smt-Switch formula representing $I(X)$; and v) trans: an Smt-Switch formula representing $T(X, X')$.

There are two kinds of transition systems: RelationalTransitionSystem and FunctionalTransitionSystem. The former has no restrictions on the form of the transition relation, while the latter is restricted to only functional updates: an equality (update assignment) with a next-state variable on the left and a function of current-state and input variables on the right. Some model checking algorithms take advantage of this structure [46,47]. Built-in checks ensure compliance with the restrictions.

A Property is an Smt-Switch formula representing a property to check for invariance.[1] A ProverResult is an enum which can be one of the following: i) UNKNOWN (result could not be determined, including incompleteness due to checking only up to some bound); ii) FALSE (the property does not hold); iii) TRUE (the property holds); and iv) ERROR (there was an internal error). The Unroller is a class for producing unrolled transition systems, i.e., encoding a finite-length symbolic execution by introducing fresh variables for each timestep.

[1] Pono currently supports invariant checking. Support for temporal properties is left to future work.

Engines. Model checking algorithms are implemented as subclasses of the abstract class `Prover` and stored in the `engines` directory. We cover the current suite of engines in more detail in Sect. 3.

Frontends. Although users can manually build transition systems through the API, it is also convenient to generate transition systems from structured input formats. `Pono` includes the following frontends: i) `BTOR2Encoder`: uses the open-source `btor2tools` [2] library to read the `BTOR2` [66] format for hardware model checking; ii) `SMVEncoder`: supports a subset of `nuXmv`'s [30] SMT-based theory extension of `SMV` [61], which added support for infinite-state systems; iii) `CoreIREncoder`: encodes the `CoreIR` [11] circuit intermediate representation. Note that Verilog [10] can be supported by using a translator from Verilog to either `BTOR2` or SMV. Examples of translators include Yosys [72] and Verilog2SMV [53], both of which are open-source.

Printers. `Pono` prints witness traces when a property does not hold. The supported formats are the `BTOR2` witness format and the `VCD` standard format used by EDA tools [10]. For theories such as arithmetic that are not supported by these formats, `Pono` implements simple extensions, ensuring that all variable assignments are included in witness traces.

Modifiers and Refiners. `Pono` includes functions that perform various transformations on transition systems, including: adding an auxiliary variable [14]; building an implicit predicate abstraction [70]; and computing a static cone-of-influence reduction for a functional transition system under a given property. It also includes functions for refining an abstract transition system.

Utils and Options. `utils` contains a collection of general-purpose classes and functions for manipulating and analyzing `Smt-Switch` terms and transition systems. `options` contains a single class, `PonoOptions`, for managing command-line options.

API. `Pono`'s native API is in C++. In addition, `Pono` has Python bindings that interact with the `Smt-Switch` Python bindings, both written in *Cython* [20]. These bindings behave very similarly to "pure" Python objects, allowing introspection and *pythonic* use of the API.

We follow best practices for modern C++ development and code quality maintenance, including issue tracking, code reviews, and continuous integration (via *GitHub Actions*). The build infrastructure is written in CMake [3] and is configurable. The `Pono` repository also provides helper scripts for installing its dependencies. We support `GoogleTest` [5] for unit testing and `gperftools` [12] for code profiling. Tests can be parameterized by both the SMT solver and the algorithm or type of transition system. We utilize *PyTest* [9] to manage and parameterize unit tests for the python bindings.

3 Capabilities

In this section, we highlight some key capabilities of `Pono`. The design makes use of abstract interfaces and inheritance to make it easy to add or extend functionality. Base class implementations of core functionality are provided but are kept simple to prioritize readability and transparency. And, of course, they can be overridden using inheritance and virtual functions.

We start by describing the interface and engines provided for push-button verification. Next, we take a closer look at two ways that the basic architecture can be extended. We then show how to use `Pono` to reason about a transition system using algebraic datatypes, demonstrating the expressive power provided by the SMT back-end.

Main Engines. All model checking algorithms in `Pono` are derived classes of the abstract base class `Prover`. The base class defines a simple public interface through a set of virtual functions:

- `initialize` initializes any objects and data structures the prover needs.
- `check_until` takes a non-negative integer parameter, k (the effort level), and calls the prover engine (the meaning of k is algorithm-dependent: in BMC [21] and k-induction [68], k is the unrolling length and in IC3-style [25] algorithms, it is the number of *frames*). The interface allows `check_until` to be called repeatedly with increasing values of k. An incremental algorithm can take advantage of this to reuse proof effort from previous calls. Engines that produce full proofs can do so as long as they do it within the provided effort level.
- `prove` attempts to prove a property without any limit on the bound.
- `witness` is called after a failed call to `prove` or `check_until`. It provides variable assignments for each step in a counterexample trace.
- `invar` is called after a successful full proof; it returns an inductive invariant that implies the property. The invariant is an `Smt-Switch` `Term` over current-state variables. Not all algorithms support this functionality.

`Pono` has several engines, all of which have been lifted to the SMT-level. We now list the main engines and include the corresponding lines of code (LoC) in the primary source file (the LoC includes all comments and license headers): 1. Bounded Model Checking [21] (88 LoC); 2. K-Induction [68] (161 LoC); 3. Interpolant-based Model Checking [62] (230 LoC); 4. IC3-style algorithms [25] (see below for LoC). The engines leverage the reusable infrastructure described in Sect. 2 (e.g., the `Unroller` for the unrolling based techniques).

IC3 Variants. IC3 is widely recognized as one of the best-performing algorithms for SAT-based model checking [43]. Liftings to SMT are an area of active research and have produced several variations with promising results [23,24,34,35,47,51,54,55,71]. To support this active research direction, `Pono` includes a special IC3 base class `IC3Base`, which implements a framework common to all variations of the algorithm.[2] The framework has several parameters that can be provided by specific instances of the algorithm: `IC3Formula` is a configurable data structure used to represent formulas constraining IC3 frames; `inductive_generalization` is the method used for inductive generalization; `predecessor_generalization`

[2] For details on how the IC3 algorithm works, we refer the reader to [25,43].

is the method used for predecessor generalization; and `abstract` and `refine` are methods that can be implemented for abstraction-refinement approaches to IC3 [35,47]. The implementation of `IC3Base` is 1086 lines of code. Current instantiations of `IC3Base` implemented in `Pono` include: i) IC3: a standard Boolean IC3 implementation [25,43] (152 LoC); ii) IC3Bits: a simple extension of IC3 to bit-vectors, which learns clauses over the individual bits (113 LoC); iii) Model-based IC3: a naive implementation of IC3 lifted to SMT, which learns clauses of equalities between variables and model values (397 LoC); iv) IC3IA: IC3 via Implicit Predicate Abstraction [35] (456 LoC); v) IC3SA: a basic implementation of IC3 with Syntax-Guided Abstraction for hardware verification [47] (984 LoC); vi) SyGuS-PDR: a syntax-guided synthesis approach for inductive generalization targeting hardware designs [73] (1047 LoC).

Counterexample-Guided Abstraction Refinement (CEGAR). CEGAR [57] is a popular framework for iteratively solving difficult model checking problems. It is typically parameterized by the underlying model checking algorithm, which operates on an abstract system that is iteratively refined as needed. `Pono` provides a generic `CEGAR` base class, parameterized by a model checking engine through a template argument. We describe two example uses of the CEGAR infrastructure implemented in `Pono`.

Operator Abstraction. This simple CEGAR algorithm uses uninterpreted functions (UF) to abstract potentially expensive theory operators (e.g. multiplication). The implementation is parameterized by the set of operators to replace with UFs. The refinement step analyzes a counterexample trace by restoring the concrete theory operator semantics. If the trace is found to be spurious, constraints are added to enforce the real semantics for the abstracted operators (e.g., equalities between certain abstract UFs and their theory operator counterparts), thus ruling out the spurious counterexample.

Counterexample-Guided Prophecy. This CEGAR approach replaces array variables with initially memoryless variables of uninterpreted sort and replaces the `select` and `store` array operators with UFs [58]. Due to the array theory semantics, it is not always possible to remove spurious counterexamples with quantifier-free refinement axioms over existing variables. However, instead of using potentially expensive quantifiers, the algorithm adds auxiliary variables (history and prophecy variables) [14], which can rule out spurious counterexamples of a given finite length. This approach has the effect of removing the need for array solving and can sometimes prove properties using prophecy variables that would otherwise require a universally quantified invariant.

Case Study with Algebraic Datatypes. To illustrate the flexibility of `Pono`'s SMT-based formalism, we next describe a case study with generalized algebraic theories (GATs) [29]. GATs are a rich formalism which can be used for high-level specifications of software or mathematical constructs. While the equality of two terms in a GAT is undecidable, one can ask the bounded question: "Does there exist a path of up to n rewrites to take a source term to a target term?"

To model this question, we use algebraic datatypes to represent dependently-typed abstract syntax trees (ASTs), paths through an AST (e.g., the 2nd argument of the 3rd argument of a term's 1st argument), and rewrite rules (e.g., $succ(n+1) = succ(m+1) \equiv succ(n) = succ(m)$). Smt-Switch supports algebraic datatypes through the CVC4 [18] back-end. A rewrite function is encoded as a transition relation. The decision of which rule to apply and at which subpath to apply it is controlled by input variables, and a state variable represents the current AST term (initially set to the source term). We check the property that the target term is not reachable from the source term. Consequently, any discovered counterexample is a valid rewrite sequence, serving as a proof of an equality that holds in the theory.

The workflow accepts a GAT input, produces an SMT encoding optimized for that particular theory, and then parses user-provided source and target terms into this theory before running bounded model checking. We used Pono to successfully find equalities in the theories of Boolean algebras, preorders, monoids, categories, and read-over-write arrays. This case study demonstrates Pono's ability to model and model check unconventional systems.

4 Related Work

Existing academic model checkers span a wide range of supported theories, modeling capabilities, and implemented algorithms. An important early model checker was SMV [61], which pioneered *symbolic* model checking of temporal logic properties [67] through BDDs [28]. NuSMV [32] and NuSMV2 [33] refined and extended the tool, followed by nuXmv [30] – a closed-source tool which added support for various SMT-based verification techniques using the SMT solver MathSAT5 [36]. Spin [52] is a well-known explicit-state model checker with extensive support for partial order reduction and other optimizations.

Several model checkers specifically target hardware verification. ABC [26] is a well-established, state-of-the-art bit-level hardware model checker based on SAT solving. CoSA [60] is an open-source model checker implemented in Python using the Python solver-agnostic SMT solving library, PySMT [45]. Although CoSA also relies on a generic API similar to Smt-Switch, the Python implementation introduces significant overhead, limiting its ability to include efficient procedures that must be implemented outside of the underlying SMT solver (e.g., CEGAR loops and some IC3 variants). AVR [48] is a state-of-the-art SMT-based hardware model checker supporting several standard model checking algorithms. It also implements a novel technique: IC3 via syntax-guided abstraction [47]. Importantly, AVR won the hardware model checking competition in 2020 [22], outperforming the previous state-of-the-art SAT-based model checker, ABC. AVR is currently closed-source, making it unsuitable for several of the use-cases targeted by our work, but a binary is available on GitHub [1].

There are several SMT-based model checkers focused on parameterized protocols. MCMT [46], the open-source extension Cubicle [49], and related systems [15, 16] perform backward-reachability analysis over infinite-state arrays.

Other open-source SMT-based model checkers include: i) `ic3ia` [13] – an example implementation of IC3IA built on MathSAT [36]; ii) `Kind2` [31] – a model checker for Lustre programs; iii) `Sally` [42] – a model checker for infinite-state systems that uses the SAL language [65] and MCMT, an extension of the SMT-LIB text format for declaring transition systems; iv) `Spacer` [56] – a Constrained Horn Clauses (CHC) solver built into the open-source Z3 [64] SMT solver, also based on an IC3-style algorithm; and v) `Intrepid` [27] – a model checker focusing primarily on the control engineering domain.

`Pono` is open-source, SMT-based, and implements a variety of model checking algorithms over transition systems. Furthermore, in contrast to the tools which focus on more limited domains, it has support for a wide set of SMT theories including fixed-width bit-vectors, arithmetic, arrays, and algebraic datatypes. To our knowledge all current open-source SMT-based model checkers tie the implementation directly to an existing SMT solver or use PySMT or the SMT-LIB text format to interact with arbitrary solvers. In contrast, `Pono` makes use of the C++ API of `Smt-Switch` to efficiently manipulate SMT terms and solvers in memory without a need for a textual interface. This allows `Pono` to provide both flexibility and performance. Finally, like the new model checker `Intrepid`, `Pono` provides an extensive API, which can be adapted and extended as needed. However, the focus is broader than `Intrepid` in terms of application domains.

5 Evaluation

In this section, we evaluate `Pono`[3] against current state-of-the-art model checkers across several domains. Our evaluation is not intended to be exhaustive. Rather, we highlight the breadth of `Pono` by selecting four sets of benchmarks in three diverse categories and a few reasonable competitors for each. The benchmarks are drawn from the following theories: i) unbounded quantifier-free arrays indexed by integers; ii) quantifier-free linear arithmetic over reals and integers; and iii) hardware verification over quantifier-free bit-vectors and (finite, bit-vector indexed) arrays. We ran all experiments on a 3.5 GHz Intel Xeon E5-2637 v4 CPU with a timeout of 1 h and a memory limit of 16 Gb. For all results, we also include the average runtime of solved instances in seconds. For portfolio solving, we ran each configuration in its own process with the full time and memory resources. In the first two categories, `Pono` used MathSAT5 [36] as the underlying SMT solver and interpolant [37,40,62] producer. For the hardware benchmarks, it used MathSAT5, Boolector [66], or both, depending on the configuration.

Arrays. We evaluate `Pono` on the integer-indexed array benchmark set of [44]. These are Constrained Horn Clauses (CHC) benchmarks inspired by software verification problems. Although there are no quantifiers in the benchmarks themselves, most cannot be proved safe without strengthening the property with quantified invariants. We compare against: i) `freqhorn` [44], a state-of-the-art CHC solver for this type of problem; ii) `prophic3` [8], a recent method that

[3] GitHub commit c175a302857ff00229a0919d5cc8fc3f78d04a26.

	`Pono`	`prophic3`	`prophic3-SA`	`freqhorn`	`nuXmv`
solved	**71** (16s)	**71** (20s)	66 (31s)	69 (6s)	4 (51s)

Fig. 2. Results on Freqhorn Array benchmarks (81 total), all expected to be safe.

result	SystemC (43 total)		Lustre (951 total)		
	`Pono`	`nuXmv`	`Pono`	`nuXmv`	`kind2`
safe	18 (673s)	**21** (571s)	**521** (10s)	516 (8s)	506 (2s)
unsafe	14 (325s)	**15** (479s)	412 (5s)	412 (1s)	409 (0.2s)
total	32 (521s)	**36** (533s)	**933** (8s)	928 (5s)	915 (1s)

Fig. 3. Results on arithmetic benchmarks.

outperforms `freqhorn` [58]; and iii) `nuXmv`, which does not support quantified invariants, to illustrate that most of these benchmarks do require them; `freqhorn` takes the CHC format natively, and we used scripts from the `ic3ia` and `nuXmv` distributions to translate the CHC input to SMV and the Verification Modulo Theories (VMT) format [38] – an annotated SMT-LIB file representing a transition system – for the other tools. We ran `Pono` with Counterexample-Guided Prophecy using IC3IA as the underlying model checking technique. We ran `prophic3` with both of the option sets used in their paper, and we ran the default configuration of `freqhorn`. Our results are shown in Fig. 2. We observe that `Pono` solves the same number of benchmarks as the reference implementation `prophic3` and is a bit faster.

Arithmetic. We next evaluate `Pono` on two sets of arithmetic benchmarks, both from the `nuXmv` distribution's example directory. The first uses linear real arithmetic, and the second uses linear integer arithmetic. Figure 3 displays the results on both benchmark sets.

Linear Real Arithmetic. We chose the `systemc QF_LRA` example benchmarks, because this is the largest set of linear real arithmetic benchmarks in the subset of SMV supported by `Pono`.[4] We ran both `nuXmv` and `Pono` with BMC and IC3IA in a portfolio. For both model checkers, BMC did not contribute any unique solves. We observe that `Pono` is quite competitive with `nuXmv` on `nuXmv`'s own benchmarks.

Linear Integer Arithmetic. We also evaluate `Pono` on a set of Lustre benchmarks which use quantifier-free linear integer arithmetic. We obtained the Lustre benchmarks from the `Kind` [50] website [6] and the SMV translation of the benchmarks from the distribution of `nuXmv`. We compare against both `nuXmv` and `Kind2` [31], the latest version of `Kind`. We ran all tools with a portfolio of techniques. For `Pono` and `nuXmv` we ran BMC and IC3IA. For `Kind2` we ran two configurations suggested by the authors: the default configuration with Z3 [64] and the default configuration, but with Yices2 [41] as the main SMT solver. Since the default

[4] `Pono` does not yet support enumeration types.

result	BV (324 total)				BV + Array (315 total)		
	Pono	AVR	CoSA2	sygus-apdr	Pono	AVR	CoSA2
safe	183 (283s)	**215** (115s)	98 (283s)	115 (545s)	252 (224s)	**274** (63s)	209 (299s)
unsafe	47 (314s)	47 (220s)	41 (232s)	15 (279s)	19 (208s)	19 (352s)	19 (204s)
total	230 (289s)	**262** (134s)	139 (268s)	130 (514s)	271 (223s)	**293** (82s)	228 (291s)

Fig. 4. Results on HWMCC2020 benchmarks.

configurations of Kind2 run 8 techniques in parallel, we gave each configuration 8 cores. Additionally, we ran Kind2's BMC and IC3 implementations using MathSAT5 as the SMT solver, because this is closest to the other model checkers' configurations. The default with Z3 was the best configuration of Kind2. We observe that Pono solves the most benchmarks overall. Once again, BMC contributed no unique solves for any model checker.

Hardware Verification. Finally, we evaluate Pono on the 2020 Hardware Model Checking Competition (HWMCC) benchmarks. The benchmarks are split into bitvector-only and bitvector plus array categories. We evaluate against AVR [1,48] and CoSA2 [4] (a previous name and version of Pono), the winners of HWMCC 2020 and HWMCC 2019, respectively. We also compare against sygus-apdr (the reference implementation of SyGuS-PDR [73]) on the bitvector benchmarks (as sygus-apdr targets bitvectors). We ran all 16 configurations of AVR from their HWMCC 2020 entry: several configurations of BMC and k-induction, and 11 configurations of IC3SA. We ran the 4 configurations of CoSA2 from the HWMCC 2019 entry: two BMC configurations, k-induction, and interpolant-based model checking. We ran sygus-apdr with 4 different parameters controlling the grammar for lemmas. For the bitvector-only benchmarks, we ran Pono with 10 configurations: 3 configurations of IC3IA, 2 configurations of IC3SA, 2 configurations of SyGuS-PDR, IC3Bits, k-induction, and BMC. For the array benchmarks, we ran 5 configurations: 3 configurations of IC3IA (one with Counterexample-Guided Prophecy), k-induction, and BMC. We show our results on the HWMCC 2020 benchmarks in Fig. 4. AVR wins in both categories, although Pono is fairly competitive, outperforming the other tools.

These results show that Pono is well on its way to being both widely applicable and performance-competitive. The arithmetic experiments demonstrate the capabilities of its IC3IA engine, but other engines have some room for improvement. In particular, both IC3SA and SyGuS-PDR were recently added to Pono, and its implementation of these algorithms still lags the corresponding implementations in AVR and sygus-apdr, respectively. There are also some features that are known to help performance and are not yet implemented in Pono. For example, the best configurations of AVR use UF data abstraction. This differs from our UF operator abstraction in that it replaces all abstracted data with uninterpreted sorts and learns targeted data refinement axioms.

6 Conclusion

We have presented Pono: a new open-source, SMT-based, and solver-agnostic model checker. We described its capabilities, design, and the emphasis on flexibility and extensibility in addition to performance. We demonstrated empirically that the suite of model checking algorithms is competitive with state-of-the-art tools. Pono has already been used in several research projects and two graduate-level classes. With this promising start, we believe that Pono is poised to have an enduring and beneficial impact on research, education, and model checking applications. Future work includes adding support for temporal properties [67] and improving and adding to Pono's engines, in particular the IC3 variants.

Acknowledgements. This work was partially supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE-1656518. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. This work was also supported by the Defense Advanced Research Projects Agency, grants FA8650-18-1-7818 and FA8650-18-2-7854. We thank these sponsors and our industry collaborators for their support.

References

1. AVR distribution. https://github.com/aman-goel/avr
2. btor2tools. https://github.com/Boolector/btor2tools
3. CMake. https://cmake.org
4. cosa2. https://github.com/upscale-project/cosa2
5. GoogleTest. https://github.com/google/googletest
6. Kind site. http://clc.cs.uiowa.edu/Kind/index.php?page=experimental-results
7. Pono. https://github.com/upscale-project/pono
8. ProphIC3 (commit: 497e2fbfb813bcf0a2c3bcb5b55ad47b2a678611). https://github.com/makaimann/prophic3
9. pytest 5.4.2. https://github.com/pytest-dev/pytest
10. IEEE Std 1364–2005, pp. 1–590 (2006)
11. CoreIR (2017). https://github.com/rdaly525/coreir
12. Google Perftools (2017). https://github.com/gperftools/gperftools
13. ic3ia. https://es-static.fbk.eu/people/griggio/ic3ia/index.html. Accessed 2020
14. Abadi, M., Lamport, L.: The existence of refinement mappings. In: Proceedings of LICS, pp. 165–175, July 1988
15. Alberti, F., Bruttomesso, R., et al.: SAFARI: SMT-based abstraction for arrays with interpolants. In: Proceedings of CAV, pp. 679–685 (2012)
16. Alberti, F., Ghilardi, S., Sharygina, N.: Booster: an acceleration-based verification framework for array programs. In: Proceedings of ATVA, pp. 18–23 (2014)
17. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB) (2016). www.smt-lib.org
18. Barrett, C.W., et al.: CVC4. In: Proceedings of CAV, pp. 171–177 (2011)

19. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Satisfiability, pp. 825–885 (2009)
20. Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D.S., Smith, K.: Cython: the best of both worlds. Comput. Sci. Eng. **2**, 31–39 (2011)
21. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Proceedings of TACAS, pp. 193–207 (1999)
22. Biere, A., Froleyks, N., Preiner, M.: Hardware model checking competition (2020). http://fmv.jku.at/hwmcc20/
23. Birgmeier, J., Bradley, A., Weissenbacher, G.: Counterexample to induction-guided abstraction-refinement (CTIGAR). In: Proceedings of CAV, pp. 831–848 (2014)
24. Bjørner, N., Gurfinkel, A.: Property directed polyhedral abstraction. In: Proceedings of VMCAI, pp. 263–281 (2015)
25. Bradley, A.: SAT-based model checking without unrolling. In: Proceedings of VMCAI, pp. 70–87 (2011)
26. Brayton, R., Mishchenko, A.: ABC: An academic industrial-strength verification tool. In: Proceedings of CAV, pp. 24–40 (2010)
27. Bruttomesso, R.: Intrepid: An SMT-based model checker for control engineering and industrial automation. In: SMT Workshop, August 2019
28. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. Comput. **8**, 677–691 (1986)
29. Cartmell, J.: Generalised algebraic theories and contextual categories. Ann. Pure Appl. Logic 209–243 (1986)
30. Cavada, R., Cimatti, A., et al.: The nuXmv symbolic model checker. In: Proceedings of CAV, pp. 334–342 (2014)
31. Champion, A., Mebsout, A., Sticksel, C., Tinelli, C.: The Kind 2 model checker. In: Proceedings of CAV, pp. 510–517 (2016)
32. Cimatti, A., Clarke, E.M., Giunchiglia, F., Roveri, M.: NUSMV: A new symbolic model verifier. In: Proceedings of CAV, pp. 495–499 (1999)
33. Cimatti, A., Clarke, E.M., et al.: NuSMV 2: an opensource tool for symbolic model checking. In: Proceedings of CAV, pp. 359–364 (2002)
34. Cimatti, A., Griggio, A., Irfan, A., et al.: Incremental linearization for satisfiability and verification modulo nonlinear arithmetic and transcendental functions. ACM Trans. Comput. Log. 19:1–19:52 (2018)
35. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: Infinite-state invariant checking with IC3 and predicate abstraction. FMSD **3**, 190–218 (2016)
36. Cimatti, A., Griggio, A., Schaafsma, B., Sebastiani, R.: The MathSAT5 SMT Solver. In: Piterman, N., Smolka, S. (eds.) Proceedings of TACAS (2013)
37. Cimatti, A., Griggio, A., Sebastiani, R.: Efficient generation of Craig interpolants in satisfiability modulo theories. ACM Trans. Comput. Log. (1), 7:1–7:54 (2010)
38. Cimatti, A., et al.: Verification Modulo Theories (2011). http://www.vmt-lib.org
39. Clarke, E., Henzinger, T., et al.: Handbook of Model Checking (2018)
40. Craig, W.: Linear reasoning. A new form of the Herbrand-Gentzen theorem. J. Symb. Log. (3), 250–268 (1957)
41. Dutertre, B.: Yices 2.2. In: Proceedings of CAV, pp. 737–744 (2014)
42. Dutertre, B., Jovanovic, D., Navas, J.A.: Verification of fault-tolerant protocols with Sally. In: Proceedings of NFM, pp. 113–120 (2018)
43. Eén, N., Mishchenko, A., Brayton, R.K.: Efficient implementation of property directed reachability. In: Proceedings of FMCAD, pp. 125–134 (2011)
44. Fedyukovich, G., Prabhu, S., Madhukar, K., Gupta, A.: Quantified invariants via syntax-guided synthesis. In: Proceedings of CAV, pp. 259–277 (2019)

45. Gario, M., Micheli, A.: PySMT: A solver-agnostic library for fast prototyping of SMT-based algorithms. In: Proceedings of SMT Workshop, pp. 373–384 (2015)
46. Ghilardi, S., Ranise, S.: MCMT: a model checker modulo theories. In: Automated Reasoning, pp. 22–29 (2010)
47. Goel, A., Sakallah, K.A.: Model checking of Verilog RTL using IC3 with syntax-guided abstraction. In: Proceedings of NFM, pp. 166–185 (2019)
48. Goel, A., Sakallah, K.A.: AVR: abstractly verifying reachability. In: Proceedings of TACAS, pp. 413–422 (2020)
49. Goel, A., Krstic, S., Leslie, R., Tuttle, M.R.: SMT-based system verification with DVF. In: Proceedings of SMT Workshop, pp. 32–43 (2012)
50. Hagen, G., Tinelli, C.: Scaling up the formal verification of Lustre programs with SMT-based techniques. In: Proceedings of FMCAD, pp. 1–9 (2008)
51. Ho, Y., Mishchenko, A., Brayton, R.K.: Property directed reachability with word-level abstraction. In: Proceedings of FMCAD, pp. 132–139 (2017)
52. Holzmann, G.J.: The SPIN Model Checker - primer and reference manual (2004)
53. Irfan, A., Cimatti, A., Griggio, A., Roveri, M., Sebastiani, R.: Verilog2SMV: a tool for word-level verification. In: Proceedings of DATE, pp. 1156–1159 (2016)
54. Jovanovic, D., Dutertre, B.: Property-directed k-induction. In: Proceedings of FMCAD, pp. 85–92 (2016)
55. K., H.G.V., Fedyukovich, G., Gurfinkel, A.: Word level property directed reachability. In: Proceedings of ICCAD, pp. 107:1–107:9 (2020)
56. Komuravelli, A., Gurfinkel, A., et al.: Automatic abstraction in SMT-based unbounded software model checking. In: Proceedings of CAV, pp. 846–862 (2013)
57. Kroening, D., Groce, A., Clarke, E.M.: Counterexample guided abstraction refinement via program execution. In: Proceedings of ICFEM, pp. 224–238 (2004)
58. Mann, M., Irfan, A., et al.: Counterexample-guided prophecy for model checking modulo the theory of arrays. In: Proceedings of TACAS, pp. 113–132 (2021)
59. Mann, M., Wilson, A., et al.: SMT-Switch: a Solver-agnostic C++ API for SMT Solving. In: Proceedings of SAT (2021)
60. Mattarei, C., Mann, M., Barrett, C., et al.: CoSA: Integrated verification for agile hardware design. In: Proceedings of FMCAD, pp. 1–5 (2018)
61. McMillan, K.: Symbolic model checking - an approach to the state explosion problem. Ph.D. thesis, Carnegie Mellon University (1992)
62. McMillan, K.L.: Interpolants and symbolic model checking. In: Proceedings of VMCAI, pp. 89–90 (2007)
63. McMillan, K.L., Padon, O.: Ivy: a multi-modal verification tool for distributed algorithms. In: Proceedings of CAV, pp. 190–202 (2020)
64. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Proceedings of TACAS, pp. 337–340 (2008)
65. de Moura, L., et al.: Sal 2. In: Proceedings of CAV, pp. 496–500 (2004)
66. Niemetz, A., Preiner, M., Wolf, C., Biere, A.: Btor2, BtorMC and Boolector 3.0. In: Proceedings of CAV, pp. 587–595 (2018)
67. Pnueli, A.: The temporal logic of programs. In: Proceedings of FOCS, pp. 46–57 (1977)
68. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: Proceedings of FMCAD, pp. 108–125 (2000)
69. Silva, J.P.M., Lynce, I., Malik, S.: Conflict-driven clause learning SAT solvers. In: Handbook of Satisfiability, pp. 131–153 (2009)
70. Tonetta, S.: Abstract model checking without computing the abstraction. In: Proceedings of FM, pp. 89–105 (2009)

71. Welp, T., Kuehlmann, A.: QF BV model checking with property directed reachability. In: Proceedings of DATE, pp. 791–796 (2013)
72. Wolf, C., Glaser, J., Kepler, J.: Yosys-a free Verilog synthesis suite. In: Proceedings of Austrochip Workshop (2013)
73. Zhang, H., Gupta, A., Malik, S.: Syntax-guided synthesis for lemma generation in hardware model checking. In: Proceedings of VMCAI (2021)

Logical Foundations

Towards a Trustworthy Semantics-Based Language Framework via Proof Generation

Xiaohong Chen[1](✉), Zhengyao Lin[1], Minh-Thai Trinh[2], and Grigore Roşu[1]

[1] University of Illinois at Urbana-Champaign, Champaign, USA
{xc3,zl38,grosu}@illinois.edu
[2] Advanced Digital Sciences Center, Illinois at Singapore, Singapore, Singapore
trinhmt@illinois.edu

Abstract. We pursue the vision of an *ideal language framework*, where programming language designers only need to define the formal *syntax* and *semantics* of their languages, and all language tools are automatically generated by the framework. Due to the complexity of such a language framework, it is a big challenge to ensure its trustworthiness and to establish the correctness of the autogenerated language tools. In this paper, we propose an innovative approach based on *proof generation*. The key idea is to generate proof objects as correctness certificates for each individual task that the language tools conduct, on a case-by-case basis, and use a trustworthy proof checker to check the proof objects. This way, we avoid formally verifying the entire framework, which is practically impossible, and thus can make the language framework both *practical* and *trustworthy*. As a first step, we formalize program execution as mathematical proofs and generate their complete proof objects. The experimental result shows that the performance of our proof object generation and proof checking is very promising.

Keywords: Semantic framework · Proof generation · Proof checking

1 Introduction

Unlike natural languages that allow vagueness and ambiguity, programming languages must be precise and unambiguous. Only with rigorous definitions of programming languages, called the *formal semantics*, can we guarantee the reliability, safety, and security of computing systems.

Our vision is thus an *ideal language framework* based on the formal semantics of programming languages. Shown in Fig. 1, an ideal language framework is one where language designers only need to define the formal syntax and semantics of their language, and all language tools are automatically generated by the framework. The *correctness* of these language tools is established by generating complete mathematical proofs as certificates that can be automatically machine-checked by a trustworthy proof checker.

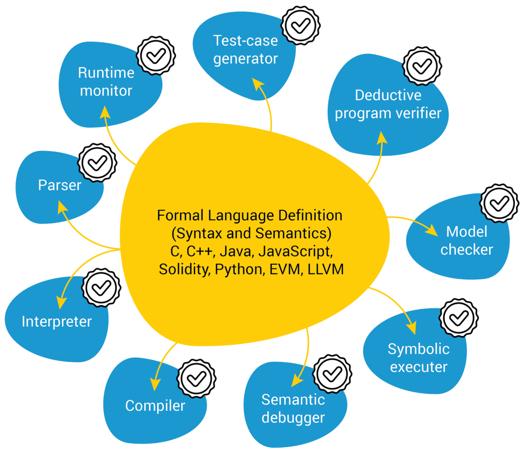

Runtime monitor
Test-case generator
Deductive program verifier
Parser
Formal Language Definition (Syntax and Semantics) C, C++, Java, JavaScript, Solidity, Python, EVM, LLVM
Model checker
Interpreter
Symbolic executer
Compiler
Semantic debugger

Fig. 1. An ideal language framework vision; language tools are autogenerated, with machine-checkable mathematical proofs as correctness certificates.

The $\mathbb{K}$ language framework (https://kframework.org) is in pursuit of the above ideal vision. It provides a simple and intuitive front end language (i.e., a meta-language) for language designers to define the formal syntax and semantics of other programming languages. From such a formal language definition, the framework automatically generates a set of language tools, including a parser, an interpreter, a deductive verifier, a program equivalence checker, among many others [9,24]. $\mathbb{K}$ has obtained much success in practice, and has been used to define the complete executable formal semantics of many real-world languages, such as C [12], Java [2], JavaScript [21], Python [13], Ethereum virtual machines byte code [15], and x86-64 [10], from which their implementations and formal analysis tools are automatically generated. Some commercial products [14,18] are powered by these autogenerated implementations and/or tools.

What is *missing* in $\mathbb{K}$ (compared to the ideal vision in Fig. 1) is its ability to generate proof objects as correctness certificates. The current $\mathbb{K}$ implementation is a complex artifact with over 500,000 lines of code written in 4 programming languages, with new code committed on a weekly basis. Its code base includes complex data structures, algorithms, optimizations, and heuristics to support the various features such as defining formal language syntax using BNF grammar, defining computation configurations as constructor terms, defining formal semantics using rewrite rules, specifying arbitrary evaluation strategies, and defining the binding behaviors of binders (Sect. 3). The large code base and rich features make it challenging to formally verify the correctness of $\mathbb{K}$.

Our **main contribution** is the proposal of a *practical approach* to establishing the correctness of a complex language framework, such as $\mathbb{K}$, via *proof object generation*. Our approach consists of the following main components:

1. A small *logical foundation* of $\mathbb{K}$;
2. *Proof parameters* that are provided by $\mathbb{K}$ as the hints for proof generation;

3. A *proof object generator* that generates *proof objects* from proof parameters;
4. A fast and trustworthy third-party *proof checker* that verifies proof objects.

The key idea that makes our approach practical is that we establish the correctness not for the entire framework, but for each individual language tasks that it conducts, on a case-by-case basis. This idea is not limited to $\mathbb{K}$ but also applicable to the existing language frameworks and/or formal semantics approaches.

As a first step, we formalize *program execution* as mathematical proofs and generate their complete proof objects. The experimental result (Table 1) shows promising performance of the proof object generation and proof checking. For example, for a 100-step program execution trace, its complete proof object has 1.6 million lines of code that takes only 5.6 s to proof-check.

We organize the rest of the paper as follows. We give an overview of our approach in Sect. 2. We introduce $\mathbb{K}$ and discuss the generation of proof parameters in Sect. 3. We discuss *matching logic*—the logical foundation of $\mathbb{K}$—in Sect. 4. We then compile $\mathbb{K}$ to matching logic in Sect. 5, and discuss *proof object generation* in Sect. 6. We discuss the limitations of our current implementation and show the experiment results in Sects. 7 and 8, respectively. Finally, we discuss related work in Sect. 9 and conclude the paper in Sect. 10.

2 Our Approach Overview

We give an overview of our approach via the following four main components: (1) a logical foundation of $\mathbb{K}$, (2) proof parameters, (3) proof object generation, and (4) a trustworthy proof checker.

Logical Foundation of $\mathbb{K}$. Our approach is based on *matching logic* [5,22]. Matching logic is the *logical foundation* of $\mathbb{K}$, in the following sense:

1. The $\mathbb{K}$ definition (i.e., the language definition in Fig. 1) of a programming language L corresponds to a *matching logic theory* Γ^L, which, roughly speaking, consists of a set of logical symbols that represents the formal syntax of L, and a set of logical axioms that specify the formal semantics.
2. All language tools in Fig. 1 and all language tasks that $\mathbb{K}$ conducts are formally specified by matching logic formulas. For example, *program execution* is specified (in our approach) by the following matching logic formula:

$$\varphi_{init} \Rightarrow \varphi_{final} \tag{1}$$

   where φ_{init} is the formula that specifies the initial state of the execution, φ_{final} specifies the final state, and "$\Rightarrow$" states the rewriting/reachability relation between states (see Sect. 5.1).
3. There exists a matching logic *proof system* that defines the provability relation $\vdash$ between theories and formulas. For example, the correctness of the above execution from φ_{init} to φ_{final} is witnessed by the formal proof:

$$\Gamma^L \vdash \varphi_{init} \Rightarrow \varphi_{final} \tag{2}$$

Therefore, matching logic is the logical foundation of $\mathbb{K}$. The *correctness* of $\mathbb{K}$ conducting one language task is reduced to the *existence of a formal proof* in matching logic. Such formal proofs are encoded as proof objects, discussed below.

Proof Parameters. A proof parameter is the necessary information that $\mathbb{K}$ should provide to help generate proof objects. For program execution, such as Eq. (2), the proof parameter includes the following information:

- the complete execution trace $\varphi_0, \varphi_1, \ldots, \varphi_n$, where $\varphi_0 \equiv \varphi_{init}$ and $\varphi_n \equiv \varphi_{final}$; we call $\varphi_0, \ldots, \varphi_n$ the intermediate *snapshots* of the execution;
- for each step from φ_i to φ_{i+1}, the *rewriting information* that consists of the rewrite/semantic rule $\varphi_{lhs} \Rightarrow \varphi_{rhs}$ that is applied, and the corresponding substitution θ such that $\varphi_{lhs}\theta \equiv \varphi_i$.

In other words, a proof parameter of a program execution trace contains the complete information about how such an execution is carried out by $\mathbb{K}$. The proof parameter, once generated by $\mathbb{K}$, is passed to the proof object generator to generate the corresponding proof object, discussed below.

Proof Object Generation. In our approach, a proof object is an encoding of matching logic formal proofs, such as Eq. (2). Proof objects are generated by a proof object generator from the proof parameters provided by $\mathbb{K}$. At a high level, a proof object for program execution, such as Eq. (2), consists of:

1. the formalization of matching logic and its provability relation $\vdash$;
2. the formalization of the formal semantics Γ^L as a logical theory, which includes axioms that specify the rewrite/semantic rules $\varphi_{lhs} \Rightarrow \varphi_{rhs}$;
3. the formal proofs of all one-step executions, i.e., $\Gamma^L \vdash \varphi_i \Rightarrow \varphi_{i+1}$ for all i;
4. the formal proof of the final proof goal $\Gamma^L \vdash \varphi_{init} \Rightarrow \varphi_{final}$.

Our proof objects have a *linear structure*, which implies a nice separation of concerns. Indeed, Item 1 is only about matching logic and is *not specific* to any programming languages/language tasks, so we only need to develop and proof-check it *once and for all.* Item 2 is specific to the language semantics Γ^L but is independent of the actual program executions, so it can be reused in the proof objects of various language executions for the same programming language L.

A Trustworthy Proof Checker. A proof checker is a small program that checks whether the formal proofs encoded in a proof object are correct. The proof checker is the main trust base of our work. In this paper, we use Metamath [20]—a third-party proof checking tool that is simple, fast, and trustworthy—to formalize matching logic and encode its formal proofs.

```
module IMP-SYNTAX
  imports DOMAINS-SYNTAX
  syntax Exp ::=
      Int
    | Id
    | Exp "+" Exp    [left, strict]
    | Exp "-" Exp    [left, strict]
    | "(" Exp ")"    [bracket]
  syntax Stmt ::=
      Id "=" Exp ";" [strict(2)]
    | "if" "(" Exp ")"
        Stmt Stmt    [strict(1)]
    | "while" "(" Exp ")" Stmt
    | "{" Stmt "}"   [bracket]
    | "{" "}"
    > Stmt Stmt      [left, strict(1)]
  syntax Pgm ::= "int" Ids ";" Stmt
  syntax Ids ::= List{Id,","}
endmodule
```

```
module IMP imports IMP-SYNTAX
  imports DOMAINS
  syntax KResult ::= Int
  configuration
   <T> <k> $PGM:Pgm </k>
       <state> .Map </state> </T>
  rule <k> X:Id => I ...</k>
       <state>... X |-> I ...</state>
  rule I1 + I2 => I1 +Int I2
  rule I1 - I2 => I1 -Int I2
  rule <k> X = I:Int => I ...</k>
       <state>... X |-> (_ => I) ...</state>
  rule {} S:Stmt => S
  rule if(I) S _ => S requires I =/=Int 0
  rule if(0) _ S => S
  rule while(B) S => if(B) {S while(B) S} {}
  rule <k> int (X, Xs => Xs) ; S </k>
       <state>... (. => X |-> 0) </state>
  rule int .Ids ; S => S
endmodule
```

Fig. 2. The complete $\mathbb{K}$ formal definition of an imperative language IMP.

Summary. Our approach to establishing the correctness of $\mathbb{K}$ is based on its logical foundation—matching logic. We formalize language semantics as logical theories, and program executions as formulas and proof goals, whose proof objects are automatically generated and proof-checked. Our proof objects have a linear structure that allows easy reuse of their components. The key characteristics of our logical-based approach are the following:

– It is *faithful* to the real $\mathbb{K}$ implementation because proof objects are generated from proof parameters, which include all execution snapshots and the actual rewriting information, provided by $\mathbb{K}$.
– It is *practical* because proof objects are generated for each program executions on a case-by-case bases, avoiding the verification of the entire $\mathbb{K}$.
– It is *trustworthy* because the autogenerated proof objects are checked using the trustworthy third-party Metamath proof checker.

3 $\mathbb{K}$ Framework and Generation of Proof Parameters

3.1 $\mathbb{K}$ Overview

$\mathbb{K}$ is an effort in realizing the ideal language framework vision in Fig. 1. An easy way to understand $\mathbb{K}$ is to look at it as a meta-language that can define other programming languages. In Fig. 2, we show an example $\mathbb{K}$ language definition of an imperative language IMP. In the 39-line definition, we *completely* define the formal syntax and the (executable) formal semantics of IMP, using a front end language that is easy to understand. From this language definition, $\mathbb{K}$ can generate all language tools for IMP, including its parser, interpreter, verifier, etc.

We use IMP as an example to illustrate the main $\mathbb{K}$ features. There are two *modules*: `IMP-SYNTAX` defines the syntax and `IMP` defines the semantics using rewrite rules. Syntax is defined as BNF grammars. The keyword `syntax` leads production rules that can have attributes that specify the additional syntactic and/or semantic information. For example, the syntax of `if`-statements is defined in lines 11–12 and has the attribute `[strict(1)]`, meaning that the evaluation order is strict in the first argument, i.e., the condition of an `if`-statement.

In the module `IMP`, we define the *configurations* of IMP and its formal semantics. A configuration (lines 23–25) is a constructor term that has all semantic information needed to execute programs. IMP configurations are simple, consisting of the IMP code and a program state that maps variables to values. We organize configurations using *(semantic) cells*: `</k>` is the cell of IMP code and `</state>` is the cell of program states. In the initial configuration (lines 24–25), `</state>` is empty and `</k>` contains the IMP program that we pass to $\mathbb{K}$ for execution (represented by the special $\mathbb{K}$ variable `$PGM`).

We define formal semantics using *rewrite rules*. In lines 26–27, we define the semantics of variable lookup, where we match on a variable `X` in the `</k>` cell and look up its value `I` in the `</state>` cell, by matching on the binding `X` $\mapsto$ `I`. Then, we rewrite `X` to `I`, denoted by `X` $\Rightarrow$ `I` in the `</k>` cell in line 26. Rewrite rules in $\mathbb{K}$ are similar to those in the rewrite engines such as Maude [7].

A Running Example. IMP is too complex as a running example so we introduce a simpler one: `TWO-COUNTERS`. Although simple, `TWO-COUNTERS` still uses the core features of defining formal syntax as grammars and formal semantics as rewrite rules.

```
module TWO-COUNTERS
  imports INT
  syntax State ::= "<" Int "," Int ">"
  configuration <T> $PGM:State </T>
  rule <M, N> => <M -Int 1, N +Int M>
       requires M >Int 0
endmodule
```

Fig. 3. Running example `TWO-COUNTERS`.

`TWO-COUNTERS` is a tiny language that defines a state machine with two counters. Its computation configuration is simply a pair $\langle m, n\rangle$ of two integers m and n, and its semantics is defined by the following (conditional) rewrite rule:

$$\langle m, n\rangle \Rightarrow \langle m-1, n+m\rangle \qquad \text{if } m > 0 \tag{3}$$

Therefore, `TWO-COUNTERS` adds n by m and reduces m by 1. Starting from the initial state $\langle m, 0\rangle$, `TWO-COUNTERS` carries out m execution steps and terminates at the final state $\langle 0, m(m+1)/2\rangle$, where $m(m+1)/2 = m + (m-1) + \cdots + 1$.

3.2 Program Execution and Proof Parameters

In the following, we show a concrete program execution trace of `TWO-COUNTERS` starting from the initial state $\langle 100, 0\rangle$:

$$\langle 100, 0\rangle, \langle 99, 100\rangle, \langle 98, 199\rangle, \ldots, \langle 1, 5049\rangle, \langle 0, 5050\rangle \tag{4}$$

To make $\mathbb{K}$ generate the above execution trace, we need to follow these steps:

1. Prepare the initial state $\langle 100, 0\rangle$ in a source file, say `100.two-counters`.
2. Compile the formal semantics `TWO-COUNTERS` into a matching logic theory, explained in Sect. 5.

3. Use the $\mathbb{K}$ execution tool `krun` and pass the source file to it:
```
$ krun 100.two-counters --depth N
```

The option `--depth N` tells $\mathbb{K}$ to execute for `N` steps and output the (intermediate) snapshot. By letting `N` be 1, 2, ..., we collect all snapshots in Eq. (4).

The *proof parameter* of Eq. (4) includes the additional rewriting information for each execution step. That is, we need to know the rewrite rule that is applied and the corresponding substitution. In `TWO-COUNTERS`, there is only one rewrite rule, and the substitution can be easily obtained by pattern matching, where we simply match the snapshot with the left-hand side of the rewrite rule.

Note that we regard $\mathbb{K}$ as a "black box". We are not interested in its complex internal algorithms. Instead, we hide such complexity by letting $\mathbb{K}$ generate proof parameters that include enough information for proof object generation. This way, we create a separation of concerns between $\mathbb{K}$ and proof object generation. $\mathbb{K}$ can aim at optimizing the performance of the autogenerated language tools, *without* making proof object generation more complex.

4 Matching Logic and Its Formalization

We review the syntax and proof system of matching logic—the logical foundation of $\mathbb{K}$. Then, we discuss its formalization, which is our main technical contribution and is a critical component of the proof objects we generate for $\mathbb{K}$ (see Sect. 2).

4.1 Matching Logic Overview

Matching logic was proposed in [23] as a means to specify and reason about programs compactly and modularly. The key concept is its formulas, called *patterns*, which are used to specify program syntax and semantics in a uniform way. Matching logic is known for its simplicity and rich expressiveness. In [4–6,22], the authors developed matching logic theories that capture FOL, FOL-lfp, separation logic, modal logic, temporal logics, Hoare logic, λ-calculus, type systems, etc. In Sect. 5, we discuss the matching logic theories that capture $\mathbb{K}$.

The *syntax* of matching logic is parametric in two sets of variables EV and SV. We call EV the set of *element variables*, denoted $x, y, \ldots$, and SV the set of *set variables*, denoted $X, Y, \ldots$.

Definition 1. *A* (matching logic) signature Σ *is a set of* (constant) symbols. *The set of* Σ-patterns, *denoted* $\textsc{Pattern}(\Sigma)$, *is inductively defined as follows:*

$$\varphi ::= x \mid X \mid \sigma \mid \varphi_1\ \varphi_2 \mid \bot \mid \varphi_1 \to \varphi_2 \mid \exists x.\, \varphi \mid \mu X.\, \varphi$$

where in $\mu X.\, \varphi$ *we require that* φ *has no negative occurrences of* X.

Thus, element variables, set variables, and symbols are patterns. $\varphi_1\, \varphi_2$ is a pattern, called *application*, where the first argument is applied to the second. We

$$\begin{array}{ll}
x[\psi/x] \equiv \psi & y[\psi/x] \equiv y \text{ if } y \not\equiv x \\
\sigma[\psi/x] \equiv \sigma & (\varphi_1 \to \varphi_2)[\psi/x] \equiv \varphi[\psi_1/x] \to \varphi_2[\psi/x] \\
\bot[\psi/x] \equiv \bot & (\varphi_1\, \varphi_2)[\psi/x] \equiv (\varphi_1[\psi/x])\ (\varphi_2[\psi/x]) \\
(\exists x.\, \varphi)[\psi/x] \equiv \exists x.\, \varphi & (\exists x.\, \varphi)[\psi/y] \equiv \exists z.\, \varphi[z/x][\psi/y] \text{ for fresh } z \\
 & (\mu X.\, \varphi)[\psi/x] \equiv \mu Z.\, \varphi[Z/X][\psi/x] \text{ for fresh } Z
\end{array}$$

Fig. 4. Capture-free substitution are defined in the usual way and formalized later in Sect. 4.2 as a part of our proof objects.

have propositional connectives $\bot$ and $\varphi_1 \to \varphi_2$, existential quantification $\exists x.\, \varphi$, and the least fixpoints $\mu X.\, \varphi$, from which the following *notations* are defined:

$$\begin{array}{lll}
\neg\varphi \equiv \varphi \to \bot & \top \equiv \neg\bot & \varphi_1 \land \varphi_2 \equiv \neg(\neg\varphi_1 \lor \neg\varphi_2) \\
\varphi_1 \lor \varphi_2 \equiv \neg\varphi_1 \to \varphi_2 & \forall x.\, \varphi \equiv \neg\exists x.\, \neg\varphi & \nu X.\, \varphi \equiv \neg\mu X.\, \neg\varphi[\neg X/X]
\end{array}$$

We use $\text{FV}(\varphi)$ to denote the free variables of φ, and $\varphi[\psi/x]$ and $\varphi[\psi/X]$ to denote capture-free substitution. Their (usual) definitions are listed in Fig. 4.

Matching logic has a *pattern matching semantics*, where a pattern φ is interpreted as the set of elements that match it. For example, $\varphi_1 \land \varphi_2$ is the pattern that is matched by those matching both φ_1 and φ_2. Matching logic semantics is not needed for proof object generation, so we exile it to [5,22].

We show the *matching logic proof system* in Fig. 5, which defines the provability relation, written $\Gamma \vdash \varphi$, meaning that φ can be proved using the proof system, with patterns in Γ added as additional axioms. We call Γ a *matching logic theory*. The proof system is a main component of proof objects. To understand it, we first need to define *application contexts*.

Definition 2. *A* context *is a pattern C with a hole variable $\square$. We write $C[\varphi] \equiv C[\varphi/\square]$ as the result of context plugging. We call C an* application context*, if*

1. $C \equiv \square$ is the identity context; or
2. $C \equiv \varphi\ C'$ or $C \equiv C'\ \varphi$, where C' is an application context and $\square \notin \text{FV}(\varphi)$.

That is, the path from the root to $\square$ in C has only applications.

The proof rules are sound and can be divided into 4 categories: FOL reasoning, frame reasoning, fixpoint reasoning, and some technical rules. The FOL reasoning rules provide (complete) FOL reasoning (see, e.g., [25]). The frame reasoning rules state that application contexts are commutative with disjunctive connectives such as $\lor$ and $\exists$. The fixpoint reasoning rules support the standard fixpoint reasoning as in modal μ-calculus [17]. The technical proof rules are needed for some completeness results (see [5] for details).

4.2 Formalizing Matching Logic

We discuss the formalization of matching logic, which is our first main contribution and forms an important component in our proof objects (see Sect. 2).

FOL Rules	(Propositional 1)	$\varphi \to (\psi \to \varphi)$
	(Propositional 2)	$(\varphi \to (\psi \to \theta)) \to ((\varphi \to \psi) \to (\varphi \to \theta))$
	(Propositional 3)	$((\varphi \to \bot) \to \bot) \to \varphi$
	(Modus Ponens)	$\dfrac{\varphi \quad \varphi \to \psi}{\psi}$
	($\exists$-Quantifier)	$\varphi[y/x] \to \exists x.\, \varphi$
	($\exists$-Generalization)	$\dfrac{\varphi \to \psi}{(\exists x.\, \varphi) \to \psi}\, x \notin FV(\psi)$
Frame Rules	(Propagation$_\bot$)	$C[\bot] \to \bot$
	(Propagation$_\vee$)	$C[\varphi \vee \psi] \to C[\varphi] \vee C[\psi]$
	(Propagation$_\exists$)	$C[\exists x.\, \varphi] \to \exists x.\, C[\varphi]$ with $x \notin FV(C)$
	(Framing)	$\dfrac{\varphi \to \psi}{C[\varphi] \to C[\psi]}$
Fixpoint Rules	(Substitution)	$\dfrac{\varphi}{\varphi[\psi/X]}$
	(Prefixpoint)	$\varphi[(\mu X.\, \varphi)/X] \to \mu X.\, \varphi$
	(Knaster-Tarski)	$\dfrac{\varphi[\psi/X] \to \psi}{(\mu X.\, \varphi) \to \psi}$
Technical Rules	(Existence)	$\exists x.\, x$
	(Singleton)	$\neg(C_1[x \wedge \varphi] \wedge C_2[x \wedge \neg\varphi])$

Fig. 5. Matching logic proof system (where C, C_1, C_2 are application contexts).

Metamath [20] is a tiny language to state abstract mathematics and their proofs in a machine-checkable style. In our work, we use Metamath to formalize matching logic and to encode our proof objects. We choose Metamath for its simplicity and fast proof checking: Metamath proof checkers are often hundreds lines of code and can proof-check thousands of theorems in a second.

Our formalization follows closely Sect. 4.1. We formalize the syntax of patterns and the proof system. We also need to formalize some metalevel operations such as free variables and capture-free substitution. An *innovative* contribution is a generic way to handling *notations* (such as $\neg$ and $\wedge$) in matching logic. The resulting formalization has only 245 lines of code, which we show in [16]. This formalization of matching logic is the main trust base of our proof objects.

Metamath Overview. We use an extract of our formalization of matching logic (Fig. 6) to explain the basic concepts in Metamath. At a high level, a Metamath source file consists of a list of *statements*. The main ones are:

1. *constant statements* (`$c`) that declare Metamath constants;

```
 1  $c \imp ( ) #Pattern |- $.
 2
 3  $v ph1 ph2 ph3 $.
 4  ph1-is-pattern $f #Pattern ph1 $.
 5  ph2-is-pattern $f #Pattern ph2 $.
 6  ph3-is-pattern $f #Pattern ph3 $.
 7  imp-is-pattern
 8    $a #Pattern ( \imp ph1 ph2 ) $.
 9
10  axiom-1
11    $a |- ( \imp ph1 ( \imp ph2 ph1 ) ) $.
12
13  axiom-2
14    $a |- ( \imp ( \imp ph1 ( \imp ph2 ph3 ) )
15          ( \imp ( \imp ph1 ph2 )
16                ( \imp ph1 ph3 ) ) ) $.
17
18  ${
19    rule-mp.0 $e |- ( \imp ph1 ph2 ) $.
20    rule-mp.1 $e |- ph1 $.
21    rule-mp   $a |- ph2 $.
22  $}
23  imp-refl $p |- ( \imp ph1 ph1 )
24  $=
25    ph1-is-pattern ph1-is-pattern
26    ph1-is-pattern imp-is-pattern
27    imp-is-pattern ph1-is-pattern
28    ph1-is-pattern imp-is-pattern
29    ph1-is-pattern ph1-is-pattern
30    ph1-is-pattern imp-is-pattern
31    ph1-is-pattern imp-is-pattern
32    imp-is-pattern ph1-is-pattern
33    ph1-is-pattern ph1-is-pattern
34    imp-is-pattern imp-is-pattern
35    ph1-is-pattern ph1-is-pattern
36    imp-is-pattern imp-is-pattern
37    ph1-is-pattern ph1-is-pattern
38    ph1-is-pattern imp-is-pattern
39    ph1-is-pattern axiom-2
40    ph1-is-pattern ph1-is-pattern
41    ph1-is-pattern imp-is-pattern
42    axiom-1 rule-mp ph1-is-pattern
43    ph1-is-pattern axiom-1 rule-mp
44  $.
```

Fig. 6. An extract of the Metamath formalization of matching logic.

2. *variable statements* (`$v`) that declare Metamath variables, and *floating statements* (`$f`) that declare their intended ranges;
3. *axiomatic statements* (`$a`) that declare Metamath axioms, which can be associated with some *essential statements* (`$e`) that declare the premises;
4. *provable statements* (`$p`) that states a Metamath theorem and its proof.

Figure 6 defines the fragment of matching logic with only implications. We declare five constants in a row in line 1, where `\imp`, `(`, and `)` build the syntax, `#Pattern` is the type of patterns, and `|-` is the provability relation. We declare three metavariables of patterns in lines 3–6, and the syntax of implication $\varphi_1 \to \varphi_2$ as `( \imp ph1 ph2 )` in line 7. Then, we define matching logic proof rules as Metamath axioms. For example, lines 18–22 define the rule (Modus Ponens).

In line 23, we show an example (meta-)theorem and its formal proof in Metamath. The theorem states that $\vdash \varphi_1 \to \varphi_1$ holds, and its proof (lines 25–43) is a sequence of labels referring to the previous axiomatic/provable statements.

Metamath proofs are very easy to proof-check, which is why we use it in our work. The proof checker reads the labels in order and push them to a *proof stack* S, which is initially empty. When a label l is read, the checker pops its premise statements from S and pushes l itself. When all labels are consumed, the checker checks whether S has exactly one statement, which should be the original proof goal. If so, the proof is checked. Otherwise, it fails.

As an example, we look at the first 5 labels of the proof in Fig. 6, line 25:

```
                  // Initially, the proof stack S is empty
ph1-is-pattern    // S = [ #Pattern ph1 ]
ph1-is-pattern    // S = [ #Pattern ph1 ; #Pattern ph1 ]
ph1-is-pattern    // S = [ #Pattern ph1 ; #Pattern ph1 ; #Pattern ph1 ]
imp-is-pattern    // S = [ #Pattern ph1 ; #Pattern ( \imp ph1 ph1 ) ]
imp-is-pattern    // S = [ #Pattern ( \imp ph1 ( \imp ph1 ph1 ) ) ]
```

where we show the stack status in comments. The first label `ph1-is-pattern` refers to a `$f`-statement without premises, so nothing is popped off, and the corresponding statement `#Pattern ph1` is pushed to the stack. The same happens, for the second and third labels. The fourth label `imp-is-pattern` refers to a `$a`-statement with two metavariables of patterns, and thus has 2 premises. Therefore, the top two statements in S are popped off, and the corresponding conclusion `#Pattern ( \imp ph1 ph1 )` is pushed to S. The last label does the same, popping off two premises and pushing `#Pattern ( \imp ph1 ( \imp ph1 ph1 ) )` to S. Thus, these five proof steps prove the *wellformedness* of $\varphi_1 \to (\varphi_1 \to \varphi_1)$.

Formalizing Matching Logic Syntax. Now, we go through the formalization of matching logic and emphasize some highlights. See [5,6,22] for full detail.

The syntax of patterns is formalized below, following Definition 1:

```
$c \bot \imp \app \exists \mu ( ) $.
var-is-pattern      $a #Pattern xX $.
symbol-is-pattern   $a #Pattern sg0 $.
bot-is-pattern      $a #Pattern \bot $.
imp-is-pattern      $a #Pattern ( \imp ph0 ph1 ) $.
app-is-pattern      $a #Pattern ( \app ph0 ph1 ) $.
exists-is-pattern   $a #Pattern ( \exists x ph0 ) $.
${  mu-is-pattern.0 $e #Positive X ph0 $.
    mu-is-pattern   $a #Pattern ( \mu X ph0 ) $.  $}
```

Note that we omit the declarations of metavariables (such as `xX`, `sg0`, ...) because their meaning can be easily inferred. The only nontrivial case above is `mu-is-pattern`, where we require that `ph0` is positive in `X`, discussed below.

Metalevel Assertions. To formalize matching logic, we need the following metalevel operations and/or assertions:

1. positive (and negative) occurrences of variables;
2. free variables;
3. capture-free substitution;
4. application contexts;
5. notations.

Item 1 is needed to define the syntax of $\mu X.\varphi$, while Items 2–5 are needed to define the proof system (Fig. 5). Here, we show how to define capture-free substitution as an example. Notations are discussed in the next section.

To formalize capture-free substitution, we first define a Metamath constant

```
$c #Substitution $.
```

that serves as an assertion symbol: `#Substitution ph ph’ ph” xX` holds iff `ph` $\equiv$ `ph’`[`ph”` / `xX`]. Then, we can define substitution following Fig. 4. The only nontrivial case is when `ph’` is $\exists x.\varphi$ or $\mu X.\varphi$, in which case α-renaming is required to avoid variable capture. We show the case when `ph’` is $\exists x.\varphi$ below:

```
substitution-exists-shadowed
   $a #Substitution ( \exists x ph1 ) ( \exists x ph1 ) ph0 x $.
${ $d xX x  $.
   $d y ph0 $.
   substitution-exists.0 $e #Substitution ph2 ph1 y x $.
   substitution-exists.1 $e #Substitution ph3 ph2 ph0 xX $.
   substitution-exists
     $a #Substitution ( \exists y ph3 ) ( \exists x ph1 ) ph0 xX $. $}
```

There are two cases, as expected from Fig. 4. `substitution-exists-shadowed` is when the substitution is shadowed. `substitution-exists` is the general case, where we first rename `x` to a fresh variable `y` and then continue the substitution. The `$d`-statements state that the substitution is not shadowed and `y` is fresh.

Supporting Notations. Notations (e.g., $\neg$ and $\wedge$) play an important role in matching logic. Many proof rules such as (Propagation$_\vee$) and (Singleton) use notations (see Fig. 5). However, Metamath has no built-in support for notations. To define a notation, say $\neg\varphi \equiv \varphi \to \bot$, we need to (1) declare a constant `\not` and add it to the pattern syntax; (2) define the equivalence relation $\neg\varphi \equiv \varphi \to \bot$; and (3) add a new case for `\not` to *every metalevel assertions.* While (1) and (2) are reasonable, we want to avoid (3) because there are many metalevel assertions and thus it creates duplication.

Therefore, we implement an innovative and generic method that allows us to define *any notations* in a compact way. Our method is to declare a new constant `#Notation` and use it to capture the *congruence relation of sugaring/desugaring.* Using `#Notation`, it takes only three lines to define the notation $\neg\varphi \equiv \varphi \to \bot$:

```
$c \not $.
not-is-pattern $a #Pattern  ( \not ph0 ) $.
not-is-sugar   $a #Notation ( \not ph0 ) ( \imp ph0 \bot ) $.
```

To make the above work, we need to state that `#Notation` is a congruence relation with respect to the syntax of patterns and all the other metalevel assertions. Firstly, we state that it is reflexive, symmetric, and transitive:

```
notation-reflexivity $a #Notation ph0 ph0 $.
${ notation-symmetry.0 $e #Notation ph0 ph1 $.
   notation-symmetry   $a #Notation ph1 ph0 $. $}
${ notation-transitivity.0 $e #Notation ph0 ph1 $.
   notation-transitivity.1 $e #Notation ph1 ph2 $.
   notation-transitivity   $a #Notation ph0 ph2 $. $}
```

And the following is an example where we state that `#Notation` is a congruence with respect to provability:

```
${ notation-provability.0 $e #Notation ph0 ph1 $.
   notation-provability.1 $e |- ph0 $.
   notation-provability   $a |- ph1 $. $}
```

This way, we only need a *fixed* number of statements that state that `#Notation` is a congruence, making it more compact and less duplicated to define notations.

Formalizing Proof System. With metalevel assertions and notations, it is now straightforward to formalize matching logic proof rules. We have seen the formalization of (Modus Ponens) in Fig. 6. In the following, we formalize the fixpoint proof rule (Kanaster-Tarski), whose premises use capture-free substitution:

```
${ rule-kt.0 $e #Substitution ph0 ph1 ph2 X $.
   rule-kt.1 $e |- ( \imp ph0 ph2 ) $.
   rule-kt   $a |- ( \imp ( \mu X ph1 ) ph2 ) $. $}
```

5 Compiling $\mathbb{K}$ into Matching Logic

To execute programs using $\mathbb{K}$, we need to compile the $\mathbb{K}$ language definition for language L into a matching logic theory, written Γ^L (see Sect. 3.2). In this section, we discuss this compilation process and show how to formalize Γ^L.

5.1 Basic Matching Logic Theories

Firstly, we discuss the basic matching logic theories that are required by Γ^L. We discuss the theories of equality, sorts (and sorted functions), and rewriting.

Theory of Equality. By equality, we mean a (predicate) pattern $\varphi_1 = \varphi_2$ that holds (i.e., equals to $\top$) iff φ_1 equals to φ_2, and fails (i.e., equals to $\bot$) otherwise. We first need to define *definedness* $\lceil \varphi \rceil$, which is a predicate pattern that states that φ is *defined*, i.e., φ is matched by at least one element: φ is not $\bot$.

Definition 3. *Consider a symbol* $\lceil \_ \rceil \in \Sigma$*, called the* definedness symbol. *We write* $\lceil \varphi \rceil$ *for the application* $\lceil \_ \rceil\ \varphi$*. In addition, we define the following axiom:*

$$\text{(Definedness)} \quad \lceil x \rceil \tag{5}$$

(Definedness) states that any element x is *defined*. Using the definedness symbol, we can define many important mathematical instruments, including equality, as the following notations:

$$\lfloor \varphi \rfloor \equiv \neg \lceil \neg \varphi \rceil \quad // \text{ Totality} \qquad \varphi_1 = \varphi_2 \equiv \lfloor \varphi_1 \leftrightarrow \varphi_2 \rfloor \quad // \text{ Equality}$$
$$\varphi_1 \subseteq \varphi_2 \equiv \lfloor \varphi_1 \rightarrow \varphi_2 \rfloor \quad // \text{ Inclusion} \qquad x \in \varphi \equiv \lceil x \wedge \varphi \rceil \quad // \text{ Membership}$$

[22, Section 5.1] shows that the above indeed capture the intended semantics.

Theory of Sorts. Matching logic is not sorted, but $\mathbb{K}$ is. To compile $\mathbb{K}$ into matching logic, we need a systematic way to dealing with sorts. We follow the "sort-as-predicate" paradigm to handle sorts and sorted functions in matching logic, following [4,6]. The main idea is to define a symbol $[\![\_]\!] \in \Sigma$, called the *inhabitant symbol*, and use the *inhabitant pattern* $[\![s]\!]$ (abbreviated for the application $[\![\_]\!]\, s$) to represent the *inhabitant set* of sort s. For example, to define a sort Nat, we define a corresponding symbol Nat that represents the sort name, and use $[\![Nat]\!]$ to represent the set of all natural numbers.

Sorted functions can be axiomatized as special matching logic symbols. For example, the successor function *succ* of natural numbers is a symbol with axiom:

$$\forall x.\, x \in [\![Nat]\!] \to \exists y.\, y \in [\![Nat]\!] \wedge succ\, x = y \tag{6}$$

In other words, for any x in the inhabitant set of Nat, there exists a y in the inhabitant set of Nat such that $succ\, x$ equals to y. Thus, *succ* is a sorted function from Nat to Nat.

Theory of Rewriting. Recall that in $\mathbb{K}$, the formal language semantics is defined using rewrite rules, which essentially define a *transition system* over computation configurations. In matching logic, a transition system can be captured by only one symbol $\bullet \in \Sigma$, called *one-path next*, with the intuition that for any configuration γ, $\bullet\gamma$ is matched by all configurations that can go to γ in one step. In other words, γ is reached on *one-path* in the *next* configuration.

Program execution is the reflexive and transitive closure of one-path next. Formally, we define program execution (i.e., rewriting) as follows:

$$\begin{aligned} \diamond\varphi &\equiv \mu X.\, \varphi \vee \bullet X && \text{// Eventually; equals to } \varphi \vee \bullet\varphi \vee \bullet\bullet\varphi \vee \dots \\ \varphi_1 \Rightarrow \varphi_2 &\equiv \varphi_1 \to \diamond\varphi_2 && \text{// Rewriting} \end{aligned}$$

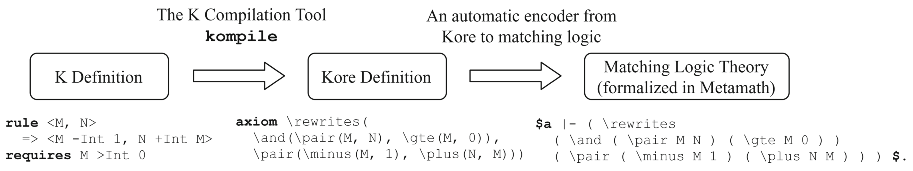
5.2 Kore: The Intermediate Between $\mathbb{K}$ and Matching Logic

The $\mathbb{K}$ compilation tool `kompile` (explained shortly) is what compiles a $\mathbb{K}$ language definition into a matching logic theory Γ^L, written in a formal language called Kore. For legacy reasons, the Kore language is not the same as the syntax of matching logic (Definition 1), but an axiomatic extension with equality, sorts, and rewriting. Thus, to formalize Γ^L in proof objects, we need to (1) formalize the matching logic theories of equality, sorts, and rewriting; and (2) automatically translate Kore definitions into the corresponding matching logic theories. Figure 7 shows the 2-phase translation from $\mathbb{K}$ to matching logic, via Kore.

Phase 1: From $\mathbb{K}$ to Kore. To compile a $\mathbb{K}$ definition such as `two-counters.k` in Fig. 3, we pass it to the $\mathbb{K}$ compilation tool `kompile` as follows:

```
$ kompile two-counters.k
```

The result is a compiled Kore definition `two-counters.kore`. We show the auto-generated Kore axiom in Fig. 7 that corresponds to the rewrite rule in Eq. (3). As we can see, Kore is a much lower-level language than $\mathbb{K}$, where the programming language concrete syntax and $\mathbb{K}$'s front end syntax are parsed and replaced by the abstract syntax trees, represented by the constructor terms.



```
rule <M, N>
  => <M -Int 1, N +Int M>
requires M >Int 0
```

```
axiom \rewrites(
  \and(\pair(M, N), \gte(M, 0)),
  \pair(\minus(M, 1), \plus(N, M)))
```

```
$a |- ( \rewrites
  ( \and ( \pair M N ) ( \gte M 0 ) )
  ( \pair ( \minus M 1 ) ( \plus N M ) ) ) $.
```

Fig. 7. Automatic translation from $\mathbb{K}$ to matching logic, via Kore

Phase 2: From Kore to Matching Logic. We develop an automatic encoder that translates Kore syntax into matching logic patterns. Since Kore is essentially the theory of equality, sorts, and rewriting, we can define the syntactic constructs of the Kore language as *notations*, using the basic theories in Sect. 5.1.

6 Generating Proof Objects for Program Execution

In this section, we discuss how to generate proof objects for program execution, based on the formalization of matching logic and $\mathbb{K}$/Kore in Sects. 4 and 5. The key step is to generate proof objects for *one-step executions*, which are then put together to build the proof objects for multi-step executions using the transitivity of the rewriting relation. Thus, we focus on the process of generating proof objects for one-step executions from the proof parameters provided by $\mathbb{K}$.

6.1 Problem Formulation

Consider the following $\mathbb{K}$ definition that consists of K (conditional) rewrite rules:

$$S = \{t_k \wedge p_k \Rightarrow s_k \mid k = 1, 2, \ldots, K\}$$

where t_k and s_k are the left- and right-hand sides of the rewrite rule, respectively, and p_k is the rewriting condition. Consider the following execution trace:

$$\varphi_0, \varphi_1, \ldots, \varphi_n \tag{7}$$

where $\varphi_0, \ldots, \varphi_n$ are snapshots. We let $\mathbb{K}$ generate the following proof parameter:

$$\Theta \equiv (k_0, \theta_0), \ldots, (k_{n-1}, \theta_{n-1}) \tag{8}$$

where for each $0 \leq i < n$, k_i denotes the rewrite rule that is applied on φ_i ($1 \leq k_i \leq K$) and θ_i denotes the corresponding substitution such that $t_{k_i}\theta_i = \varphi_i$.

As an example, the rewrite rule of `TWO-COUNTERS`, restated below:

$$\langle m, n \rangle \Rightarrow \langle m - 1, n + m \rangle \quad \text{if } m > 0 \qquad // \text{ Same as Eq. (3)}$$

has the left-hand side $t_k \equiv \langle m, n \rangle$, the right-hand side $s_k \equiv \langle m - 1, n + m \rangle$, and the condition $p_k \equiv m \geq 0$. Note that the right-hand side pattern s_k contains the arithmetic operations "$+$" and "$-$" that can be further evaluated to a value, if concrete instances of the variables m and n are given. Generally speaking, the right-hand side of a rewrite rule may include (built-in or user-defined) functions that are not constructors and thus can be further evaluated. We call such evaluation process a *simplification.*

6.2 Applying Rewrite Rules and Applying Simplifications

In the following, we list all proof objects for one-step executions.

$$\begin{array}{ll}
\Gamma^L \vdash \varphi_0 \Rightarrow s_{k_0}\theta_0 & // \text{ by applying } t_{k_0} \wedge p_{k_0} \Rightarrow s_{k_0} \text{ using } \theta_0 \\
\Gamma^L \vdash s_{k_0}\theta_0 = \varphi_1 & // \text{ by simplifying } s_{k_0}\theta_0 \\
\quad \cdots & \\
\Gamma^L \vdash \varphi_{n-1} \Rightarrow s_{k_{n-1}}\theta_{n-1} & // \text{ by applying } t_{k_{n-1}} \wedge p_{k_{n-1}} \Rightarrow s_{k_{n-1}} \text{ using } \theta_{n-1} \\
\Gamma^L \vdash s_{k_{n-1}}\theta_{n-1} = \varphi_n & // \text{ by simplifying } s_{k_{n-1}}\theta_{n-1}
\end{array}$$

As we can see, there are two types of proof objects: one that proves the results of *applying rewrite rules* and one that *applies simplification*.

Applying Rewrite Rules. The main steps in proving $\Gamma^L \vdash \varphi_i \Rightarrow s_{k_i}\theta_i$ are (1) to *instantiate* the rewrite rule $t_{k_i} \wedge p_{k_i} \Rightarrow s_{k_i}$ using the substitution

$$\theta_i = [c_1/x_1, \ldots, c_m/x_m]$$

given in the proof parameter, and (2) to show that the (instantiated) rewriting condition $p_{k_i}\theta_i$ holds. Here, $x_1, \ldots, x_m$ are the variables that occur in the rewrite rule and $c_1, \ldots, c_m$ are terms by which we instantiate the variables. For (1), we need to first prove the following lemma, called (Functional Substitution) in [5], which states that $\forall$-quantification can be instantiated by functional patterns:

$$\frac{\forall \boldsymbol{x}.\, t_{k_1} \wedge p_{k_i} \Rightarrow s_{k_i} \quad \exists y_1.\, \varphi_1 = y_1 \quad \cdots \quad \exists y_m.\, \varphi_m = y_m}{t_{k_i}\theta_i \wedge p_{k_i}\theta_i \Rightarrow s_{k_i}\theta_i} \; y_1, \ldots, y_m \text{ fresh}$$

Intuitively, the premise $\exists y_1.\, \varphi_1 = y_1$ states that φ_1 is a functional pattern because it equals to some element y_1.

If Θ in Eq. (8) is the correct proof parameter, θ_i is the correct substitution and thus $t_{k_i}\theta_i \equiv \varphi_i$. Therefore, to prove the original proof goal for one-step execution, i.e. $\Gamma^L \vdash \varphi_i \Rightarrow s_{k_i}\theta_i$, we only need to prove that $\Gamma^L \vdash p_{k_i}\theta_i$, i.e., the rewriting condition p_{k_i} holds under θ_i. This is done by *simplifying* $p_{k_i}\theta_i$ to $\top$, discussed together with the simplification process in the following.

Applying Simplifications. $\mathbb{K}$ carries out simplification exhaustively before trying to apply a rewrite rule, and simplifications are done by applying (oriented) equations. Generally speaking, let s be a functional pattern and $p \to t = t'$ be a (conditional) equation, we say that s can be *simplified* w.r.t. $p \to t = t'$, if there is a sub-pattern s_0 of s (written $s \equiv C[s_0]$ where C is a context) and a substitution θ such that $s_0 = t\theta$ and $p\theta$ holds. The resulting *simplified pattern* is denoted $C[t'\theta]$. Therefore, a proof object of the above simplification consists of two proofs: $\Gamma^L \vdash s = C[t'\theta]$ and $\Gamma^L \vdash p\theta$. The latter can be handled recursively, by simplifying $p\theta$ to $\top$, so we only need to consider the former.

The main steps of proving $\Gamma^L \vdash s = C[t'\theta]$ are the following:

1. to find C, s_0, θ, and $t = t'$ in Γ^L such that $s \equiv C[s_0]$ and $s_0 = t\theta$; in other words, s can be simplified w.r.t. $t = t'$ at the sub-pattern s_0;
2. to prove $\Gamma^L \vdash s_0 = t'\theta$ by instantiating $t = t'$ using the substitution θ, using the same (Functional Substitution) lemma as above;
3. to prove $\Gamma^L \vdash C[s_0] = C[t']$ using the transitivity of equality.

Finally, we repeat the above one-step simplifications until no sub-patterns can be simplified further. The resulting proof objects are then put together by the transitivity of equality.

7 Discussion on Implementation

As discussed in Sect. 2, a complete proof object for program execution (i.e., $\Gamma^L \vdash \varphi_{init} \Rightarrow \varphi_{final}$) consists of (1) the formalization of matching logic and its basic theories; (2) the formalization of Γ^L; and (3) the proofs of one-step and multi-step program executions. In our implementation, (1) is developed manually because it is fixed for all programming languages and program executions. (2) and (3) are automatically generated by the algorithms in Sect. 6.

During the (manual) development of (1), we needed to prove many basic matching logic (meta-)theorems as lemmas, such as (Functional Substitution) in Sect. 6.2. To ease the manual work, we developed an *interactive theorem prover* (ITP) for matching logic, which allows us to carry out higher-level interactive proofs that are later automatically translated into the lower-level Metamath proofs. We show the highlights of our ITP for matching logic in Sect. 7.1.

In Sect. 7.2, we discuss the main limitations of our current preliminary implementation. These limitations are planned to be addressed in future work.

7.1 An Interactive Theorem Prover for Matching Logic

Metamath proofs are low-level and not human readable (see, e.g., the proof of $\vdash \varphi \to \varphi$ in Fig. 6). Metamath has its own interactive theorem prover (ITP), but it is for general purposes and does not have specific support for matching logic. Therefore, we developed a new ITP for matching logic that has the following characteristic features:

– Our ITP understands the syntax of matching logic patterns and has proof tactics to *desugar* notations in the proof goals;
– Our ITP has an automatic proof tactic for propositional tautologies, based on the *resolution* method;
– Our ITP allows *dynamic proofs*, meaning that new lemmas can be dynamically added during an interactive proof; this makes our ITP easier to use.

When an interactive proof is finished, our ITP will translate the higher-level proof tactics into real Metamath formal proofs, and thus ease the manual development. It is not our interest to fully introduce ITP in this paper, as more detail about the ITP is to be found in future publications.

7.2 Limitations and Threats to Validity

We discuss the trust base of the autogenerated proof objects by pointing out the main threats to validity, caused by the limitations of our preliminary implementation. It should be noted that these limitations are about the implementation, and *not* our approach. We shall address these limitations in future work.

Limitation 1: Need to Trust Kore. Our current implementation is based on the existing $\mathbb{K}$ compilation tool `kompile` that compiles $\mathbb{K}$ into Kore definitions. Recall that Kore is a (legacy) formal language with built-in support for equality, sorts, and rewriting, and thus is different (and more complex) than the syntax of matching logic. By using Kore as the intermediate between $\mathbb{K}$ and matching logic (Fig. 7), we need to trust Kore and the $\mathbb{K}$ complication tool `kompile`.

In the future, we will eliminate Kore entirely from the picture and formalize $\mathbb{K}$ *directly*. To do that, we need to formalize the "front end matters" of $\mathbb{K}$, such as concrete programming language syntax and $\mathbb{K}$ attributes, currently handled by `kompile`. That is, we need to formalize and generate proof objects for `kompile`.

Limitation 2: Need to Trust Domain Reasoning. $\mathbb{K}$ has built-in support for domain reasoning such as integer arithmetic. Our current proof objects do not include the formal proofs of such domain reasoning, but instead regard them as assumed lemmas. In the future, we will incorporate the existing research on generating proof objects for SMT solvers [1] into our implementation, in order to generate proof objects also for domain reasoning; see also Sect. 9.

Limitation 3: Do Not Support More Complex $\mathbb{K}$features. Our current implementation only supports the core $\mathbb{K}$ features of defining programming language syntax and of defining formal semantics as rewrite rules. Some more complex features are not supported; the main ones are (1) the `[strict]` attributes that specify evaluation orders; and (2) the use of built-in collection datatypes, such as lists, sets, and maps.

To support (1), we should handle the so-called *heating/cooling rules* that are autogenerated rewrite rules that implement the specified evaluation orders. Our current implementation does not support these heating/cooling rules because they are conditional rules, and their conditions are those that state that an element is *not* a computation result. To prove such conditions, we need additional constructors axioms for the sorts/types that represent results of computation. To support (2), we should extend our algorithms in Sect. 6 with *unification* modulo these collection datatypes.

8 Evaluation

In this section, we evaluate the performance of our implementation and discuss the experiment results, summarized in Table 1. We use two sets of benchmarks.

Table 1. Performance of proof generation/checking (time measured in seconds).

Programs	Proof generation			Proof checking			Proof size	
	Sem	Rewrite	Total	Logic	Task	Total	kLOC	MB
`10.two-counters`	5.95	12.19	18.13	3.26	0.19	3.44	963.8	77
`20.two-counters`	6.31	24.33	30.65	3.41	0.38	3.79	1036.5	83
`50.two-counters`	6.48	73.09	79.57	3.52	0.98	4.50	1259.2	100
`100.two-counters`	6.75	177.55	184.30	3.50	2.10	5.60	1635.6	130
`add8`	11.59	153.34	164.92	3.40	3.09	6.48	1986.8	159
`factorial`	3.84	34.63	38.46	3.57	0.90	4.47	1217.9	97
`fibonacci`	4.50	12.51	17.01	3.44	0.21	3.65	971.7	77
`benchexpr`	8.41	53.22	61.62	3.61	0.80	4.41	1191.3	95
`benchsym`	8.79	47.71	56.50	3.53	0.72	4.25	1163.4	93
`benchtree`	8.80	26.86	35.66	3.47	0.32	3.80	1021.5	81
`langton`	5.26	23.07	28.33	3.46	0.40	3.86	1048.0	84
`mul8`	14.39	279.97	294.36	3.48	7.18	10.66	3499.2	280
`revelt`	4.98	51.83	56.81	3.35	1.10	4.45	1317.4	105
`revnat`	4.81	123.44	128.25	3.37	5.28	8.65	2691.9	215
`tautologyhard`	5.16	400.89	406.05	3.55	14.50	18.04	6884.7	550

The first is our running example `TWO-COUNTERS` with different inputs (10, 20, 50, and 100). The second is REC [11], which is a popular performance benchmark for rewriting engines. We evaluate both the performance of proof object *generation* and that of proof *checking*. Our implementation can be found in [16] and [3].

The main takeaways of our experiments are:

1. Proof checking is efficient and takes a few seconds; in particular, the *task-specific* checking time is often less than one second ("task" column in Table 1).
2. Proof object generation is slower and takes several minutes.
3. Proof objects are huge, often of millions LOC (wrapped at 80 characters).

Proof Object Generation. We measure the proof object generation time as the time to generate complete proof objects following the algorithms in Sect. 6, from the compiled language semantics (i.e., Kore definitions) and proof parameters. As shown in Table 1, proof generation takes around 17–406 s on the benchmarks, and the average is 107 s.

Proof object generation can be divided into two parts: that of the language semantics Γ^L and that of the (one-step and multi-step) program executions. Both parts are shown in Table 1 under columns "sem" and "rewrite", respectively. For the same language, the time to generate language semantics Γ^L is the same (up to experimental error). The time for executions is linear to the number of steps.

Proof Checking. Proof checking is efficient and takes a few seconds on our benchmarks. We can divide the proof checking time into two parts: that of the logical foundation and that of the actual program execution tasks. Both parts are shown in Table 1 under columns "logic" and "task". The "logic" part includes formalization of matching logic and its basic theories, and thus is *fixed* for any programming language and program and has the same proof checking time (up to experimental error). The "task" part includes the language semantics and proof objects for the one-step and multi-step executions. Therefore, the time to check the "task" part is a *more valuable and realistic* measure, and according to our experiments, it is often less than 1 s, making it acceptable in practice.

As a pleasant surprise, the time for "task-specific"proof checking is roughly the same as the time that it takes $\mathbb{K}$ to parse and execute the programs. In other words, there is *no significant performance difference* on our benchmarks between running the programs directly in $\mathbb{K}$ and checking the proof objects.

There exists much potential to optimize the performance of proof checking and make it even faster than program execution. For example, in our approach proof checking is an *embarrassingly parallel problem*, because each meta-theorems can be proof-checked entirely independently. Therefore, we can significantly reduce the proof checking time by running multiple checkers in parallel.

9 Related Work

The idea of using proof generation to address the functional correctness of complicated systems has been introduced a long time ago.

Interactive theorem provers such as Coq [19] and Isabelle [26] are often used to formalize programming language semantics and to reason about program properties. These provers often provide a high-level proof script language that allows the users to develop human-readable proofs, which are then automatically translated into lower-level proof objects that can be checked by the corresponding proof checkers. For example, the proof objects of Coq are of the form $t : t'$, where t' is a term that represents the proposition to be proved and t' represents a formal proof. The typing claim $t : t'$ can then be proof-checked by a proof checker that implements the typing rules of the calculus of inductive constructions (CIC) [8], which is the logical foundation of Coq.

There are two main differences between provers such as Coq and our technique. Firstly, Coq is not regarded as a language framework in the sense of Fig. 1 because no language tools are autogenerated from the formal semantics. In our case, we need to be able to handle the correctness of individual tasks on a case-by-case basis to reduce the complexity. Secondly, Coq proof checking is based on CIC, which is arguably more complex than matching logic—the logical foundation of $\mathbb{K}$ as demonstrated in this paper. Indeed, the formalization of matching logic requires only 245 LOC which we display entirely in [16].

Another application of proof generation is to ensure the correctness of SMT solvers. These are popular tools to check the satisfiability of FOL formulas, written in a formal language containing interpreted functions and predicates. SMT solvers often implement complex data structures and algorithms, putting their correctness at risk. There is recent work such as [1] studying proof generation for SMT solvers. The research has been incorporated in theorem provers such as Lean, which attempts to bridge the gap between SMT reasoning and proof assistants more directly by building a proof assistant with efficient and sophisticated built-in SMT capabilities. As discussed in Sect. 7, our current implementation does not generate proofs for domain reasoning. So, we plan to incorporate the above SMT proof generation work into our future implementation.

10 Conclusion

We propose an innovative approach based on proof generation. The key idea is to generate proof objects as *proof certificates* for each individual task that the language tools conduct, on a case-by-case basis. This way, we avoid formally verifying the entire framework, which is practically impossible, and thus can make the language framework both *practical* and *trustworthy*.

Acknowledgment. The work presented in this paper was supported in part by NSF CNS 16-19275 and an IOHK grant. This material is based upon work supported by the United States Air Force and DARPA under Contract No. FA8750-18-C-0092.

References

1. Barrett, C., De Moura, L., Fontaine, P.: Proofs in satisfiability modulo theories. In: All About Proofs, Proofs for All, vol. 55, no. 1, pp. 23–44 (2015)
2. Bogdănaş, D., Roşu, G.: K-Java: a complete semantics of Java. In: Proceedings of the 42nd Symposium on Principles of Programming Languages (POPL 2015), Mumbai, India, pp. 445–456. ACM (2015)
3. Chen, X., Lin, Z., Trinh, MT, Roşu, G.: Towards a trustworthy semantics-based language framework via proof generation (artifact image). https://zenodo.org/record/4701997#.YIAywHX0mso (2021)
4. Chen, X., Lucanu, D., Roşu, G.: Initial algebra semantics in matching logic. Technical Report, University of Illinois at Urbana-Champaign, July 2020. http://hdl.handle.net/2142/107781
5. Chen, X., Roşu, G.: Matching μ-logic. In: Proceedings of the 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2019), Vancouver, Canada, pp. 1–13. IEEE (2019)
6. Chen, X., Roşu, G.: A general approach to define binders using matching logic. In: Proceedings of the 25th ACM SIGPLAN International Conference on Functional Programming (ICFP 2020), New Jersey, USA, pp. 1–32. ACM (2020)
7. Clavel, M., et al.: Maude Manual (version 3.0). SRI International (2020)
8. Coq Team: Coq documents: calculus of inductive constructions. https://coq.inria.fr/refman/language/cic.html (2020)

9. Ştefănescu, A., Park, D., Yuwen, S., Li, Y., Roşu, G.: Semantics-based program verifiers for all languages. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016), pp. 74–91. ACM (2016)
10. Dasgupta, S., Park, D., Kasampalis, T., Adve, V.S., Roşu, G.: A complete formal semantics of x86–64 user-level instruction set architecture. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019), Phoenix, Arizona, USA, pp. 1133–1148. ACM (2019)
11. Durán, F., Garavel, H.: The rewrite Engines competitions: a RECtrospective. In: Beyer, D., Huisman, M., Kordon, F., Steffen, B. (eds.) TACAS 2019. LNCS, vol. 11429, pp. 93–100. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17502-3_6
12. Ellison, C., Rosu, G.: An executable formal semantics of C with applications. ACM SIGPLAN Not. **47**(1), 533–544 (2012)
13. Guth, D.: A formal semantics of Python 3.3 (2013)
14. Guth, D., Hathhorn, C., Saxena, M., Roşu, G.: RV-Match: practical semantics-based program analysis. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 447–453. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_24
15. Hildenbrandt, E.: KEVM: a complete semantics of the Ethereum virtual machine. In: Proceedings of the 2018 IEEE Computer Security Foundations Symposium (CSF 2018), Oxford, UK, pp. 204–217. IEEE (2018). http://jellopaper.org
16. K Team: Matching logic proof checker. GitHub page (2021). https://github.com/kframework/matching-logic-proof-checker
17. Kozen, D.: Results on the propositional μ-calculus. Theor. Comput. Sci. **27**(3), 333–354 (1983)
18. Luo, Q., et al.: RV-monitor: efficient parametric runtime verification with simultaneous properties. In: Bonakdarpour, B., Smolka, S.A. (eds.) RV 2014. LNCS, vol. 8734, pp. 285–300. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11164-3_24
19. Coq Team: The Coq proof assistant. LogiCal Project (2020)
20. Megill, N., Wheeler, D.A.: Metamath: a computer language for mathematical proofs. Lulu.com (2019)
21. Park, D., Ştefănescu, A., Roşu, G.: KJS: a complete formal semantics of JavaScript. In: Proceedings of the 36th Annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2015), Portland, OR, pp. 346–356. ACM (2015)
22. Roşu, G.: Matching logic. Log. Methods Comput. Sci. **13**(4), 1–61 (2017)
23. Roşu, G., Schulte, W.: Matching logic–extended report. Technical Report Department of Computer Science UIUCDCS-R-2009-3026, University of Illinois at Urbana-Champaign, January 2009
24. Rosu, G.: K-A semantic framework for programming languages and formal analysis tools. Dependable Softw. Syst. Eng. **50**, 186 (2017)
25. Shoenfield, J.R.: Mathematical Logic. Addison-Wesley Pub. Co, Boston (1967)
26. The Isabelle Development Team. Isabelle (2018). https://isabelle.in.tum.de/

Foundations of Fine-Grained Explainability

Sylvain Hallé(✉) and Hugo Tremblay(✉)

Laboratoire d'informatique formelle,
Université du Québec à Chicoutimi, Saguenay, Canada
shalle@acm.org, Hugo_Tremblay2@uqac.ca

Abstract. Explainability is the process of linking part of the inputs given to a calculation to its output, in such a way that the selected inputs somehow "cause" the result. We establish the formal foundations of a notion of explainability for arbitrary abstract functions manipulating nested data structures. We then establish explanation relationships for a set of elementary functions, and for compositions thereof. A fully functional implementation of these concepts is finally presented and experimentally evaluated.

1 Introduction

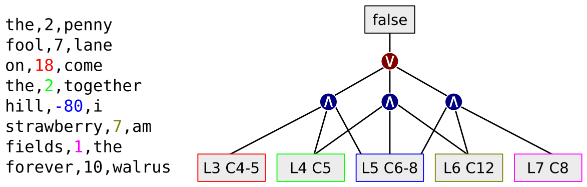
Developers of information systems in all disciplines are facing increasing pressure to come up with mechanisms to describe how or why a specific result is produced—a concept called *explainability*. For example, a web application testing tool that discovers a layout bug can be asked to pinpoint the elements of the page actually responsible for the bug [24]. A process mining system finding a compliance violation inside a business process log can extract a subset of the log's sequence of events that causes the violation [46]. Similarly, an event stream processing system can monitor the state of a server room and, when raising an alarm, identify what machines in the room are the cause of the alarm [40]. All these situations have in common that one is not only interested in an oracle that produces a simple Boolean pass/fail verdict from a given input, but also additional information that somehow links parts of this input to the result.

Explainability is currently handled by *ad hoc* means, if at all. Hence, a developer may write a script that checks a complex condition on some input object; however, for this script to provide an explanation, and not just a verdict, extra code must be written in order to identify, organize, and format the relevant input elements that form an explanation for the result. This extra code is undesirable: it represents additional work, is specific to the condition being evaluated, and relies completely on the developer's intuition as to what constitutes a suitable "explanation". Better yet would be a formal framework where this notion would be defined for arbitrary abstract functions, and accompanied by a generic and systematic way of constructing an explanation.

A. Silva and K. R. M. Leino (Eds.): CAV 2021, LNCS 12760, pp. 500–523, 2021.
https://doi.org/10.1007/978-3-030-81688-9_24

In this paper, we present the theoretical foundations of a notion of explainability. Our model focuses on abstract functions whose input arguments and output values can be composite objects—that is, data structures that may be composed of multiple parts, such as lists. In contrast with existing works on the subject, which consider relationships between inputs and outputs as a whole, our framework is fine-grained: it is possible to point to a specific *part* of an output result, and construct an explanation that refers to precise *parts* of the input.

Figure 1 illustrates the approach on a simple example. On the left is an input, which in this case is a text log of comma-separated values. Suppose that on this log, one wishes to extract the second element of each line, and check that the average of any three successive values is always greater than 3. It is easy to see that the condition is false, but where in the input log are the locations that cause it to be violated? By applying the systematic mechanism described in this paper, one can construct an explanation that is represented by the graph at the right. We can observe that the false output of the condition (the graph's root) is linked to several leaves that designate parts of the input (character locations in each line of the file, identified by colors). Moreover, the graph contains Boolean nodes, indicating that the explanation may involve multiple elements ("and"), and that alternate explanations are possible ("or").



```
the,2,penny
fool,7,lane
on,18,come
the,2,together
hill,-80,i
strawberry,7,am
fields,1,the
forever,10,walrus
```

Fig. 1. Left: a simple text file. Right: an explanation graph obtained from the evaluation of a function on this input.

First, in Sect. 2, we review existing works related to the concept of causality, provenance and taint propagation, which are the notions closest to our concerns. Section 3 lays out the theoretical foundations of our framework: it introduces the notion of parts of composite objects, and formally defines a relation between parts of inputs and outputs of an arbitrary function, called *explanation*. Moreover, it shows how an explanation can be constructed for a function that is a composition of basic functions, by composing their respective explanations. Section 4 then illustrates these definitions by demonstrating what constitutes an explanation on a small yet powerful set of basic functions. Taken together, these results make it possible to easily construct explanations for a wide range of computations.

To showcase the feasibility of this approach, Sect. 5 presents *Petit Poucet*, a fully-functioning Java library that implements these concepts. The library allows users to create their own complex functions by composing built-in primitives, and can automatically produce an explanation for any result computed by these functions on a given input. Experiments on a few examples provide insight on the time and memory overhead incurred by the handling of explanations. Finally, Sect. 6 identifies a number of exciting open theoretical questions that arise from our formal framework.

2 Related Work

Explainability can be seen as a particular case of a more general concept called *lineage*, where inputs and outputs of a calculation are put in relation according to various definitions. Related works around this notion can be separated into a few categories, which we briefly describe below.

2.1 Causality

In the field of testing and formal verification of software systems, lineage often takes the form of *causality*. A classical definition is given by Halpern and Pearl [28], based on what is called "counterfactual dependence": A is a cause of B if the absence of A entails the absence of B. According to this principle, some feature of an object causes the failure of a verification or testing procedure if the absence of this feature instead causes the procedure to emit a passing verdict. This notion can be transposed for various types of systems. When a condition is expressed as a propositional logic formula, the cause of the true or false value of the formula can be constructed in the form of an *explanatory sub-model*; informally, it can be thought of as the smallest set of propositional variables whose value implies the value of the formula.

In the case of conditions expressed on state-based systems, the cause of a violation has been taken either as the shortest prefix that guarantees the violation of the property regardless of what follows [23], or as a minimal set of word-level predicates extracted from a failed execution [50]. A platform for hardware formal verification, called RuleBasePE, expands on this latter definition to identify components of a system that are responsible for the violation of a safety property on a single execution trace [5].

Causality has been criticized as an all-or-nothing notion; *responsibility* has been introduced a refinement on this concept, where the involvement of some element A as the cause of B can be quantified [12]. The problem of deciding causality also has a high computational complexity; as a matter of fact, determining if A is a cause of B in a model only allowing Boolean values to variables is already NP-complete [18]. Tight automata [31,43] have also been developed to produce minimal counter-examples, which in this case, are sequences of states produced by a traversal of the finite-state machine. Explanatory sub-models can also be extended to the temporal case [19]. Another approach involves the computation of a so-called "minimal debugging window", which is a small segment of the input trace that contains the discovered violation [36].

Finally, distance-based approaches compare a faulty trace with the closest valid trace (according to a given distance metric) [22]; the differences between the two traces are defined as the cause of the failure. A similar principle is applied in a software testing technique called *delta debugging* to identify values of variables in a program that are responsible for a failure [15].

2.2 Provenance in Information Systems

In a completely different direction, a large amount of work on lineage has been done in the field of databases, where this notion is often called *provenance*. A thorough survey of related approaches on provenance [9] reveals that this concept has been studied in several different areas of data management, such as scientific data processing and database management systems.

Research in this field typically distinguishes between three types of provenance. The first type is called *why-provenance* [17]: to each tuple t in the output of a relational database query, why-provenance associates a set of tuples present in the input of the query that helped to "produce" t. *How-provenance*, as its name implies, keeps track not only of what input tuples contribute to the input, but also in which way these tuples have been combined to form the result [21]. For example, a symbolic polynomial like $t^2 + t \cdot t'$ indicates that an output tuple can be explained in two alternative ways: either by using tuple t twice, or by combining t and t'. Finally, *where-provenance* describes where a piece of data is copied from [8]. It is typically expressed at a finer level of granularity, by allowing to link individual values inside an output tuple to individual values of one or more input tuple. One possible way of doing it is through a technique called annotation-propagation, where each part of the input is given symbolic "annotations", which are then percolated and tracked all the way to the output [7].

A recent survey reveals the existence of more than two dozen provenance-aware systems [38]. Where-provenance has been implemented into Polygen [51], DBNotes [11], Mondrian [20], MXQL [48] and Orchestra [29]. The Spider system performs a slightly different task, by showing to a user the "route" from input to output that is being taken by data when a specific database query is executed [10]. The foundations for all these systems are relational databases, where sets of tuples are manipulated by operators from relational algebra, or extensions of SQL.

Taken in a broad sense, we also list in this category various works that aim to develop *explainable* Artificial Intelligence [42]. Models used in AI vary widely, ranging from deep neural networks to Bayesian rules and decision trees; consequently, the notion of what constitutes an explanation for each of them is also very variable, and is at times only informally stated.

2.3 Taint Analysis and Information Flow

A last line of works this time considers the linkage between the inputs and the outputs produced by a piece of procedural code, mostly for considerations of security. Dynamic *taint analysis* consists in marking and tracking certain data in a program at run-time. A typical use case for taint analysis is to check whether sensitive information (such as a password) is being leaked into an unprotected memory location, or if a "tainted" piece of input such as a user-provided string is being passed to a function like a database query without having been sanitized first (opening the door to injection attacks).

In this category, TaintCheck has been developed into a system where each memory byte is associated with a 4-byte pointer to a taint data structure [37]; program inputs are marked as tainted, and the system propagates taint markers to other memory locations during the execution of a program; this concept has been extended to the operating system as a whole in an implementation called Asbestos [47]. Hardware implementations of this principle have also been proposed [16,44]. Dytan [14] is a notable system for taint propagation and analysis on programs written in assembly language. We shall also mention the Gift system, as well as a compiler based on it called Aussum [32]. On its side, Rifle focuses on the information flow [45], while TaintBochs is a system that has been used to track the lifetime of sensitive data inside the memory of a program [13].

The capability of following taint markings on the inputs of a program can be used in many ways. For example, a system called Comet uses taint propagation to improve the coverage of an existing test suite [33]. Taint analysis can also be used to quantify the amount of information leak in a system [34]. Stated simply, the propagation of taint markings can be seen as a form of how-provenance, applied to variables and procedural code instead of tuples and relational queries. Note however that it operates in a top-down fashion: mark the inputs, and track these markings on their way to the output. In contrast, we shall see that our notion of explanation is bottom-up: point at a part of the output, and retrace the parts of the input that are related to it.

3 A Formal Definition of Explanation

The problem we consider can be simply stated. Given an abstract function f and an input argument x, establish a formal relationship that explains $f(x)$ based on x. While this is more or less closely related to the works we presented in the previous section, the solution we propose has a few distinguishing features.

First, x and $f(x)$ may be composite objects made of multiple "parts", and it is possible to relate specific parts of an output to specific parts of an input. Second, if f is itself a composition $g \circ h$, it is possible to construct the input-output relationships of f from the individual input-output relationships of g and h. Third, these relations are not defined *ad hoc* for each individual function, but come as consequences of a general definition. Finally, given x and $f(x)$, determining the parts of x in relation with a given part of $f(x)$ is tractable.

In this section, we establish the formal foundations of our proposed approach. We start by defining an abstract "part-of" relation between abstract objects, based on the notion of *designator*, and establish some properties of this relation. We then propose a definition of *explanation* for arbitrary functions, and discuss how it differs from existing causality and lineage relationships mentioned earlier.

3.1 Object Parts

Let $\mathfrak{U} = \bigcup_i \mathcal{O}_i$ be a union of sets of *objects*; the sets $\mathcal{O}_i$ are called *types*. We suppose that $\mathfrak{U}$ contains a special object, noted $\boxslash$, that represents "nothing". Types are disjoint sets, with the exception of $\boxslash$ which, for convenience, is assumed to be part of every type.

Each type $\mathcal{O}$ is associated with a set $\Pi_\mathcal{O}$, whose elements are called *parts*. The set $\Pi_\mathcal{O}$ contain functions of the form $\pi : \mathcal{O} \rightarrow \mathcal{O}'$; it is expected that $\pi(\boxslash) = \boxslash$ for every such function. In addition, we impose that $\Pi_\mathcal{O}$ always contains two other functions defined as $\mathbf{1} : x \mapsto x$, and $\mathbf{0} : x \mapsto \boxslash$. We shall override the term "part" and say that an object $o' \in \mathcal{O}'$ is a part of some oth er object $o \in \mathcal{O}$ if it is the result of applying some $\pi \in \Pi_\mathcal{O}$ to o. A *proper part* is any part other than $\mathbf{1}$ and $\mathbf{0}$; we shall use $\Pi^*_\mathcal{O}$ to designate the set of proper parts for a given type. A type $\mathcal{O}$ is called *scalar* if $\Pi^*_\mathcal{O} = \emptyset$; otherwise it is called *composite*.

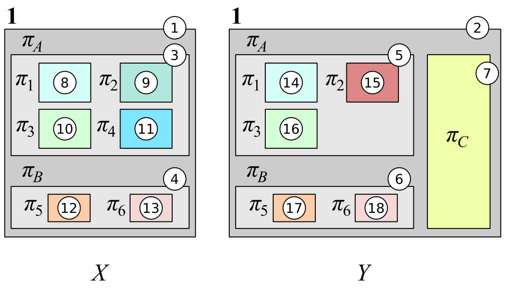


Fig. 2. Illustration of two abstract composite objects of the same type. Composite parts are represented by gray rectangles; scalar parts are represented by colored rectangles. (Color figure online)

Figure 2 shows an example of two abstract composite objects X and Y of the same type $\mathcal{O}$, which we suppose has a set of three proper parts called π_A, π_B and π_C. Parts of an object can themselves be of a composite type; we assume here that type A has four parts $\pi_1, \dots, \pi_4$, type B has two parts π_5, π_6, and type C is a scalar. For example, $\pi_B(X)$ corresponds to the rectangle numbered 4, and $\pi_C(Y)$ is the rectangle numbered 7. Consistent with our definitions, $\mathbf{1}$ designates the whole object, hence $\mathbf{1}(X)$ is the rectangle 1. These parts are not all present in all objects of a given type; for example we see that $\pi_C(X) = \boxslash$.

Parts can be composed in the usual way: if $\pi : \mathcal{O} \rightarrow \mathcal{O}'$ and $\pi' : \mathcal{O}' \rightarrow \mathcal{O}''$, their composition $\pi \circ \pi'$ is defined as $o \mapsto \pi(\pi'(o))$. This corresponds intuitively to the notion of the part of some part of an object, and will allow us to point to arbitrarily fine-grained portions of input arguments or output values. For example, in Fig. 2, we have that $(\pi_1 \circ \pi_A)(X)$ corresponds to the rectangle numbered 8, which indeed corresponds to part π_1 of the part π_A of object X. If $\Pi = \{\pi_1, \dots, \pi_n\}$ is a set of parts and π is another part, we will abuse notation and write $\Pi \circ \pi$ to mean the set $\{\pi_1 \circ \pi, \dots, \pi_n \circ \pi\}$.

If π is a part of an object, we shall say that $\pi' \circ \pi$ is a *refinement* of π; inversely, π is a *generalization* of $\pi' \circ \pi$, as it corresponds to a "greater" part of the object. Remark that since $(\mathbf{1} \circ \pi) = (\pi \circ \mathbf{1}) = \pi$, any part π is simultaneously a refinement and a generalization of itself. If π is a proper part of o, all its refinements are also considered as parts of o. The notions of refinement and generalization can be extended to sets of parts. Given two sets $\Pi = \{\pi_1, \dots, \pi_m\}$ and $\Pi' = \{\pi'_1, \dots, \pi'_n\}$, Π is a refinement if there exists an injection μ between Π and Π' such that $\mu(\pi) = \pi'$ if and only if π is a refinement of π'; conversely, Π' is a generalization of Π. Again, a refinement of a set of parts Π picks fewer parts, and more selective parts of an object than Π. In the example of Fig. 2, the set $\Pi = \{\pi_1 \circ \pi_A, \pi_C\}$ is a refinement of $\Pi' = \{\pi_A, \pi_5 \circ \pi_B, \pi_C\}$ (the injection here being the two associations $\pi_1 \circ \pi_A \mapsto \pi_A$ and $\pi_C \mapsto \pi_C$).

Given a part π, two objects $o, o' \in \mathcal{O}$ are said to *differ on* π if $\pi(o) \neq \pi(o')$. This can be illustrated in Fig. 2. Objects X and Y differ on π_C, since $\pi_C(X) \neq \pi_C(Y)$. They also differ on π_A (since rectangles 3 and 5 are different) and on $\pi_4 \circ \pi_A$ (which produces rectangle 11 for X and $\boxslash$ for Y). However, they do not differ on π_B (rectangles 4 and 6 are identical), and they do not differ on $\pi_3 \circ \pi_A$ (rectangles 10 and 16 are identical). This example shows that if two objects differ on a part π, they do not necessarily differ on a refinement of π (compare π_A and $\pi_1 \circ \pi_A$). Given a set of parts $\Pi = \{\pi_1, \dots, \pi_n\}$, two objects differ on Π if they differ in some $\pi_i \in \Pi$. Obviously, if objects differ on Π, they also differ on any of its generalizations.

Finally, two parts π and π' *intersect* if there exist two parts π_I and π'_I (different from $\mathbf{0}$) such that whenever $(\pi_I \circ \pi)(o) \neq \boxslash$ and $(\pi'_I \circ \pi')(o) \neq \boxslash$, then $(\pi_I \circ \pi)(o) = (\pi'_I \circ \pi')(o)$. Two sets of parts Π and Π' intersect if at least one of their respective parts intersect. In Fig. 2, the sets $\{\pi_A\}$ and $\{\pi_2 \circ \pi_A, \pi_C\}$ intersect (they have in common the part $\pi_2 \circ \pi_A$), while the sets $\{\pi_A\}$ and $\{\pi_5 \circ \pi_B\}$ do not.

3.2 A Definition of Explanation

We are now interested in relations between a part of a function's output, and one or more parts of that function's input. We shall focus on the set of unary functions $f : \mathcal{O} \to \mathcal{O}'$. For the sake of simplicity, functions of multiple input arguments will be modeled as unary functions whose input is a composite type that contains the arguments. These composite types will be used informally to illustrate the notions, and will be formally defined in Sect. 4.

Definition 1. Let $f : \mathcal{O} \to \mathcal{O}'$ be a function, $\Pi \subseteq \Pi_{\mathcal{O}}$ be a set of parts of the function's input type, and $\pi \in \Pi_{\mathcal{O}'}$ be a part of the function's output type. Consider a set Π such that there exists an object o' that differs from o only on Π, and for which $f(o)$ and $f(o')$ differ on π. We say that Π *explains* π if Π is minimal, meaning that no refinement of Π satisfies the previous condition.

As an example, consider the function $f : \langle x, y \rangle \mapsto xy$, with $x = 1$, $y = 1$. For this particular input object, the part designating the first element of the input is an explanation, as changing it to any other value changes the result of the function; the same argument can be made for the part that designates the second element of the input. Consider now the case where $x = 0$, $y = 1$. This time, the value of the second element is irrelevant: changing it to anything else still produces 0. Therefore, the second element of the input is not an explanation for the output; the first element of the input alone "explains" the result of 0 in the output.

Based on these simple definitions, we can already put our framework in contrast with some of the papers we surveyed in Sect. 2. First, note how this definition is different from counterfactual causality [1,28]. This can be illustrated by considering the previous function, in the case where $x = 0$ and $y = 0$. Argument x is not a counterfactual cause of the output value, as changing it to anything else still produces 0; the same argument shows that y is not a cause of the output either. Therefore, one ends up with the counter-intuitive conclusion that none of the inputs cause the output.

In contrast, there exists a minimal set of parts that satisfies our explanation property, which is the set that contains *both* the first and the second element. Indeed, there exists another input object that differs on both elements, and which produces a different result. This is in line with the intuition that either element is sufficient to explain the null value produced by the function, and that therefore both need to be changed to have an impact. It highlights a first distinguishing feature of our approach: the presence of multiple parts inside a set indicates a form of "disjunction" or "alternate" explanations, something that cannot be easily accounted for in many definitions of causality.[1]

Why-provenance is expressed on tuples manipulated by a relational query [17], but our simple case can easily be adapted by assuming that x, y and $f(\langle x, y\rangle)$ each are tuples with a dummy attribute a. The definition then leads to the conclusion that both x and y are considered to "produce" the output, whereas explanation rather concludes that *either* explains the output; the use of how-provenance [21] would produce a similar verdict. Where-provenance [8] is even less appropriate here, as it makes little sense to ask whether the product of two numbers "copies" any of the input arguments to its output.

Since a single explanation is a set of parts, the set of all explanations is therefore a set of sets of parts. As we have shown, a set of parts intuitively represents an alternative (either part is an explanation). In turn, elements of a set of set of parts represents the fact that each of them is an explanation. Therefore, a concise graphical notation for representing sets of sets of parts are and-or trees, such as the one shown in Fig. 3. In the present case, leaves of the tree each represent a (single) object part, while non-leaf nodes are labeled either with "and" ($\wedge$) or "or" ($\vee$). For example, this tree represents the set of sets $\{\{\pi_1\}, \{\pi_2 \circ \pi_3, \pi_5 \circ \pi_6, \pi_4\}, \{\pi_2 \circ \pi_3, \pi_7 \circ \pi_6, \pi_4\}\}$.[2]

[1] Case in point, in [1] a cause is assumed to be a *conjunction* of assertions of the form $X = x$.

[2] Obviously, there exist multiple equivalent trees for the same set of sets of parts.

4 Building Explanations for Functions

Equipped with these abstract definitions, we shall now establish properties of a handful of elementary functions, namely logical and arithmetic operations, and list manipulations. The reader may find that this section is stating the obvious, as many of the results we present correspond to very intuitive notions. The main interest of our approach is that these seemingly trivial conclusions are not defined *ad hoc*, but rather come as consequences of the general definition of explanation introduced in the previous section.

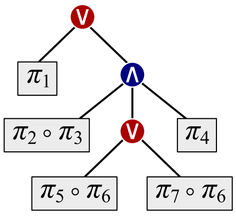


Fig. 3. An example of and-or tree where leaves are object parts.

We first consider as *scalar* types the sets of Boolean values $\mathbb{B}$, the set of real numbers $\mathbb{R}$, and the set of characters $\mathbb{S}$. Then, we shall denote by $\mathbb{V}\langle\mathcal{O}_1,\dots,\mathcal{O}_n\rangle$ the set of vectors of size n, where the i-th element is of type $\mathcal{O}$. Its set of parts $\Pi_{\mathbb{V}\langle\mathcal{O}_1,\dots,\mathcal{O}_n\rangle}$ contains **1** and **0**, as well as all functions $[i] : \mathbb{V}\langle\mathcal{O}_1,\dots,\mathcal{O}_n\rangle \to \mathcal{O}_i$ defined as:

$$[i] : \langle o_1,\dots,o_n\rangle \mapsto \begin{cases} o_i & \text{if } 1 \leq i \leq n \\ \boxslash & \text{otherwise.} \end{cases}$$

In other words, the proper parts of a vector are each of its elements. We shall designate for finite vectors of variable length and uniform type $\mathcal{O}$ by $\mathbb{V}\langle\mathcal{O}^*\rangle$. Finally, character strings will be viewed as the type $\mathbb{V}\langle\mathbb{S}^*\rangle$, i.e. finite words over the alphabet of symbols $\mathbb{S}$. We stress that, although our concept of explanation is illustrated on a small set of functions operating on these types, it is by no means limited to these functions or these types.

4.1 Conservative Generalizations

Some of the functions we shall consider return objects that may be composite; these functions introduce the additional complexity that one may want to refer not only to the whole output of the function, but also to a single part of that function's output. What is more, the inputs of these functions can also be composite objects, and explicitly enumerating all their minimal sets of parts for explanation may not be possible.

Take for example the function $f : \mathbb{V}\langle\mathcal{O}\rangle\to\mathbb{V}\langle\mathcal{O}\rangle$, which simply returns its input vector as is. Suppose that we focus on $\pi = [1]$, the first element of the output vector. Clearly, the set $\{[1]\}$, pointing to the first element of the input vector, should be recognized as the only one that explains this output. However, if $\mathcal{O}$ is a composite type, this set is not minimal, and should be further broken down into all the parts of $\mathcal{O}$. Besides being unmanageable, this enumeration also misses the intuition that what explains the first element of the output is simply the first element of the input.

In the following, we employ an alternate approach, which will be to define a set of *conservative generalizations* of the function's input parts. For most functions, the principle will be the same: given an output part π, we shall define a set of sets of input parts $\Pi = \{\Pi_1, \ldots, \Pi_n\}$, and demonstrate that any input part Π' that explains π on some input intersects with one of the Π_i. It follows that any minimal input part that explains the output is a refinement of one of the Π_i.

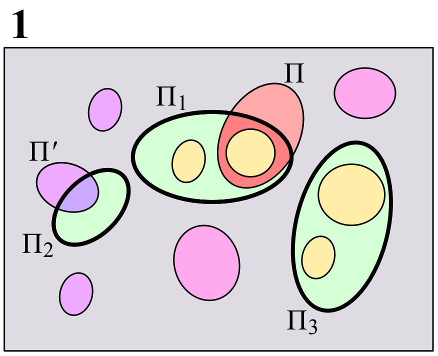


Fig. 4. The sets of parts Π_1, Π_2 and Π_3 (in green) represent a conservative generalization of the minimal sets of explanation parts (yellow circles) (Color figure online).

This is illustrated in Fig. 4, where the minimal sets of explanations of an input are illustrated in yellow. Here, the set $\{\Pi_1, \Pi_2, \Pi_3\}$ has been identified as the target set of sets of input parts. It is possible to see that if one establishes that any set lying outside of the green ovals is not an explanation, it follows that the minimal sets of parts for explanation are all contained inside one of Π_1, Π_2 and Π_3. Note that this is a generalization, as, for example, Π_2 does not contain any minimal set. However, this generalization is conservative, in the sense that all minimal sets are indeed contained within a green oval.

The goal is therefore to come up with generalizations that are, in a sense, as tight as possible. In the example of function f above, we could easily demonstrate that any set of input parts Π that has an impact on the first element of the output must contain a refinement of the first element of the input, and therefore identify $\Pi = \{\{[1]\}\}$ as a sufficient set that "covers" all the minimal explanation input parts. It so happens that this set corresponds exactly to the intuitive result we expected in the first place: the first element of the input vector contains all the parts that impact the output, and no other part of the input has this property.

4.2 Explanation for Scalar Functions

In the following, we provide the formal definition of conservative generalizations of the explanation relationship for a number of elementary functions. We start with functions performing basic arithmetic operations returning a scalar value. Establishing explanation for addition over a vector of numbers is trivial.

Theorem 1. Let $f : \mathbb{V}\langle\mathbb{R}^*\rangle \rightarrow \mathbb{R}$ be the function defined as $\langle x_1, \dots, x_n\rangle \mapsto x_1 + \dots + x_n$. For any input $\langle x_1, \dots, x_n\rangle$, $\varPi$ explains $\mathbf{1}$ on $\langle x_1, \dots, x_n\rangle$ if and only if $\varPi = \{[i]\}$ for some $1 \leq i \leq n$.

In other words, any single element of the input vector explains the result; the case of subtraction is defined identically. Multiplication, however, has a different definition. This is caused by the fact that 0 is an absorbing element, hence its presence suffices for a product to yield zero, as is explained by the following theorem.

Theorem 2. Let $f : \mathbb{R}^n \rightarrow \mathbb{R}$ be the function defined as $\langle x_1, \dots, x_n\rangle \mapsto x_1 \cdots x_n$. For a given input $\langle x_1, \dots, x_n\rangle$, $\varPi$ explains $\mathbf{1}$ if and only if:

- $\varPi = \{[i]\}$ for some $1 \leq i \leq n$, if for all $1 \leq j \leq n$, $x_j \neq 0$
- $\varPi = \bigcup_{i \in \{j : 1 \leq j \leq n \text{ and } x_j = 0\}} \{[i]\}$ otherwise.

Proof. Suppose that all elements of the input vector are non-null. Let $\varPi = \{[i]\}$ for some $1 \leq i \leq n$. Clearly, any input that differs from $\langle x_1, \dots, x_n\rangle$ only in the i-th element produces a different product, and hence is a minimal explanation. Suppose now that at least one element of the vector is null; in such a case, the function returns 0. Let $S = \{j : 1 \leq j \leq n \text{ and } x_j = 0\}$ be the set of all such vector indices. A vector must differ from the input in at least all these positions in order to produce a different output, otherwise the function still returns zero. The only refinements of this set are its strict subsets, but none is sufficient to change the output, and hence the defined set is the only minimal set satisfying the explanation property. □

The same argument can be made of the Boolean function that computes the conjunction of a vector of Boolean values. If all elements of the vector are $\top$, changing any of them produces a change of value, and hence each $\{[i]\}$ explains the output. Otherwise, the set $\varPi = \bigcup_{i \in \{j : 1 \leq j \leq n \text{ and } x_j = \bot\}} \{[i]\}$ is the only minimal set of input parts that explains the output, by a reasoning similar to the case of multiplication above. A dual argument can be done for disjunction by swapping the roles of $\top$ and $\bot$.

The case of the remaining usual arithmetic and Boolean functions can be dispatched easily. Functions taking a single argument (abs, etc.) obviously have this single argument as their only minimal explanation part.

We finally turn to the case of a function that extracts the k-th element of a vector. We recall that vectors can be nested, and hence this element may itself be composite. The intuition here is that what explains a part of the output is that same part in the k-th element of the input.

Theorem 3. Let $f_k : \mathbb{V}\langle\mathcal{O}^n\rangle \to \mathcal{O}$ be the function defined as $\langle x_1,\dots,x_n\rangle \mapsto x_k$ if $1 \le k \le n$, and $\langle x_1,\dots,x_n\rangle \mapsto \boxslash$ otherwise. Let $\pi \in \Pi_{\mathcal{O}}$ be an arbitrary part of $\mathcal{O}$. For any input $\langle x_1,\dots,x_n\rangle$, Π explains π for $\langle x_1,\dots,x_n\rangle$ if and only if $\Pi = \{\pi \circ [k]\}$ and $1 \le k \le n$.

Proof. If $k < 0$ or $k > n$, f_k produces $\boxslash$ regardless of its argument, and so no set of input explains the result. Otherwise, it suffices to observe that $(\pi \circ [i])(\langle x_1,\dots,x_n\rangle) = \pi(x_i) = \pi(f_k(\langle x_1,\dots,x_n\rangle))$, and hence $\{\pi \circ [i]\}$ explains the output. No other set satisfies this condition. $\square$

Table 1. Definition of elementary vector functions studied in this paper.

$$\begin{aligned}
[i](\langle o_1 \dots o_n\rangle) &= \begin{cases} \langle o_i\rangle \text{ if } 1 \le i \le n \\ \boxslash \quad \textit{otherwise} \end{cases} \\
\alpha_f(\langle o_1,\dots,o_n\rangle) &= \langle f(o_1),\dots,f(o_n)\rangle \\
\omega_f^{k,o}(\langle o_1,\dots,o_n\rangle) &= \langle f(\langle o_1,\dots,o_k\rangle),\dots,f(\langle o_{n-k},\dots,o_n\rangle)\rangle \\
¿(\langle b,o,o'\rangle) &= \begin{cases} o \quad \text{if } b = \top \\ o' \; \textit{otherwise} \end{cases}
\end{aligned}$$

4.3 Explanation for Vector Functions

We shall delve into more detail on basic functions that produce a value that may be of non-scalar type, summarized in Table 1 (function $[i]$ has already been discussed earlier). Here, we must consider the fact that an explanation may refer to a part of an element of their output, i.e. designators of the form $\pi \circ [i]$, with π some arbitrary designator.

The first function, noted α_f, applies a function f on each element of an input vector, resulting in an output vector of same cardinality.

Theorem 4. On a given input $\langle x_1,\dots,x_n\rangle$, Π explains $\pi \circ [i]$ of α_f if and only if $\Pi = \{\Pi' \circ [i]\}$ for some set of parts Π', and Π' explains π for x_i of f.

Proof. The i-th element of the output of α_f is $f(x_i)$; if Π' explains x_i of f, then $\Pi' \circ [i]$ explains $\langle x_1,\dots,x_n\rangle$ of α_f. Conversely, suppose that $\Pi' \circ [i]$ explains $\langle x_1,\dots,x_n\rangle$ of α_f; by definition, there exists another input $\langle x'_1,\dots,x'_n\rangle$ that differs on $\Pi' \circ [i]$ and such that the output of α_f differs on $\pi \circ [i]$. Then x_i and x'_i differ on Π', and by the definition of α_f, $f(x_i)$ and $f(x'_i)$ differ on π. If Π' admits a proper part that satisfies this property, then Π is not an explanation, which contradicts the hypothesis. Hence Π' is minimal, and it therefore explains π for x_i of f. $\square$

Function $\omega_f^{k,o}$ applies a function f on a sliding window of width k to the input vector. That is, the first element of the output vector is the result of evaluating f on the first k elements; the second element is the evaluation of f on elements at positions 2 to $k+1$, and so on. If the input vector has fewer than k elements, the function is defined to return a predefined value o. To establish the set of minimal explanations, we define a special function σ_i; given a set of parts Π such that all parts are of the form $\pi \circ [j]$, replaces each of them by $\pi \circ [j - i]$. In other words, parts pointing to a part π of the j-th element of a vector end up pointing to the same part π of the $(j-i)$-th element of a vector.

Theorem 5. *On a given input $\langle x_1, \dots, x_n\rangle$, Π explains $\pi \circ [i]$ of $\omega_f^{k,o}$ if and only if $n \geq k$, $i \geq n-k$, $\Pi = \{\Pi' \circ [i]\}$ for some set of parts Π'. and $\sigma_i(\Pi')$ explains π for x_i of f.*

Proof. The proof is almost identical as for α_f, with the added twist that in the i-th window, an explanation for f referring to a part of the j-th element of its input vector actually refers to the $(j-i)$-th element of the input vector given to ω_f; this explains the presence of σ_i. We omit the details. $\square$

Finally, function "¿" acts as a form of if-then-else construct: depending on the value of its (Boolean) first argument, it returns either its second or its third argument (which can be arbitrary objects). Defining its explanation requires a few cases, depending on whether the second and third element of the input are equal.

Theorem 6. *On a given input $\langle x_1, x_2, x_3\rangle$, if $x_2 \neq x_3$, then $\{[1]\}$ always explains π of ¿; moreover $\{\pi \circ [2]\}$ explains π of ¿ if $x_1 = \top$, and $\{\pi \circ [3]\}$ explains π of ¿ if $x_1 = \bot$. However, if $x_2 = x_3$, then: 1. if $x_1 = \top$, Π explains π of ¿ if and only if $\Pi = \{\pi \circ [2]\}$ or $\Pi = \{[1], \pi \circ [3]\}$; 2. if $x_1 = \bot$, Π explains π of ¿ if and only if $\Pi = \{\pi \circ [3]\}$ or $\Pi = \{[1], \pi \circ [2]\}$.*

Proof. Direct from the fact that $¿(\langle \top, x_2, x_3\rangle) = x_2$, and $¿(\langle \bot, x_2, x_3\rangle) = x_3$. The only corner case is when $x_2 = x_3$; if $x_1 = \top$, one must change either x_2, or *both* x_1 and x_3 in order to produce a different result (and dually when $x_1 = \bot$). $\square$

4.4 Explanation for Composed Functions

Defining and proving conservative generalizations is a task that can quickly become tedious for complex functions, as the previous examples have shown us. Moreover, this process must be done from scratch for each new function one wishes to consider, as the proofs for each of them are quite different. In this section, we consider the situation where a complex function f is built through the composition of simpler functions.

We first demonstrate a recipe for building conservative generalizations for compositions of functions. In such a case, it is possible to derive a conservative generalization for f by chaining and combining the generalizations already obtained for the simpler functions it is made of. To ease notation, we shall write $\Pi \Vdash_o \pi$ to indicate that Π is a conservative generalization of all minimal input parts that explain π for input o. We extend the notion of generalization to sets of output parts; for a set of parts Π', we have that $\Pi \Vdash_o \Pi'$ if $\Pi \Vdash_o \pi'$ for every $\pi' \in \Pi'$. We first trivially observe the following:

Theorem 7. *For a given function $f : \mathcal{O} \to \mathcal{O}'$ and a given input $o \in \mathcal{O}$, if $\Pi_1 \Vdash_o \pi_1$ and $\Pi_2 \Vdash_o \pi_2$, then $\Pi_1 \cup \Pi_2 \Vdash_o \{\pi_1, \pi_2\}$.*

Thus, given a set of output parts Π', a conservative generalization can be obtained by taking the union of the generalizations for each individual part in Π'. We can then establish a result for the composition of two functions.

Theorem 8. Let π be an output part of some function f, and a given input $o \in \mathcal{O}$. Let Π_f be a set of sets of parts such that $\Pi_f \Vdash_o \pi$ for function f. Let Π_g be a set of sets of parts such that $\Pi_g \Vdash_{f(o)} \Pi_f$ for some function g. Then $\Pi_g \Vdash_o \pi$ for function $f \circ g$.

Proof. Suppose that there is a set of parts Π that does not intersect with Π_g, and such that for two inputs o and o' that differ on π_g, $\pi(f \circ g)(o) \neq \pi(f \circ g)(o')$. Let $x = g(o)$ and $y = g(o')$; since $\pi(f(x)) \neq \pi(f(y))$, then x and y differ on a set of parts Π'. By definition, a refinement of Π', called Π'', is also a refinement of Π_f. Since $\Pi_g \Vdash_{f(o)} \Pi_f$, this implies that o and o' differ on a part that intersects with Π_g, which contradicts the hypothesis. It follows that all sets of parts of $f \circ g$ that explain o for π intersect with Π_g, and hence $\Pi_g \Vdash_{f(o)} \pi$ for function $f \circ g$. $\square$

Thanks to this result, one can obtain a conservative generalization for $f \circ g$ by first finding a conservative generalization Π of π for g, and then finding a conservative generalization of Π' of Π for f. This spares us from defining an input-output explanation relation for each possible function, at the price of obtaining a conservative approximation of the actual relation. When expressing these explanations as and-or trees, this simply amounts to appending the root of an explanation (the output of a function) to the leaf designating the corresponding input of the function it is composed with.

We recall that one of the claimed features of our proposed approach was tractability. The theorems stated throughout this section give credence to this claim. One can see that, for each of the elementary functions we studied in Sects. 4.2 and 4.3, determining the sets of input parts that are (conservative) explanations of an output part can be done by applying simple rules that require no particular calculation. Then, building an explanation for a composed function is not much harder, and requires properly matching the output parts of a function to the input parts of the one it calls.

5 Implementation and Experiments

Combined, the previous results make it possible to systematically construct explanations for a wide range of computations. It suffices to observe that nested lists-of-lists, coupled with the functions defined in Sect. 4, represent a significant fragment of a functional programming language such as Lisp.

To illustrate this point, the concepts introduced above have been concretely implemented into *Petit Poucet*[3], an open source Java library.[4] The library allows

[3] In English *Hop-o'-My-Thumb*, a fairy tale by French writer Charles Perrault where the main character uses stones to mark a trail that enables him to successfully lead his lost brothers back home.

[4] https://github.com/liflab/petitpoucet. Version 1.0 is considered in this paper.

users to create complex functions by composing the elementary functions studied earlier, to evaluate these functions on inputs, and to generate the corresponding explanation graphs. This library is meant as a proof of concept that serves two goals:1. show the feasibility of our proposed theoretical framework and provide initial results on its running time and memory consumption; 2. provide a test bench allowing us to study the explanation graphs of various functions for various inputs.

5.1 Library Overview

Petit Poucet provides a set of ready-made `Function` objects; in its current implementation, it contains all the functions defined in Sect. 4, in addition to a few others for number comparison, type conversion, descriptive statistics (i.e. average), basic I/O (reading and writing to files) and string and list manipulation. Composed functions are created by adding elementary functions into an object called `CircuitFunction`, and manually connecting the output of each function instance to the input argument of another.

Figure 5 shows a graphical representation of a complex function that can be created by instantiating and composing elementary functions of the library. Each white box represents an elementary function; composition is illustrated by lines connecting the output (dark square) of a function to the input (light square) of another. For functions taking other functions as parameters, namely α_f and ω_f, the parameterized function is represented by a rectangle attached with a dotted line (such as box A attached to box 2). The composed function shown here is exactly the one from our example in the introduction: from a CSV file (box 1), the second element of each line is extracted and cast to a number (boxes 2 and A), the average over a sliding window is taken (boxes 3 and B), each value is checked to be greater than 3 (boxes 4 and C), and the logical conjunction of all these values is taken (box 5).

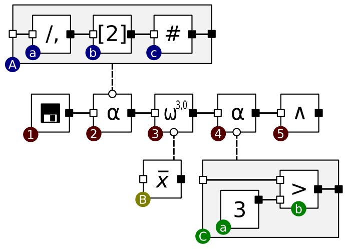


Fig. 5. Evaluating a condition on the average of values over a sliding window.

Once created, a function can be evaluated with input arguments. When this happens, it returns a special object called a `Queryable`. The purpose of the queryable is to retain the information about the function's evaluation necessary to answer "queries" about it at a later time. Each evaluation of the function produces a distinct queryable object with its own memory. Given a designator pointing to a part of the function's output, calling a `Queryable`'s method `query` produces the corresponding explanation and-or tree. Typically, one is interested in a simplified rendition displaying only the root, leaves and Boolean nodes, hiding the intermediate nodes made of the input/output of the intermediate functions in the explanation. On the CSV file shown in the introduction, the library produces the tree that is shown in Fig. 1.[5]

One can see that the `false` result produced by the function admits three alternate explanations (the three sub-trees under the "or" node). The first explanation involves the numerical values in lines 3-4-5 (children of the first "and" node), the second includes lines 4-5-6, and the third explanation is made of the numbers in lines 5-6-7. This corresponds exactly to the three windows of successive value whose average is not greater than 3. Indeed, the presence of either of these three windows, and nothing else, suffices for our global condition to evaluate to false. Note how the explanation generation mechanism correctly and automatically identifies these, and also how, thanks to our concept of designator, the explanation can refer to specific locations inside specific lines of the input object.

In Petit Poucet, lineage capabilities are *built-in*. The user is not required to perform any special task in order to keep track of provenance information. In addition, it shall be noted that one does not need to declare in advance what designation graph will be asked for. The construction of the `Queryable` objects is the same, regardless of the output part being used as the starting point. Finally, the library follows a modular architecture where the set of available functions can easily be extended by creating packages defining new descendants of `Function`. It suffices for each function to produce a `Queryable` object that computes its specific input-output relationships; an explanation can then be computed for any composed function that uses it.

5.2 Experiments

To test the performance of the library, we selected various data processing tasks and implemented them as composed functions in Petit Poucet.

Get All Numbers. Represents a simple operation that takes an input comma-separated list of elements, and produces a vector containing only the elements of the file that are numerical values. The explanation we ask is to point at a given element of the output vector, and retrace the location in the input string that corresponds to this number.

Sliding Window Average. Given a CSV file, this task extracts the numerical value in each line, compute the average of each set of n successive values and check that it is below some threshold t (similar to the example we discussed

[5] Or more precisely a directed acyclic graph, since leaf nodes with the same designator are not repeated.

earlier). Computing an aggregation over a sliding window is a common task in the field of event stream processing [26] and runtime verification [4], and is also provided by most statistical software, such as R's `smooth` package. It can be seen as a basic form of trend deviation detection [41], where the end result of the calculation is an "alarm" indicating that the expected trend has not been followed across the whole data file; a classical example of this is the detection of temperature peaks in a server rack [40]. The explanation we ask is to point at the output Boolean value (true or false), and retrace the locations in the file corresponding to the numbers explaining the result.

Triangle Areas. Given a list of arbitrary vectors, this task checks that each vector contains the lengths of the three sides of a valid triangle; if so, it computes their area using Heron's formula[6] and sums the area of all valid triangles. It was chosen because it involves multiple if-then-else cases to verify the sides of a triangle. It also involves a slightly more involved arithmetical calculation to get the area, which is completely implemented by composing basic arithmetic operators in a composed function. The whole function is the composition of 59 elementary functions.

This example is notable, because the explanations it generates may take different forms. An explanation for the output value (total area) includes an explanation for each vector: if it represents a valid triangle, it refers to its three sides; if it does not, the condition can fail for different reasons: the vector may not have three elements, or have one of its elements that is not a number, or contain a negative value, or violate the triangle inequality. Each condition, when violated, produces different explanations pointing at different elements of the vector, or the vector as a whole. Moreover, each vector in the list may not be a valid triangle for different reasons, and hence a different explanation will be built for each of them. For example, given the list $[\langle a, 4, 2\rangle, \langle 3, 5, 6\rangle, \langle 2, 3\rangle]$, a tree will be produced that describes an explanation involving three elements: the first points at the element a of the first vector (not a number), the second points at all three components of the second vector (valid triangle), and the last points at the whole third vector (wrong number of elements).

Nested Bounding Boxes. Given a DOM tree[7], this task checks that each element has a bounding box (width and height) larger than all of its children. This condition is the symptom of a layout bug which shows visually as an element protruding from its parent box inside the web browser's window. This corresponds to one of the properties that is evaluated by web testing tools such as Cornipickle [24] and ReDeCheck [49] on real web pages. In this task, trees are represented as nested lists-of-lists, with each DOM node corresponding to a triplet made of its width, height, and a list of its children nodes. The explanation we ask is to point at the Boolean output of the condition, and retrace the nodes of the tree that violate it.

[6] $A = \sqrt{s(s-a)(s-b)(s-c)}$, where $s = \frac{a+b+c}{2}$.

[7] A DOM tree represents the structure of elements in an HTML document [2].

In all these tasks, the inputs given to the function are randomly generated structures of the corresponding type. The experiments were implemented using the LabPal testing framework [25], which makes it possible to bundle all the necessary code, libraries and input data within a single self-contained executable file, such that anyone can download and independently reproduce the experiments. A downloadable lab instance containing all the experiments of this paper can be obtained online [27]. Overall, our empirical measurements involve 56 individual experiments, which together generated 224 distinct data points. All the experiments were run on a AMD Athlon II X4 640 1.8 GHz running Ubuntu 18.04, inside a Java 8 virtual machine with 3566 MB of memory.

Memory Consumption. The first experiment aims to measure the amount of memory used up by the `Queryable` objects generated by the evaluation of a function, and the impact of the size of the input on global memory consumption. To this end, we ran various functions on inputs of different size; for each, we measured the amount of memory consumed, with explainability successively turned on and off. This is possible thanks to a switch provided by Petit Poucet, and which allows users to completely disable tracking if desired.

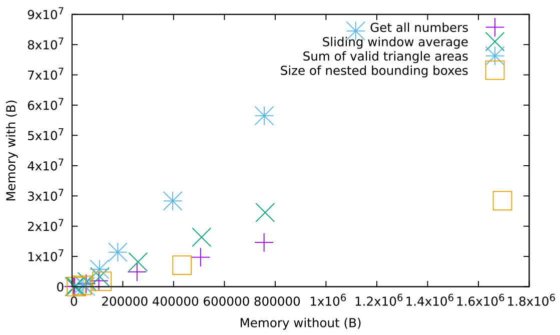


Fig. 6. Impact of explainability on memory consumption.

Figure 6 shows a plot that compares the amount of memory consumed by `Function` objects. Each point in the plot corresponds to a pair of experiments: the x coordinate corresponds to the memory consumed by a function without explainability, and the y coordinate corresponds to the memory consumed by the same function on the same input, but with explainability enabled. All the points for the same task have been grouped into a category and are given the same color.

Analyzing this plot brings both bad news and good news. The "bad" news is that the additional memory required for explainability is high when expressed in relative terms. For example, a composed that requires 498 KB to be evaluated on an input requires close to 15 MB once explainability is enabled. The "good" news is twofold. First, this consumption is still reasonable in the absolute: at this rate, it takes an input file of 42 million lines before filling up the available RAM in a 64 GB machine. Second, and most importantly, the relationship between memory consumption with and without lineage is linear: for all the tasks we tested, if m is the memory used without lineage, then the memory m' used when lineage is enabled is in $O(m)$, i.e. the ratio m'/m does not depend on the size of the input.

These figures should be put in context by comparing the overhead incurred by other systems mentioned in Sect. 2. Related systems for provenance in databases (namely Polygen [51], MONDRIAN [20], MXQL [48], DBNotes [11] pSQL [7] and ORCHESTRA [29]) do not divulge their storage overhead for provenance data. A recent technical report on a provenance-aware database management system measures an overhead ranging between 19% and 702% [3]. Dynamic taint propagation systems report a memory overhead reaching 4× for TaintCheck [37], 240× [14] for Dytan, and "an enormity" of logging information for RIFLE ([13], authors' quote). Although these systems operate at a different level of abstraction, this shows that explainability is inherently costly regardless of the approach chosen.

Computation Time. We performed the same experiments, but this time measuring computation time. The results are shown in Fig. 7; similar to memory consumption, they compare the running time of the same function on the same input, both with and without explanation tracking. The largest slowdown observed across all instances is 6.7×. For a task like *Sliding Window Average*, the average slowdown observed is 1.93× across all inputs. Although this slowdown is non-negligible, it is reasonable nevertheless, adding at most a few hundreds of milliseconds on the problems considered in our benchmark.

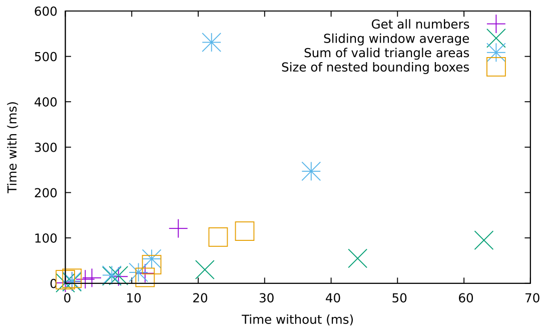


Fig. 7. Impact of explainability on computation time.

Again, these results should be put in context with respect to existing works that include a form of lineage. The MONDRIAN system reports an average slowdown of 3×; pSQL ranges between 10× and 1,000×; the remaining tools do not report CPU overhead. For taint analysis tools, Dytan reports a 30–50× slowdown; GIFT-compiled programs are slowed down by up to 12×; TaintCheck has a slowdown of around 20×, 1–2× for Rifle and around 20× for TaintCheck. Of course, these various systems compute different types of lineage information, but these figures give an outlook of the order of magnitude one should expect from such systems.

6 Conclusion

This paper provided the formal foundations for a generic and granular explainability framework. An important highlight of this model is its capability to handle abstract composite data structures, including character strings or lists of elements. The paper then defined the notion of designator, which are functions that can point to and extract *parts* of these data structures. An explainability relationship on functions has been formally defined, and conservative approximations of this relation have been proved for a set of elementary functions. A point in favor of this approach is that explanations of composed functions can be built by composing the explanations for elementary functions. Combined, these concepts make it possible to automatically extract the explanation of a result for generic functions at a fine level of granularity. These concepts have been implemented into a proof-of-concept, yet fully functional library called *Petit Poucet*, and evaluated experimentally on a number of data processing tasks. These experiments revealed that the amount of memory required to track explainability metadata is relatively high, but more importantly, showed that it is *linear* in the size of the memory required to evaluate the function in the first place.

Obviously, Petit Poucet is not intended to replace programs written using other languages and following different paradigms. However, it could be used as a library by other tools that could benefit from its explanation features. In particular, testing libraries such as JUnit could be extended by assertions written as Petit Poucet functions, and provide a detailed explanation of a test failure without requiring extra code. Explainability functionalities could also easily be retrofitted into existing (Java) software, with minimal interference on their current code. Case in point, we already identified the Cornipickle web testing tool [24] and the BeepBeep event stream processing engine [26] as some of the first targets for the addition of explainability based on Petit Poucet. A lineage-aware version of the GRAL plotting library[8] is also considered.

The existence of a definition of fine-grained explainability opens the way to multiple exciting theoretical questions. For example: for a given function, is there a part of the input that is present in all explanations? We can see an example of this in Fig. 1, with the leaf pointing to value −80. Intuitively, this tends to indicate that some parts of an input have a greater "responsibility" than others in the result, and could provide an alternate way of quantifying this notion than what has been studied so far [12]. On the contrary, is there a part of the input that never explains the production of the output, regardless of the input? This latter question could shed a different light on an existing notion called *vacuity* [6], expressed not in terms of elements of the specification, but on the parts of

[8] https://github.com/eseifert/gral.

the input it is evaluated on. More generally, explainability can be viewed as a particular form of static analysis for functions; it would therefore be interesting to recast our model in the abstract interpretation framework [35,39] in order to further assess its strengths and weaknesses.

Finally, explanations could also prove useful from a testing and verification standpoint. The explanation graph could be used for log trace and bug triaging [30]: if two execution traces violate the same condition, one could keep one trace instance for each distinct explanation they induce, as representatives of traces that fail for different reasons. This could help reduce the amount of log data that needs to be preserved, by keeping only one log instance of each type of failure.

References

1. Aleksandrowicz, G., Chockler, H., Halpern, J.Y., Ivrii, A.: The computational complexity of structure-based causality. J. Artif. Intell. Res. **58**, 431–451 (2017)
2. Apparao, V., et al.: Document object model (DOM) level 1 specification. Technical report, World Wide Web Consortium (1998). https://www.w3.org/DOM/. Accessed 17 Nov 2019
3. Arab, B., Gawlick, D., Krishnaswamy, V., Radhakrishnan, V., Glavic, B.: Formal foundations of reenactment and transaction provenance. Technical Report IIT/CS-DB-2016-01, Illinois Institute of Technology (2016)
4. Basin, D.A., Klaedtke, F., Marinovic, S., Zalinescu, E.: Monitoring of temporal first-order properties with aggregations. Formal Methods Syst. Des. **46**(3), 262–285 (2015)
5. Beer, I., Ben-David, S., Chockler, H., Orni, A., Trefler, R.J.: Explaining counterexamples using causality. Formal Methods Syst. Des. **40**(1), 20–40 (2012)
6. Ben-David, S., Copty, F., Fisman, D., Ruah, S.: Vacuity in practice: temporal antecedent failure. Formal Methods Syst. Des. **46**(1), 81–104 (2015). https://doi.org/10.1007/s10703-014-0221-0
7. Bhagwat, D., Chiticariu, L., Tan, W.C., Vijayvargiya, G.: An annotation management system for relational databases. VLDB J. **14**(4), 373–396 (2005). https://doi.org/10.1007/s00778-005-0156-6
8. Buneman, P., Khanna, S., Wang-Chiew, T.: Why and where: a characterization of data provenance. In: Van den Bussche, J., Vianu, V. (eds.) ICDT 2001. LNCS, vol. 1973, pp. 316–330. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44503-X_20
9. Cheney, J., Chiticariu, L., Tan, W.-C.: Provenance in databases: why, how, and where. Found. Trends Databases **1**(4), 379–474 (2007)
10. Chiticariu, L., Tan, W.C.: Debugging schema mappings with routes. In: Dayal, U., et al. (eds.) Proceedings of the VLDB 2006, pp. 79–90. ACM (2006)
11. Chiticariu, L., Tan, W.C., Vijayvargiya, G.: DBNotes: a post-it system for relational databases based on provenance. In: Özcan, F. (ed.) Proceedings of the SIGMOD 2005, pp. 942–944. ACM (2005)
12. Chockler, H., Halpern, J.Y.: Responsibility and blame: A structural-model approach. J. Artif. Intell. Res. **22**, 93–115 (2004)

13. Chow, J., Pfaff, B., Garfinkel, T., Christopher, K., Rosenblum, M.: Understanding data lifetime via whole system simulation. In: Blaze, M. (ed.) Proceedings of the USENIX Security 2004, pp. 321–336. USENIX (2004)
14. Clause, J.A., Li, W., Orso, A.: Dytan: a generic dynamic taint analysis framework. In: Rosenblum, D.S., Elbaum, S.G. (eds.) Proceedings of the ISSTA 2007, pp. 196–206. ACM (2007)
15. Cleve, H., Zeller, A.: Locating causes of program failures. In: Roman, G., Griswold, W.G., Nuseibeh, B. (eds.) Proceedings of the ICSE 2005, pp. 342–351. ACM (2005)
16. Crandall, J.R., Chong, F.T.: Minos: control data attack prevention orthogonal to memory model. In: Proceedings of the MICRO-37, pp. 221–232. IEEE Computer Society (2004)
17. Cui, Y., Widom, J., Wiener, J.L.: Tracing the lineage of view data in a warehousing environment. ACM Trans. Database Syst. **25**(2), 179–227 (2000)
18. Eiter, T., Lukasiewicz, T.: Complexity results for structure-based causality. Artif. Intell. **142**(1), 53–89 (2002)
19. Ferrère, T., Maler, O., Ničković, D.: Trace diagnostics using temporal implicants. In: Finkbeiner, B., Pu, G., Zhang, L. (eds.) ATVA 2015. LNCS, vol. 9364, pp. 241–258. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24953-7_20
20. Geerts, F., Kementsietsidis, A., Milano, D.: MONDRIAN: annotating and querying databases through colors and blocks. In: Liu, L., Reuter, A., Whang, K., Zhang, J. (eds.) Proceedings of the ICDE 2006, pp. 82. IEEE Computer Society (2006)
21. Green, T.J., Karvounarakis, G., Tannen. Provenance semirings. In: Libkin, L. (ed.) Proceedings of the PODS 2007, pp. 31–40. ACM (2007)
22. Groce, A., Chaki, S., Kroening, D., Strichman, O.: Error explanation with distance metrics. STTT **8**(3), 229–247 (2006)
23. Hallé, S.: Causality in message-based contract violations: a temporal logic "whodunit". In: Proceedings of the EDOC 2011, pp. 171–180. IEEE Computer Society (2011)
24. Hallé, S., Bergeron, N., Guérin, F., Le Breton, G., Beroual, O.: Declarative layout constraints for testing web applications. J. Log. Algebraic Meth. Program. **85**(5), 737–758 (2016)
25. Hallé, S., Khoury, R., Awesso, M.: Streamlining the inclusion of computer experiments in a research paper. IEEE Comput. **51**(11), 78–89 (2018)
26. Hallé, S.: Event Stream Processing With BeepBeep 3: Log Crunching and Analysis Made Easy. Presses de l'Université du Québec (2018)
27. Hallé, S., Tremblay, H.: Measuring the impact of lineage tracking in the Petit Poucet library (version v1.0), April 2021
28. Halpern, J.Y., Pearl, J.: Causes and explanations: a structural-model approach, part I: causes. Brit. J. Philos. Sci. **56**(4), 843–887 (2005)
29. Karvounarakis, G., Ives, Z.G., Tannen, V.: Querying data provenance. In: Elmagarmid, A.K., Agrawal, D. (eds.) Proceedings of the SIGMOD 2010, pp. 951–962. ACM (2010)
30. Khoury, R., Gaboury, S., Hallé, S.: Three views of log trace triaging. In: Cuppens, F., Wang, L., Cuppens-Boulahia, N., Tawbi, N., Garcia-Alfaro, J. (eds.) FPS 2016. LNCS, vol. 10128, pp. 179–195. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-51966-1_12
31. Kupferman, O., Vardi, M.Y.: Model checking of safety properties. Formal Methods Syst. Des. **19**(3), 291–314 (2001). https://doi.org/10.1023/A:1011254632723
32. Lam, L., Chiueh, T.-C.: A General dynamic information flow tracking framework for security applications. In Proceedings of the ACSAC 2006, pp. 463–472, Miami Beach, FL, USA, December 2006. IEEE

33. Leek, T., Brown, R., Zhivich, M., Lippmann, R.: Coverage maximization using dynamic taint tracing. Technical Report 1112, Massachusetts Institute of Technology (2007)
34. McCamant, S., Ernst, M.D.: Quantitative information flow as network flow capacity. In: Gupta, R., Amarasinghe, S.P. (eds.) Proceedings of the PLDI 2008, pp. 193–205. ACM (2008)
35. Møller, A., Schwartzbach, M.I.: Static program analysis, October 2018. Department of Computer Science, Aarhus University. http://cs.au.dk/~amoeller/spa/
36. Mukherjee, S., Dasgupta, P.: Computing minimal debugging windows in failure traces of AMS assertions. IEEE Trans. CAD Integr. Circ. Syst. **31**(11), 1776–1781 (2012)
37. Newsome, J., Song, D.X.: Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In: Proceedings of the NDSS 2005. The Internet Society (2005)
38. Pérez, B., Rubio, J., Sáenz-Adán, C.: A systematic review of provenance systems. Knowl. Inf. Syst. **57**(3), 495–543 (2018). https://doi.org/10.1007/s10115-018-1164-3
39. Rival, X., Yi, K.: Introduction to Static Analysis: An Abstract Interpretation Perspective. MIT Press, Cambridge (2020)
40. Rohrmann, T.: Introducing complex event processing (CEP) with Apache Flink, 2016. https://flink.apache.org/news/2016/04/06/cep-monitoring.html. Accessed 17 Nov 2019
41. Roudjane, M., Rebaïne, D., Khoury, R., Hallé, S.: Detecting trend deviations with generic stream processing patterns. Inf. Syst. **101446**, 1–24 (2019). https://doi.org/10.1016/j.is.2019.101446
42. Samek, W., Wiegand, T., Müller, K.-R.: Explainable artificial intelligence: understanding, visualizing and interpreting deep learning models. ITU J. **1** (2017). arXiv: 1708.08296
43. Schuppan, V., Biere, A.: Shortest counterexamples for symbolic model checking of LTL with past. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 493–509. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31980-1_32
44. Suh, G.E., Lee, J.W., Zhang, D., Devadas, S.: Secure program execution via dynamic information flow tracking. In: Mukherjee, S., McKinley, K.S., (eds.) Proceedings of the ASPLOS 2004, pp. 85–96. ACM (2004)
45. Vachharajani, N., et al.: RIFLE: an architectural framework for user-centric information-flow security. In: Proceedings of the MICRO-37, pp. 243–254. IEEE Computer Society (2004)
46. van Zelst, S.J., Bolt, A., Hassani, M., van Dongen, B.F., van der Aalst, W.M.P.: Online conformance checking: relating event streams to process models using prefix-alignments. Int. J. Data Sci. Anal. **8**(3), 269–284 (2019). https://doi.org/10.1007/s41060-017-0078-6
47. Vandebogart, S., et al.: Labels and event processes in the Asbestos operating system. ACM Trans. Comput. Syst. **25**(4), 11 (2007)
48. Velegrakis, Y., Miller, R.J., Mylopoulos, J.: Representing and querying data transformations. In: Aberer, K., Franklin, M.J., Nishio, S. (eds.) Proceedings of the ICDE 2005, pp. 81–92. IEEE Computer Society (2005)
49. Walsh, T.A., McMinn, P., Kapfhammer, G.M.: Automatic detection of potential layout faults following changes to responsive web pages. In: Proceedings of the ASE 2015, pp. 709–714. ACM (2015)

50. Wang, C., Yang, Z., Ivančić, F., Gupta, A.: Whodunit? Causal analysis for counterexamples. In: Graf, S., Zhang, W. (eds.) ATVA 2006. LNCS, vol. 4218, pp. 82–95. Springer, Heidelberg (2006). https://doi.org/10.1007/11901914_9
51. Wang, Y.R., Madnick, S.E.: A polygen model for heterogeneous database systems: the source tagging perspective. In: McLeod, D., Sacks-Davis, R., Schek, H. (eds.) Proceedings of the VLDB 1990, pp. 519–538. Morgan Kaufmann (1990)

Latticed k-Induction with an Application to Probabilistic Programs

Kevin Batz[1](✉), Mingshuai Chen[1](✉), Benjamin Lucien Kaminski[2](✉), Joost-Pieter Katoen[1](✉), Christoph Matheja[3](✉), and Philipp Schröer[1]

[1] RWTH Aachen University, Aachen, Germany
{kevin.batz,chenms,katoen}@cs.rwth-aachen.de
[2] University College London, London, UK
b.kaminski@ucl.ac.uk
[3] ETH Zürich, Zürich, Switzerland
cmatheja@inf.ethz.ch

Abstract. We revisit two well-established verification techniques, *k-induction* and *bounded model checking* (BMC), in the more general setting of fixed point theory over complete lattices. Our main theoretical contribution is *latticed k-induction*, which (i) generalizes classical k-induction for verifying transition systems, (ii) generalizes Park induction for bounding fixed points of monotonic maps on complete lattices, and (iii) extends from naturals k to transfinite ordinals κ, thus yielding *κ-induction*.

The lattice-theoretic understanding of k-induction and BMC enables us to apply both techniques to the *fully automatic verification of infinite-state probabilistic programs*. Our prototypical implementation manages to automatically verify non-trivial specifications for probabilistic programs taken from the literature that—using existing techniques—cannot be verified without synthesizing a stronger inductive invariant first.

Keywords: k-induction · Bounded model checking · Fixed point theory · Probabilistic programs · Quantitative verification

1 Introduction

Bounded model checking (BMC) [12,17] is a successful method for analyzing models of hardware and software systems. For checking a *finite-state* transition system (TS) against a safety property ("bad states are unreachable"), BMC unrolls the transition relation until it either finds a counterexample and hence refutes the property, or reaches a pre-computed completeness threshold on the unrolling depth and accepts the property as verified. For *infinite-state* systems, however, such completeness thresholds need not exist (cf. [64]), rendering BMC a *refutation-only* technique. To *verify* infinite-state systems, BMC is typically combined with the search for an *inductive invariant*, i.e., a superset of the reachable

This work has been partially funded by the ERC Advanced Project FRAPPANT under grant No. 787914.

states which is closed under the transition relation. Proving a—not necessarily inductive—safety property then amounts to *synthesizing* a sufficiently strong, often complicated, inductive invariant that excludes the bad states. A plethora of techniques target computing or approximating inductive invariants, including IC3 [14], induction [13,20], interpolation [50,51], and predicate abstraction [27,36]. However, invariant synthesis may burden full automation, as it either relies on user-supplied annotations or confines push-button technologies to semi-decision or approximate procedures.

k-induction [65] generalizes the principle of simple induction (aka 1-induction) by considering k consecutive transition steps instead of only a single one. It is more powerful: an invariant can be k-inductive for some $k > 1$ but not 1-inductive. Following the seminal work of Sheeran et al. [65] which combines k-induction with SAT solving to check safety properties, k-induction has found a broad spectrum of applications in the realm of hardware [29,37,45,65] and software verification [10,21–23,55,63]. Its success is due to (1) being a foundational yet potent reasoning technique, and (2) integrating well with SAT/SMT solvers, as also pointed out in [45]: "*the simplicity of applying k-induction made it the go-to technique for SMT-based infinite-state model checking*". This paper explores whether k-induction can have a similar impact on the *fully automatic verification* of infinite-state *probabilistic programs*. That is, we aim to verify that the *expected value* of a specified *quantity*—think: "quantitative postcondition"—after the execution of a probabilistic program is bounded by a specified threshold.

Example 1 (Bounded Retransmission Protocol [19,32]*).* The loop

```
while ( $sent < toSend \wedge fail < maxFail$ ) {
    { $fail := 0$ ; $sent := sent + 1$ } [0.9] { $fail := fail + 1$ ; $totalFail := totalFail + 1$ }
}
```

models a simplified version of the bounded retransmission protocol, which attempts to transmit *toSend* packages via an unreliable channel (that fails with probability 0.1) allowing for at most *maxFail* retransmissions per package.

Using our generalization of k-induction, we can fully automatically verify that the *expected total number of failed transmissions* is at most 1, if the number of packages we want to (successfully) send is at most 3. In terms of weakest preexpectations [38,44,49], this quantitative property reads

$$\mathsf{wp}[\![C]\!]\,(\mathit{totalFail}) \quad\preceq\quad [\mathit{toSend} \leq 3] \cdot (\mathit{totalFail} + 1) + [\mathit{toSend} > 3] \cdot \infty .$$

The bound on the right-hand-side of the inequality is 4-inductive, but *not* 1-inductive; verifying the same bound using 1-induction requires finding a non-trivial—and far less perspicuous—inductive invariant. Moreover, if we consider an arbitrary number of packages to send, i.e., we drop $[\mathit{toSend} \leq 3]$, this bound becomes invalid. In this case, our BMC procedure produces a counterexample, i.e., values for *toSend* and *maxFail*, proving that the bound does not hold. $\triangleleft$

Lifting the classical formalization (and SAT encoding) of k-induction over TSs to the probabilistic setting is non-trivial. We encounter the following challenges:

(A) *Quantitative reachability.* In a TS, a state reachable within k steps remains reachable on increasing k. In contrast, reachability *probabilities* in Markov chains—a common operational model for probabilistic programs [28]—may increase on increasing k. Hence, proving that the probability of reaching a bad state remains below a given threshold is more intricate than reasoning about qualitative reachability.

(B) *Counterexamples are subsystems.* In a TS, an acyclic path from an initial to a bad state suffices as a witness for refuting safety, i.e., non-reachability. SAT encodings of k-induction rely on this by expressing the absence of witnesses up to a certain path-length. In the probabilistic setting, however, witnesses are no longer single paths [30]. Rather, a witness for the probability of reaching a bad state to exceed a threshold is a *subsystem* [15], i.e., a set of possibly cyclic paths.

(C) *Symbolic encodings.* To enable fully automated verification, we need a suitable encoding such that our lifting integrates well into SMT solvers. Verifying probabilistic programs involves reasoning about execution *trees*, where each (weighted) branch corresponds to a probabilistic choice. A suitable encoding needs to capture such trees which requires more involved theories than encoding paths in classical k-induction.

We address challenges (A) and (B) by developing *latticed k-induction*, which is a proof technique in the rather general setting of fixed point theory over complete lattices. Latticed k-induction generalizes classical k-induction in three aspects: (1) it works with any monotonic map on a complete lattice instead of being confined to the transition relation of a transition system, (2) it generalizes the Park induction principle for bounding fixed points of such monotonic maps, and (3) it extends from natural numbers k to (possibly transfinite) ordinals κ, hence its short name: *κ-induction.*

It is this lattice-theoretic understanding that enables us to lift both k-induction and BMC to reasoning about quantitative properties of probabilistic programs. To enable *automated* reasoning, we address challenge (C) by an incremental SMT encoding based on the theory of quantifier-free mixed integer and real arithmetic with uninterpreted functions (QF_UFLIRA). We show how to effectively compute all needed operations for κ-induction using the SMT encoding and, in particular, how to decide *quantitative entailments.*

A prototypical implementation of our method demonstrates that κ-induction for (linear) probabilistic programs manages to automatically verify non-trivial specifications for programs taken from the literature which—using existing techniques—cannot be verified without synthesizing a stronger inductive invariant.

Due to space restrictions, most proofs and details about individual benchmarks have been omitted; they are found in an extended version of this paper [8].

Related Work. Besides the aforementioned related work on k-induction, we briefly discuss other automated analysis techniques for probabilistic systems and other approaches for bounding fixed points. Symbolic engines exist for exact inference [26] and sensitivity analysis [33]. Other automated approaches focus on bounding expected costs [56], termination analysis [2,16], and static analysis [3,67]. BMC has been applied in a rather rudimentary form to the on-the-fly verification of finite unfoldings of probabilistic programs [35], and the enumerative generation of counterexamples in finite Markov chains [68]. (Semi-)automated invariant-synthesis techniques can be found in [6,24,41]. A recent variant of IC3 for probabilistic programs called PrIC3 [7] is restricted to finite-state systems. When applied to finite-state Markov chains, our κ-induction operator is related to other operators that have been employed for determining reachabilitiy probabilities through value iteration [4,31,61]. In particular, when iterated on the candidate upper bound, the κ-induction operator coincides with the (upper value iteration) operator in interval iteration [4]; the latter operator can be used together with the up-to techniques (cf. [53,58,59]) to prove our κ-induction rule sound (in contrast, we give an elementary proof). However, the κ-induction operator avoids comparing current and previous iterations. It is thus easier to implement and more amenable to SMT solvers. Finally, the proof rules for bounding fixed points recently developed in [5] are restricted to finite-state systems.

2 Verification as a Fixed Point Problem

We start by recapping some fundamentals on fixed points of monotonic operators on complete lattices before we state our target verification problem.

Fundamentals. For the next three sections, we fix a *complete lattice* $(E, \sqsubseteq)$, i.e. a carrier set E together with a partial order $\sqsubseteq$, such that every subset $S \subseteq E$ has a *greatest lower bound* $\bigsqcap S$ (also called the *meet* of S) and a *least upper bound* $\bigsqcup S$ (also called the *join* of S). For just two elements $\{g, h\} \subseteq E$, we denote their meet by $g \sqcap h$ and their join by $g \sqcup h$. Every complete lattice has a *least* and a *greatest* element, which we denote by $\bot$ and $\top$, respectively.

In addition to $(E, \sqsubseteq)$, we also fix a *monotonic operator* $\Phi\colon E \to E$. By the Knaster-Tarski theorem [43,47,66], every monotonic operator Φ admits a *complete lattice of (potentially infinitely many) fixed points.* The least fixed point $\mathsf{lfp}\ \Phi$ and the greatest fixed point $\mathsf{gfp}\ \Phi$ are moreover constructible by (possibly transfinite) *fixed point iteration* from $\bot$ and $\top$, respectively: Cousot & Cousot [18] showed that there exist ordinals α and β, such that[1]

$$\mathsf{lfp}\ \Phi \;=\; \Phi^{\lceil \alpha \rceil}(\bot) \quad \text{and} \quad \mathsf{gfp}\ \Phi \;=\; \Phi^{\lfloor \beta \rfloor}(\top)\,, \tag{†}$$

where $\Phi^{\lceil \delta \rceil}(g)$ denotes the *upper δ-fold iteration* and $\Phi^{\lfloor \delta \rfloor}(g)$ denotes the *lower δ-fold iteration* of Φ on g, respectively. Formally, $\Phi^{\lceil \delta \rceil}(g)$ is given by[2]

[1] We use lowercase greek letters α, β, γ, δ, etc. to denote arbitrary (possibly transfinite) ordinals and i, j, k, m, n, etc. to denote natural (finite) numbers in $\mathbb{N}$.

[2] To ensure well-definedness of transfinite iterations, we fix an *ambient ordinal* ν and *tacitly assume $\delta < \nu$ for all ordinals δ considered throughout this paper.* Formally, ν is the smallest ordinal such that $|\nu| > |E|$. Intuitively, ν then upper-bounds the length of any repetition-free sequence over elements of E.

$$\Phi^{\lceil\delta\rceil}(g) \;=\; \begin{cases} g & \text{if } \delta = 0, \\ \Phi\left(\Phi^{\lceil\gamma\rceil}(g)\right) & \text{if } \delta = \gamma + 1 \text{ is a successor ordinal,} \\ \bigsqcup\left\{\Phi^{\lceil\gamma\rceil}(g) \;\middle|\; \gamma < \delta\right\} & \text{if } \delta \text{ is a limit ordinal.} \end{cases}$$

Intuitively, if δ is the successor of γ, then we simply do another iteration of Φ. If δ is a limit ordinal, then $\Phi^{\lceil\delta\rceil}(g)$ can also be thought of as a limit, namely of iterating Φ on g. However, simply iterating Φ on g need not always converge, especially if the iteration does not yield an ascending chain. To remedy this, we take as limit the join over the whole (possibly transfinite) iteration sequence, i.e., the least upper bound over all elements that occur along the iteration. The lower δ-fold iteration $\Phi^{\lfloor\delta\rfloor}(g)$ is defined analogously to $\Phi^{\lceil\delta\rceil}(g)$, except that we take a meet instead of a join whenever δ is a limit ordinal.

An important special case for fixed point iteration (see (†)) is when the operator Φ is *Scott-continuous* (or simply *continuous*), i.e., if $\Phi\left(\bigsqcup\{g_1 \sqsubseteq g_2 \sqsubseteq \ldots\}\right) = \bigsqcup\Phi\left(\{g_1 \sqsubseteq g_2 \sqsubseteq \ldots\}\right)$. In this case, α in (†) coincides with the first infinite limit ordinal ω (which can be identified with the set $\mathbb{N}$ of natural numbers). This fact is also known as the Kleene fixed point theorem [1].

Problem Statement. Fixed points are ubiquitous in computer science. Prime examples of properties that can be conveniently characterized as least fixed points include both the set of reachable states in a transition system and the function mapping each state in a Markov chain to the probability of reaching some goal state (cf. [60]). However, least and greatest fixed points are often difficult or even impossible [39] to compute; it is thus desirable to *bound* them.

For example, it may be sufficient to prove that a system modeled as a Markov chain reaches a bad state from its initial state with probability *at most* 10^{-6}, instead of computing *precise* reachability probabilities for each state. Moreover, if said probability is *not* bounded by 10^{-6}, we would like to witness that as well.

In general lattice-theoretic terms, our problem statement reads as follows:

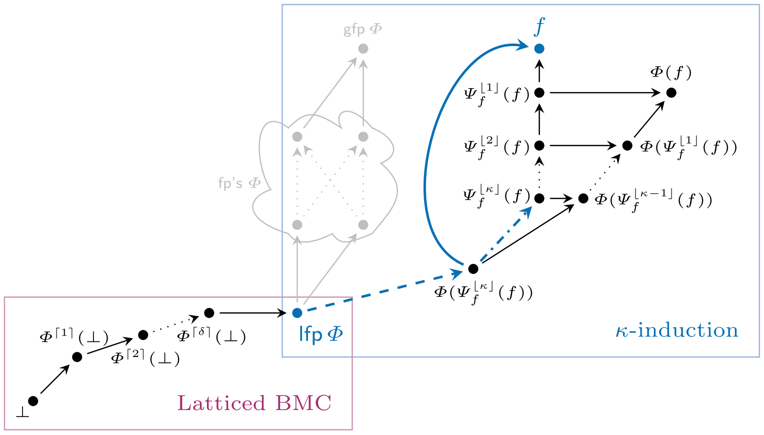
Given a complete lattice $(E, \sqsubseteq)$, a monotonic operator $\Phi\colon E \to E$,
and a candidate upper bound $f \in E$ on $\mathsf{lfp}\ \Phi$,

prove or *refute* that $\mathsf{lfp}\ \Phi \sqsubseteq f$.

For *proving*, we will present *latticed k-induction*; for *refuting*, we will present *latticed bounded model checking*. Running both in parallel may (and under certain conditions: *will*) lead to a decision of the above problem.

3 Latticed *k*-Induction

In this section, we generalize the well-established k-induction verification technique [23,29,37,45,55,65] to *latticed k-induction* (for short: κ*-induction*; reads:



gfp Φ
fp's Φ
f
$\Phi(f)$
$\Psi_f^{\lfloor 1 \rfloor}(f)$
$\Psi_f^{\lfloor 2 \rfloor}(f)$
$\Phi(\Psi_f^{\lfloor 1 \rfloor}(f))$
$\Psi_f^{\lfloor \kappa \rfloor}(f)$
$\Phi(\Psi_f^{\lfloor \kappa-1 \rfloor}(f))$
$\Phi(\Psi_f^{\lfloor \kappa \rfloor}(f))$
lfp Φ
κ-induction
$\Phi^{\lceil 1 \rceil}(\bot)$
$\Phi^{\lceil 2 \rceil}(\bot)$
$\Phi^{\lceil \delta \rceil}(\bot)$
$\bot$
Latticed BMC

Fig. 1. κ-induction and latticed BMC in case that $\mathsf{lfp}\ \Phi \sqsubseteq f$. An arrow from g to h indicates $g \sqsubseteq h$. The solid blue arrow from $\Phi(\Psi_f^{\lfloor \kappa \rfloor}(f))$ to f is the premise of κ-induction, i.e., the LHS of Lemma 2, which implies the dash-dotted blue arrow from $\Phi(\Psi_f^{\lfloor \kappa \rfloor}(f))$ to $\Psi_f^{\lfloor \kappa \rfloor}(f)$, i.e., the RHS of Lemma 2. The dashed blue arrow from $\mathsf{lfp}\ \Phi$ to $\Phi(\Psi_f^{\lfloor \kappa \rfloor}(f))$ is a consequence of the dash-dotted arrow (by Park induction, Theorem 1) and ultimately proves that $\mathsf{lfp}\ \Phi \sqsubseteq f$.

"kappa induction"). With κ-induction, our aim is to *prove* that $\mathsf{lfp}\ \Phi \sqsubseteq f$. To this end, we attempt "ordinary" induction, also known as *Park induction*:

Theorem 1 (Park Induction [57]). *Let $f \in E$. Then*

$$\Phi(f) \sqsubseteq f \quad \text{implies} \quad \mathsf{lfp}\ \Phi \sqsubseteq f.$$

Intuitively, this principle says: if pushing our candidate upper bound f through Φ takes us *down* in the partial order $\sqsubseteq$, we have verified that f is indeed an upper bound on $\mathsf{lfp}\ \Phi$. The true power of Park induction is that applying Φ *once* tells us something about iterating Φ possibly *transfinitely often* (see (†) in Sect. 2).

Park induction, unfortunately, does *not* work in the reverse direction: If we are unlucky, $f \sqsupseteq \mathsf{lfp}\ \Phi$ *is* an upper bound on $\mathsf{lfp}\ \Phi$, but nevertheless $\Phi(f) \not\sqsubseteq f$. In this case, we say that f is *not inductive*. But how can we verify that f is indeed an upper bound in such a non-inductive scenario? We search *below* f for a *different, but inductive*, upper bound on $\mathsf{lfp}\ \Phi$, that is, we

$$\text{search for an } h \in E \qquad \text{such that} \qquad \mathsf{lfp}\ \Phi \sqsubseteq \Phi(h) \sqsubseteq h \sqsubseteq f.$$

In order to perform a *guided* search for such an h, we introduce the κ-induction operator—a modified version of Φ that is parameterized by our candidate f:

Definition 1 (κ-Induction Operator). *For $f \in E$, we call*

$$\Psi_f\colon \quad E \to E, \qquad g \mapsto \Phi(g) \sqcap f$$

the κ-induction operator *(with respect to f and Φ).*

What does Ψ_f do? As illustrated in Fig. 1, if $\Phi(f) \not\sqsubseteq f$ (i.e. f is non-inductive) then "*at least some part of $\Phi(f)$ is greater than f*". If the whole of $\Phi(f)$ is greater than f, then $f \sqsubset \Phi(f)$; if only some part of $\Phi(f)$ is greater and some is smaller than f, then f and $\Phi(f)$ are incomparable. The κ-induction operator Ψ_f now *rectifies* $\Phi(f)$ being (partly) greater than f by *pulling* $\Phi(f)$ *down* via the meet with f (i.e., via $\_\_ \sqcap f$), so that the result is in no part greater than f. Applying Ψ_f to f hence always yields something below or equal to f.

Together with the observation that Ψ_f is monotonic, iterating Ψ_f on f necessarily *descends* from f downwards in the direction of $\mathsf{lfp}\ \Phi$ (and never below):

Lemma 1 (Properties of the κ-Induction Operator). *Let $f \in E$ and let Ψ_f be the κ-induction operator with respect to f and Φ. Then*

(a) Ψ_f is monotonic, i.e., $\forall g_1, g_2 \in E\colon\ g_1 \sqsubseteq g_2$ implies $\Psi_f(g_1) \sqsubseteq \Psi_f(g_2)$.
(b) Iterations of Ψ_f starting from f are descending, i.e., for all ordinals γ, δ,

$$\gamma < \delta \quad \text{implies} \quad \Psi_f^{\lfloor\delta\rfloor}(f) \sqsubseteq \Psi_f^{\lfloor\gamma\rfloor}(f).$$

(c) Ψ_f is dominated by Φ, i.e., $\forall g \in E\colon\ \Psi_f(g) \sqsubseteq \Phi(g)$.

(d) If $\mathsf{lfp}\ \Phi \sqsubseteq f$, then for any ordinal δ,

$$\mathsf{lfp}\ \Phi \sqsubseteq \ldots \sqsubseteq \Psi_f^{\lfloor\delta\rfloor}(f) \sqsubseteq \ldots \sqsubseteq \Psi_f^{\lfloor 2\rfloor}(f) \sqsubseteq \Psi_f(f) \sqsubseteq f.$$

The descending sequence $f \sqsupseteq \Psi_f(f) \sqsupseteq \Psi_f^{\lfloor 2\rfloor}(f) \sqsupseteq \ldots$ constitutes our guided search for an inductive upper bound on $\mathsf{lfp}\ \Phi$. For each ordinal κ (hence the short name: κ-induction), $\Psi_f^{\lfloor\kappa\rfloor}(f)$ is a potential candidate for Park induction:

$$\Phi\left(\Psi_f^{\lfloor\kappa\rfloor}(f)\right) \overset{\text{potentially}}{\sqsubseteq} \Psi_f^{\lfloor\kappa\rfloor}(f). \tag{‡}$$

For efficiency reasons, e.g., when offloading the above inequality check to an SMT solver, we will not check the inequality (‡) directly but a property equivalent to (‡), namely whether $\Phi(\Psi_f^{\lfloor\kappa\rfloor}(f))$ is below f instead of $\Psi_f^{\lfloor\kappa\rfloor}(f)$:

Lemma 2 (Park Induction from κ-Induction). *Let $f \in E$. Then*

$$\Phi\left(\Psi_f^{\lfloor\kappa\rfloor}(f)\right) \sqsubseteq f \quad \text{iff} \quad \Phi\left(\Psi_f^{\lfloor\kappa\rfloor}(f)\right) \sqsubseteq \Psi_f^{\lfloor\kappa\rfloor}(f).$$

Proof. The if-direction is trivial, as $\Psi_f^{\lfloor\kappa\rfloor}(f) \sqsubseteq f$ (Lemma 1(d)). For only-if:

$$\begin{aligned}
\Psi_f^{\lfloor\kappa\rfloor}(f) &\sqsupseteq \Psi_f^{\lfloor\kappa+1\rfloor}(f) && \text{(by Lemma 1(b))}\\
&= \Psi_f\left(\Psi_f^{\lfloor\kappa\rfloor}(f)\right) && \text{(by definition of } \Psi_f^{\lfloor\kappa+1\rfloor}(f))\\
&= \Phi\left(\Psi_f^{\lfloor\kappa\rfloor}(f)\right) \sqcap f && \text{(by definition of } \Psi_f)\\
&\sqsupseteq \Phi\left(\Psi_f^{\lfloor\kappa\rfloor}(f)\right). && \text{(by the premise)} \quad \square
\end{aligned}$$

Algorithm 1: Latticed k-induction

```
input: $\Phi\colon E \to E$ and $f \in E$.
output: "verify" if $f$ is a $k$-inductive invariant, diverge otherwise.
1 $g \leftarrow f$ ;
2 while $\Phi(g) \not\sqsubseteq f$ do
3     $g \leftarrow \Psi_f(g)$ ;
      // recall: $\Psi_f(g) = \Phi(g) \sqcap f$
4 return verify ;
```

Algorithm 2: Latticed BMC

```
input: $\Phi\colon E \to E$ and $f \in E$.
output: "refute" if there exists $k \in \mathbb{N}$ with $\Phi^{\lceil k \rceil}(\bot) \not\sqsubseteq f$, diverge otherwise.
1 $g \leftarrow \bot$ ;
2 repeat
3     $g \leftarrow \Phi(g)$ ;
4 until $g \not\sqsubseteq f$ ;
5 return refute ;
```

If $\Phi\big(\Psi_f^{\lfloor\kappa\rfloor}(f)\big) \sqsubseteq f$, then Lemma 2 tells us that $\Psi_f^{\lfloor\kappa\rfloor}(f)$ is Park inductive and thereby an upper bound on $\mathsf{lfp}\ \Phi$. Since iterating Ψ_f on f yields a descending iteration sequence (see Lemma 1(b)), $\Psi_f^{\lfloor k\rfloor}(f)$ is below f and therefore f is also an upper bound on $\mathsf{lfp}\ \Phi$. Put in more traditional terms, we have shown that $\Psi_f^{\lfloor\kappa\rfloor}(f)$ is an inductive invariant stronger than f. Formulated as a proof rule, we obtain the following induction principle:

Theorem 2 ($\boldsymbol{\kappa}$-Induction). *Let $f \in E$ and let κ be an ordinal. Then*

$$\Phi\left(\Psi_f^{\lfloor\kappa\rfloor}(f)\right) \sqsubseteq f \quad \text{implies} \quad \mathsf{lfp}\ \Phi \sqsubseteq f.$$

Proof. Following the argument above, for details see [8, Appx. A.2]. □

An illustration of κ-induction is shown in (the right frame of) Fig. 1. For every ordinal κ, if $\Phi(\Psi_f^{\lfloor\kappa\rfloor}(f)) \sqsubseteq f$, then we call f $(\kappa{+}1)$-*inductive* (for Φ). In particular, κ-induction generalizes Park induction, in the sense that 1-induction *is* Park induction and, $(\kappa > 1)$-induction is a *more general principle of induction.*

Algorithm 1 depicts a (semi-)algorithm that performs *latticed k-induction* (for $k < \omega$) in order to prove $\mathsf{lfp}\ \Phi \sqsubseteq f$ by iteratively increasing k. For implementing this algorithm, we require, of course, that both Φ and Ψ_f are computable and that $\sqsubseteq$ is decidable. Notice that the loop (lines 2–3) never terminates if $f \sqsubset \Phi(f)$—a condition that can easily be checked before entering the loop. Even with this optimization, however, Algorithm 1 is a *proper* semi-algorithm: even if $\mathsf{lfp}\ \Phi \sqsubseteq f$, then f is still not guaranteed to be k-inductive for some $k < \omega$. And even if an algorithm *could* somehow perform transfinitely many iterations, then f is still not guaranteed to be κ-inductive for some ordinal κ:

Counterexample 1 (Incompleteness of $\boldsymbol{\kappa}$-Induction). *Consider the carrier set $\{0,1,2\}$, partial order $0 \sqsubset 1 \sqsubset 2$, and the monotonic operator Φ with $\Phi(0) = 0 = \mathsf{lfp}\ \Phi$, and $\Phi(1) = 2$, and $\Phi(2) = 2 = \mathsf{gfp}\ \Phi$. Then $\mathsf{lfp}\ \Phi \sqsubseteq 1$, but for any ordinal κ, $\Psi_1^{\lfloor\kappa\rfloor}(1) = 1$ and $\Phi(1) = 2 \not\sqsubseteq 1$. Hence 1 is not κ-inductive.* ◁

Despite its incompleteness, we now provide a *sufficient* criterion which ensures that *every* upper bound on $\mathsf{lfp}\ \Phi$ is κ-inductive for some ordinal κ.

Theorem 3 (Completeness of κ-Induction for Unique Fixed Point). *If* $\mathsf{lfp}\ \Phi = \mathsf{gfp}\ \Phi$ *(i.e. Φ has* exactly one *fixed point), then, for every $f \in E$,*

$$\mathsf{lfp}\ \Phi \ \sqsubseteq\ f \qquad \text{implies} \qquad f \textit{ is } \kappa\textit{-inductive for some ordinal } \kappa.$$

Proof. By the Knaster-Tarski theorem, we have $\Phi^{\lfloor \beta \rfloor}(\top) = \mathsf{gfp}\ \Phi$ for some ordinal β. We then show that f is (β+1)-inductive; see [8, Appx A.3] for details. □

The proof of the above theorem immediately yields that, if the unique fixed point can be reached through *finite* fixed point iterations starting at $\top$, then f is k-inductive for some *natural* number k; Algorithm 1 thus eventually terminates.

Corollary 1. *If $\Phi^{\lfloor n \rfloor}(\top) = \mathsf{lfp}\ \Phi$ for some $n \in \mathbb{N}$, then, for every $f \in E$,*

$$\mathsf{lfp}\ \Phi \ \sqsubseteq\ f \qquad \text{implies} \qquad f \textit{ is } n\textit{-inductive for some } n \in \mathbb{N}.$$

4 Latticed vs. Classical *k*-Induction

We show that our purely lattice-theoretic κ-induction from Sect. 3 generalizes classical k-induction for hardware- and software verification. To this end, we first recap how k-induction is typically formalized in the literature [10,23,29,37]: Let TS $= (S, I, T)$ be a transition system, where S is a (countable) set of *states*, $I \subseteq S$ is a non-empty set of *initial states*, and $T \subseteq S \times S$ is a *transition relation*. As in the seminal work on k-induction [65], we require that T is a *total* relation, i.e., every state has at least one successor. This requirement is sometimes overlooked in the literature, which renders the classical SAT-based formulation of k-induction ((1a) and (1b) below) unsound in general.

Our goal is to verify that a given *invariant property* $P \subseteq S$ covers all states reachable in TS from some initial state. Suppose that I, T and P are characterized by logical formulae $I(s)$, $T(s, s')$ and $P(s)$ (over the free variables s and s'), respectively. Then, achieving the above goal with classical k-induction amounts to proving the validity of

$$I(s_1) \wedge T(s_1, s_2) \wedge \ldots \wedge T(s_{k-1}, s_k) \implies P(s_1) \wedge \ldots \wedge P(s_k), \quad \text{and} \tag{1a}$$

$$P(s_1) \wedge T(s_1, s_2) \wedge \ldots \wedge P(s_k) \wedge T(s_k, s_{k+1}) \implies P(s_{k+1}). \tag{1b}$$

Here, the *base case* (1a) asserts that P holds for *all states reachable within k transition steps from some initial state*; the *induction step* (1b) formalizes that P is *closed under taking up to k transition steps*, i.e., if we start in P and stay in P for up to k steps, then we also end up in P after taking the (k+1)-st step. If both (1a) and (1b) are valid, then classical k-induction tells us that the property P holds for *all* reachable states of TS. How is the above principle reflected in *latticed* k-induction (cf. Sect. 3)? For that, we choose the complete lattice $(2^S, \subseteq)$, where 2^S denotes the powerset of S; the least element is $\bot = \emptyset$ and the meet operation is standard intersection $\cap$.

Moreover, we define a monotonic operator Φ whose least fixed point precisely characterizes the set of reachable states of the transition system TS:

$$\Phi\colon \quad 2^S \;\to\; 2^S, \qquad F \;\mapsto\; I \cup \mathsf{Succs}(F),$$

That is, Φ maps any given set of states $F \subseteq S$ to the union of the initial states I and of those states $\mathsf{Succs}(F)$ that are reachable from F using a single transition.[3]

Using the κ-induction operator Ψ_P constructed from Φ and P according to Definition 1, the principle of κ-induction (cf. Theorem 2) then tells us that

$$\Phi\left(\Psi_P^{\lfloor\kappa\rfloor}(P)\right) \;\subseteq\; P \qquad \text{implies} \qquad \underbrace{\mathsf{lfp}\,\Phi}_{\text{reachable states of TS}} \;\subseteq\; P.$$

For our above choices, the premise of κ-induction equals the classical formalization of k-induction—formulae (1a) and (1b)—because the set of initial states I is "baked into" the operator Φ. More concretely, for the base case (1a), we have

$$\underbrace{\underbrace{\underbrace{\underbrace{I(s_1)}_{\Phi(\emptyset)} \wedge T(s_1, s_2)}_{\Phi^{\lceil 2\rceil}(\emptyset)} \wedge \ldots \wedge T(s_{k-1}, s_k)}_{\Phi^{\lceil k\rceil}(\emptyset)} \;\Longrightarrow\; P(s_1) \wedge \ldots \wedge P(s_k)}_{\text{meaning} \quad \Phi^{\lceil k\rceil}(\emptyset) \,\subseteq\, P}\,.$$

In other words, formula (1a) captures those states that are reachable from I via at most k transitions. If we assume that (1a) is valid, then P contains all initial states and formula (1b) coincides with the premise of κ-induction:

$$\underbrace{\underbrace{\underbrace{\underbrace{\underbrace{P(s_1) \wedge T(s_1, s_2)}_{\Phi(P)} \wedge P(s_2)}_{\Psi_P(P) \,=\, \Phi(P) \cap P} \wedge T(s_2, s_3) \wedge \ldots \wedge P(s_k)}_{\Psi_P^{\lfloor k-1\rfloor}(P)} \wedge T(s_k, s_{k+1})}_{\Phi\left(\Psi_P^{\lfloor k-1\rfloor}(P)\right)} \;\Longrightarrow\; P(s_{k+1})}_{\text{meaning} \quad \Phi\left(\Psi_P^{\lfloor k-1\rfloor}(P)\right) \,\subseteq\, P}\,.$$

It follows that, when considering transition systems, our (latticed) κ-induction is equivalent to the classical notion of k-induction for $\kappa < \omega$:

Theorem 4. *For every natural number $k \geq 1$,*

$$\Phi\left(\Psi_P^{\lfloor k-1\rfloor}(P)\right) \;\subseteq\; P \quad \text{iff} \quad \text{formulae (1a) and (1b) are valid.}$$

[3] Formally, $\mathsf{Succs}(F) \;\triangleq\; \{\, t' \mid t \in F,\, (t, t') \in T \,\}$.

$$\begin{array}{rcl} C & ::= & \texttt{skip} \\ & \mid & x \coloneqq e \\ & \mid & C\,;C \\ & \mid & \{\,C\,\}\,[p]\,\{\,C\,\} \\ & \mid & \texttt{if}\ (\varphi)\ \{\,C\,\}\ \texttt{else}\ \{\,C\,\} \\ & \mid & \texttt{while}\,(\varphi)\,\{\,C\,\} \end{array}$$

(a) pGCL programs

$$\begin{array}{rcl} e & ::= & n \\ & \mid & x \\ & \mid & n \cdot e \\ & \mid & e + e \\ & \mid & e \dot{-} e \ \ (\text{monus } \max\{0, e - e\}) \end{array}$$

(b) Linear expressions

$$\begin{array}{rcl} \varphi & ::= & e < e \\ & \mid & \varphi \wedge \varphi \\ & \mid & \neg\varphi \end{array}$$

(c) Linear guards

Fig. 2. Syntax of pGCL programs, linear expressions, and guards, where x is a variable taken from a countable set Vars of program variables (evaluating to natural numbers), $p \in [0,1] \cap \mathbb{Q}$ is a rational probability, and $n \in \mathbb{N}$ is a constant.

5 Latticed Bounded Model Checking

We complement κ-induction with a latticed analog of bounded model checking [11,12] for *refuting* that $\mathsf{lfp}\ \Phi \sqsubseteq f$. In lattice-theoretic terms, bounded model checking amounts to a *fixed point iteration* of Φ on $\bot$ while continually checking whether the iteration exceeds our candidate upper bound f. If so, then we have indeed refuted $\mathsf{lfp}\ \Phi \sqsubseteq f$:

Theorem 5 (Soundness of Latticed BMC). *Let $f \in E$. Then*

$$\exists \text{ ordinal } \delta\colon \quad \Phi^{\lceil \delta \rceil}(\bot) \ \not\sqsubseteq\ f \qquad \text{implies} \qquad \mathsf{lfp}\ \Phi \ \not\sqsubseteq\ f.$$

Furthermore, if we were actually able to perform transfinite iterations of Φ on $\bot$, then latticed bounded model checking is also complete: If f is in fact *not* an upper bound on $\mathsf{lfp}\ \Phi$, this *will* be witnessed at some ordinal:

Theorem 6 (Completeness of Latticed BMC). *Let $f \in E$. Then*

$$\mathsf{lfp}\ \Phi \ \not\sqsubseteq\ f \qquad \text{implies} \qquad \exists \text{ ordinal } \delta\colon \quad \Phi^{\lceil \delta \rceil}(\bot) \ \not\sqsubseteq\ f.$$

More practically relevant, if Φ is continuous (which is the case for Bellman operators characterizing reachability probabilities in Markov chains), then a simple *finite* fixed point iteration, see Algorithm 2, is sound and complete for refutation:

Corollary 2 (Latticed BMC for Continuous Operators). *Let $f \in E$ and let Φ be continuous. Then*

$$\exists\, n \in \mathbb{N}\colon \quad \Phi^{n}(\bot) \ \not\sqsubseteq\ f \qquad \text{iff} \qquad \mathsf{lfp}\ \Phi \ \not\sqsubseteq\ f.$$

6 Probabilistic Programs

In the remainder of this article, we employ latticed k-induction and BMC to verify imperative programs with access to discrete probabilistic choices—branching on the outcomes of coin flips. In this section, we briefly recap the necessary background on formal reasoning about probabilistic programs (cf. [44,49] for details).

6.1 The Probabilistic Guarded Command Language

Syntax. Programs in the *probabilistic guarded command language* pGCL adhere to the grammar in Fig. 2a. The semantics of most statements is standard. In particular, the *probabilistic choice* $\{\, C_1 \,\} \; [p] \; \{\, C_2 \,\}$ flips a coin with bias $p \in [0,1] \cap \mathbb{Q}$. If the coin yields heads, it executes C_1; otherwise, C_2. In addition to the syntax in Fig. 2, we admit standard expressions that are definable as syntactic sugar, e.g., true, false, $\varphi_1 \vee \varphi_2$, $e_1 = e_2$, $e_1 \leq e_2$, etc.

Program States. A *program state* σ maps every variable in Vars to its value, i.e., a natural number in $\mathbb{N}$.[4] To ensure that the set of program states Σ remains countable[5], we restrict ourselves to states in which only finitely many variables—those that appear in a given program—evaluate to non-zero values. Formally,

$$\Sigma \;\triangleq\; \Big\{ \sigma \colon \mathsf{Vars} \to \mathbb{N} \;\Big|\; \big|\{\, x \in \mathsf{Vars} \mid \sigma(x) \neq 0 \,\}\big| < \infty \Big\}.$$

The evaluation of expressions e and guards φ under a state σ, denoted by $e(\sigma)$ and $\varphi(\sigma)$, is standard. For example, we define the evaluation of "monus" as

$$(e_1 \dot{-} e_2)(\sigma) \;\triangleq\; \max\{\, 0,\; e_1(\sigma) - e_2(\sigma) \,\}\,.$$

6.2 Weakest Preexpectations

Expectations. An *expectation* $f \colon \Sigma \to \mathbb{R}^{\infty}_{\geq 0}$ is a map from program states to the non-negative reals extended by infinity. We denote by $\mathbb{E}$ the set of all expectations. Moreover, $(\mathbb{E}, \preceq)$ forms a complete lattice, where the partial order $\preceq$ is given by the pointwise application of the canonical ordering $\leq$ on $\mathbb{R}^{\infty}_{\geq 0}$, i.e.,

$$f \;\preceq\; g \qquad \text{iff} \qquad \forall \sigma \in \Sigma\colon \quad f(\sigma) \;\leq\; g(\sigma).$$

To conveniently describe expectations evaluating to some $r \in \mathbb{R}^{\infty}_{\geq 0}$ for every state, we slightly abuse notation and denote by r the constant expectation $\lambda\sigma\text{.}\; r$. Similarly, given an arithmetic expression e, we denote by e the expectation $\lambda\sigma\text{.}\; e(\sigma)$.

4 We prefer unsigned integers because our quantitative "specifications" (aka *expectations*) must evaluate to non-negative numbers. Otherwise, expectations like $x+y$ are not well-defined, and, as a remedy, we would frequently have to take the absolute value of every program variable. Restricting ourselves to unsigned variables does not decrease expressive power as signed variables can be emulated (cf. [9, Sec. 11.2]).

5 In order to avoid any technical issues pertaining to measurability.

Table 1. Rules defining the weakest preexpectation transformer.

C	$\mathsf{wp}\ \llbracket C \rrbracket\ (g)$
$\mathtt{skip}$	g
$x := e$	$g\,[x/e]$
$C_1\, ;\, C_2$	$\mathsf{wp}\llbracket C_1 \rrbracket\ (\mathsf{wp}\llbracket C_2 \rrbracket\ (g))$
$\{\, C_1 \,\}\ [\, p\,]\ \{\, C_2 \,\}$	$p \cdot \mathsf{wp}\llbracket C_1 \rrbracket\ (g) + (1-p) \cdot \mathsf{wp}\llbracket C_2 \rrbracket\ (g)$
$\mathtt{if}\ (\, \varphi\,)\ \{\, C_1 \,\}\ \mathtt{else}\ \{\, C_2 \,\}$	$[\varphi] \cdot \mathsf{wp}\llbracket C_1 \rrbracket\ (g) + [\neg\varphi] \cdot \mathsf{wp}\llbracket C_2 \rrbracket\ (g)$
$\mathtt{while}\, (\, \varphi\,)\, \{\, C' \,\}$	$\mathsf{lfp}\ h \bullet\ [\neg\varphi] \cdot g + [\varphi] \cdot \mathsf{wp}\llbracket C' \rrbracket\ (h)$

The least element of $(\mathbb{E}, \preceq)$ is 0 and the greatest element is ∞. We employ the *Iverson bracket* notation to cast Boolean expressions into expectations, i.e.,

$$[\varphi] \quad = \quad \lambda\sigma \bullet \begin{cases} 1 & \text{if } \varphi(\sigma) = \mathsf{true}, \\ 0 & \text{if } \varphi(\sigma) = \mathsf{false}. \end{cases}$$

The *weakest preexpectation transformer* $\mathsf{wp}\colon \mathsf{pGCL} \to (\mathbb{E} \to \mathbb{E})$ is defined in Table 1, where $g\,[x/e]$ denotes the substitution of variable x by expression e, i.e.,

$$g\,[x/e] \triangleq \lambda\sigma \bullet\ g(\sigma\,[x \mapsto e(\sigma)]), \quad \text{where} \quad \sigma\,[x \mapsto e(\sigma)] \triangleq \lambda y \bullet \begin{cases} e(\sigma) & \text{if } y = x, \\ \sigma(y) & \text{otherwise.} \end{cases}$$

We call $\mathsf{wp}\llbracket C \rrbracket\ (g)$ the *weakest preexpectation* of program C w.r.t. postexpectation g. The weakest preexpectation $\mathsf{wp}\llbracket C \rrbracket\ (g)$ is itself an expectation of type $\mathbb{E}$, which maps each initial state σ to the expected value of g after running C on σ. More formally, if μ_C^σ is the distribution over final states obtained by executing C on initial state σ, then for any postexpectation g [44],

$$\mathsf{wp}\llbracket C \rrbracket\ (g)\ (\sigma) \ = \ \sum\nolimits_{\tau \in \Sigma} \mu_C^\sigma(\tau) \cdot g(\tau).$$

For a gentle introduction to weakest preexpectations, see [38, Chap. 2 and 4].

7 BMC and k-Induction for Probabilistic Programs

We now instantiate latticed κ-induction and BMC (as developed in Sects. 2 to 5) to enable verification of loops written in pGCL; we discuss practical aspects later in Sects. 7.1 to 7.3 and Sect. 8. For the next two sections, we fix a loop

$$C_{\text{loop}} \quad = \quad \mathtt{while}\,(\,\varphi\,)\,\{\, C \,\}\,.$$

For simplicity, we assume that the loop body C is loop-free (every probabilistic program can be rewritten as a single while loop with loop-free body [62]).

Given an expectation $g \in \mathbb{E}$ and a candidate upper bound $f \in \mathbb{E}$ on the expected value of g after executing C_{loop} (i.e. $\mathsf{wp}[\![C_{\text{loop}}]\!]\,(g)$), we will apply latticed verification techniques to check whether f indeed upper-bounds $\mathsf{wp}[\![C_{\text{loop}}]\!]\,(g)$.

To this end, we denote by Φ the *characteristic functional* of C_{loop} and g, i.e.,

$$\Phi\colon \quad \mathbb{E} \;\to\; \mathbb{E}, \qquad h \;\mapsto\; [\neg\varphi]\cdot g + [\varphi]\cdot \mathsf{wp}[\![C]\!]\,(h)\,,$$

whose least fixed point defines $\mathsf{wp}[\![C_{\text{loop}}]\!]\,(g)$ (cf. Table 1). We remark that Φ is a monotonic—and in fact even continuous—operator over the complete lattice $(\mathbb{E},\,\preceq)$ (cf. Sect. 6.2). In this lattice, the meet is a pointwise minimum, i.e.,

$$h \sqcap h' \;=\; h \;\mathsf{min}\; h' \;\triangleq\; \lambda\sigma\text{.}\;\; \min\{\, h(\sigma), h'(\sigma)\,\}\,.$$

By Definition 1, Φ and g then induce the (continuous) κ-induction operator

$$\Psi_f\colon \quad \mathbb{E} \;\to\; \mathbb{E}, \qquad h \;\mapsto\; \Phi\,(h)\;\mathsf{min}\; f.$$

With this setup, we obtain the following proof rule for reasoning about probabilistic loops as an immediate consequence of Theorem 2:

Corollary 3 (k-Induction for pGCL). *For every natural number $k \in \mathbb{N}$,*

$$\Phi\left(\Psi_f^{\lfloor k\rfloor}(f)\right) \;\preceq\; f \qquad \text{implies} \qquad \mathsf{wp}[\![C_{\text{loop}}]\!]\,(g) \;\preceq\; f.$$

Analogously, refuting that f upper-bounds the expected value of g after execution of C_{loop} via bounded model checking is an instance of Corollary 2:

Corollary 4 (Bounded Model Checking for pGCL).

$$\exists\, n \in \mathbb{N}\colon \quad \Phi^n(0) \;\not\preceq\; f \qquad \text{iff} \qquad \mathsf{wp}[\![C_{\text{loop}}]\!]\,(g) \;\not\preceq\; f.$$

Example 2 (Geometric Loop). The pGCL program

$$C_{\text{geo}} \quad = \quad \mathtt{while}\,(\,x = 1\,)\,\{\,\{\,x \coloneqq 0\,\}\,[\,0.5\,]\,\{\,c \coloneqq c+1\,\}\,\}$$

keeps flipping a fair coin x until it flips heads, sets x to 0, and terminates. Whenever it flips tails instead, it increments the counter c and continues. We refer to C_{geo} as the "geometric loop" because after its execution, the counter variable c is distributed according to a geometric distribution.

What is a (preferably small) upper bound on the expected value $\mathsf{wp}[\![C_{\text{geo}}]\!]\,(c)$ of c after execution of C_{geo}? Using 2-induction, we can (automatically) verify that $c+1$ is indeed an upper bound: Since $\Phi\,(\Psi_{c+1}(c+1)) \preceq c+1$, where Φ denotes the characteristic functional of C_{geo}, Corollary 3 yields $\mathsf{wp}[\![C_{\text{geo}}]\!]\,(c) \preceq c+1$.

However, $c+1$ *cannot* be proven an upper bound using Park induction as it is *not* inductive. Moreover, it is indeed the *least* upper bound, i.e., any smaller bound is refutable using BMC (cf. Corollary 4). For example, we have $\mathsf{wp}[\![C_{\text{geo}}]\!]\,(c) \not\preceq c+0.99$, since $\Phi^{\lceil 11\rceil}\,(0) \not\preceq c+0.99$. Finally, we remark that some correct upper bounds only become κ-inductive for *transfinite* ordinals κ. For instance, the innocuous-looking bound $2\cdot c+1$ is not k-inductive for any natural number k, but it is $(\omega+1)$-inductive, since $\Phi\big(\Psi_{2\cdot c+1}^{\lfloor\omega\rfloor}(2\cdot c+1)\big) \preceq 2\cdot c+1$. $\triangleleft$

In principle, we can semi-decide whether $\mathsf{wp}[\![C_{\text{loop}}]\!]\,(g) \not\preceq f$ holds or whether f is k-inductive for some k: it suffices to run Algorithms 1 and 2 in parallel. However, for these two algorithms to actually be semi-decision procedures, we cannot admit arbitrary expectations. Rather, we restrict ourselves to a suitable subset Exp of expectations in $\mathbb{E}$ satisfying all of the following requirements:

1. Exp is closed under computing the characteristic functional Φ, i.e.,
$$\forall\, h \in \mathsf{Exp}\colon \quad \Phi\,(h) \text{ is computable and belongs to } \mathsf{Exp}.$$
2. Quantitative entailments between expectations in Exp are decidable, i.e.,
$$\forall\, h, h' \in \mathsf{Exp}\colon \quad \text{it is decidable whether } h \;\preceq\; h'.$$
3. (For k-induction) Exp is closed under computing meets, i.e.,
$$\forall\, h, h' \in \mathsf{Exp}\colon \quad h \;\mathsf{min}\; h' \text{ is computable and belongs to } \mathsf{Exp}.$$

Below, we show that *linear expectations* meet all of the above requirements.

7.1 Linear Expectations

Recall from Fig. 2b that we assume all expressions appearing in pGCL programs to be linear. For our fragment of syntactic expectations, we consider *extended* linear expressions $\tilde{e}$ that (1) are defined over *rationals* instead of natural numbers and (2) admit ∞ as a constant (but not as a *sub*expression). Formally, the set of extended linear expressions is given by the following grammar:

$$\tilde{e} \;::=\; e \mid \infty \qquad\qquad e \;::=\; r \mid x \mid r \cdot e \mid e + e \mid e \dot{-} e \qquad\qquad (r \in \mathbb{Q}_{\geq 0})$$

Similarly, we admit extended linear expressions (without ∞) in linear guards φ.[6] With these adjustments to expressions and guards in mind, the set LinExp of *linear expectations* is defined by the grammar

$$h \;::=\; \tilde{e} \quad\mid\quad [\varphi] \cdot h \quad\mid\quad h + h.$$

We write $h = h'$ if h and h' are *<u>syntactically</u> identical*; and $h \equiv h'$ if they are *<u>semantically</u> equivalent*, i.e., if for all states σ, we have $h(\sigma) = h'(\sigma)$.

Furthermore, the *rescaling* $c \cdot h$ of a linear expectation h by a constant $c \in \mathbb{Q}_{\geq 0}$ is syntactic sugar for rescaling suitable[7] arithmetic subexpressions of h, e.g.,

$$1/2 \cdot ([x = 1] \cdot 4 + 1/3 \cdot x + \infty) \;\equiv\; 1/2 \cdot [x = 1] \cdot 4 + 1/2 \cdot 1/3 \cdot x + \infty \in \mathsf{LinExp}.$$

A formal definition of the rescaling $c \cdot h$ is found in [8, Appx A.5].

[6] We do not admit ∞ in guards for convenience. In principle, all comparisons with ∞ in guards can be removed by a simple preprocessing step.

[7] We do not rescale every subexpression to account for the corner cases $c \cdot \infty = \infty$ and $0 \cdot \infty = 0$.

If we choose a linear expectation h as a postexpectation, then a quick inspection of Table 1 reveals that the weakest preexpectation $\mathsf{wp}[\![C]\!]\,(h)$ of any *loop-free* pGCL program C and h yields a linear expectation again. Hence, linear expectations are closed under applying Φ— Requirement 1 above—because

$$\forall\, g, h \in \mathsf{LinExp}\colon \quad \Phi\,(h) \;=\; \underbrace{\underbrace{[\neg\varphi]\cdot g}_{\in\ \mathsf{LinExp}} + [\varphi]\cdot \underbrace{\mathsf{wp}[\![C]\!]\,(h)}_{\in\ \mathsf{LinExp}}}_{\in\ \mathsf{LinExp}}\,.$$

7.2 Deciding Quantitative Entailments Between Linear Expectations

To prove that linear expectations meet Requirement 2—decidability of quantitative entailments—we effectively reduce the question of whether an entailment $h \preceq h'$ holds to the decidable satisfiability problem for QF_LIRA—quantifier-free mixed linear integer and real arithmetic (cf. [42]).

As a first step, we show that every linear expectation can be represented as a sum of mutually exclusive extended arithmetic expressions—a representation we refer to as the *guarded normal form* (similar to [41, Lem. 1], [9, Lem. A.2]).

Definition 2 (Guarded Normal Form (GNF)). *$h \in \mathsf{LinExp}$ is in GNF if*

$$h \;=\; \sum\nolimits_{i=1}^{n} [\varphi_i]\cdot \tilde{e}_i,$$

where $\tilde{e}_1,\ldots,\tilde{e}_n$ are extended linear expressions, $n \in \mathbb{N}$ is some natural number, and $\varphi_1,\ldots,\varphi_n$ are linear Boolean expressions that partition the set of states, i.e., for each $\sigma \in \Sigma$ there exists exactly one *$i \in \{1,\ldots,n\}$ such that $\varphi_i(\sigma) = \mathsf{true}$.*

Lemma 3. *Every linear expectation $h \in \mathsf{LinExp}$ can effectively be transformed into an equivalent linear expectation $\mathsf{GNF}\,(h) \equiv h$ in guarded normal form.*

The number of summands $|\mathsf{GNF}\,(h)\,|$ in $\mathsf{GNF}\,(h)$ is, in general, exponential in the number of summands in h. In practice, however, this exponential blow-up can often be mitigated by pruning summands with unsatisfiable guards. Throughout the remainder of this paper, we denote the components of $\mathsf{GNF}\,(h)$ and $\mathsf{GNF}\,(h')$, where h and h' are arbitrary linear expectations, as follows:

$$\mathsf{GNF}\,(h) \;=\; \sum\nolimits_{i=1}^{n} [\varphi_i]\cdot \tilde{e}_i \quad \text{and} \quad \mathsf{GNF}\,(h') \;=\; \sum\nolimits_{j=1}^{m} [\psi_j]\cdot \tilde{a}_j.$$

We now present a decision procedure for the *quantitative entailment* over LinExp.

Theorem 7. (Decidability of Quantitative Entailment over LinExp). *For $h, h' \in \mathsf{LinExp}$, it is decidable whether $h \preceq h'$ holds.*

Proof. Let $h, h' \in \mathsf{LinExp}$. By Lemma 3, we have $h \preceq h'$ iff $\mathsf{GNF}\,(h) \preceq \mathsf{GNF}\,(h')$.

Let σ be some state. By definition of the GNF, σ satisfies exactly one guard φ_i and exactly one guard ψ_j. Hence, the inequality $\mathsf{GNF}\,(h)\,(\sigma) \leq \mathsf{GNF}\,(h')\,(\sigma)$

does *not* hold iff $\tilde{e}_i(\sigma) > \tilde{a}_j(\sigma)$ holds for the expressions $\tilde{e}_i$ and $\tilde{a}_j$ guarded by φ_i and ψ_j, respectively. Based on this observation, we construct a QF_LIRA formula $\mathsf{cex}_{\preceq}(h, h')$ that is *unsatisfiable* iff there is no counterexample to $h \preceq h'$:

$$\mathsf{cex}_{\preceq}(h, h') \triangleq \bigvee\nolimits_{i=1}^{n} \bigvee\nolimits_{j=1,\, \tilde{a}_j \neq \infty}^{m} (\varphi_i \ \wedge \ \psi_j \ \wedge \ \mathsf{encodeInfty}(\tilde{e}_i) > \tilde{a}_j)\,.$$

Here, we identify every program variable in h or h' with an $\mathbb{N}$-valued SMT variable. Moreover, to account for comparisons with ∞, we rely on the fact that our (extended) arithmetic expressions either evaluate to ∞ for *every* state or *never* evaluate to ∞. To deal with the case $\tilde{e}_i > \infty$, which is always false, we can thus safely exclude cases in which $\tilde{a}_j = \infty$ holds. To deal with the case $\infty > \tilde{a}_j$, we represent ∞ by some unbounded number, i.e., we introduce a fresh, unconstrained $\mathbb{N}$-valued SMT variable infty and set $\mathsf{encodeInfty}(\tilde{e})$ to infty if $\tilde{e} = \infty$; otherwise, $\mathsf{encodeInfty}(\tilde{e}) = \tilde{e}$. Since QF_LIRA is decidable (cf. [42]), we conclude that the quantitative entailment problem is decidable. □

Since quantitative entailments are decidable, we can already conclude that, for linear expectations, Algorithm 2 is a semi-decision procedure.

7.3 Computing Minima of Linear Expectations

To ensure that latticed k-induction on pGCL programs (cf. Algorithm 1 and Sect. 7) is a semi-decision procedure when considering linear expectations, we have to consider Requirement 3—the expressability and computability of meets:

Theorem 8. LinExp *is effectively closed under taking minima.*

Proof. For $k \in \mathbb{N}$, let $\mathbf{k} \triangleq \{1, \ldots, k\}$. Then, for two linear expectations h, h', the linear expectation $\mathsf{GNF}(h)\ \mathsf{min}\ \mathsf{GNF}(h') \in \mathsf{LinExp}$ is given by:

$$\sum_{(i,j)\, \in\, \mathbf{n} \times \mathbf{m}} \begin{cases} [\varphi_i \wedge \psi_j] \cdot \tilde{a}_j, & \text{if } \tilde{e}_i = \infty, \\ [\varphi_i \wedge \psi_j] \cdot \tilde{e}_i, & \text{if } \tilde{a}_i = \infty, \\ [\varphi_i \wedge \psi_j \wedge \tilde{e}_i \leq \tilde{a}_j] \cdot \tilde{e}_i + [\varphi_i \wedge \psi_j \wedge \tilde{e}_i > \tilde{a}_j] \cdot \tilde{a}_j & \text{otherwise,} \end{cases}$$

where we exploit that, for every state, exactly one guard φ_i and exactly one guard ψ_j is satisfied (cf. Lemma 3). Notice that in the last case we indeed obtain a linear expectation since neither $\tilde{e}$ nor $\tilde{a}$ are equal to ∞. □

8 Implementation

We have implemented a prototype called KIPRO2—k-Induction for PRObabilistic PROgrams—in Python 3.7 using the SMT solver Z3 [54] and the solver-API PySMT [25]. Our tool, its source code, and our experiments are available online.[8]

[8] https://github.com/moves-rwth/kipro2.

KIPRO2 performs in parallel latticed k-induction and BMC to fully automatically verify upper bounds on expected values of pGCL programs as described in Sect. 7. In addition to reasoning about expected values, KIPRO2 supports verifying bounds on *expected runtimes* of pGCL programs, which are characterized as least fixed points à la [40]. Rather than fixing a specific runtime model, we took inspiration from [56] and added a statement $\texttt{tick}(n)$ that does not affect the program state but consumes $n \in \mathbb{N}$ time units.

To discharge quantitative entailments and compute the meet, we use the constructions in Theorems 7 and 8, respectively. As an additional optimization, we do not iteratively apply the k-induction operator Ψ_f directly but use an *incremental encoding.* We briefly sketch our encoding for k-induction (Algorithm 2); the encoding for BMC is similar. In both cases, we employ uninterpreted functions on top of mixed integer and real arithmetic, i.e., QF_UFLIRA.

Recall Example 2, the geometric loop C_{geo}, where we used k-induction to prove $\mathsf{wp}[\![C_{\text{geo}}]\!]\,(c) \preceq c+1$. For every $k \in \mathbb{N}$, $\Phi(\Psi_{c+1}^{\lfloor k \rfloor}(c+1))$ is given by

$$\underbrace{[x=1]\cdot\Big(0.5\cdot\underbrace{\Psi_{c+1}^{\lfloor k\rfloor}(c+1)}_{Q_k}[x/0] \;+\; 0.5\cdot\underbrace{\Psi_{c+1}^{\lfloor k\rfloor}(c+1)}_{Q_k}[c/c+1]\Big) \;+\; [x\neq 1]\cdot c\,.}_{P_k}$$

To obtain an incremental encoding, we introduce an uninterpreted function $P_k\colon \mathbb{N}\times\mathbb{N}\to\mathbb{R}_{\geq 0}$ and a formula $\rho_k(c,x)$ specifying that $P_k(c,x)$ characterizes $\Phi(\Psi_{c+1}^{\lfloor k\rfloor}(c+1))$, i.e., for all $\sigma\in\Sigma$ and $r\in\mathbb{R}_{\geq 0}$ with $\Phi(\Psi_{c+1}^{\lfloor k\rfloor}(c+1))(\sigma)<\infty$,[9]

$$\rho_k(\sigma(c),\sigma(x)) \wedge P_k(\sigma(c),\sigma(x)) = r \text{ is satisfiable} \quad \text{iff} \quad r = \Phi\Big(\Psi_{c+1}^{\lfloor k\rfloor}(c+1)\Big)(\sigma).$$

If $\Phi(\Psi_{c+1}^{\lfloor k\rfloor}(c+1))(\sigma)=\infty$, our construction of $\rho_k(x,c)$ ensures that the above conjunction is satisfiable for arbitrarily large r. Analogously, we introduce an uninterpreted function $Q_k\colon \mathbb{N}\times\mathbb{N}\to\mathbb{R}_{\geq 0}$ that characterizes $\Psi_{c+1}^{\lfloor k\rfloor}(c+1)$.

In particular, may use all uninterpreted functions introduced for smaller or equal values of k—not just the function $P_k(c,x)$ it needs to characterize. This enables an incremental encoding, i.e., $\rho_k(c,x)$ can be computed on top of $\rho_{k-1}(c,x)$ by reusing $P_{k-1}(c,x)$, $Q_k(c,x)$, and the construction in Theorem 8.

Moreover, we can reuse $\rho_k(c,x)$ to avoid computing the (expensive) GNF for deciding certain quantitative entailments (cf. Theorem 7): For example, to check whether $\Phi(\Psi_{c+1}^{\lfloor k\rfloor}(c+1)) \not\preceq h'$ holds, we only need to transform the right-hand side into GNF (cf. Sect. 7.2), i.e., if $\mathsf{GNF}\,(h') = \sum_{j=1}^{m}[\psi_j]\cdot\tilde{a}_j$, then

$$\Phi\Big(\Psi_{c+1}^{\lfloor k\rfloor}(c+1)\Big) \not\preceq g \quad \text{iff} \quad \rho_k \wedge \bigvee\nolimits_{j=1,\,\tilde{a}_j\neq\infty}^{m} \psi_j \wedge P_k(c,x) > \tilde{a}_j \text{ is satisfiable.}$$

[9] Notice that we do *not* axiomatize in $\rho_k(c,x)$ that $\Phi(\Psi_{c+1}^{\lfloor k\rfloor}(c+1))$ and $P_k(c,x)$ are the same function because we have no access to universal quantifiers. Rather, we specify that both functions coincide for any fixed concrete values assigned to c and x. This weaker notion is *not* robust against formal modifications of the parameters, e.g., through substitution. For example, to assign the correct interpretation to $P_k(c,x)\,[c/c+1]$, we have to construct a (second) formula $\rho_k(c,x)\,[c/c+1]$.

9 Experiments

We evaluate KIPRO2 on two sets of benchmarks. The first set, shown in Table 2, consists of four (infinite-state) probabilistic systems compiled from the literature; each benchmark is evaluated on multiple variants of candidate upper bounds:

(1) `brp` is a `pGCL` variant of the bounded retransmission protocol [19,32]. The goal is to transmit *toSend* many packages via an unreliable channel allowing for at most *maxFail* many retransmissions per package (cf. Example 1). The variable *totalFail* keeps track of the total number of failed attempts to send a package. We verified upper bounds on the expected outcome of *totalFail* (variants 1–4). In doing so, we bound the number of packages to send by 4 (10, 20, 70) while keeping *maxFail unbounded*, i.e., we still verify an infinite-state system. We notice that $k > 1$ is required for proving any of the candidate bounds; for up to $k = 11$, KIPRO2 manages to prove non-trivial bounds within a few seconds. However, unsurprisingly, the complexity increases rapidly with larger k. While KIPRO2 can prove variant 3, it needs to increase k to 23; we observe that the complexity grows rapidly both in terms of the size of formulae and in terms of runtime with increased k. Furthermore, variants 5–7 correspond to (increasing) incorrect candidate bounds ($\mathit{totalFail} + 1$, $\mathit{totalFail} + 1.5$, $\mathit{totalFail} + 3$) that are refuted (or time out) when not imposing any restriction on *toSend*.

(2) `geo` corresponds to the geometric loop from Example 2. We verify that $c+1$ upper-bounds the expected value of c for every initial state (variant 1); we refute the incorrect candidates $c + 0.99$ and $c + 0.999999999999$ (variants 2–3).

(3) `rabin` is a variant of Rabin's mutual exclusion algorithm [46] taken from [34]. We aim to verify that the probability of obtaining a unique winning process is at most $2/3$ for at most 2 (3, 4) participants (variants 1–3) and refute both $1/3$ (variant 4) and $3/5$ (variant 5) for an unbounded number of participants.

(4) `unif_gen` implements the algorithm in [48] for generating a discrete uniform distribution over some interval $\{l, \ldots, l+n-1\}$ using only fair coin flips. We aim to verify that $1/n$ upper-bounds the probability of sampling a particular element from *any* such interval of size at most $n = 2$ (3, 4, 5, 6) (variants 1–5).

Our second set of benchmarks, shown in Table 3, confirms the correctness of (1-inductive) bounds on the expected runtime of `pGCL` programs synthesized by the runtime analyzers ABSYNTH [56] and (later) KoAT [52]; this gives a baseline for evaluating the performance of our implementation. Moreover, it demonstrates the flexibility of our approach as we effortlessly apply the expected runtime calculus [40] instead of the weakest preexpectation calculus for verification.

Setup. We ran Algorithms 1 and 2 in parallel using an AMD Ryzen 5 3600X processor with a shared memory limit of 8GB and a 15-minute timeout. For every benchmark finishing within the time limit, KIPRO2 either finds the smallest k required to prove the candidate bound by k-induction or the smallest unrolling depth k to refute it. If KIPRO2 refutes, the SMT solver provides a concrete initial state witnessing that violation. In Tables 2 and 3, column #formulae gives the maximal number of conjuncts on the solver stack; formulae_t, sat_t, and total_t give the amount of time spent on (1) computing formulae, (2) satisfiability checking, and (3) everything (including preprocessing), respectively. The input consists of a program, a candidate upper bound, and a postexpectation; in Table 3, the latter is fixed to "postruntime" 0 and thus omitted.

Table 2. Empirical results for the first benchmark set (time in seconds).

	postexpectation	variant	result	k	#formulae	formulae_t	sat_t	total_t
`brp`	*totalFail*	1	ind	5	285	0.15	0.01	0.28
		2	ind	11	2812	1.77	0.12	2.03
		3	ind	23	26284	17.68	28.09	45.94
		4	TO	–	–	–	–	–
		5	ref	13	949	0.84	14.39	15.28
		6	TO	–	–	–	–	–
		7	TO	–	–	–	–	–
`geo`	c	1	ind	2	18	0.01	0.00	0.08
		2	ref	11	103	0.04	0.01	0.09
		3	ref	46	1223	0.39	0.04	0.48
`rabin`	$[i = 1]$	1	ind	1	21	0.01	0.00	0.15
		2	ind	5	1796	1.27	0.03	1.44
		3	TO	–	–	–	–	–
		4	ref	4	458	0.31	0.03	0.40
		5	ref	8	10508	8.76	2.85	11.68
`unif_gen`	$[c = i]$	1	ind	2	267	0.27	0.02	0.56
		2	ind	3	1402	1.45	0.10	1.81
		3	ind	3	1402	1.48	0.11	1.86
		4	ind	5	40568	47.31	15.70	63.28
		5	TO	–	–	–	–	–

Evaluation of Benchmark Set 1. Table 2 empirically underlines that probabilistic program verification can benefit from k-induction to the same extent as classical software verification: KIPRO2 *fully automatically* verifies relevant properties of *infinite-state* randomized algorithms and stochastic processes from the literature that require *k to be strictly larger than* 1. That is, proving these properties using (1-)inductive invariants requires either non-trivial invariant synthesis or additional user annotations. This indicates that k-induction mitigates the need for complicated specifications in probabilistic program verification (cf. [40]).

We observe that k-induction tends to succeed if *some* variable is bounded in the candidate upper bound under consideration (cf. `brp`, `rabin`, `unif_gen`). However, k-induction can also succeed without any bounds (cf. `geo`). The time and formulae required for checking k-inductivity increases rapidly for larger k; this is particularly striking for `rabin` and `unif_gen`. When refuting candidate bounds with BMC, we obtain a similar picture. Both the time and formulae required for refutation increase if the candidate bound increases (cf. `brp`, `geo`, `rabin`).

For both k-induction and BMC, we observe a direct correlation between the complexity of the loop, i.e., the number of possible traces through the loop from some fixed initial state after some bounded number of iterations, and the required time and space (number of formulae). Whereas for `geo` and `brp`—which exhibit a rather simple structure—these checks tend to be fast, this is not the case for `rabin` and `unif_gen`, which have more complex loop bodies. For such complex loops, k-induction and BMC quickly become infeasible as k increases.

Table 3. Empirical results for (a subset of) the ERTs [56] (time in *milliseconds*).

	runtime bound candidate	result	k	#formulae	formulae_t	sat_t	total_t
`2drwalk`	$2 \cdot (n + 1 \dot{-} d)$	TO	–	–	–	–	–
`bayesian_network`	$5 \cdot n$	TO	–	–	–	–	–
`ber`	$2 \cdot (n \dot{-} x)$	ind	1	9	7.22	0.44	88.12
`C4B_t303`	$0.5 \cdot (x + 2) + 0.5 \cdot (y + 2)$	ind	3	129	91.38	10.01	216.11
`condand`	$m + n$	ind	1	10	7.10	0.43	76.21
`fcall`	$2 \cdot (n \dot{-} x)$	ind	1	9	6.73	0.41	75.73
`hyper`	$5 \cdot (n \dot{-} x)$	ind	1	11	7.24	0.46	97.52
`linear01`	$0.6 \cdot x$	ind	1	11	7.19	0.49	74.38
`prdwalk`	$1.14286 \cdot (n + 4 \dot{-} x)$	ind	1	17	7.64	0.72	194.44
`prspeed`	$2 \cdot (m \dot{-} y) + 0.6666667 \cdot (n \dot{-} x)$	ind	1	18	7.64	0.81	145.13
`race`	$0.666667 \cdot (t + 9 \dot{-} h)$	ind	1	30	9.21	0.86	695.89
`rdspeed`	$2 \cdot (m \dot{-} y) + 0.666667 \cdot (n \dot{-} x)$	ind	1	19	7.70	0.78	143.45
`rdwalk`	$2 \cdot (n + 1 \dot{-} x)$	ind	1	12	10.22	0.75	85.03
`sprdwalk`	$2 \cdot (n \dot{-} x)$	ind	1	9	7.28	0.42	83.40

Evaluation of Benchmark Set 2. From Table 3, we observe that—in almost every case—verification is instantaneous and requires very few formulae. The programs we verify are equivalent to the programs provided in [56] up to interpreting minus as *monus* and using $\mathbb{N}$-typed (instead of $\mathbb{Z}$) variables. A manual inspection reveals that this matters for `C4B_t303` and `rdwalk`, which is the reason why the runtime bound for `C4B_t303` is 3-inductive rather than 1-inductive.

There are two timeouts (`2drwalk`, `bayesian_network`) due to the GNF construction from Lemma 3, which exhibits a runtime exponential in the number of possible execution branches through the loop body. We conjecture that further preprocessing (by pruning infeasible branches upfront) can mitigate this, rendering `2drwalk` and `bayesian_network` tractable as well. We consider a thorough investigation of suitable preprocessing strategies for GNF construction, which is outside the scope of this paper, a worthwhile direction for future research.

10 Conclusion

We presented κ-induction, a generalization of classical k-induction to arbitrary complete lattices, and—together with a complementary bounded model checking approach—obtained a fully automated technique for verifying infinite-state probabilistic programs. Experiments showed that this technique can prove non-trivial properties in an automated manner that using existing techniques cannot be proven—at least not without synthesizing a stronger inductive invariant. If a given candidate bound is k-inductive for some k, then our prototypical tool will find that k for linear programs and linear expectations. In theory, our tool is also applicable to non-linear programs at the expense of an undecidability quantitative entailment problem. It is left for future work to consider (positive) real-valued program variables for non-linear expectations.

Acknowledgements. B. L. Kaminski thanks Larry Fischer for his linguistic advice.

References

1. Abramsky, Jung: Domain theory. In: Handbook of Logic in Computer Science, vol. 3 (1994)
2. Agrawal, Chatterjee, Novotný: Lexicographic ranking supermartingales. PACMPL **2**(POPL) (2018)
3. Amtoft, Banerjee: A theory of slicing for imperative probabilistic programs. TOPLAS **42**(2) (2020)
4. Baier, C., Klein, J., Leuschner, L., Parker, D., Wunderlich, S.: Ensuring the reliability of your model checker: interval iteration for Markov decision processes. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 160–180. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_8
5. Baldan, et al.: Fixpoint theory - upside down. In: FoSSaCS (2021)
6. Barthe, G., Espitau, T., Ferrer Fioriti, L.M., Hsu, J.: Synthesizing probabilistic invariants via Doob's decomposition. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 43–61. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_3
7. Batz, K., Junges, S., Kaminski, B.L., Katoen, J.-P., Matheja, C., Schröer, P.: PrIC3: property directed reachability for MDPs. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12225, pp. 512–538. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53291-8_27
8. Batz, et al.: Latticed k-induction with an application to probabilistic programs (extended version). arXiv (2021)
9. Batz, et al.: Relatively complete verification of probabilistic programs. PACMPL **5**(POPL) (2021)
10. Beyer, D., Dangl, M., Wendler, P.: Boosting k-induction with continuously-refined invariants. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 622–640. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_42
11. Biere: Bounded model checking. In: Handbook of Satisfiability (2009)
12. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-49059-0_14

13. Biere, A., Clarke, E., Raimi, R., Zhu, Y.: Verifying safety properties of a PowerPC – microprocessor using symbolic model checking without BDDs. In: Halbwachs, N., Peled, D. (eds.) CAV 1999. LNCS, vol. 1633, pp. 60–71. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48683-6_8
14. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_7
15. Chadha, Viswanathan: A counterexample-guided abstraction-refinement framework for Markov decision processes. TOCL **12**(1) (2010)
16. Chakarov, A., Sankaranarayanan, S.: Probabilistic program analysis with martingales. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 511–526. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_34
17. Clarke, et al.: Bounded model checking using satisfiability solving. Formal Methods Syst. Des. **19**(1) (2001)
18. Cousot, Cousot: Constructive versions of Tarski's fixed point theorems. Pacific J. Math. **82**(1) (1979)
19. D'Argenio, P.R., Jeannet, B., Jensen, H.E., Larsen, K.G.: Reachability analysis of probabilistic systems by successive refinements. In: de Alfaro, L., Gilmore, S. (eds.) PAPM-PROBMIV 2001. LNCS, vol. 2165, pp. 39–56. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44804-7_3
20. Déharbe, D., Moreira, A.M.: Using induction and BDDs to model check invariants. In: Advances in Hardware Design and Verification. IAICT, vol. 105, pp. 203–213. Springer, Boston, MA (1997). https://doi.org/10.1007/978-0-387-35190-2_13
21. Donaldson, A.F., Kroening, D., Rümmer, P.: Automatic analysis of scratch-pad memory code for heterogeneous multicore processors. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 280–295. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12002-2_24
22. Donaldson, Kroening, Rümmer: Automatic analysis of DMA races using model checking and k-induction. Formal Methods Syst. Des. **39**(1) (2011)
23. Donaldson, A.F., Haller, L., Kroening, D., Rümmer, P.: Software verification using *k*-induction. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 351–368. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23702-7_26
24. Feng, Y., Zhang, L., Jansen, D.N., Zhan, N., Xia, B.: Finding polynomial loop invariants for probabilistic programs. In: D'Souza, D., Narayan Kumar, K. (eds.) ATVA 2017. LNCS, vol. 10482, pp. 400–416. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68167-2_26
25. Gario, Micheli: PySMT: a solver-agnostic library for fast prototyping of SMT-based algorithms. In: SMT Workshop (2015)
26. Gehr, T., Misailovic, S., Vechev, M.: PSI: exact symbolic inference for probabilistic programs. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 62–83. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_4
27. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-63166-6_10
28. Gretz, Katoen: McIver: operational versus weakest pre-expectation semantics for the probabilistic guarded command language. Perform. Eval. **73** (2014)
29. Gurfinkel, Ivrii: K-induction without unrolling. In: FMCAD (2017)
30. Han, Katoen, Damman: Counterexample generation in probabilistic model checking. IEEE Trans. Softw. Eng. **35**(2) (2009)

31. Hartmanns, A., Kaminski, B.L.: Optimistic value iteration. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12225, pp. 488–511. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53291-8_26
32. Helmink, L., Sellink, M.P.A., Vaandrager, F.W.: Proof-checking a data link protocol. In: Barendregt, H., Nipkow, T. (eds.) TYPES 1993. LNCS, vol. 806, pp. 127–165. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58085-9_75
33. Huang, Z., Wang, Z., Misailovic, S.: PSense: automatic sensitivity analysis for probabilistic programs. In: Lahiri, S.K., Wang, C. (eds.) ATVA 2018. LNCS, vol. 11138, pp. 387–403. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01090-4_23
34. Hurd, McIver, Morgan: Probabilistic guarded commands mechanized in HOL. Theor. Comput. Sci. **346**(1) (2005)
35. Jansen, N., Dehnert, C., Kaminski, B.L., Katoen, J.-P., Westhofen, L.: Bounded model checking for probabilistic programs. In: Artho, C., Legay, A., Peled, D. (eds.) ATVA 2016. LNCS, vol. 9938, pp. 68–85. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46520-3_5
36. Jhala, R., McMillan, K.L.: A practical and complete approach to predicate refinement. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 459–473. Springer, Heidelberg (2006). https://doi.org/10.1007/11691372_33
37. Jovanović, Dutertre: Property-directed k-induction. In: FMCAD (2016)
38. Kaminski: Advanced weakest precondition calculi for probabilistic programs. Ph.D. thesis, RWTH Aachen University, Germany (2019)
39. Kaminski, Katoen, Matheja: On the hardness of analyzing probabilistic programs. Acta Inform. **56**(3) (2019)
40. Kaminski, et al.: Weakest precondition reasoning for expected runtimes of randomized algorithms. J. ACM **65**(5) (2018)
41. Katoen, J.-P., McIver, A.K., Meinicke, L.A., Morgan, C.C.: Linear-invariant generation for probabilistic programs: automated support for proof-based methods. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 390–406. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15769-1_24
42. King, Barrett, Tinelli: Leveraging linear and mixed integer programming for SMT. In: SMT (2014)
43. Knaster: Un théorème sur les functions d'ensembles. Ann. Soc. Pol. Math. **6** (1928)
44. Kozen: A probabilistic PDL. J. Comput. Syst. Sci. **30**(2) (1985)
45. Vediramana Krishnan, H.G., Vizel, Y., Ganesh, V., Gurfinkel, A.: Interpolating strong induction. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11562, pp. 367–385. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25543-5_21
46. Kushilevitz, Rabin: Randomized mutual exclusion algorithms revisited. In: PODC (1992)
47. Lassez, Nguyen, Sonenberg: Fixed point theorems and semantics. Inf. Process. Lett. **14**(3) (1982)
48. Lumbroso: Optimal discrete uniform generation from coin flips, and applications. arXiv (2013)
49. McIver, Morgan: Abstraction, refinement and proof for probabilistic systems (2005)
50. McMillan, K.L.: Interpolation and SAT-based model checking. In: Hunt, W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45069-6_1
51. McMillan: An interpolating theorem prover. Theor. Comput. Sci. **345**(1) (2005)
52. Meyer, Hark, Giesl: Inferring expected runtimes of probabilistic integer programs using expected sizes. In: TACAS (2021, to appear)

53. Milner: Communication and concurrency (1989)
54. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
55. de Moura, L., Rueß, H., Sorea, M.: Bounded model checking and induction: from refutation to verification. In: Hunt, W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 14–26. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45069-6_2
56. Ngo, Carbonneaux, Hoffmann: Bounded expectations: resource analysis for probabilistic programs. In: PLDI (2018)
57. Park: Fixpoint induction and proofs of program properties. Mach. Intell. **5** (1969)
58. Pous, D.: Complete lattices and up-to techniques. In: Shao, Z. (ed.) APLAS 2007. LNCS, vol. 4807, pp. 351–366. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-76637-7_24
59. Pous, Sangiorgi: Enhancements of the bisimulation proof method. In: Advanced Topics in Bisimulation and Coinduction, vol. 52 (2012)
60. Puterman: Markov Decision Processes (1994)
61. Quatmann, T., Katoen, J.-P.: Sound value iteration. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 643–661. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_37
62. Rabehaja, Sanders: Refinement algebra with explicit probabilism. In: TASE (2009)
63. Rocha, W., Rocha, H., Ismail, H., Cordeiro, L., Fischer, B.: DepthK: a k-induction verifier based on invariant inference for C programs. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10206, pp. 360–364. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54580-5_23
64. Schüle, Schneider: Bounded model checking of infinite state systems. Formal Methods Syst. Des. **30**(1) (2007)
65. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: Hunt, W.A., Johnson, S.D. (eds.) FMCAD 2000. LNCS, vol. 1954, pp. 127–144. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-40922-X_8
66. Tarski: A lattice-theoretical fixpoint theorem and its applications. Pacific J. Math. **5**(2) (1955)
67. Wang, Hoffmann, Reps: PMAF: an algebraic framework for static analysis of probabilistic programs. In: PLDI (2018)
68. Wimmer, R., Braitling, B., Becker, B.: Counterexample generation for discrete-time Markov chains using bounded model checking. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 366–380. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-93900-9_29

Stochastic Systems

Runtime Monitors for Markov Decision Processes

Sebastian Junges(✉), Hazem Torfah, and Sanjit A. Seshia

University of California at Berkeley, Berkeley, USA
sjunges@berkeley.edu

Abstract. We investigate the problem of monitoring partially observable systems with nondeterministic and probabilistic dynamics. In such systems, every state may be associated with a risk, e.g., the probability of an imminent crash. During runtime, we obtain partial information about the system state in form of observations. The monitor uses this information to estimate the risk of the (unobservable) current system state. Our results are threefold. First, we show that extensions of state estimation approaches do not scale due the combination of nondeterminism and probabilities. While exploiting a geometric interpretation of the state estimates improves the practical runtime, this cannot prevent an exponential memory blowup. Second, we present a tractable algorithm based on model checking conditional reachability probabilities. Third, we provide prototypical implementations and manifest the applicability of our algorithms to a range of benchmarks. The results highlight the possibilities and boundaries of our novel algorithms.

1 Introduction

Runtime assurance is essential in the deployment of safety-critical (cyber-physical) systems [12,29,45,49,50]. Monitors observe system behavior and indicate when the system is at risk to violate system specifications. A critical aspect in developing reliable monitors is their ability to handle noisy or missing data. In cyber-physical systems, monitors observe the system state via sensors, i.e., sensors are an interface between the system and the monitor. A monitor has to base its decision solely on the obtained sensor output. These sensors are not perfect, and not every aspect of a system state can be measured.

This paper considers a model-based approach to the construction of monitors for systems with imprecise sensors. Consider Fig. 1(b). We assume a model for the environment together with the controller. Typically, such a model contains both nondeterministic and probabilistic behavior, and thus describes a Markov decision process (MDP): In particular, the sensor is a stochastic process [56] that

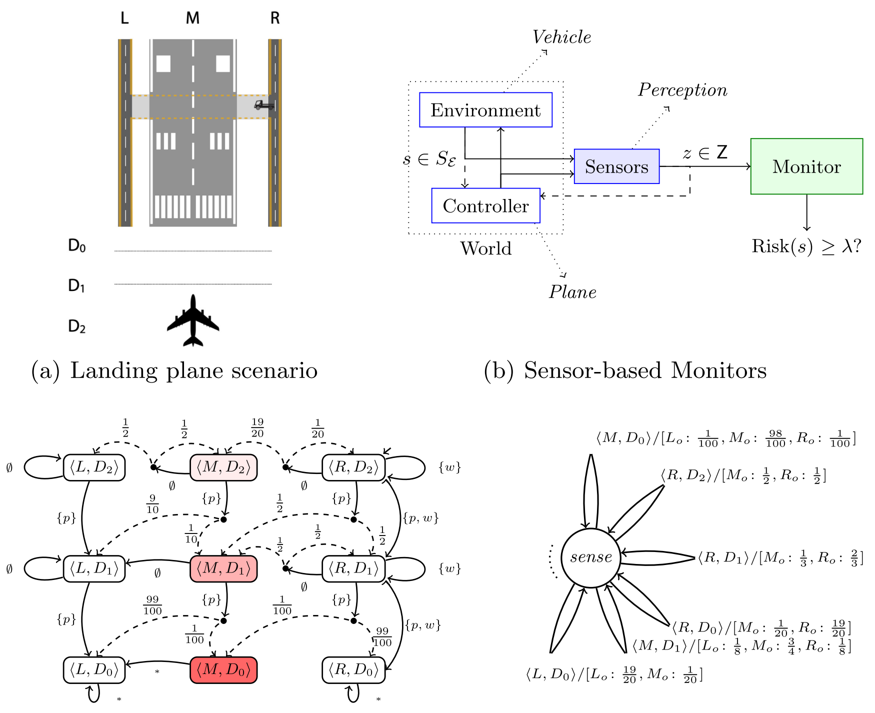
This work is partially supported by NSF grants 1545126 (VeHICaL), 1646208 and 1837132, by the DARPA contracts FA8750-18-C-0101 (AA) and FA8750-20-C-0156 (SDCPS), by Berkeley Deep Drive, and by Toyota under the iCyPhy center.

translates the environment state into an observation. For example, this could be a perception module on a plane that during landing estimates the movements of an on-ground vehicle, as depicted in Fig. 1(a). Due to lack of precise data, the vehicle movements itself may be most accurately described using nondeterminism.

We are interested in the associated *state risk* of the current system state. The state risk may encode, e.g., the probability that the plane will crash with the vehicle within a given number of steps, or the expected time until reaching the other side of the runway. The challenge is that the monitor cannot directly observe the current system state. Instead, the monitor must infer from a trace of observations the current state risk. This cannot be done perfectly as the system state cannot be inferred precisely. Rather, we want a sound, conservative estimate of the system state. More concretely, for a fixed resolution of the nondeterminism, the *trace risk* is the weighted sum over the probability of being in a state having observed the trace, times the risk imposed by this state. The monitoring problem is to decide whether for any possible scheduler resolving the nondeterminism the trace risk of a given trace exceeds a threshold.

Monitoring of systems that contain either only probabilistic or only nondeterministic behavior is typically based on *filtering*. Intuitively, the monitor then estimates the current system states based on the model. For purely nondeterministic systems (without probabilities) a set of states needs to be tracked, and purely probabilistic systems (without nondeterminism) require tracking a distribution over states. This tracking is rather efficient. For systems that contain both probabilistic and nondeterministic behavior, filtering is more challenging. In particular, we show that filtering on MDPs results in an exponential memory blowup as the monitor must track sets of distributions. We show that a reduction based on the geometric interpretation of these distributions is essential for practical performance, but cannot avoid the worst-case exponential blowup. As a tractable alternative to filtering, we rephrase the monitoring problem as the computation of conditional reachability probabilities [9]. More precisely, we unroll and transform the given MDP, and then model check this MDP. This alternative approach yields a polynomial-time algorithm. Indeed, our experiments show the feasibility of computing the risk by computing conditional probabilities. We also show benchmarks on which filtering is a competitive option.

Contribution and Outline. This paper presents the first runtime monitoring for systems that can be adequately abstracted by a combination of *probabilities and nondeterminism* and where the system state is *partially observable*. We describe the use case, show that typical filtering approaches in general fail to deal with this setting, and show that a tractable alternative solution exists. In Sect. 3, we investigate *forward filtering*, used to estimate the possible system states in partially observable settings. We show that this approach is tractable for systems that have probabilistic *or* nondeterministic uncertainty, but not for systems that have both. To alleviate the blowup, Sect. 4 discusses an (often) efficacious pruning strategy and its limitations. In Sect. 5 we consider model checking as a more tractable alternative. This result utilizes constructions from the analysis



(a) Landing plane scenario (b) Sensor-based Monitors

(c) World Model (d) Partial Sensor Model

Fig. 1. A probabilistic world and sensor model represented by two MDPs for the scenario of an airplane in landing approach with on-ground vehicle movements.

of *partially observable MDPs* and model checking MDPs with *conditional properties*. In Sect. 6 we present baseline implementations of these algorithms, on top of the open-source model checker STORM, and evaluate their performance. The results show that the implementation allows for monitoring of a variety of MDPs, and reveals both strengths and weaknesses of both algorithms. We start with a motivating example and review related work at the end of the paper.

Motivating Example. Consider a scenario where an autonomous airplane is in its final approach, i.e., lined up with a designated runway and descending for landing, see Fig. 1(a). On the ground, close to the runway, maintenance vehicles may cross the runway. The airplane tracks the movements of these vehicles and has to decide, depending on the movements of the vehicles, whether to abort the landing. To simplify matters, assume that the airplane (P) is tracking the movement of one vehicle (V) that is about to cross the runway. Let us further assume that P tracks V using a perception module that can only determine the position of the vehicle with a certain accuracy [33], i.e., for every position of V, the perception module reports a noisy variant of the position of V. However, it is important to know that the plane obtains a sequence of these measurements.

Figure 1 illustrates the dynamics of the scenario. The world model describing the movements of V and P is given in Fig. 1(c), where D_2, D_1, and D_0 define how close P is to the runway, and R, M, and L define the position of V. Depending on what information V perceives about P, given by the atomic proposition $\{(p)\mathit{rogress}\}$, and what commands it receives $\{(w)\mathit{ait}\}$, it may or may not cross the runway. The perception module receives the information about the state of the world and reports with a certain accuracy (given as a probability) the position of V. The (simple) model of the perception module is given in Fig. 1(d). For example, if P is in zone D_2 and V is in R then there is high chance that the perception module returns that V is on the runway. The probability of incorrectly detecting V's position reduces significantly when P is in D_0.

A monitor responsible for making the decision to land or to perform a go-around based on the information computed by the perception module, must take into consideration the accuracy of this returned information. For example, if the sequence of sensor readings passed to the monitor is the sequence $\tau = R_o{\cdot}R_o{\cdot}M_o$, and each state is mapped to a certain risk, then how risky is it to land after seeing τ? If, for instance, the world is with high probability in state $\langle M, D_0 \rangle$, a very risky state, then the plane should go around. In the paper, we address the question of computing the risk based on this observation sequence. We will use this example as our running example.

2 Monitoring with Imprecise Sensors

In this section, we formalize the problem of monitoring with imprecise sensors when both the world and sensor models are given by MDPs. We start with a recap of MDPs, define the monitoring problem for MDPs, and finally show how the dynamics of the system under inspection can be modeled by an MDP defined by the composition of two MDPs of the sensors and world model of the system.

2.1 Markov Decision Processes

For a countable set X, let $\mathsf{Distr}(X) \subset (X \to [0,1])$ define the set of all distributions over X, i.e., for $d \in \mathsf{Distr}(X)$ it holds that $\Sigma_{x \in X} d(x) = 1$. For $d \in \mathsf{Distr}(X)$, let the *support* of d be defined by $\mathsf{supp}(d) := \{x \mid d(x) > 0\}$. We call a distribution d *Dirac*, if $|\mathsf{supp}(d)| = 1$.

Definition 1 (Markov decision process). *A* Markov decision process *is a tuple* $\mathcal{M} = \langle S, \iota, \mathsf{Act}, P, \mathsf{Z}, \mathsf{obs} \rangle$*, where* S *is a finite set of* states*,* $\iota \in \mathsf{Distr}(S)$ *is an* initial distribution*,* Act *is a finite set of* actions*,* $P \colon S \times \mathsf{Act} \to \mathsf{Distr}(S)$ *is a* partial transition function*,* Z *is a finite set of* observations*, and* $\mathsf{obs} \colon S \to \mathsf{Distr}(\mathsf{Z})$ *is a* observation function*.*

Remark 1. The observation function can also be defined as a state-action observation function $\mathsf{obs} \colon S \times \mathsf{Act} \to \mathsf{Distr}(\mathsf{Z})$. MDPs with state-action observation function can be easily transformed into equivalent MDPs with a state observation function using auxiliary states [19]. Throughout the paper we use state-action observations to keep (sensor) models concise.

For a state $s \in S$, we define $\mathsf{AvAct}(s) = \{\alpha \mid P(s,\alpha) \neq \bot\}$. W.l.o.g., $|\mathsf{AvAct}(s)| \geq 1$. If all distributions in $\mathcal{M}$ are Dirac, we refer to $\mathcal{M}$ as a *Kripke structure* (KS). If $|\mathsf{AvAct}(s)| = 1$ for all $s \in S$, we refer to $\mathcal{M}$ as a *Markov chain* (MC). When $\mathsf{Z} = S$, we refer to $\mathcal{M}$ as *fully observable* and omit Z and obs from its definition. A *finite path* in an MDP $\mathcal{M}$ is a sequence $\pi = s_0 a_0 s_1 \dots s_n \in S \times \left(\mathsf{Act} \times S\right)^*$ such that for every $0 \leq i < n$ it holds that $P(s_i, a_i)(s_{i+1}) > 0$ and $\iota(s_0) > 0$. We denote the set of finite paths of $\mathcal{M}$ by $\varPi_{\mathcal{M}}$. The *length* of the path is given by the number of actions along the path. The set $\varPi^n_{\mathcal{M}}$ for some $n \in \mathbb{N}$ denotes the set of finite paths of length n. We use $\pi_{\downarrow}$ to denote the last state in π. We omit $\mathcal{M}$ whenever it is clear from the context. A *trace* is a sequence of observations $\tau = z_0 \dots z_n \in \mathsf{Z}^+$. Every path induces a distribution over traces.

As standard, any nondeterminism is resolved by means of a scheduler.

Definition 2 (Scheduler). *A* scheduler *for an MDP $\mathcal{M}$ is a function $\sigma \colon \varPi_{\mathcal{M}} \to \mathsf{Distr}(\mathsf{Act})$ with $\mathsf{supp}(\sigma(\pi)) \subseteq \mathsf{AvAct}(\pi_{\downarrow})$ for every $\pi \in \varPi_{\mathcal{M}}$.*

We use $\mathit{Sched}(\mathcal{M})$ to denote the set of schedulers. For a fixed scheduler $\sigma \in \mathit{Sched}(\mathcal{M})$, the probability $\mathsf{Pr}_{\sigma}(\pi)$ of a path π (under the scheduler σ) is the product of the transition probabilities in the induced Markov chain. For more details we refer the reader to [8].

2.2 Formal Problem Statement

Our goal is to determine the risk that a system is exposed to having observed a trace $\tau \in \mathsf{Z}^+$. Let $r \colon S \to \mathbb{R}_{\geq 0}$ map states in $\mathcal{M}$ to some risk in $\mathbb{R}_{\geq 0}$. We call r a *state-risk function* for $\mathcal{M}$. This function maps to the risk that is associated with being in every state. For example, in our experiments, we flexibly define the state risk using the (expected reward extension of the) temporal logic PCTL [8], to define the probability of reaching a fail state. For example, we can define risk as the probability to crash within H steps. The use of expected rewards allows for even more flexible definitions.

Intuitively, to compute this risk of the system we need to determine the current system state having observed τ, considering both the probabilistic and nondeterministic context. To this end, we formalize the (conditional) probabilities and risks of paths and traces. Let $\mathsf{Pr}_{\sigma}(\pi \mid \tau)$ define the probability of a path π, under a scheduler σ, having observed τ. Since a scheduler may define many paths that induce the observation trace τ, we are interested in the weighted risk over all paths, i.e., $\sum_{\pi \in \varPi^{|\tau|}_{\mathcal{M}}} \mathsf{Pr}_{\sigma}(\pi \mid \tau) \cdot r(\pi_{\downarrow})$. The monitoring problem for MDPs then conservatively over-approximates the risk of a trace by assuming an adversarial scheduler, that is, by taking the supremum risk estimate over all schedulers[1].

[1] We later see in Lemma 8 that this is indeed a maximum.

The Monitoring Problem. Given an MDP $\mathcal{M}$, a state-risk $r\colon S \to \mathbb{R}_{\geq 0}$, an observation trace $\tau \in \mathsf{Z}^+$, and a threshold $\lambda \in [0, \infty)$, decide $R_r(\tau) > \lambda$, where the *weighted risk function* $R_r\colon \mathsf{Z}^+ \to \mathbb{R}_{\geq 0}$ is defined as

$$R_r(\tau) \quad := \quad \sup_{\sigma \in Sched(\mathcal{M})} \sum_{\pi \in \Pi_{\mathcal{M}}^{|\tau|}} \mathsf{Pr}_\sigma(\pi \mid \tau) \cdot r(\pi_\downarrow).$$

The conditional probability $\mathsf{Pr}_\sigma(\pi \mid \tau)$ can be characterized using Bayes' rule[2]:

$$\mathsf{Pr}_\sigma(\pi \mid \tau) = \frac{\mathsf{Pr}(\tau \mid \pi) \cdot \mathsf{Pr}_\sigma(\pi)}{\mathsf{Pr}_\sigma(\tau)}.$$

The probability $\mathsf{Pr}(\tau \mid \pi)$ of a trace τ for a fixed path π is $\mathsf{obs}_{\mathsf{tr}}(\pi)(\tau)$, where

$$\mathsf{obs}_{\mathsf{tr}}(s) := \mathsf{obs}(s), \quad \mathsf{obs}_{\mathsf{tr}}(\pi\alpha s') := \{\tau \cdot z \mapsto \mathsf{obs}_{\mathsf{tr}}(\pi)(\tau) \cdot \mathsf{obs}(s')(z)\},$$

when $|\pi| = |\tau|$, and $\mathsf{obs}_{\mathsf{tr}}(\pi)(\tau) = 0$ otherwise. The probability $\mathsf{Pr}_\sigma(\tau)$ of a trace τ is $\sum_\pi \mathsf{Pr}_\sigma(\pi) \cdot \mathsf{Pr}(\tau \mid \pi)$.

We call the special variant with $\lambda = 0$ the *qualitative monitoring problem.* The problems are (almost) equivalent on Kripke structures, where considering a single path to an adequate state suffices. Details are given in [36, Appendix].

Lemma 1. *For Kripke structures the monitoring and qualitative monitoring problems are logspace interreducible.*

In the next sections we present two types of algorithms for the monitoring problem. The first algorithm is based on the widespread (forward) filtering approach [44]. The second is new algorithm based on model checking conditional probabilities. While filtering approaches are efficacious in a purely nondeterministic or a purely probabilistic setting, it does not scale on models such as MDPs that are both probabilistic and nondeterministic. In those models, model checking provides a tractable alternative. Before going into details, we first connect the problem statement more formally to our motivating example.

2.3 An MDP Defining the System Dynamics

We show how the weighted risk for a system given by a world and sensor model can be formalized as a monitoring problem for MDPs. To this end, we define the dynamics of the world and sensors that we use as basis for our monitor as the following joint MDP.

For a fully observable world MDP $\mathcal{E} = \langle S_\mathcal{E}, \iota_\mathcal{E}, \mathsf{Act}_\mathcal{E}, P_\mathcal{E}\rangle$ and a sensor MDP $\mathcal{S} = \langle S_\mathcal{S}, \iota_\mathcal{S}, S_\mathcal{E}, P_\mathcal{S}, \mathsf{Z}, \mathsf{obs}\rangle$, where obs is state-action based, the *inspected system* is defined by an MDP $[\![\langle \mathcal{E}, \mathcal{S}\rangle]\!] = \langle S_\mathcal{J}, \iota_\mathcal{J}, \mathsf{Act}_\mathcal{E}, P_\mathcal{J}, \mathsf{Z}, \mathsf{obs}_\mathcal{J}\rangle$ being the synchronous composition of $\mathcal{E}$ and $\mathcal{S}$:

[2] For conciseness we assume throughout the paper that $\frac{0}{0} = 0$.

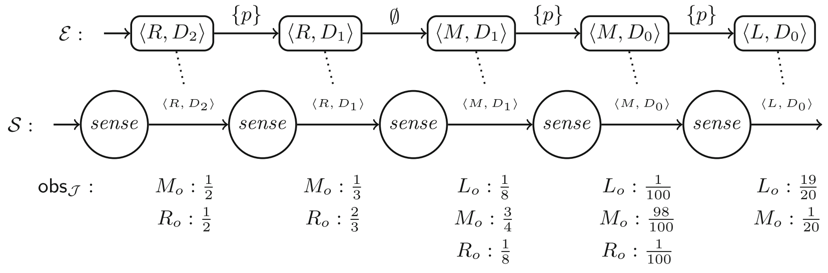


Fig. 2. A run with its observations of the inspected system $[\![\langle \mathcal{E}, \mathcal{S}\rangle]\!]$ where $\mathcal{E}$ and $\mathcal{S}$ are the models given in Fig. 1.

- $S_{\mathcal{J}} := S_{\mathcal{E}} \times S_{\mathcal{S}}$,
- $\iota_{\mathcal{J}}$ is defined as $\iota_{\mathcal{J}}(\langle u, s\rangle) := \iota_{\mathcal{E}}(u) \cdot \iota_{\mathcal{S}}(s)$ for each $u \in S_{\mathcal{E}}$ and $s \in S_{\mathcal{S}}$,
- $P_{\mathcal{J}} \colon S_{\mathcal{J}} \times \mathsf{Act}_{\mathcal{E}} \to \mathsf{Distr}(S_{\mathcal{J}})$ such that for all $\langle u, s\rangle \in S_{\mathcal{J}}$ and $\alpha \in \mathsf{Act}_{\mathcal{E}}$;

$$P_{\mathcal{J}}(\langle u, s\rangle, \alpha) = d_{u,s} \in \mathsf{Distr}(S_{\mathcal{J}}),$$

  where for all $u' \in S_{\mathcal{E}}$ and $s' \in S_{\mathcal{S}}$: $d_{u,s}(\langle u', s'\rangle) = P_{\mathcal{E}}(u, \alpha)(u') \cdot P_{\mathcal{S}}(s, u)(s')$,
- $\mathsf{obs}_{\mathcal{J}} \colon S_{\mathcal{J}} \to \mathsf{Distr}(\mathsf{Z})$ with $\mathsf{obs}_{\mathcal{J}} \colon \langle u, s\rangle \mapsto \mathsf{obs}(s, u)$.

In Fig. 2 we illustrate a run of $[\![\langle \mathcal{E}, \mathcal{S}\rangle]\!]$ for the world and sensor MDPs presented in Fig. 1. We particularly show the observations of the joint MDP given by the distributions over the observations for each transition in the run (we omitted the probabilistic transitions for simplicity). The observations of the MDP $\mathcal{M}$ present the output of the sensor upon a path through $\mathcal{M}$. These observations in turn are the inputs to a monitor on top of the system. The role of the monitor is then to compute the risk of being in a critical state based on the received observations.

3 Forward Filtering for State Estimation

We start by showing why standard forward filtering does not scale well on MDPs. We briefly show how filtering can be used to solve the monitoring problem for purely nondeterministic systems (Kripke structures) or purely probabilistic systems (Markov Chains). Then, we show why for MDPs, the forward filtering needs to manage, although finite but an exponential set of distributions. In Sect. 4 we present a new improved variant of forward filtering for MDPs based on filtering with vertices of the convex hull. In Sect. 5 we present a new polynomial-time model checking-based algorithm for solving the problem.

3.1 State Estimators for Kripke Structures.

For Kripke structures, we maintain a set of possible states that agree with the observed trace. This set of states is inductively characterized by the function

$\mathsf{est}_{\mathsf{KS}} \colon \mathsf{Z}^+ \to 2^S$ which we define formally below. For an observation trace τ, $\mathsf{est}_{\mathsf{KS}}(\tau)$ defines the set of states that can be reached with positive probability. This set can be computed by a forward state traversal [31]. To illustrate how $\mathsf{est}_{\mathsf{KS}}(\tau)$ is computed for τ, consider the underlying Kripke structure of the inspected system $[\![\langle \mathcal{E}, \mathcal{S} \rangle]\!]$ for our running example in Fig. 1 (to make this a Kripke structure, we remove the probabilities). Consider further the observation trace $\tau = R_o \cdot M_o \cdot L_o$. Since $[\![\langle \mathcal{E}, \mathcal{S} \rangle]\!]$ has only one initial state $\langle \langle R, D_2 \rangle, \mathit{sense} \rangle$ and R_o is observable with a positive probability in this state, $\mathsf{est}_{\mathsf{KS}}(R_o) = \{\langle \langle R, D_2 \rangle, \mathit{sense} \rangle\}$. As M_o is observed next, $\mathsf{est}_{\mathsf{KS}}(R_o \cdot M_o)$ computes the states reached from $\langle \langle R, D_2 \rangle, \mathit{sense} \rangle$ and where M_o can be observed with a positive probability, i.e., $\mathsf{est}_{\mathsf{KS}}(R_o \cdot M_o) = \{\langle \langle R, D_1 \rangle, \mathit{sense} \rangle, \langle \langle R, M_1 \rangle, \mathit{sense} \rangle\}$. Finally, the current state having observed $R_o \cdot M_o \cdot L_o$ may be one of the states $\mathsf{est}_{\mathsf{KS}}(\tau) = \{\langle \langle M, D_1 \rangle, \mathit{sense} \rangle, \ \langle \langle L, D_1 \rangle, \mathit{sense} \rangle, \ \langle \langle L, D_0 \rangle, \mathit{sense} \rangle, \ \langle \langle M, D_0 \rangle, \mathit{sense} \rangle\}$, which especially shows that we might be in the high-risk world state $\langle M, D_0 \rangle$.

Definition 3 (KS state estimator). *For* $\mathsf{KS} = \langle S, \iota, \mathsf{Act}, P, \mathsf{Z}, \mathsf{obs} \rangle$*, the state estimation function* $\mathsf{est}_{\mathsf{KS}} \colon \mathsf{Z}^+ \to 2^S$ *is defined as*

$$\mathsf{est}_{\mathsf{KS}}(z) := \{ s \in S \mid \iota(s) > 0 \land \mathsf{obs}(s)(z) > 0 \}$$
$$\mathsf{est}_{\mathsf{KS}}(\tau \cdot z) := \Big\{ s' \in S \mid \exists s \in \mathsf{est}_{\mathsf{KS}}(\tau), \exists \alpha \in \mathsf{Act}, P(s, \alpha)(s') > 0 \land \mathsf{obs}(s')(z) > 0 \Big\}.$$

For a Kripke structure KS and a given trace τ, the monitoring problem can be solved by computing $\mathsf{est}_{\mathsf{KS}}(\tau)$, using [31] and Lemma 1.

Lemma 2. *For a Kripke stucture* $\mathsf{KS} = \langle S, \iota, \mathsf{Act}, P, \mathsf{Z}, \mathsf{obs} \rangle$*, a trace* $\tau \in \mathsf{Z}^+$*, and a state-risk function* $r \colon S \to \mathbb{R}_{\geq 0}$*, it holds that* $R_r(\tau) = \max_{s \in \mathsf{est}_{\mathsf{KS}}(\tau)} r(s)$*. Computing* $R_r(\tau)$ *requires time* $\mathcal{O}(|\tau| \cdot |P|)$ *and space* $\mathcal{O}(|S|)$*.*

A proof can be found in [36, Appendix]. The time and space requirements follow directly from the inductive definition of $\mathsf{est}_{\mathsf{KS}}$ which resembles solving a forward state traversal problem in automata [31]. In particular, the algorithm allows updating the result after extending τ in $\mathcal{O}(|P|)$.

3.2 State Estimators for Markov Chains

For Markov chains, in addition to tracking the potential reachable system states, we also need to take the transition probabilities into account. When a system is (observation-)deterministic, we can adapt the notion of beliefs, similar to RVSE [54], and similar to the construction of belief MDPs for *partially observable MDPs*, cf. [53]:

Definition 4 (Belief). *For an MDP* $\mathcal{M}$ *with a set of states* S*, a belief* bel *is a distribution in* $\mathsf{Distr}(S)$*.*

In the remainder of the paper, we will denote the function $S \to \{0\}$ by $\mathbf{0}$ and the set $\mathsf{Distr}(S) \cup \{\mathbf{0}\}$ by Bel. A state estimator based on Bel is then defined as follows [51,54,57][3]:

[3] For the deterministic case, we omit the unique action for brevity.

Definition 5 (MC state estimator). *For* $\mathsf{MC} = \langle S, \iota, \mathsf{Act}, P, \mathsf{Z}, \mathsf{obs}\rangle$*, a trace* $\tau \in \mathsf{Z}^+$ *the state estimation function* $\mathsf{est}_{\mathsf{MC}} \colon \mathsf{Z}^+ \to \mathsf{Bel}$ *is defined as*

$$\mathsf{est}_{\mathsf{MC}}(z) := \begin{cases} \left\{ s \mapsto \frac{\iota(s)\cdot\mathsf{obs}(s)(z)}{\sum_{\hat{s}\in S}\iota(\hat{s})\cdot\mathsf{obs}(\hat{s})(z)} \right\} & \exists s \in S.\ \iota(s)\cdot\mathsf{obs}(z) > 0, \\ \mathbf{0} & \textit{otherwise.} \end{cases}$$

$$\mathsf{est}_{\mathsf{MC}}(\tau \cdot z) := \left\{ s' \mapsto \frac{\sum_{s\in S} \mathsf{est}_{\mathsf{MC}}(\tau)(s)\cdot P(s,s')\cdot\mathsf{obs}(s')(z)}{\sum_{s\in S}\mathsf{est}_{\mathsf{MC}}(\tau)(s)\cdot\big(\sum_{\hat{s}\in S}P(s,\hat{s})\cdot\mathsf{obs}(\hat{s})(z)\big)} \right\}$$

To illustrate how $\mathsf{est}_{\mathsf{MC}}$ is computed, consider again our system in Fig. 1 and assume that the MDP has only the actions labeled with $\{p\}$ (reducing it to the Markov chain induced by the a scheduler that only performs the $\{p\}$ actions). Again we consider the observation trace $\tau = R_o \cdot M_o \cdot L_o$ and compute $\mathsf{est}_{\mathsf{MC}}(\tau)$. For the first observation R_o, and since there is only one initial state, it follows that $\mathsf{est}_{\mathsf{MC}}(R_o) = \{\langle R, D_2\rangle \mapsto 1\}$[4]. From $\langle R, D_2\rangle$ and having observed M_o we can reach the states $\langle R, D_1\rangle$ and $\langle M, D_1\rangle$ with probabilities $\mathsf{est}_{\mathsf{MC}}(R_o \cdot M_o) = \{\langle R, D_1\rangle \mapsto \frac{\frac{1}{2}\cdot\frac{1}{3}}{\frac{1}{2}\cdot\frac{1}{3}+\frac{1}{2}\cdot\frac{3}{4}} = \frac{4}{13}, \langle M, D_1\rangle \mapsto \frac{\frac{1}{2}\cdot\frac{3}{4}}{\frac{1}{2}\cdot\frac{1}{3}+\frac{1}{2}\cdot\frac{3}{4}} = \frac{9}{13}\}$. Finally, from the later two states, when observing L_o, the states $\langle M, D_0\rangle$ and $\langle L, D_0\rangle$ can be reached with probabilities $\mathsf{est}_{\mathsf{MC}}(R_o \cdot M_o \cdot L_o) = \{\langle M, D_0\rangle \mapsto 0.0001, \langle L, D_0\rangle \mapsto 0.999\}$. Notice that although the state $\langle R, D_0\rangle$ can be reached from $\langle R, D_1\rangle$, the probability of being in this state is 0 since the probability of observing L_o in this state is $\mathsf{obs}(\langle R, D_0\rangle)(L_o) = 0$.

Lemma 3. *For a Markov chain* $\mathsf{MC} = \langle S, \iota, \mathsf{Act}, P, \mathsf{Z}, \mathsf{obs}\rangle$*, a trace* $\tau \in \mathsf{Z}^+$*, and a state-risk function* $r \colon S \to \mathbb{R}_{\geq 0}$*, it holds that* $R_r(\tau) = \sum_{s\in S}\mathsf{est}_{\mathsf{MC}}(\tau)(s)\cdot r(s)$*. Computing* $R_r(\tau)$ *can be done in time* $\mathcal{O}(|\tau|\cdot|S|\cdot|P|)$ *, and using* $|S|$ *many rational numbers. The size of the rationals*[5] *may grow linearly in* τ*.*

Proof Sketch. Since the system is deterministic, there is a unique scheduler σ, thus $R_r(\tau) = \sum_{\pi\in\Pi_{\mathsf{MC}}^{|\tau|}}\mathsf{Pr}_\sigma(\pi \mid \tau)\cdot r(\pi_\downarrow)$ by definition. We can show by induction over the length of τ that $\mathsf{Pr}_\sigma(\pi \mid \tau) = \mathsf{est}_{\mathsf{MC}}(\tau)(\pi_\downarrow)$ and conclude that $R_r(\tau) = \sum_{\pi\in\Pi_{\mathcal{M}}^{|\tau|}}\mathsf{est}_{\mathsf{MC}}(\tau)(\pi_\downarrow)\cdot r(\pi_\downarrow) = \sum_{s\in S}\mathsf{est}_{\mathsf{MC}}(\tau)(s)\cdot r(s)$ because $\mathsf{est}_{\mathsf{MC}}(\tau)(s) = 0$ for all $s \in S$ for which there is no path $\pi \in \Pi_{\mathcal{M}}^{|\tau|}$ with $\pi_\downarrow = s$. The complexity follows from the inductive definition of $\mathsf{est}_{\mathsf{MC}}$ that requires in each inductive step to iterate over all transitions of the system and maintain a belief over the states of the system. □

3.3 State Estimators for Markov Decision Processes

In an MDP, we have to account for every possible resolution of nondeterminism, which means that a belief can evolve into a set of beliefs:

[4] We omit the (single) sensor state for conciseness.

[5] To avoid growth, one may use fixed-precision numbers that over-approximate the probability of being in any state—inducing a growing (but conservative) error.

Definition 6 (MDP state estimator). *For an MDP* $\mathcal{M} = \langle S, \iota, \mathsf{Act}, P, \mathsf{Z}, \mathsf{obs} \rangle$*, a trace* $\tau \in \mathsf{Z}^+$*, and a state-risk function* $r \colon S \to \mathbb{R}_{\geq 0}$*, the state estimation function* $\mathsf{est}_{\mathsf{MDP}} \colon \mathsf{Z}^+ \to 2^{\mathsf{Bel}}$ *is defined as*

$$\begin{aligned} \mathsf{est}_{\mathsf{MDP}}(z) &= \{\mathsf{est}_{\mathsf{MC}}(z)\}, \\ \mathsf{est}_{\mathsf{MDP}}(\tau \cdot z) &= \left\{ \mathsf{bel}' \in \mathsf{Bel} \;\middle|\; \exists \mathsf{bel} \in \mathsf{est}_{\mathsf{MDP}}(\tau).\ \mathsf{bel}' \in \mathsf{est}^{\mathsf{up}}_{\mathsf{MDP}}(\mathsf{bel}, z) \right\}, \end{aligned}$$

and where $\mathsf{bel}' \in \mathsf{est}^{\mathsf{up}}_{\mathsf{MDP}}(\mathsf{bel}, z)$ *if there exists* $\varsigma_{\mathsf{bel}} \colon S \to \mathsf{Distr}(\mathsf{Act})$ *such that:*

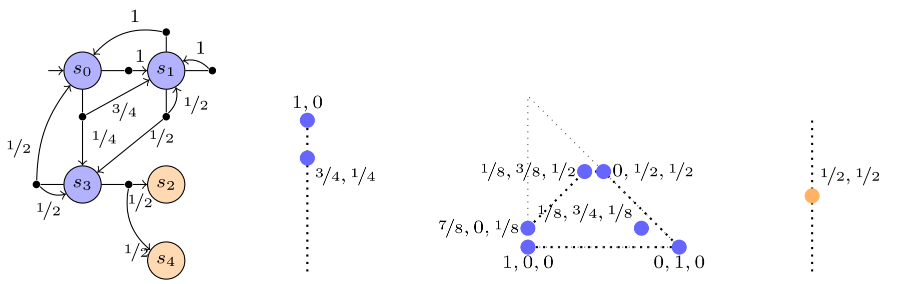
$$\forall s'.\mathsf{bel}'(s') = \frac{\sum\limits_{s \in S} \mathsf{bel}(s) \cdot \sum\limits_{\alpha \in \mathsf{Act}} \varsigma_{\mathsf{bel}}(s)(\alpha) \cdot P(s, \alpha, s') \cdot \mathsf{obs}(s')(z)}{\sum\limits_{s \in S} \mathsf{bel}(s) \cdot \sum\limits_{\alpha \in \mathsf{Act}} \varsigma_{\mathsf{bel}}(s)(\alpha) \cdot \sum\limits_{\hat{s} \in S} P(s, \alpha, \hat{s}) \cdot \mathsf{obs}(\hat{s})(z)}.$$

The definition conservatively extends both Definition 3 and Definition 5. Furthermore, we remark that we do not restrict how the nondeterminism is resolved: any distribution over actions can be chosen, and the distributions may be different for different traces.

Consider our system in Fig. 1. For the trace $\tau = R_o \cdot M_o \cdot L_o$, $\mathsf{est}_{\mathsf{MDP}}(\tau)$ is computed as follows. First, when observing R_o, the state estimator computes the initial belief set $\mathsf{est}_{\mathsf{MDP}}(R_o) = \{\{\langle R, D_2 \rangle \mapsto 1\}\}$. From this set of beliefs, when observing M_o, a set $\mathsf{est}_{\mathsf{MDP}}(R_o \cdot M_o)$ can be computed since all transitions $\emptyset, \{p\}, \{w\}, \{p, w\}$ (as well as their convex combinations) are possible from $\langle R, D_2 \rangle$. One of these beliefs is for example $\{\langle R, D_1 \rangle \mapsto \frac{4}{13}, \langle M, D_1 \rangle \mapsto \frac{9}{13}\}$ when a scheduler takes the transition $\{p\}$ (as was computed in our example for the Markov chain case). Having additionally observed L_o a new set $\mathsf{est}_{\mathsf{MDP}}(R_o M_o L_o)$ of beliefs can be computed based on the beliefs in $\mathsf{est}_{\mathsf{MDP}}(R_o M_o)$. For example from the belief $\{\langle R, D_1 \rangle \mapsto \frac{4}{13}, \langle M, D_1 \rangle \mapsto \frac{9}{13}\}$, two of the new beliefs are $\{\langle L, D_0 \rangle \mapsto 0.999, \langle M, D_0 \rangle \mapsto 0.0001\}$ and $\{\langle M, D_1 \rangle \mapsto 0.0287, \langle M, D_0 \rangle \mapsto 0.0001, \langle L, D_0 \rangle \mapsto 0.9712\}$. The first belief is reached by a scheduler that takes a transition $\{p\}$ at both $\langle R, D_1 \rangle$ and $\langle M, D_1 \rangle$. Notice that the belief does not give a positive probability to the state $\langle R, D_0 \rangle$ because L_o cannot be observed in this state. The second belief is reached by considering a scheduler that takes transition $\{p\}$ at $\langle M, D_1 \rangle$ and transition $\emptyset$ at $\langle R, D_1 \rangle$.

Theorem 1. *For an MDP* $\mathcal{M} = \langle S, \iota, \mathsf{Act}, P, \mathsf{Z}, \mathsf{obs} \rangle$*, a trace* $\tau \in \mathsf{Z}^+$*, and a state-risk function* $r \colon S \to \mathbb{R}_{\geq 0}$*, it holds that* $R_r(\tau) = \sup_{\mathsf{bel} \in \mathsf{est}_{\mathsf{MDP}}(\tau)} \sum_{s \in S} \mathsf{bel}(s) \cdot r(s)$.

Proof Sketch. For a given trace τ, each (history-dependent, randomizing) scheduler induces a belief over the states of the Markov chain induced by the scheduler. Also, each belief in $\mathsf{est}_{\mathsf{MDP}}(\tau)$ corresponds to a fixed scheduler, namely that one used to compute the belief recursively (i.e., an arbitrary randomizing memoryless scheduler for every time step). Once a scheduler σ and its corresponding belief bel is fixed, or vice versa, we can show using induction over the length of τ that $\sum_{\pi \in \Pi^{|\tau|}_{\mathcal{M}}} \mathsf{Pr}_\sigma(\pi \mid \tau) \cdot r(\pi_\downarrow) = \sum_{s \in S} \mathsf{bel}(s) \cdot r(s)$. $\square$



(a) MDP $\mathcal{M}$ (b) Over s_1, s_3 (c) Over s_0, s_1, s_3 (d) Over s_2, s_3

Fig. 3. Beliefs in $\mathbb{R}^n$ on $\mathcal{M}$ for $\tau = z_0 z_0$, $z_0 z_0 z_0$ and $z_0 z_0 z_1$, respectively.

4 Convex Hull-Based Forward Filtering

In this section, we show that we can use a finite representation for $\mathsf{est}_{\mathsf{MDP}}(\tau)$, but that this representation is exponentially large for some MDPs.

4.1 Properties of $\mathsf{est}_{\mathsf{MDP}}(\tau)$.

First, observe that $\mathbf{0}$ never maximizes the risk. Furthermore, $\mathbf{0}$ is closed under updates, i.e., $\mathsf{est}^{\mathsf{up}}_{\mathsf{MDP}}(\mathbf{0}, z) = \{\mathbf{0}\}$. We can thus w.l.o.g. assume that $\mathbf{0} \notin \mathsf{est}_{\mathsf{MDP}}(\tau)$. Second, observe that $\mathsf{est}_{\mathsf{MDP}}(\tau) \neq \emptyset$ if $\mathsf{Pr}_\sigma(\tau) > 0$.

We can interpret a belief $\mathsf{bel} \in \mathsf{Bel}$ as point in (a bounded subset of) $\mathbb{R}^{(|S|-1)}$. We are in particular interested in convex sets of beliefs. A set $B \subseteq \mathsf{Bel}$ is convex if the convex hull $\mathsf{CH}(B)$ of B, i.e. all convex combination of beliefs in B[6], coincides with B, i.e., $\mathsf{CH}(B) = B$. For a set $B \subseteq \mathsf{Bel}$, a belief $\mathsf{bel} \in B$ is an interior belief if it can be expressed as convex combination of the beliefs in $B \setminus \{\mathsf{bel}\}$. All other beliefs are (extremal) points or *vertices*. Let the set $\mathcal{V}(B) \subseteq B$ denote the set of *vertices of the convex hull* of B.

Example 1. Consider Fig. 3(a). All observation are Dirac, and only states s_2 and s_4 have observation z_1. The beliefs having observed $z_0 z_0$ are distributions over s_1, s_3, and can thus be depicted in a one-dimensional simplex. In particular, we have $\mathcal{V}(\mathsf{est}_{\mathsf{MDP}}(z_0 z_0)) = \{\{s_1 \mapsto 1\}, \{s_1 \mapsto 3/4, s_3 \mapsto 1/4\}\}$, as depicted in Fig. 3(b). The six beliefs having observed $z_0 z_0 z_0$ are distributions over s_0, s_1, s_3, depicted in Fig. 3(c). Five out of six beliefs are vertices. The belief having observed $z_0 z_0 z_1$ is in Fig. 3(d).

Remark 2. Observe that we illustrate the beliefs over only the states $\mathsf{est}_{\mathsf{KS}}(\tau)$. We therefore call $|\mathsf{est}_{\mathsf{KS}}(\tau)|$ the dimension of $\mathsf{est}_{\mathsf{MDP}}(\tau)$.

From the fundamental theorem of linear programming [47, Ch. 7] it immediately follows that the trace risk R_τ is obtained at a vertex of the beliefs of $\mathsf{est}_{\mathsf{MDP}}\tau$. We obtain the following refinement over Theorem 1:

[6] That is, $\mathsf{CH}(B) = \{\sum_{\mathsf{bel} \in B} w(\mathsf{bel}) \cdot \mathsf{bel} \mid \text{for all } w \in \mathbb{R}^B_{\geq 0} \text{ with } \sum w(\mathsf{bel}) = 1\}$.

Theorem 2. *For every τ and r: $R_r(\tau) = \max_{\mathsf{bel} \in \mathcal{V}(\mathsf{est}_{\mathsf{MDP}}(\tau))} \sum_{s \in S} \mathsf{bel}(s) \cdot r(s)$.*

Lemma 5 below clarifies that this maximum indeed exists.

We make some observations that allow us to compute the vertices more efficiently: Let $\mathsf{est}^{\mathsf{up}}_{\mathsf{MDP}}(B, z)$ denote $\bigcup_{\mathsf{bel} \in B} \mathsf{est}^{\mathsf{up}}_{\mathsf{MDP}}(\mathsf{bel}, z)$. From the properties of convex sets [18, Ch. 2], we make the following observations: If B is convex, $\mathsf{est}^{\mathsf{up}}_{\mathsf{MDP}}(B, z)$ is convex, as all operations in computing a new belief are convex-set preserving[7]. Furthermore, if B has a finite set of vertices, then $\mathsf{est}^{\mathsf{up}}_{\mathsf{MDP}}(B, z)$ has a finite set of vertices. The following lemma which is based on the observations above clarifies how to compute the vertices:

Lemma 4. *For a convex set of beliefs B with a finite set of vertices and an observation z:*

$$\mathcal{V}(\mathsf{est}^{\mathsf{up}}_{\mathsf{MDP}}(B, z)) = \mathcal{V}(\mathsf{est}^{\mathsf{up}}_{\mathsf{MDP}}(\mathcal{V}(B), z)).$$

By induction and using the facts above we obtain:

Lemma 5. *Any $\mathcal{V}(\mathsf{est}_{\mathsf{MDP}}(\tau))$ is finite.*

A monitor thus only needs to track the vertices. Furthermore, $\mathsf{est}^{\mathsf{up}}_{\mathsf{MDP}}(B, z)$ can be adapted to compute only vertices by limiting ς_{bel} to $S \to \mathsf{Act}$.

4.2 Exponential Lower Bounds on the Relevant Vertices

We show that a monitor in general cannot avoid an exponential blow-up in the beliefs it tracks. First observe that updating bel yields up to $\prod_s |\mathsf{Act}(s)|$ new beliefs (vertex or not), a prohibitively large number. The number of vertices is also exponential:

Lemma 6. *There exists a family of MDPs $\mathcal{M}_n$ with $2n+1$ states such that $|\mathcal{V}(\mathsf{est}_{\mathsf{MDP}}(\tau))| = 2^n$ for every τ with $|\tau| > 2$.*

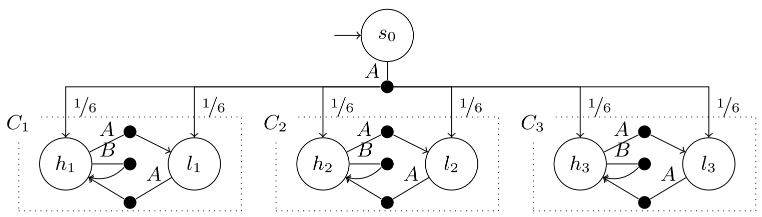
Proof Sketch. We construct $\mathcal{M}_n$ with $n = 3$, that is, $\mathcal{M}_3$ in Fig. 4(a). For this MDP and $\tau = AAA$, $|\mathcal{V}(\mathsf{est}_{\mathsf{MDP}}(\tau))| = 2^3$. In particular, observe how the belief factorizes into a belief within each component $C_i = \{h_i, l_i\}$ and notice that $\mathcal{M}_n$ has components C_1 to C_n. In particular, for each component, the belief being that we are with probability mass $1/n$ (for $n = 3$, $1/3$) in the 'low' state l_i or the 'high' state h_i. We depict the beliefs in Fig. 4(b,c,d). Thus, for any τ with $|\tau| > 2$ we can compactly represent $\mathcal{V}(\mathsf{est}_{\mathsf{MDP}}(\tau))$ as bit-strings of length n. Concretely, the belief

$$\{h_1, l_2, l_3 \mapsto 1/3, l_1, h_2, h_3 \mapsto 0\} \text{ maps to } 100, \text{ and}$$
$$\{h_1, l_2, h_3 \mapsto 1/3, l_1, h_2, l_3 \mapsto 0\} \text{ maps to } 101.$$

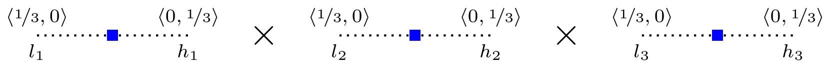
These are exponentially many beliefs for bit strings of length n. □

One might ask whether a symbolic encoding of an exponentially large set may result in a more tractable approach to filtering. While Theorem 2 allows

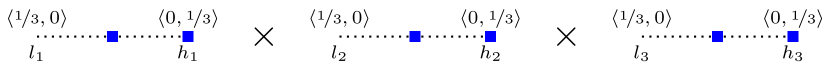
[7] The scaling is called a *projection*.



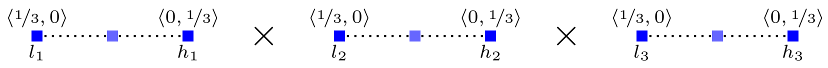
(a) $\mathcal{M}_3$. Observations $\mathsf{Z} = \mathsf{Act}$ with $\mathsf{obs}(s, \alpha) = \alpha$ if $\alpha \neq B$ and $\mathsf{obs}(s, B) = A$ for every s. Initial belief is A, all probabilities are 1, unless stated otherwise.



(b) Beliefs after AA



(c) Beliefs after AAA



(d) Beliefs after $AAAA^+$

Fig. 4. Construction for the correctness of Lemma 6.

to compute the associated risk from a set of linear constraints with standard techniques, it is not clear whether the concise set of constraints can be efficiently constructed and updated in every step. We leave this concern for future work.

In the remainder we investigate whether we need to track all these beliefs. First, when the monitor is unaware of the state-risk, this is trivially unavoidable. More precisely, all vertices may induce the maximal weighted trace risk by choosing an appropriate state-risk:

Lemma 7. *For every τ and every $\mathsf{bel} \in \mathcal{V}(\mathsf{est}_{\mathsf{MDP}}(\tau))$ there exists an r s.t.*

$$\sum_{s \in S} \mathsf{bel}(s) \cdot r(s) \geq \max_{\mathsf{bel}' \in \mathcal{V}(\mathsf{est}_{\mathsf{MDP}}(\tau)) \setminus \{\mathsf{bel}\}} \sum_{s \in S} \mathsf{bel}'(s) \cdot r(s) \ \textit{with} \ \max_{\mathsf{bel} \in \emptyset} = -\infty.$$

Proof Sketch. We construct r such that $r(s) > r(s')$ if $\mathsf{bel}(s) > \mathsf{bel}(s')$. $\square$

Second, even if the monitor is aware of the state risk r, it may not be able to prune enough vertices to avoid exponential growth. The crux here is that while some of the current beliefs may induce a smaller risk, an extension of the trace may cause the belief to evolve into a belief that induces the maximal risk.

Theorem 3. *There exist MDPs $\mathcal{M}_n$ a τ with $B := \mathcal{V}(\mathsf{est}_{\mathsf{MDP}}(\tau))$ and a state-risk r such that $|B| = 2^n$ and for all $\mathsf{bel} \in B$ exists $\tau' \in \mathsf{Z}^+$ with $R_r(\tau \cdot \tau') > \sup_{\mathsf{bel} \in B'} \sum_s \mathsf{bel}(s) \cdot r(s)$, where $B' = \mathsf{est}^{\mathsf{up}}_{\mathsf{MDP}}(B \setminus \{\mathsf{bel}\}, \tau')$.*

It is helpful to understand this theorem as describing the outcome of a game between monitor and environment: The statement says if the monitor decides to drop some vertices from $\mathsf{est}_{\mathsf{MDP}}\tau$, the environment may produce an observation trace τ' that will lead the monitor to underestimate the weighted risk at $R_r(\tau{\cdot}\tau')$.

Proof Sketch. We extend the construction of Fig. 4(a) with choices to go to a final state. The full proof sketch can be found in [36, Appendix].

4.3 Approximation by Pruning

Finally, we illustrate that we cannot simply prune small probabilities from beliefs. This indicates that an approximative version of filtering for the monitoring problem is nontrivial. Reconsider observing $z_0 z_0$ in the MDP of Fig. 3, and, for the sake of argument, let us prune the (small) entry $s_3 \mapsto 1/4$ to 0. Now, continuing with the trace $z_0 z_0 z_1$, we would update the beliefs from before and then conclude that this trace cannot be observed with positive probability. With pruning, there is no upper bound on the difference between the *computed* R_τ and the *actual* R_τ. Thus, forward filtering is, in general, not tractable on MDPs.

5 Unrolling with Model Checking

We present a tractable algorithm for the monitoring problem. Contrary to filtering, this method incorporates the state risk. We briefly consider the qualitative case. An algorithm that solves that problem iteratively guesses a successor such that the given trace has positive probability, and reaches a state with sufficient risk. The algorithm only stores the current and next state and a counter.

Theorem 4. *The Monitoring Problem with $\lambda = 0$ is in NLOGSPACE.*

This result implies the existence of a polynomial time algorithm, e.g., using a graph-search on a graph growing in $|\tau|$. There also is a deterministic algorithm with space complexity $\mathcal{O}(log^2(|\mathcal{M}|+|\tau|))$, which follows from applying Savitch's Theorem [46] , but that algorithm has exponential time complexity.

We now present a tractable algorithm for the quantitative case, where we need to store all paths. We do this efficiently by storing an unrolled MDP with these paths using ideas from [9,19]. In particular, on this MDP, we can efficiently obtain the scheduler that optimizes the risk by model checking rather than enumerating over all schedulers explicitly. We give the result before going into details.

Theorem 5. *The Monitoring Problem (with $\lambda > 0$) is P-complete.*

The problem is P-hard, as unary-encoded step-bounded reachability is P-hard [41]. It remains to show a P-time algorithm[8], which is outlined below. Roughly, the algorithm constructs an MDP $\mathcal{M}'''$ from $\mathcal{M}$ in three conceptual steps, such that the

[8] On first sight, this might be surprising as step-bounded reachability in MDPs is PSPACE-hard and only quasi-polynomial. However, our problem gets a trace and therefore (assuming that the trace is not compressed) can be handled in time polynomial in the length of the trace.

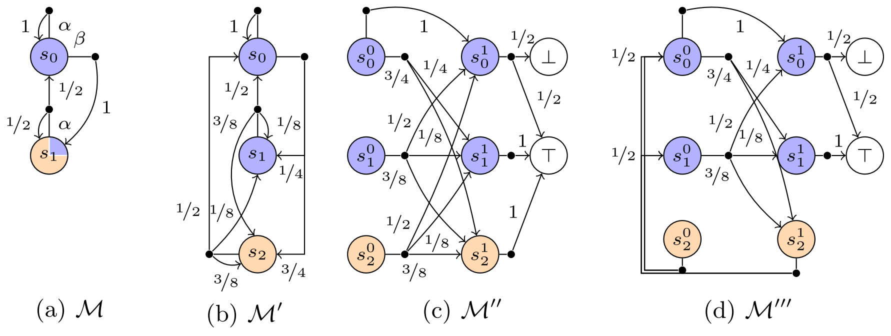


Fig. 5. Polynomial-time algorithm for solving Problem 1 illustrated.

maximal probability of reaching a state in $\mathcal{M}'''$ coincides with the $R_r(\tau)$. The former can be solved by linear programming in polynomial time. The downside is that even in the best case, the memory consumption grows linearly in $|\tau|$.

We outline the main steps of the algorithm and exemplify them below: First, we transform $\mathcal{M}$ into an MDP $\mathcal{M}'$ with *deterministic state* observations, i.e., with $\mathsf{obs}' : S \to \mathsf{Z}$. This construction is detailed in [19, Remark 1], and runs in polynomial time. The new initial distribution takes into account the initial observation and the initial distribution. Importantly, for each path π and each trace τ, $\mathsf{obs}_{\mathsf{tr}}(\pi)(\tau)$ is preserved. From here, the idea for the algorithm is a tailored adaption of the construction for conditional reachability probabilities in [9]. We ensure that $r(s) \in [0,1]$ by scaling r and λ accordingly. Now, we construct a new MDP $\mathcal{M}'' = \langle S'', \iota'', \mathsf{Act}'', P'' \rangle$ with state space $S'' := (S' \times \{0, \dots, |\tau|-1\}) \cup \{\bot, \top\}$ and an n-times unrolled transition relation. Furthermore, from the states $\langle s, |\tau|-1 \rangle$, there is a single outgoing action that with probability $r(s)$ leads to $\top$ and with probability $1 - r(s)$ leads to $\bot$. Observe that the risk is now the supremum of conditioned reachability probabilities over paths that reach $\top$, conditioned by the trace τ. The MDP $\mathcal{M}''$ is only polynomially larger. Then, we construct MDP $\mathcal{M}'''$ by copying $\mathcal{M}''$ and replacing (part of) the transition relation P'' by P''' such that paths π with $\tau \notin \mathsf{obs}_{\mathsf{tr}}(\pi)$ are looped back to the initial state (resembling rejection sampling). Formally,

$$P'''(\langle s, i \rangle, \alpha) = \begin{cases} P''(\langle s, i \rangle, \alpha) & \text{if } \mathsf{obs}'(s) = \tau_i, \\ \iota & \text{otherwise.} \end{cases}$$

The maximal conditional reachability probability in $\mathcal{M}''$ is the maximal reachability probability in M''' [9]. Maximal reachability probabilities can be computed by solving a linear program [43], and can thus be computed in polynomial time.

Example 2. We illustrate the construction in Fig. 5. In Fig. 5(a), we depict an MDP $\mathcal{M}$, with $\iota = \{s_0, s_1 \mapsto 1/2\}$. Furthermore, let $\tau = z_0 z_0$ and let $r(s_0) = 1$ and $r(s_1) = 2$. Let $\mathsf{obs}(s_0) = \{z_0 \mapsto 1\}$ and $\mathsf{obs}(s_1) = \{z_0 \mapsto 1/4, z_1 \mapsto 3/4\}$. State s_1 has two possible observations, so we split s_1 into s_1 and s_2 in MDP

$\mathcal{M}'$, each with their own observations. Any transition into s_1 is now split. As $|\tau| = 2$, we unroll the MDP $\mathcal{M}'$ into MDP $\mathcal{M}''$ to represent two steps, and add goal and sink states. After rescaling, we obtain that $r(s_0) = 1/2$, whereas $r(s_1) = r(s_2) = 2/2 = 1$, and we add the appropriate outgoing transitions to the states s^1_*. In a final step, we create MDP $\mathcal{M}'''$ from $\mathcal{M}''$: we reroute all probability mass that does not agree with the observations to the initial states. Now, $R_r(z_0 z_0)$ is given by the probability to reach, in $\mathcal{M}'''$, in an unbounded number of steps, $\top$.

The construction also implies that maximizing over a finite set of schedulers, namely the deterministic schedulers with a counter from 0 to $|\tau|$, suffices. We denote this class $\Sigma_{\mathrm{DC}}(|\tau|)$. Formally, a scheduler is in $\Sigma_{\mathrm{DC}}(k)$ if for all π, π':

$$\Big(\pi_\downarrow = \pi'_\downarrow \wedge \big(|\pi| = |\pi'| \vee (|\pi| > k \wedge |\pi'| > k)\big)\Big) \text{ implies } \sigma(\pi) = \sigma(\pi').$$

Lemma 8. *For every τ, it holds that*

$$R_r(\tau) \quad = \quad \max_{\sigma \in \Sigma_{DC}(|\tau|)} \sum_{\pi \in \Pi_M^{|\tau|}} \mathsf{Pr}_\sigma(\pi \mid \tau) \cdot r(\pi_\downarrow).$$

The crucial idea underpinning this lemma is that memoryless schedulers suffice for the unrolling, and that the states of the unrolling can be uniquely mapped to a state and the length of the history for every π through $\mathcal{M}$. By reducing step-bounded reachability we can also show that this set of schedulers is necessary [4].

6 Empirical Evaluation

Implementation. We provide prototype implementations for both filtering- and model-checking-based approaches from Sect. 3, built on top of the probabilistic model checker STORM [30]. We provide a schematic setup of our implementation in Fig. 6. As input, we consider a symbolic description of MDPs with state-based observation labels, based on an extended dialect of the Prism language. We define the state risk in this MDP via a temporal property (given as a PCTL formula), and obtain the concrete state-risk by model checking. We take a seed that yields a trace using the simulator. For the experiments, actions are resolved uniformly in this simulator[9]. The simulator iteratively feeds observations into the monitor, running either of our two algorithms (implemented in C++). After each observation z_i, the monitor computes the risk R_i having observed $z_0 \ldots z_i$. We flexibly combine these components via a Python API[10].

For filtering as in Sect. 4, we provide a sparse data structure for beliefs that is updated using only deterministic schedulers. This is sufficient, see Lemma 4. To further prune the set of beliefs, we implement an SMT-driven elimination [48]

[9] This is not an assumption but rather our evaluation strategy.
[10] Available at https://github.com/monitoring-MDPs/premise.

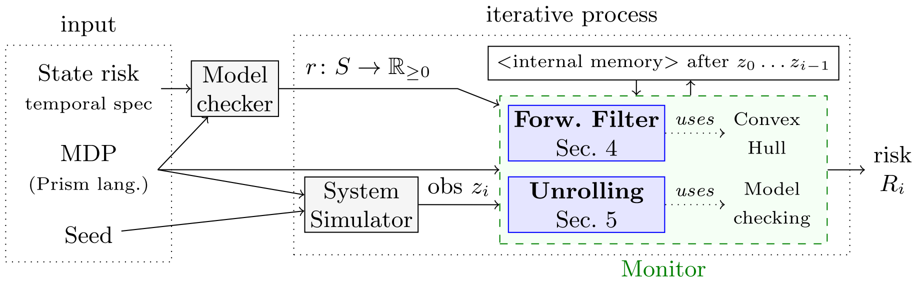


Fig. 6. Schematic setup for prototype mapping stream $z_0 \dots z_k$ to stream $R_0 \dots R_k$.

of interior beliefs, inside of the convex hull[11]. We construct the unrolling as described in Sect. 5 and apply model checking via any sparse engines in Storm.

Reproducibility. We archived a container with sources, benchmarks, and scripts to reproduce our experiments: https://doi.org/10.5281/zenodo.4724622.

Set-Up. For each benchmark described below, we sampled 50 random traces using seeds 0–49 of lengths up to $|\tau| = 500$. We are interested in the *promptness*, that is, the delay of time between getting an observation z_i and returning corresponding risk r_i, as well as the *cumulative performance* obtained by summing over the promptness along the trace. We use a timeout of 1 second for this query. We compare the forward filtering (FF) approach with and without convex hull (CH) reduction, and the model unrolling approach (UNR) with two model checking engines of Storm: exact policy iteration (EPI, [43]) and optimistic value iteration (OVI, [28]). All experiments are run on a MacBook Pro MV962LL/A, using a single core. The memory limit of 6GB was not violated. We use Z3 [38] as SMT-solver [11] for the convex hull reduction.

Benchmarks. We present three benchmark families, all MDPs with a combination of probabilities, nondeterminism and partial observability.

Airport-A is as in Sect. 1, but with a higher resolution for both ground vehicle in the middle lane and the plane. Airport-B has a two-state sensor model with stochastic transitions between them.

Refuel-A models robots with a depleting battery and recharging stations. The world model consists of a robot moving around in a $D\times D$ grid with some dedicated charging cells, where each action costs energy. The risk is to deplete the battery within a fixed horizon. Refuel-B is a two-state sensor variant.

Evade-I is inspired by a navigation task in a multi-agent setting in a $D\times D$ grid. The monitored robot moves randomly, and the risk is defined as the probability of crashing with the other robot. The other robot has an internal incentive in the form of a cardinal direction, and nondeterministically decides to move or

[11] Advanced algorithms like Quickhull [10] are not without significant adaptions applicable as the set of beliefs can be degenerate (roughly, a set without full rank).

Table 1. Performance for promptness of online monitoring on various benchmarks.

						CH Forward Filtering							Unrolling				
Id	Name	Inst	$\|S\|$	$\|P\|$	$\|\tau\|$	N	T_{avg}	T_{max}	B_{avg}	B_{max}	D_{avg}	D_{max}	N	T_{avg}	T_{max}	$\|S_u\|_{\text{avg}}$	$\|S_u\|_{\text{max}}$
1	AIRPORT-A	7,50,30	20910	114143	100	**50**	0.01	0.01	4.5	7	4.6	7	**50**	0.04	0.11	524	599
					500	**50**	0.01	0.01	1.0	1	1.0	1	**50**	0.01	0.01	1075	1258
2	AIRPORT-B	3,50,30	20232	106012	100	**0**							**50**	0.09	0.16	556	629
					500	**0**							**50**	0.01	0.01	1460	1647
3	AIRPORT-B	7,50,30	41820	308474	100	**0**							**50**	0.14	0.33	1000	1183
					500	**0**							**11**	0.02	0.02	2097	2297
4	REFUEL-A	12,50	45073	2431691	100	**50**	0.01	0.01	2.2	4	2.8	5	**50**	0.01	0.05	325	409
					500	**50**	0.01	0.01	1.5	4	1.7	5	**50**	0.01	0.19	1071	2409
5	REFUEL-B	12,50	90145	9725277	100	**50**	0.06	0.23	4.2	8	5.6	10	**50**	0.04	0.17	608	732
					500	**50**	0.01	0.01	2.9	8	3.3	10	**46**	0.04	0.09	2171	4688
6	EVADE-I	15	377101	2022295	100	**50**	0.01	0.02	2.6	10	3.3	4	**49**	0.01	0.06	332	363
					500	**50**	0.01	0.01	2.4	5	3.4	4	**45**	0.08	0.90	1655	1891
7	EVADE-V	5,3	1001	5318	100	**26**	0.01	0.01	1.0	1	1.0	1	**50**	0.00	0.02	134	241
					500	**25**	0.01	0.01	1.0	1	1.0	1	**50**	0.00	0.01	538	671
8	EVADE-V	6,3	2161	11817	100	**1**	0.01	0.01	1.0	1	1.0	1	**50**	0.02	0.32	319	861
					500	**1**	0.01	0.01	1.0	1	1.0	1	**49**	0.01	0.02	777	1484

to uniformly randomly change its incentive. The monitor observes everything except the incentive of the other robot. EVADE-V is an alternative navigation task: Contrary to above, the other robot does not have an internal state and indeed navigates nondeterministically in one of the cardinal directions. We only observe the other robot location is within the view range.

Results. We split our results in two tables. In Table 1, we give an ID for every benchmark name and instance, along with the size of the MDP (nr. of states $|S|$ and transitions $|P|$) our algorithms operate on. We consider the promptness after prefixes of length $|\tau|$. In particular, for forward filtering with the convex hull optimization, we give the number N of traces that did not time out before, and consider the average T_{avg} and maximal time T_{max} needed (over all sampled traces that did not time-out before). Furthermore, we give the average, B_{avg}, and maximal, B_{max}, number of beliefs stored (after reduction), and the average, D_{avg}, and maximal, D_{max}, dimension of the belief support. Likewise, for unrolling with exact model checking, we give the number N of traces that did not time out before, and we consider average T_{avg} and maximal time T_{max}, as well as the average size and maximal number of states of the unfolded MDP.

In Table 2, we consider for the benchmarks above the cumulative performance. In particular, this table also considers an alternative implementation for both FF and UNR. We use the IDs to identify the instance, and sum for each prefix of length $|\tau|$ the time. For filtering, we recall the number of traces N that did not time out, the average and maximal cumulative time along the trace, the average cumulative number of beliefs that were considered, and the average cumulative number of beliefs eliminated. For the case without convex hull, we do not eliminate any vertices. For unrolling, we report average T_{avg} and maximal cumulative time using EPI, as well as the time required for model building, *Bld*$^{\%}$ (relative to the total time, per trace). We compare this to the average

Table 2. Summarized performance for online monitoring

		FF w/o CH				FF w/ CH					UNR (EPI)					UNR (OVI)		
Id	$\lvert\tau\rvert$	N	T_{avg}	T_{max}	B_{avg}	N	T_{avg}	T_{max}	B_{avg}	E_{avg}	N	T_{avg}	T_{max}	$Bld^{\%}_{\text{avg}}$	$Bld^{\%}_{\text{max}}$	N	T_{avg}	T_{max}
1	100	**0**				**50**	0.9	1.1	493	241	**50**	2.9	3.6	6	56	**50**	0.0	0.1
	500	**0**				**50**	3.7	4.3	1040	316	**50**	7.5	10.7	21	24	**50**	0.4	0.8
2	100	**0**				**0**					**50**	3.7	4.7	6	54	**50**	0.1	0.1
	500	**0**				**0**					**50**	11.9	17.1	18	23	**50**	0.6	0.8
3	100	**0**				**0**					**50**	7.6	10.6	5	55	**50**	0.1	0.2
	500	**0**				**0**					**11**	21.3	28.7	19	23	**50**	0.9	1.7
4	100	**1**	0.9	0.9	1473	**50**	0.7	0.8	241	138	**50**	0.7	1.0	35	69	**50**	0.0	0.1
	500	**1**	0.9	0.9	1873	**50**	3.4	3.7	868	226	**50**	5.6	21.2	57	67	**50**	0.5	0.9
5	100	**0**				**50**	7.4	10.7	442	2267	**50**	2.5	4.4	32	57	**50**	0.1	0.2
	500	**0**				**50**	16.5	42.2	1781	4249	**46**	19.5	64.2	55	70	**50**	1.3	2.3
6	100	**13**	0.7	2.9	2055	**50**	1.1	4.8	273	160	**49**	0.5	2.0	34	65	**47**	0.0	0.1
	500	**2**	4.4	6.8	20524	**50**	5.1	11.5	1237	632	**45**	22.4	53.6	13	29	**43**	0.5	0.7
7	100	**13**	0.1	0.5	274	**26**	0.8	1.2	106	11	**50**	0.4	1.0	19	45	**48**	0.0	0.1
	500	**13**	0.1	0.5	674	**25**	3.7	4.2	505	7	**50**	1.3	4.4	46	58	**47**	0.2	0.3
8	100	**0**				**1**	1.3	1.3	124	109	**50**	1.5	7.0	15	39	**36**	0.4	5.6
	500	**0**				**1**	4.3	4.3	524	109	**49**	4.9	28.1	37	56	**35**	0.7	6.4

and maximal cumulative time for using OVI (notice that building times remain approximately the same).

Discussion. The results from our prototype show that conservative (sound) predictive modeling of systems that combine probabilities, nondeterminism and partial observability is within reach with the methods we proposed and state-of-the-art algorithms. Both forward filtering and an unrolling-based approaches have their merits. The practical results thus slightly diverge from the complexity results in Sect. 3.1, due to structural properties of some benchmarks. In particular, for AIRPORT-A and REFUEL-A, the nondeterminism barely influences the belief, and so there is no explosion, and consequentially the dimension of the belief is sufficiently small that the convex hull can be efficiently computed. Rather than the number of states, this belief dimension makes EVADE-V a difficult benchmark[12]. *If many states can be reached with a particular trace, and if along these paths there are some probabilistic states, forward filtering suffers significantly.* We see that if the benchmark allows for efficacious forward filtering, it is not slowed down in the way that unrolling is slower on longer traces. For UNR, we observe that OVI is typically the fastest, but EPI does not suffer from the numerical worst-cases as OVI does. *If an observation trace is unlikely, the unrolled MDP constitutes a numerically challenging problem, in particular for value-iteration based model checkers*, see [27]. For FF, the convex hull computation is essential for any dimension, and eliminating some vertices in every step keeps the number of belief states manageable.

[12] The max dimension =1 in EVADE-V is only over the traces that did not time-out. The dimension when running in time-outs is above 5.

7 Related Work

We are not the first to consider model-based runtime verification in the presence of partial observability and probabilities. Runtime verification with state estimation on hidden Markov models (HMM)—without nondeterminism has been studied for various types of properties [51,54,57] and has been extended to hybrid systems [52]. The tool Prevent focusses on black-box systems by learning an HMM from a set of traces. The HMM approximates (with only convergence-in-the-limit guarantees) the actual system [6], and then estimates during runtime the most likely trace rather than estimating a distribution over current states. Extensions consider symmetry reductions on the models [7]. These techniques do not make a conservative (sound) risk estimation. The recent framework for runtime verification in the presence of partial observability [23] takes a more strict black-box view and cannot provide state estimates. Finally, [26] chooses to have partial observability to make monitoring of software systems more efficient, and [58] monitors a noisy sensor to reduce energy consumption.

State beliefs are studied when verifying HMMs [59], where the question whether a sequence of observations likely occurs, or which HMM is an adequate representation of a system [37]. State beliefs are prominent in the verification of partially observable MDPs [16,32,40], where one can observe the actions taken (but the problem itself is to find the right scheduler). Our monitoring problem can be phrased as a special case of verification of partially observable stochastic games [20], but automatic techniques for those very general models are lacking. Likewise, the idea of *shielding* (pre)computes all action choices that lead to safe behavior [3,5,15,24,34,35]. For partially observable settings, shielding again requires to compute partial-information schedulers [21,39], contrary to our approach. Partial observability has also been studied in the context of diagnosability, studying if a fault has occurred (in the past) [14], or what actions uncover faults [13]. We, instead assume partial observability in which we do detect faults, but want to estimate the risk that these faults occur in the future.

The assurance framework for reinforcement learning [42] implicitly allows for stochastic behavior, but cannot cope with partial observability or nondeterminism. Predictive monitoring has been combined with deep learning [17] and Bayesian inference [22], where the key problem is that the computation of an imminent failure is too expensive to be done exactly. More generally, learning automata models has been motivated with runtime assurance [1,55]. Testing approaches statistically evaluate whether traces are likely to be produced by a given model [25]. The approach in [2] studies stochastic black-box systems with controllable nondeterminism and iteratively learns a model for the system.

8 Conclusion

We have presented the first framework for monitoring based on a trace of observations on models that combine nondeterminism and probabilities. Future work includes heuristics for approximate monitoring and for faster convex hull computations, and to apply this work to gray-box (learned) models.

Acknowledgments. The example in Fig. 1 is inspired by a challenge problem provided by Boeing in the DARPA Assured Autonomy program.

References

1. Aichernig, B.K., et al.: Learning a behavior model of hybrid systems through combining model-based testing and machine learning. In: Gaston, C., Kosmatov, N., Le Gall, P. (eds.) ICTSS 2019. LNCS, vol. 11812, pp. 3–21. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-31280-0_1
2. Aichernig, B.K., Tappler, M.: Probabilistic black-box reachability checking (extended version). Formal Methods Syst. Des. **54**(3), 416–448 (2019)
3. Alshiekh, M., Bloem, R., Ehlers, R., Könighofer, B., Niekum, S., Topcu, U.: Safe reinforcement learning via shielding. In: AAAI, pp. 2669–2678. AAAI Press (2018)
4. Andova, S., Hermanns, H., Katoen, J.-P.: Discrete-time rewards model-checked. In: Larsen, K.G., Niebert, P. (eds.) FORMATS 2003. LNCS, vol. 2791, pp. 88–104. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-40903-8_8
5. Avni, G., Bloem, R., Chatterjee, K., Henzinger, T.A., Könighofer, B., Pranger, S.: Run-time optimization for learned controllers through quantitative games. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 630–649. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_36
6. Babaee, R., Gurfinkel, A., Fischmeister, S.: $\mathcal{P}revent$: a predictive run-time verification framework using statistical learning. In: Johnsen, E.B., Schaefer, I. (eds.) SEFM 2018. LNCS, vol. 10886, pp. 205–220. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-92970-5_13
7. Babaee, R., Gurfinkel, A., Fischmeister, S.: Predictive run-time verification of discrete-time reachability properties in black-box systems using trace-level abstraction and statistical learning. In: Colombo, C., Leucker, M. (eds.) RV 2018. LNCS, vol. 11237, pp. 187–204. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03769-7_11
8. Baier, C., Katoen, J.: Principles of Model Checking. MIT Press, Cambridge (2008)
9. Baier, C., Klein, J., Klüppelholz, S., Märcker, S.: Computing conditional probabilities in Markovian models efficiently. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 515–530. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_43
10. Barber, C.B., Dobkin, D.P., Huhdanpaa, H.: The Quickhull algorithm for convex hulls. ACM Trans. Math. Softw. **22**(4), 469–483 (1996)
11. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 825–885. IOS Press (2009)
12. Bartocci, E., et al.: Specification-based monitoring of cyber-physical systems: a survey on theory, tools and applications. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification. LNCS, vol. 10457, pp. 135–175. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5_5
13. Bertrand, N., Fabre, É., Haar, S., Haddad, S., Hélouët, L.: Active diagnosis for probabilistic systems. In: Muscholl, A. (ed.) FoSSaCS 2014. LNCS, vol. 8412, pp. 29–42. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54830-7_2
14. Bertrand, N., Haddad, S., Lefaucheux, E.: A tale of two diagnoses in probabilistic systems. Inf. Comput. **269** (2019)

15. Bloem, R., Könighofer, B., Könighofer, R., Wang, C.: Shield synthesis: runtime enforcement for reactive systems. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 533–548. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_51
16. Bork, A., Junges, S., Katoen, J.-P., Quatmann, T.: Verification of indefinite-horizon POMDPs. In: Hung, D.V., Sokolsky, O. (eds.) ATVA 2020. LNCS, vol. 12302, pp. 288–304. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-59152-6_16
17. Bortolussi, L., Cairoli, F., Paoletti, N., Smolka, S.A., Stoller, S.D.: Neural predictive monitoring. In: Finkbeiner, B., Mariani, L. (eds.) RV 2019. LNCS, vol. 11757, pp. 129–147. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32079-9_8
18. Boyd, S.P., Vandenberghe, L.: Convex Optimization. Cambridge University Press, Cambridge (2014)
19. Chatterjee, K., Chmelik, M., Gupta, R., Kanodia, A.: Optimal cost almost-sure reachability in POMDPs. Artif. Intell. **234**, 26–48 (2016)
20. Chatterjee, K., Doyen, L.: Partial-observation stochastic games: how to win when belief fails. ACM Trans. Comput. Log. **15**(2), 16:1–16:44 (2014)
21. Chatterjee, K., Novotný, P., Pérez, G.A., Raskin, J., Zikelic, D.: Optimizing expectation with guarantees in POMDPs. In: AAAI, pp. 3725–3732. AAAI Press (2017)
22. Chou, Y., Yoon, H., Sankaranarayanan, S.: Predictive runtime monitoring of vehicle models using Bayesian estimation and reachability analysis. In: IROS (2020, to appear)
23. Cimatti, A., Tian, C., Tonetta, S.: Assumption-based runtime verification with partial observability and resets. In: Finkbeiner, B., Mariani, L. (eds.) RV 2019. LNCS, vol. 11757, pp. 165–184. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32079-9_10
24. Dräger, K., Forejt, V., Kwiatkowska, M.Z., Parker, D., Ujma, M.: Permissive controller synthesis for probabilistic systems. Log. Methods Comput. Sci. **11**(2) (2015)
25. Gerhold, M., Stoelinga, M.: Model-based testing of probabilistic systems. Formal Asp. Comput. **30**(1), 77–106 (2018)
26. Grigore, R., Kiefer, S.: Selective monitoring. In: CONCUR. LIPIcs, vol. 118, pp. 20:1–20:16. Dagstuhl - LZI (2018)
27. Haddad, S., Monmege, B.: Interval iteration algorithm for MDPs and IMDPs. Theor. Comput. Sci. **735**, 111–131 (2018)
28. Hartmanns, A., Kaminski, B.L.: Optimistic value iteration. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12225, pp. 488–511. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53291-8_26
29. Havelund, K., Roşu, G.: Runtime verification - 17 years later. In: Colombo, C., Leucker, M. (eds.) RV 2018. LNCS, vol. 11237, pp. 3–17. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03769-7_1
30. Hensel, C., Junges, S., Katoen, J., Quatmann, T., Volk, M.: The probabilistic model checker storm. CoRR abs/2002.07080 (2020)
31. Henzinger, T.A., Ho, P.-H., Wong-Toi, H.: Algorithmic analysis of nonlinear hybrid systems. IEEE Trans. Autom. Control **43**(4), 540–554 (1998)
32. Horák, K., Bosanský, B., Chatterjee, K.: Goal-HSVI: heuristic search value iteration for goal POMDPs. In: IJCAI, pp. 4764–4770. ijcai.org (2018)
33. Jansen, N., Humphrey, L., Tumova, J., Topcu, U.: Structured synthesis for probabilistic systems. In: Badger, J.M., Rozier, K.Y. (eds.) NFM 2019. LNCS, vol. 11460, pp. 237–254. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-20652-9_16

34. Jansen, N., Könighofer, B., Junges, S., Serban, A., Bloem, R.: Safe reinforcement learning using probabilistic shields (invited paper). In: CONCUR. LIPIcs, vol. 171, pp. 3:1–3:16. Dagstuhl - LZI (2020)
35. Junges, S., Jansen, N., Dehnert, C., Topcu, U., Katoen, J.-P.: Safety-constrained reinforcement learning for MDPs. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 130–146. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_8
36. Junges, S., Torfah, H., Seshia, S.A.: Runtime monitoring for Markov decision processes. CoRR abs/2105.12322 (2021)
37. Kiefer, S., Sistla, A.P.: Distinguishing hidden Markov chains. In: LICS, pp. 66–75. ACM (2016)
38. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
39. Nam, W., Alur, R.: Active learning of plans for safety and reachability goals with partial observability. IEEE Trans. Syst. Man Cybern. Part B **40**(2), 412–420 (2010)
40. Norman, G., Parker, D., Zou, X.: Verification and control of partially observable probabilistic systems. Real Time Syst. **53**(3), 354–402 (2017)
41. Papadimitriou, C.H., Tsitsiklis, J.N.: The complexity of Markov decision processes. Math. Oper. Res. **12**(3), 441–450 (1987)
42. Phan, D.T., Grosu, R., Jansen, N., Paoletti, N., Smolka, S.A., Stoller, S.D.: Neural simplex architecture. In: Lee, R., Jha, S., Mavridou, A., Giannakopoulou, D. (eds.) NFM 2020. LNCS, vol. 12229, pp. 97–114. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-55754-6_6
43. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley Series in Probability and Statistics. Wiley (1994)
44. Rabiner, L.R.: A tutorial on hidden Markov models and selected applications in speech recognition. Proc. IEEE **77**(2), 257–286 (1989)
45. Sánchez, C., et al.: A survey of challenges for runtime verification from advanced application domains (beyond software). Formal Methods Syst. Des. **54**(3), 279–335 (2019)
46. Savitch, W.J.: Relationships between nondeterministic and deterministic tape complexities. J. Comput. Syst. Sci. **4**(2), 177–192 (1970)
47. Schrijver, A.: Theory of Linear and Integer Programming. Wiley-Interscience Series in Discrete Mathematics and Optimization. Wiley (1999)
48. Seidel, R.: Convex hull computations. In: Handbook of Discrete and Computational Geometry, 2nd edn, pp. 495–512. Chapman and Hall/CRC (2004)
49. Seshia, S.A.: Introspective environment modeling. In: Finkbeiner, B., Mariani, L. (eds.) RV 2019. LNCS, vol. 11757, pp. 15–26. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32079-9_2
50. Seshia, S.A., Sadigh, D., Sastry, S.S.: Towards verified artificial intelligence. arXiv e-prints, July 2016
51. Sistla, A.P., Srinivas, A.R.: Monitoring temporal properties of stochastic systems. In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) VMCAI 2008. LNCS, vol. 4905, pp. 294–308. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78163-9_25
52. Sistla, A.P., Žefran, M., Feng, Y.: Runtime monitoring of stochastic cyber-physical systems with hybrid state. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 276–293. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29860-8_21

53. Spaan, M.T.J.: Partially observable Markov decision processes. In: Wiering, M., van Otterlo, M. (eds.) Reinforcement Learning, Adaptation, Learning, and Optimization, vol. 12, pp. 387–414. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-27645-3_12
54. Stoller, S.D., et al.: Runtime verification with state estimation. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 193–207. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29860-8_15
55. Tappler, M., Aichernig, B.K., Bacci, G., Eichlseder, M., Larsen, K.G.: L^*-based learning of Markov decision processes. In: ter Beek, M.H., McIver, A., Oliveira, J.N. (eds.) FM 2019. LNCS, vol. 11800, pp. 651–669. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30942-8_38
56. Thrun, S., Burgard, W., Fox, D.: Probabilistic Robotics. Intelligent Robotics and Autonomous Agents. MIT Press (2005)
57. Wilcox, C.M., Williams, B.C.: Runtime verification of stochastic, faulty systems. In: Barringer, H., et al. (eds.) RV 2010. LNCS, vol. 6418, pp. 452–459. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16612-9_34
58. Woo, H., Mok, A.K.: Real-time monitoring of uncertain data streams using probabilistic similarity. In: RTSS, pp. 288–300. IEEE CS (2007)
59. Zhang, L., Hermanns, H., Jansen, D.N.: Logic and model checking for hidden Markov models. In: Wang, F. (ed.) FORTE 2005. LNCS, vol. 3731, pp. 98–112. Springer, Heidelberg (2005). https://doi.org/10.1007/11562436_9

Model Checking Finite-Horizon Markov Chains with Probabilistic Inference

Steven Holtzen[1(✉)], Sebastian Junges[2], Marcell Vazquez-Chanlatte[2], Todd Millstein[1], Sanjit A. Seshia[2], and Guy Van den Broeck[1]

[1] University of California, Los Angeles, CA, USA
sholtzen@cs.ucla.edu
[2] University of California, Berkeley, CA, USA

Abstract. We revisit the symbolic verification of Markov chains with respect to finite horizon reachability properties. The prevalent approach iteratively computes step-bounded state reachability probabilities. By contrast, recent advances in probabilistic inference suggest symbolically representing all horizon-length paths through the Markov chain. We ask whether this perspective advances the state-of-the-art in probabilistic model checking. First, we formally describe both approaches in order to highlight their key differences. Then, using these insights we develop Rubicon, a tool that transpiles Prism models to the probabilistic inference tool `Dice`. Finally, we demonstrate better scalability compared to probabilistic model checkers on selected benchmarks. All together, our results suggest that probabilistic inference is a valuable addition to the probabilistic model checking portfolio, with Rubicon as a first step towards integrating both perspectives.

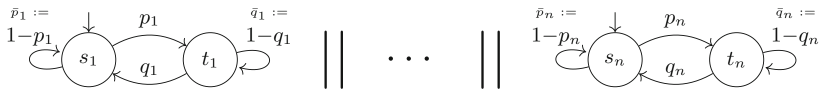
1 Introduction

Systems with probabilistic uncertainty are ubiquitous, e.g., probabilistic programs, distributed systems, fault trees, and biological models. Markov chains replace nondeterminism in transition systems with probabilistic uncertainty, and *probabilistic model checking* [4,7] provides model checking algorithms. A key property that probabilistic model checkers answer is: *What is the (precise) probability that a target state is reached (within a finite number of steps h)*? Contrary to classical *qualitative* model checking and approximate variants of probabilistic model checking, precise probabilistic model checking must find the total probability of *all* paths from the initial state to any target state.

S. Holtzen and S. Junges—Contributed equally.

This work is partially supported by NSF grants #IIS-1943641, #IIS-1956441, #CCF-1837129, DARPA grant #N66001-17-2-4032, a Sloan Fellowship, a UCLA Dissertation Year Fellowship, and gifts by Intel and Facebook Research.
This work is partially supported by NSF grants 1545126 (VeHICaL), 1646208 and 1837132, by the DARPA contracts FA8750-18-C-0101 (AA) and FA8750-20-C-0156 (SDCPS), by Berkeley Deep Drive, and by Toyota under the iCyPhy center.

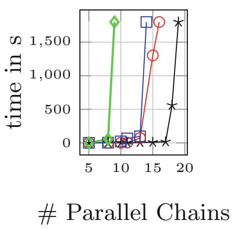
A. Silva and K. R. M. Leino (Eds.): CAV 2021, LNCS 12760, pp. 577–601, 2021.
https://doi.org/10.1007/978-3-030-81688-9_27



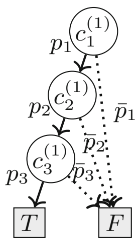
(a) Motivating factory Markov chain with $s_i = [\![c_i = 0]\!], t_i = [\![c_i = 1]\!]$.

```
const double p1, p2, p3, q1, q2, q3;
module F1
 c1 : bool init false;
 [a] !c1 -> p1: (c1'=1) +1-p1: (c1'=0);
 [a]  c1 -> q1: (c1'=0) +1-q1: (c1'=1);
endmodule
module F2 = F1[c1=c2,p1=p2,q1=q2]
module F3 = F1[c1=c3,p1=p3,q1=q3]
label "allStrike" = c1 & c2 & c3;
```

(b) A PRISM model of (a) with 3 factories.



(c) Relative scaling.



(d) BDD

Fig. 1. Motivating example. Figure 1(c) compares the performance of RUBICON (—⋆—), STORM's explicit engine (—○—), STORM's symbolic engine (—□—) and PRISM (—◇—) when invoked on a (b) with arbitrarily fixed (different) constants for p_i, q_i and horizon $h = 10$. Times are in seconds, with a time-out of 30 min.

Nevertheless, the prevalent ideas in probabilistic model checking are generalizations of qualitative model checking. Whereas qualitative model checking tracks the states that can reach a target state (or dually, that can be reached from an initial state), probabilistic model checking tracks the i-step reachability probability for each state in the chain. The i+1-step reachability can then be computed via multiplication with the *transition matrix*. The scalability concern is that this matrix grows with the state space in the Markov chain. Mature model checking tools such as STORM [36], Modest [34], and PRISM [51] utilize a variety of methods to alleviate the state space explosion. Nevertheless various natural models cannot be analyzed by the available techniques.

In parallel, within the AI community a different approach to representing a distribution has emerged, which on first glance can seem unintuitive. Rather than marginalizing out the paths and tracking reachability probabilities per state, the probabilistic AI community commonly aggregates all *paths* that reach the target state. At its core, inference is then a weighted sum over all these paths [16]. This hinges on the observation that this set of paths can often be stored more compactly, and that the probability of two paths that share the same prefix or suffix can be efficiently computed on this concise representation. This *inference technique* has been used in a variety of domains in the artificial intelligence (AI) and verification communities [9,14,27,39], but is not part of any mature probabilistic model checking tools.

This paper theoretically and experimentally compares and contrasts these two approaches. In particular, we describe and motivate RUBICON, a probabilistic model checker that *leverages the successful probabilistic inference techniques*. We begin with an example that explains the core ideas of Rubicon followed by the paper structure and key contributions.

Motivating Example. Consider the example illustrated in Fig. 1(a). Suppose there are n factories. Each day, the workers at each factory collectively decide whether or not to strike. To simplify, we model each factory (i) with two states, striking (t_i) and not striking (s_i). Furthermore, since no two factories are identical, we take the probability to begin striking (p_i) and to stop striking (q_i) to be different for each factory. Assuming that each factory transitions synchronously and in parallel with the others, we query: "what is the probability that all the factories are simultaneously striking within h days?"

Despite its simplicity, we observe that state-of-the-art model checkers like Storm and Prism do not scale beyond 15 factories.[1] For example, Fig. 1(b) provides a Prism encoding for this simple model (we show the instance with 3 factories), where a Boolean variable c_i is used to encode the state of each factory. The "`allStrike`" label identifies the target state. Figure 1(c) shows the run time for an increasing number of factories. While all methods eventually time out, Rubicon scales to systems with an order of magnitude more states.

Why is This Problem Hard? To understand the issue with scalability, observe that tools such as Storm and Prism store the transition matrix, either explicitly or symbolically using algebraic decision diagrams (ADDs). Every distinct entry of this transition matrix needs to be represented; in the case of ADDs using a unique leaf node. Because each factory in our example has a different probability of going on strike, that means each subset of factories will likely have a unique probability of jointly going on strike. Hence, the transition matrix then will have a number of distinct probabilities that is exponential in the number of factories, and its representation as an ADD must blow up in size. Concretely, for 10 factories, the size of the ADD representing the transition matrix has 1.9 million nodes. Moreover, the explicit engine fails due to the dense nature of the underlying transition matrix. We discuss this method in Sect. 3.

How to Overcome This Limitation? This problematic combinatorial explosion is often unnecessary. For the sake of intuition, consider the simple case where the horizon is 1. Still, the standard transition matrix representations blow up exponentially with the number of factories n. Yet, the probability of reaching the "`allStrike`" state is easy to compute, even when n grows: it is $p_1 \cdot p_2 \cdots p_n$.

Rubicon aims to compute probabilities in this compact *factorized* way by representing the computation as a binary decision diagram (BDD). Figure 1(d) gives an example of such a BDD, for three factories and a horizon of one. A key property of this BDD, elaborated in Sect. 3, is that it can be interpreted as a *parametric Markov chain*, where the weight of each edge corresponds with the probability of a particular factory striking. Then, the probability that the goal state is reached is given by the weighted sum of paths terminating in T: for this instance, there is a single such path with weight $p_1 \cdot p_2 \cdot p_3$. These BDDs are tree-like Markov-chains, so model checking can be performed in time linear in the size

[1] Section 6 describes the experimental apparatus and our choice of comparisons.

of the BDD using dynamic programming. Essentially, the BDD represents the set of paths that reach a target state—an idea common in probabilistic inference.

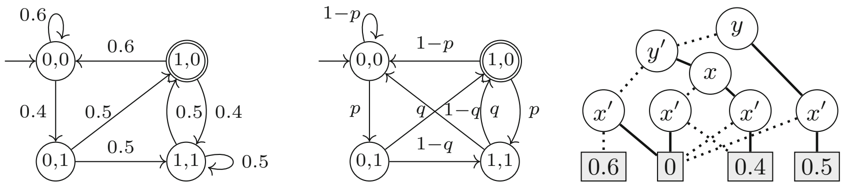
To construct this BDD, we propose to encode our reachability query symbolically as a *weighted model counting* (WMC) query on a logical formula. By compiling that formula into a BDD, we obtain a diagram where computing the query probability can be done efficiently (in the size of the BDD). Concretely for Fig. 1(d), the BDD represents the formula $c_1^{(1)} \wedge c_2^{(1)} \wedge c_3^{(1)}$, which encodes all paths through the chain that terminate in the goal state (all factories strike on day 1). For this example and this horizon, this is a single path. WMC is a well-known strategy for probabilistic inference and is currently the among the state-of-the-art approaches for discrete graphical models [16], discrete probabilistic programs [39], and probabilistic logic programs [27].

In general, the exponential growth of the number of paths might seem like it dooms this approach: for $n = 3$ factories and horizon $h = 1$, we need to only represent 8 paths, but for $h = 2$, we would need to consider 64 different paths, and so on. However, a key insight is that, for many systems – such as the factory example – the structural compression of BDDs allows a concise representation of exponentially many paths, all *while* being parametric over path probabilities (see Sect. 4). To see why, observe that in the above discussion, the state of each factory is *independent* of the other factories: independence, and its natural generalizations like *conditional* and *contextual* independence, are the driving force behind many successful probabilistic inference algorithms [47]. Succinctly, the key advantage of Rubicon is that it exploits a form of structure that has thus far been under-exploited by model checkers, which is why it scales to more parallel factories than the existing approaches on the hard task. In Sect. 6 we consider an extension to this motivating example that adds dependencies between factories. This dependency (or rather, the accompanying increase in the size of the underlying MC) significantly decreases scalability for the existing approaches but negligibly affects Rubicon.

This leads to the task: *how does one go from a* Prism *model to a concise BDD efficiently*? To do this, Rubicon leverages a novel translation from Prism models into a probabilistic programming language called `Dice` (outlined in Sect. 5).

Contribution and Structure. Inspired by the example, we contribute conceptual and empirical arguments for leveraging BDD-based probabilistic inference in model checking. Concretely:

1. We demonstrate fundamental advantages in using probabilistic inference on a natural class of models (Sect. 1 and 6).
2. We explain these advantages by showing the fundamental differences between existing model checking approaches and probabilistic inference (Sect. 3 and 4). To that end, Sect. 4 presents probabilistic inference based on an operational and a logical perspective and combines these perspectives.
3. We leverage those insights to build Rubicon, a tool that transpiles Prism to `Dice`, a probabilistic programming language (Sect. 5).



(a) Toy-example $\mathcal{M}$ (b) pMC $\mathcal{M}'$ (c) For $\mathcal{M}$: P as ADD

Fig. 2. (a) MC toy example (b) (distinct) pMC toy example (c) ADD transition matrix

4. We demonstrate that Rubicon indeed attains an order-of-magnitude scaling improvement on several natural problems including sampling from parametric Markov chains and verifying network protocol stabilization (Sect. 6).

Ultimately we argue that Rubicon makes a valuable contribution to the portfolio of probabilistic model checking backends, and brings to bear the extensive developments on probabilistic inference to well-known model checking problems.

2 Preliminaries and Problem Statement

We state the problem formally and recap relevant concepts. See [7] for details. We sometimes use $\bar{p}$ to denote $1-p$. A *Markov chain* (MC) is a tuple $\mathcal{M} = \langle S, \iota, P, T \rangle$ with S a (finite) set of *states*, $\iota \in S$ the *initial state*, $P\colon S \to \mathit{Distr}(S)$ the *transition function*, and T a set of *target states* $T \subseteq S$, where $\mathit{Distr}(S)$ is the set of distributions over a (finite) set S. We write $P(s, s')$ to denote $P(s)(s')$ and call P a *transition matrix*. The successors of s are $\mathsf{Succ}(s) = \{s' \mid P(s, s') > 0\}$. To support MCs with billions of states, we may describe MCs symbolically, e.g., with Prism [51] or as a probabilistic program [42,48]. For such a symbolic description $\mathcal{P}$, we denote the corresponding MC with $[\![\, \mathcal{P} \,]\!]$. States then reflect assignments to symbolic variables.

A *path* $\pi = s_0 \dots s_n$ is a sequence of states, $\pi \in S^+$. We use $\pi_\downarrow$ to denote the *last state* s_n, and the *length* of π above is n and is denoted $|\pi|$. Let Paths_h denote the paths of length h. The probability of a path is the product of the transition probabilities, and may be defined inductively by $\Pr(s) = 1$, $\Pr(\pi \cdot s) = \Pr(\pi) \cdot P(\pi_\downarrow, s)$. For a fixed *horizon* h and set of states T, let the set $[\![\, s {\rightarrow} \Diamond^{\leq h} T \,]\!] = \{\pi \mid \pi_0 = s \wedge |\pi| \leq h \wedge \pi_\downarrow \in T \wedge \forall i < |\pi|.\ \pi_i \notin T\}$ denote paths from s of length at most h that terminate at a state contained in T. Furthermore, let $\Pr_{\mathcal{M}}(s \models \Diamond^{\leq h} T) = \sum_{\pi \in [\![\, s \rightarrow \Diamond^{\leq h} T \,]\!]} \Pr(\pi)$ describe the probability to reach T within h steps. We simplify notation when $s = \iota$ and write $[\![\, \Diamond^{\leq h} T \,]\!]$ and $\Pr_{\mathcal{M}}(\Diamond^{\leq h} T)$, respectively. We omit $\mathcal{M}$ whenever that is clear from the context.

Formal Problem: Given an MC $\mathcal{M}$ and a horizon h, compute $\Pr_{\mathcal{M}}(\lozenge^{\leq h} T)$.

Example 1. For conciseness, we introduce a toy example MC $\mathcal{M}$ in Fig. 2(a). For horizon $h = 3$, there are three paths that reach state $\langle 1{,}0 \rangle$: For example the path $\langle 0, 0\rangle\langle 0, 1\rangle\langle 1, 0\rangle$ with corresponding reachability probability $0.4 \cdot 0.5$. The reachability probability $\Pr_{\mathcal{M}}(\lozenge^{\leq 3}\{\langle 1, 0\rangle\}) = 0.42$.

It is helpful to separate the topology and the probabilities. We do this by means of a *parametric MC* (pMC) [22]. A pMC over a fixed set of parameters $\boldsymbol{p}$ generalises MCs by allowing for a transition function that maps to $\mathbb{Q}[\boldsymbol{p}]$, i.e., to polynomials over these variables [22]. A pMC and a *valuation* of parameters $\boldsymbol{u}\colon \boldsymbol{p} \to \mathbb{R}$ describe a MC by replacing $\boldsymbol{p}$ with $\boldsymbol{u}$ in the transition function P to obtain $P[\boldsymbol{u}]$. If $P[\boldsymbol{u}](s)$ is a distribution for every s, then we call $\boldsymbol{u}$ a *well-defined* valuation. We can then think about a pMC $\mathcal{M}$ as a generator of a set of MCs $\{\mathcal{M}[\boldsymbol{u}] \mid \boldsymbol{u} \text{ well-defined}\}$. Figure 2(b) shows a pMC; any valuation $\boldsymbol{u}$ with $\boldsymbol{u}(p), \boldsymbol{u}(q) \in [0, 1]$ is well-defined. We consider the following associated problem:

Parameter Sampling: Given a pMC $\mathcal{M}$, a finite set of well-defined valuations U, and a horizon h, compute $\Pr_{\mathcal{M}[\boldsymbol{u}]}(\lozenge^{\leq h} T)$ for each $\boldsymbol{u} \in U$.

We recap binary *decision diagrams* (BDDs) and their generalization into algebraic decision diagrams (ADDs, a.k.a. multi-terminal BDDs). ADDs over a set of variables X are directed acyclic graphs whose vertices V can be partitioned into *terminal nodes* V_t without successors and *inner nodes* V_i with two successors. Each terminal node is labeled with a polynomial over some parameters $\boldsymbol{p}$ (or just to constants in $\mathbb{Q}$), $\mathsf{val}\colon V_t \to \mathbb{Q}[\boldsymbol{p}]$, and each inner node V_i with a variable, $\mathsf{var}\colon V_i \to X$. One node is the root node v_0. Edges are described by the two successor functions $E_0\colon V_i \to V$ and $E_1\colon V_i \to V$. A BDD is an ADD with exactly two terminals labeled T and F. Formally, we denote an ADD by the tuple $\langle V, v_0, X, \mathsf{var}, \mathsf{val}, E_0, E_1\rangle$. ADDs describe functions $f\colon \mathbb{B}^X \to \mathbb{Q}[\boldsymbol{p}]$ (described by a path in the underlying graph and the label of the corresponding terminal node). As finite sets can be encoded with bit vectors, ADDs represent functions from (tuples of) finite sets to polynomials.

Example 2. The transition matrix P of the MC in Fig. 2(a) maps states, encoded by bit vectors, $\langle x, y\rangle, \langle x', y'\rangle$ to the probabilities to move from state $\langle x, y\rangle$ to $\langle x', y'\rangle$. Figure 2(c) shows the corresponding ADD.[2]

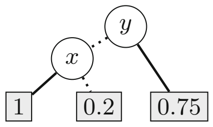
3 A Model Checking Perspective

We briefly analyze the de-facto standard approach to symbolic probabilistic model checking of finite-horizon reachability probabilities. It is an adaptation of qualitative model checking, in which we track the (backward) reachable states. This set can be thought of as a mapping from states to a Boolean indicating whether a target state can be reached. We generalize the mapping to a function that maps every state s to the probability that we reach T within i steps,

[2] The ADD also depends on the variable order, which we assume fixed for conciseness.

state	horizon h			
	0	1	2	3
⟨0,0⟩	0	0	0.2	0.42
⟨0,1⟩	0	0.5	0.75	0.875
⟨1,0⟩	1	1	1	1
⟨1,1⟩	0	0.5	0.75	0.875

(a) $\Pr_{\mathcal{M}}(\lozenge^{\leq h}\{\langle 0,1\rangle\})$



(b) $\Pr_{\mathcal{M}}(\lozenge^{\leq 2}\{\langle 0,1\rangle\})$ as ADD

Fig. 3. Bounded reachability and symbolic model checking for the MC $\mathcal{M}$ in Fig. 2(a).

denoted $\Pr_{\mathcal{M}}(s \models \lozenge^{\leq i} T)$. First, it is convenient to construct a transition relation in which the target states have been made absorbing, i.e., we define a matrix with $A(s,s') = P(s,s')$ if $s \notin T$ and $A(s,s') = [s = s']$[3] otherwise. The following *Bellman equations* characterize that aforementioned mapping:

$$\Pr_{\mathcal{M}}\left(s \models \lozenge^{\leq 0} T\right) = [s \in T],$$
$$\Pr_{\mathcal{M}}\left(s \models \lozenge^{\leq i} T\right) = \sum_{s' \in \mathsf{Succ}(s)} A(s,s') \cdot \Pr_{\mathcal{M}}(s' \models \lozenge^{\leq i-1} T) \qquad \text{with } i > 0.$$

The main aspect model checkers take from these equations is that to compute the h-step reachability from state s, one only needs to combine the $h-1$-step reachability from any state s' *and* the transition probabilities $P(s,s')$. We define a vector $\boldsymbol{T}$ with $\boldsymbol{T}(s) = [s \in T]$. The algorithm then iteratively computes and stores the i step reachability for $i = 0$ to $i = h$, e.g. by computing $A^3 \cdot \boldsymbol{T}$ using $A \cdot (A \cdot (A \cdot \boldsymbol{T}))$. This reasoning is thus *inherently backwards* and *implicitly marginalizing out paths*. In particular, rather than storing the i-step paths that lead to the target, one only stores a vector $\boldsymbol{x} = A^i \cdot \boldsymbol{T}$ that stores for every state s the sum over all i-long paths from s.

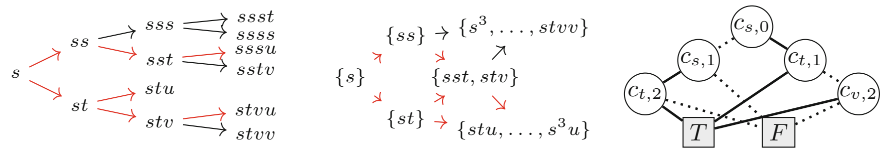
Explicit representations of matrix A and vector $\boldsymbol{x}$ require memory at least in the order $|S|$.[4] To overcome this limitation, *symbolic* probabilistic model checking stores both A and $A^i \cdot \boldsymbol{T}$ as an ADD by considering the matrix as a function from a tuple $\langle s,s' \rangle$ to $A(s,s')$, and $\boldsymbol{x}$ as a function from s to $\boldsymbol{x}(s)$ [2].

Example 3. Reconsider the MC in Fig. 2(a). The h-bounded reachability probability $\Pr_{\mathcal{M}}(\lozenge^{\leq h}\{\langle 1,0\rangle\})$ can be computed as reflected in Fig. 3(a). The ADD for P is shown in Fig. 2(c). The ADD for $\boldsymbol{x}$ when $h = 2$ is shown in Fig. 3(b).

The performance of symbolic probabilistic model checking is directly governed by the sizes of these two ADDs. The size of an ADD is bounded from below by the number of leafs. In qualitative model checking, both ADDs are in fact BDDs, with two leafs. However, for the ADD representing A, this lower bound is given by the number of different probabilities in the transition matrix. In the running example, we have seen that a small program $\mathcal{P}$ may have an underlying MC $[\![\mathcal{P}]\!]$ with an exponential state space S and equally many different transition probabilities. Symbolic probabilistic model checking also scales

[3] Where $[x]$=1 if x holds and 0 otherwise.

[4] Excluding e.g., partial exploration or sampling which typically are not exact.



(a) $\mathsf{CT}(\mathcal{M}, 3)$ (b) $\mathsf{CT}(\mathcal{M}, 3)$ compressed (c) Predicate as BDD

Fig. 4. The computation tree for $\mathcal{M}$ and horizon 3 and its compression. We label states as s=⟨0,0⟩, t=⟨0,1⟩, u=⟨1,0⟩, v=⟨1,1⟩. Probabilities are omitted for conciseness.

badly on some models where A has a concise encoding but $\boldsymbol{x}$ has too many different entries.[5] Therefore, model checkers may store $\boldsymbol{x}$ partially explicit [49].

The insights above are not new. Symbolic probabilistic model checking has advanced [46] to create small representations of both A and $\boldsymbol{x}$. In competitions, Storm often applies a bisimulation-to-explicit method that extracts an explicit representation of the bisimulation quotient [26, 36]. Finally, game-based abstraction [32, 44] can be seen as a predicate abstraction technique on the ADD level. However, these methods do not change the computation of the finite horizon reachability probabilities and thus do not overcome the inherent weaknesses of the iterative approach in combination with an ADD-based representation.

4 A Probabilistic Inference Perspective

We present four key insights into probabilistic inference. **(1)** Sect. 4.1 shows how probabilistic inference takes the classical definition as summing over the set of paths, and turns this definition into an algorithm. In particular, these paths may be stored in a computation tree. **(2)** Sect. 4.2 gives the traditional reduction from probabilistic inference to the classical weighted model counting (WMC) problem [16, 57]. **(3)** Sect. 4.3 connects this reduction to point (1) by showing that a BDD that represents this WMC is *bisimilar* to the computation tree assuming that the out-degree of every state in the MC is two. **(4)** Sect. 4.4 describes and compares the computational benefits of the BDD representation. In particular, we clarify that enforcing an out-degree of two is a key ingredient to overcoming one of the weaknesses of symbolic probabilistic model checking: the number of different probabilities in the underlying MC.

4.1 Operational Perspective

The following perspective frames (an aspect of) probabilistic inference as a model transformation. By definition, the set of all paths – each annotated with the transition probabilities – suffices to extract the reachability probability. These sets of paths may be represented in the computation tree (which is itself an MC).

[5] For an interesting example of this, see the "Queue" example in Sect. 6.

Example 4. We continue from Example 1. We put all paths of length three in a computation tree in Fig. 4(a) (cf. the caption for state identifiers). The three paths that reach the target are highlighted in red. The MC is highly redundant. We may compress to the MC in Fig. 4(b).

Definition 1. *For MC $\mathcal{M}$ and horizon h, the computation tree (CT) $\mathsf{CT}(\mathcal{M}, h) = \langle Paths_h, \iota, P', T' \rangle$ is an MC with states corresponding to paths in $\mathcal{M}$, i.e., $Paths_h^{\mathcal{M}}$, initial state ι, target states $T' = [\![\lozenge^{\leq h} T]\!]$, and transition relation*

$$P'(\pi, \pi') = \begin{cases} P(\pi_{\downarrow}, s) & \text{if } \pi_{\downarrow} \notin T \wedge \pi' = \pi.s, \\ [\pi_{\downarrow} \in T \wedge \pi' = \pi] & \text{otherwise.} \end{cases} \tag{1}$$

The CT contains (up to renaming) the same paths to the target as the original MC. Notice that after h transitions, all paths are in a sink state, and thus we can drop the step bound from the property and consider either finite or indefinite horizons. The latter considers all paths that eventually reach the target. We denote the probability mass of these paths with $\Pr_{\mathcal{M}}(s \models \lozenge T)$ and refer to [7] for formal details.[6] Then, we may compute bounded reachability probabilities in the original MC by analysing unbounded reachability in the CT:

$$\Pr_{\mathcal{M}}(\lozenge^{\leq h} T) = \Pr_{\mathsf{CT}(\mathcal{M},h)}(\lozenge^{\leq h} T') = \Pr_{\mathsf{CT}(\mathcal{M},h)}(\lozenge T').$$

The nodes in the CT have a natural topological ordering. The unbounded reachability probability is then computed (efficiently in CT's size) using dynamic programming (i.e., topological value iteration) on the Bellman equation for $s \notin T$:

$$\Pr_{\mathcal{M}}(s \models \lozenge T) = \textstyle\sum_{s' \in \mathsf{Succ}(s)} P(s, s') \cdot \Pr_{\mathcal{M}}(s' \models \lozenge T).$$

For pMCs, the right-hand side naturally is a factorised form of the *solution function* f that maps parameter values to the induced reachability probability, i.e. $f(\boldsymbol{u}) = \Pr_{\mathcal{M}[\boldsymbol{u}]}(\lozenge^{\leq h} T)$ [22,24,33]. For bounded reachability (or acyclic pMCs), this function amounts to a sum over all paths with every path reflected by a term of a polynomial, i.e., the sum is a polynomial. In sum-of-terms representation, the polynomial can be exponential in the number of parameters [5].

For computational efficiency, we need a smaller representation of the CT. As we only consider reachability of T, we may simplify [43] the notion of (weak) bisimulation [6] (in the formulation of [40]) to the following definition.

Definition 2. *For $\mathcal{M}$ with states S, a relation $\mathcal{R} \subseteq S \times S$ is a (weak) bisimulation (with respect to T) if $s \mathcal{R} s'$ implies $\Pr_{\mathcal{M}}(s \models \lozenge T) = \Pr_{\mathcal{M}}(s' \models \lozenge T)$. Two states s, s' are (weakly) bisimilar (with respect to T) if $\Pr_{\mathcal{M}}(s \models \lozenge T) = \Pr_{\mathcal{M}}(s' \models \lozenge T)$*

Two MCs $\mathcal{M}, \mathcal{M}'$ are bisimilar, denoted $\mathcal{M} \sim \mathcal{M}'$ if the initial states are bisimilar in the disjoint union of the MCs. It holds by definition that if $\mathcal{M} \sim \mathcal{M}'$, then $\Pr_{\mathcal{M}}(\lozenge T) = \Pr_{\mathcal{M}'}(\lozenge T')$. The notion of bisimulation can be lifted to pMCs [33].

[6] Alternatively, on acyclic models, a large step bound $h > |S|$ suffices.

Idea 1: Given a symbolic description $\mathcal{P}$ of a MC $[\![\mathcal{P}]\!]$, efficiently construct a concise MC $\mathcal{M}$ that is bisimilar to $\mathsf{CT}([\![\mathcal{P}]\!], h)$.

Indeed, the (compressed) CT in Fig. 4(b) and Fig. 4(a) are bisimilar. We remark that we do not necessarily compute the bisimulation quotient of $\mathsf{CT}([\![\mathcal{P}]\!], h)$.

4.2 Logical Perspective

The previous section defined weakly bisimilar chains and showed computational advantages, but did not present an algorithm. In this section we frame the finite horizon reachability probability as a logical query known as *weighted model counting* (WMC). In the next section we will show how this logical perspective yields an algorithm for constructing bisimilar MCs.

Weighted model counting is well-known as an effective reduction for probabilistic inference [16,57]. Let φ be a logical sentence over variables C. The *weight function* $W_C \colon C \to \mathbb{R}_{\geq 0}$ assigns a weight to each logical variable. A *total variable assignment* $\eta \colon C \to \{0, 1\}$ by definition has weight $\mathsf{weight}(\eta) = \prod_{c \in C} W_C(c)\eta(c) + (1 - W_C(c)) \cdot (1 - \eta(c))$. Then the *weighted model count* for φ given W is $\mathsf{WMC}(\varphi, W_C) = \sum_{\eta \models \varphi} \mathsf{weight}(\eta)$. Formally, we desire to compute a reachability query using a WMC query in the following sense:

Idea 2: Given an MC $\mathcal{M}$, efficiently construct a predicate $\varphi^C_{\mathcal{M},h}$ and a weight-function W_C such that $\Pr_{\mathcal{M}}(\lozenge^{\leq h} T) = \mathsf{WMC}(\varphi^C_{\mathcal{M},h}, W_C)$.

Consider initially the simplified case when the MC $\mathcal{M}$ is *binary*: every state has at most two successors. In this case producing $(\varphi^C_{\mathcal{M},h}, W_C)$ is straightforward:

Example 5. Consider the MC in Fig. 2(a), and note that it is binary. We introduce logical variables called *state/step coins* $C = \{c_{s,i} \mid s \in S, i < h\}$ for every state and step. Assignments to these coins denote choices of transitions at particular times: if the chain is in state s at step i, then it takes the transition to the lexicographically first successor of s if $c_{s,i}$ is true and otherwise takes the transition to the lexicographically second successor. To construct the predicate $\varphi^C_{\mathcal{M},3}$, we will need to write a logical sentence on coins whose models encode accepting paths (red paths) in the CT in Fig. 4(a).

We start in state $s = \langle 0, 0\rangle$ (using state labels from the caption of Fig. 4). We order states as $s = \langle 0, 0\rangle < t = \langle 0, 1\rangle < u = \langle 1, 0\rangle < v = \langle 1, 1\rangle$. Then, $c_{s,0}$ is true if the chain transitions into state s at time 0 and false if it transitions to state t at time 0. So, one path from s to the target node $\langle 1, 0\rangle$ is given by the logical sentence $(c_{s,0} \wedge \neg c_{s,1} \wedge c_{t,2})$. The full predicate $\varphi^C_{\mathcal{M},3}$ is therefore:

$$\varphi^C_{\mathcal{M},3} = (c_{s,0} \wedge \neg c_{s,1} \wedge c_{t,2}) \vee (\neg c_{s,0} \wedge c_{t,1}) \vee (\neg c_{s,0} \wedge \neg c_{t,1} \wedge c_{v,2}).$$

Each model of this sentence is a single path to the target. This predicate $\varphi^C_{\mathcal{M},h}$ can clearly be constructed by considering all possible paths through the chain, but later on we will show how to build it more efficiently.

Finally, we fix W_C: The weight for each coin is directly given by the transition probability to the lexicographically first successor: for $0 \leq i < h$, $W_C(c_{s,i}) = 0.6$ and $W_C(c_{t,i}) = W_C(c_{v,i}) = 0.5$. The WMC is indeed 0.42, reflecting Example 1.

When the MC is not binary, it suffices to limit the out-degree of an MC to be at most two by adding auxiliary states, hence binarizing all transitions, cf. [38].

4.3 Connecting the Operational and the Logical Perspective

Now that we have reduced bounded reachability to weighted model counting, we reach a natural question: how do we perform WMC?[7] Various approaches to performing WMC have been explored; a prominent approach is to compile the logical function into a binary decision diagram (BDD), which supports fast weighted model counting [21]. In this paper, we investigate the use of a BDD-driven approach for two reasons: (i) BDDs admit straightforward support for parametric models. (ii) BDDs provide a direct connection between the logical and operational perspectives. To start, observe that the graph of the BDD, together with the weights, can be interpreted as an MC:

Definition 3. *Let φ^X be a propositional formula over variables X and $<_X$ an ordering on X. Let $\mathsf{BDD}(\varphi^X, <_X) = \langle V, v_0, X, \mathsf{var}, \mathsf{val}, E_0, E_1\rangle$ be the corresponding BDD, and let W be a weight function on X with $0 \leq W(x) \leq 1$. We define the MC $\mathsf{BDD}_{\mathsf{MC}}(\varphi^X, <_X, W) = \langle S, \iota, P, T\rangle$ with $S = V$, $\iota = v_0$, $P(s) = \{E_0(s) \mapsto W(\mathsf{var}(s)), E_1(s) \mapsto 1 - W(\mathsf{var}(s))\}$ and $T = \{v \in V \mid \mathsf{val}(v) = 1\}$.*

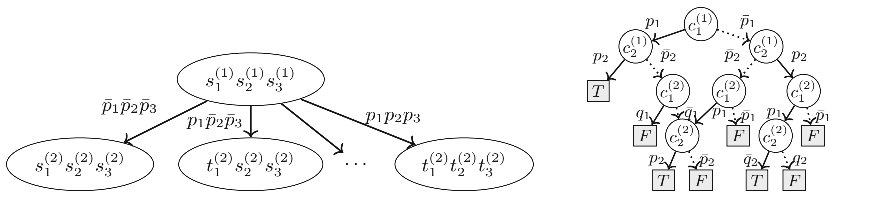
These BDDs are intimately related to the computation trees discussed before. For a binary MC $\mathcal{M}$, the tree $\mathsf{CT}(\mathcal{M}, h)$ is binary and can be considered as a (not necessarily reduced) BDD. More formally, let us construct $\mathsf{BDD}_{\mathsf{MC}}(\varphi^C_{\mathcal{M},h}, <_C,)$. We fix a total order on states. Then we fix *state/step coins* $C = \{c_{s,i} \mid s \in S, i < h\}$ and the weights as in Example 5. Finally, let $<_C$ be an order on C such that $i < j$ implies $c_{s,i} <_C c_{s,j}$. Then:

$$\mathsf{CT}(\mathcal{M}, h) \sim \mathsf{BDD}_{\mathsf{MC}}(\varphi^C_{\mathcal{M},h}, <_C, W). \tag{2}$$

In the spirit of Idea 1, we thus aim to construct $\mathsf{BDD}_{\mathsf{MC}}(\varphi^C_{\mathcal{M},h}, <_C, W)$, a representation as outlined in Idea 2, efficiently. Indeed, the BDD (as MC) in Fig. 4(c) is bisimilar to the MC in Fig. 4(b).

Idea 3: Represent a bisimilar version of the computation tree using a BDD.

[7] In this paper, we concentrate on reductions to *exact* WMC, leaving approximate approaches for future work [14].



(a) Unfactorized computation tree for (h=1, n=3). (b) Factorized (h=2, n=2).

Fig. 5. Two computation trees for the motivating example in Sect. 1.

4.4 The Algorithmic Benefits of BDD Construction

Thus far we have described how to construct a binarized MC bisimilar to the CT. Here, we argue that this construction has algorithmic benefits by filling in two details. First, the binarized representation is an important ingredient for compact BDDs. Second, we show how to choose a variable ordering that ensures that the BDDs grow linearly in the horizon. In sum,

Idea 4: WMC encodings of binarized Markov Chains may increase compression of computation trees.

To see the benefits of binarized transitions, we return to the factory example in Sect. 1. Figure 5(a) gives a bisimilar computation tree for the 3-factory $h = 1$ example. However, in this tree, the states are *unfactorized*: each node in the tree is a joint configuration of factories. This tree has 8 transitions (one for each possible joint state transition) with 8 distinct probabilities. On the other hand, the bisimilar computation tree in Fig. 1(d) has binarized transitions: each node corresponds to a single factory's state at a particular time-step, and each transition describes an update to only a single factory. This binarization enables the exploitation of new structure: in this case, the independence of the factories leads to smaller BDDs, that is otherwise lost when considering only joint configurations of factories.

Recall that the size of the ADD representation of the transition function is bounded from below by the number of distinct probabilities in the underlying MC: in this case, this is visualized by the number of distinct outgoing edge probabilities from all nodes in the unfactorized computation tree. Thus, a good binarization can have a drastically positive effect on performance. For the running example, rather than 2^n different transition probabilities (with n factories), the system now has only $4 \cdot n$ distinct transition probabilities!

Causal Orderings. Next, we explore some of the *engineering choices* RUBICON makes to exploit the sequential structure in a MC when constructing the BDD for a WMC query. First, note that the transition matrix $P(s, s')$ implicitly encodes a distribution over state transition functions, $S \to S$. To encode P as a BDD, we must encode each transition as a logical variable, similar to the situation in Sect. 4.2. In the case of binary transitions this is again easy. In the case of non-binary transitions, we again introduce additional logical variables [16,27,39,57].

This logical function has the following form:

$$f_P\colon \{0,1\}^C \to (S \to S). \tag{3}$$

Whereas the computation tree follows a fixed (temporal) order of states, BDDs can represent the same function (and the same weighted model count) using an arbitrary order. Note that the BDD's size and structure drastically depends both on the construction of the propositional formula *and* the order of the variables in that encoding. We can bound the size of the BDD by enforcing a variable order based on the temporal structure of the original MC. Specifically, given h coin collections $\boldsymbol{C} = C \times \ldots \times C$, one can generate a function f describing the h-length paths via repeated applications of f_P:

$$f\colon \{0,1\}^{\boldsymbol{C}} \to \mathrm{Paths}_h \quad f(C_1, \ldots, C_h) = \Big(f_P(C_h) \circ \ldots \circ f_P(C_1)\Big)(\iota) \tag{4}$$

Let ψ denote an indicator for the reachability property as a function over paths, $\psi\colon \mathrm{Paths}_h \to \{0,1\}$ with $\psi(\pi) = [\pi \in [\![\Diamond^{\leq h} T]\!]]$. We call predicates formed by composition with f_P, i.e., $\varphi = \psi \circ f_P$, *causal encodings* and orderings on $c_{i,t} \in \boldsymbol{C}$ that are lexicographically sorted in time, $t_1 < t_2 \implies c_{i,t_1} < c_{j,t_2}$, *causal orderings.* Importantly, causally ordered / encoded BDDs grow linearly in horizon h, [61, Corollary 1]. More precisely, let $\varphi^{\boldsymbol{C}}_{\mathcal{M},h}$ be causally encoded where $|\boldsymbol{C}| = h \cdot m$. The causally ordered BDD for $\varphi^{\boldsymbol{C}}_{\mathcal{M},h}$ has at most $h \cdot |S \times S_\psi| \cdot m \cdot 2^m$ nodes, where $|S_\psi| = 2$ for reachability properties.[8] However, while the worst-case growth is linear in the horizon, constructing that BDD may induce a super-linear cost in the size, e.g., function composition using BDDs is super-linear!

Figure 5(b) shows the motivating factory example with 2 factories and $h = 2$. The variables are causally ordered: the factories in time step 1 occur before the factories in time step 2. For n factories, a fixed number $f(n)$ of nodes are added to the BDD upon each iteration, guaranteeing growth on the order $\mathcal{O}(f(n) \cdot h)$. Note the factorization that occurs: the BDD has node sharing (node $c_2^{(2)}$ is reused) that yields additional computational benefits.

Summary and Remaining Steps. The operational view highlights that we want to compute a transformation of the original input MC $\mathcal{M}$. The logical view presents an approach to do so efficiently: By computing a BDD that stores a predicate describing all paths that reach the target, and interpreting and evaluating the (graph of the) BDD as an MC. In the following section, we discuss the two steps that we follow to create the BDD: (i) From $\mathcal{P}$ generate $\mathcal{P}'$ such that $\mathsf{CT}([\![\mathcal{P}]\!], h) \sim [\![\mathcal{P}']\!]$. (ii) From $\mathcal{P}'$ generate $\mathcal{M}$ such that $\mathcal{M} = [\![\mathcal{P}']\!]$.

5 Rubicon

We present Rubicon which follows the two steps outlined above. For exposition, we first describe a translation of *monolithic* Prism programs to `Dice` programs

[8] Generally, it is the smallest number of states required for a DFA to recognize ψ.

```
module main
  x : [0..1] init 0;
  y : [0..2] init 1;
  [] x=0 & y<2 -> 0.5:x'=1 + 0.5:y'=y+1;
  [] y=2 -> 1:y'=y-1;
  [] x=1 & y!=2 -> 1:x'=y & y'=x;
endmodule
property: P=? [F<=2 (x=0 & y=2)]
```

(a) Prism program with reachability query

$$\begin{tikzcd}
 & \arrow[d] & \\
\langle 0,0\rangle \arrow[r, "1/2"] \arrow[d, "1/2"'] & \langle 0,1\rangle \arrow[r, bend left, "1/2"] \arrow[d, "1/2"] & \langle 0,2\rangle \arrow[l, bend left, "1"] \\
\langle 1,0\rangle \arrow[ur, "1"'] & \langle 1,1\rangle \arrow[loop left, "1"] & \langle 1,2\rangle \arrow[l, "1"']
\end{tikzcd}$$

(b) Underlying MC

```
let s = init() in // init state
let T = hit(s) in  // init target
let (s, T) = if !T
   then let s' = step(s) in (s', hit(s'))
   else (s, T) in
let (s, T) = if !T then
   then let s' = step(s) in (s', hit(s'))
   else (s, T) in
T
```

(c) Main Dice program for h=2

```
fun init() { (0,1) }
fun hit((x,y)) { x ==0 &&  y == 2 }
fun step((x,y)) {
  if x==0 && y<2 then
    if flip 0.5 then (1,y) else (x,y+1)
  else if y==2 then (x,y-1)
  else if x==1 &&y!=1 then (y,x)
  else (x,y)
}
```

(d) Dice auxiliary functions

Fig. 6. From Prism to Dice using Rubicon.

and then extend this translation to admit modular programs. Technical steps and extensions are deferred to [38, Appendix].

Dice Preliminaries. We give a brief description of Dice, a probabilistic programming language (PPL) introduced in [39]. A PPL is a programming language augmented with a primitive notion of random choice: for instance, in Dice, a Bernoulli random variable is introduced by the syntax `flip 0.5`. The syntax of Dice is similar to the programming language OCaml: local variables are introduced by the syntax `let x = e`$_1$ `in e`$_2$, where `e`$_1$ and `e`$_2$ are *expressions*, i.e., sub-programs. Dice supports procedures, bounded integers, bounded loops, and standard control flow via `if`-statements.

One goal of a PPL is to perform *probabilistic inference*: compute the probability that the program returns a particular value. Inference on the tiny Dice program `let x = flip 0.1 in x` would yield that `true` is returned with probability 0.1. The Dice compiler performs probabilistic inference via weighted model counting and BDD compilation. In doing so, it accomplishes the *non-trivial* tasks of: (i) choosing a logical encoding for probabilistic programs (ii) establishing good variable orderings (iii) efficiently manipulating and constructing BDDs (iv) performing WMC . For details, we refer the reader to [39].

Rubicon uses Dice to effectively construct a BDD and perform WMC on a Dice program that reflects a description of some computation tree. This implementation exploits the structure that was described in Sect. 4.4: in particular, the BDD generated in Fig. 5(b) is exactly the BDD that will be generated by Dice from the output of Rubicon. The variable ordering used by Dice is given by the order in which program variables are introduced, and Rubicon's translation was designed with this variable ordering in mind.

Transpiling Prism to Dice. We present the core translation routine implemented in Rubicon. We note that the ultimate performance of Rubicon is heavily dependent on the quality of this translation. We evaluate the performance in the next section.

The Prism specification language consists of one or more reactive *modules* (or partially synchronized state machines) that may interact with each other. Our example in Fig. 1(b) illustrates fully synchronized state machines. While Prism programs containing multiple modules can be flattened into a single monolithic program, this yields an exponential blow-up: If one flattens the n modules in Fig. 1(b) to a single module, the resulting program has 2^n updates per command. This motivates our direct translation of PRISM programs containing multiple modules.

Monolithic Prism Programs. We explain most ideas on Prism programs that consist of a single "monolithic" module before we address the modular translation at the end of the subsection. A module has a set of bounded variables, and the valuations of these variables span the state space of the underlying MC. Its transitions are described by guarded *commands* of the form:

$$[\texttt{act}]\ \texttt{guard} \quad \rightarrow \quad p_1 : \texttt{update}_1 + \ldots\ldots + p_n : \texttt{update}_n$$

The *action* name `act` is only relevant in the modular case and can be ignored for now. The *guard* is a Boolean expression over the module's variables. If the guard evaluates to `true` for some state (a valuation), then the module evolves into one of the n successor states by updating its variables. An *update* is chosen according to the probability distribution given by the expressions $p_1, \ldots, p_n$. In every state enabling the guard, the evaluation of $p_1, \ldots, p_n$ must sum up to one. A set of guards *overlap* if they all evaluate to `true` on a given state. The semantics of overlapping guards in the monolithic setting is to first uniformly select an active guard and then apply the corresponding stochastic transition. Finally, a self-loop is implicitly added to states without an enabled guard.

Example 6. We present our translation primarily through example. In Fig. 6(a), we give a Prism program for a MC. The program contains two variables x and y, where x is either zero or one, and y between zero and two. There are thus 6 different states. We denote states as tuples with the x- and y-value. We depict the MC in Fig. 6(b). From state $\langle 0, 0\rangle$, (only) the first guard is enabled and thus there are two transitions, each with probability a half: one in which x becomes one and one in which y is increased by one. Finally, there is no guard enabled in state $\langle 1, 1\rangle$, resulting in an implicit self-loop.

Translation. All `Dice` programs consist of two parts: a *main* routine, which is run by default when the program starts, and *function declarations* that declare auxiliary functions. We first define the auxiliary functions. For simplicity let us temporarily assume that no guards overlap and that probabilities are constants, i.e., not state-dependent.

The main idea in the translation is to construct a `Dice` function `step` that, given the current state, outputs the next state. Because a monolithic Prism

```
module main
x : [0..2] init 1;
y : [0..2] init 1;
[] x>1 -> 1:x'=y&y'=x;
[] y<2 -> 1:x'=min(x+1,2);
endmodule
```

(a)

```
fun step((x,y)) {
  let aEn =(x>1)                   in
  let bEn =(y<2)                   in
  let act = selectFrom(aEn, bEn)   in
  if act==1 then (y,x)
  else if act==2 then (min(x+1,2),y)
  else (x,y)} ...
```

(b)

Fig. 7. PRISM program with overlapping guards and its translation (conceptually).

```
module m1
x : [0..1] init 0;
[a] x=1 -> 1:x'=1-y;
[b] x=0 -> 1:x'=0;
endmodule
module m2
y : [0..1] init 0;
[b] y=1 -> 0.5:y'=0 +0.5:y'=1;
[c] true -> 1:x'=1-x;
endmodule
```

(a)

```
fun step((x,y)) {
  let aEn =(x==1) in
  let bEn =(x=0 &&y=1) in
  let cEn =true in
  let act =selectFrom(aEn, bEn, cEn) in
  if act==1 then (1-y, y)
  else if act==2 then (0, flip 0.5)
  else if act==3 then (1-x, y)
  else (x,  y)
}
```

(b)

Fig. 8. Modular PRISM and resulting `Dice` step function.

program is almost a sequential program, in its most basic version, the `step` function is straightforward to construct using built-in `Dice` language primitives: we simply build a large if-else block corresponding to each command. This block iteratively considers each command's guard until it finds one that is satisfied. To perform the corresponding update we flip a coin – based on the probabilities corresponding to the updates – to determine which update to perform. If no command is enabled, we return the same state in accordance with the implicit self-loop. Figure 6(d) shows the program blocks for the PRISM program from Fig. 6(a) with target state $[\![x = 0, y = 2]\!]$. There are two other important auxiliary functions. The `init` function simply returns the initial state by translating the initialization statements from PRISM, and the `hit` function checks whether the current state is a target state that is obtained from the property.

Now we outline the main routine, given for this example in Fig. 6(c). This function first initializes the state. Then, it calls `step` 2 times, checking on each iteration using `hit` if the target state is reached. Finally, we return whether we have been in a target state. The probability to return true corresponds to the reachability probability on the underlying MC specified by the PRISM program.

Overlapping Guards. PRISM allows multiple commands to be enabled in the same state, with semantics to uniformly at random choose one of the enabled commands to evaluate. `Dice` has no primitive notion of this construct.[9] We illustrate the translation in Fig. 7(a) and Fig. 7(b). It determines which guards `aEn`, `bEn`, `cEn` are enabled. Then, we randomly select one of the commands which are enabled, i.e., we uniformly at random select a true bit from a given tuple

[9] One cannot simply condition on selecting an enabled guard as this redistributes probability mass over all paths and not only over paths with the same prefix.

of bits. We store the index of that bit and use it to execute the corresponding command.

Modular Prism Programs. For modular PRISM programs, the *action names* at the front of PRISM commands are important. In each module, there is a set of action names available. An action is *enabled* if each module that contains this action name has (at least) one command with this action whose guard is satisfied. Commands with an empty action are assumed to have a globally unique action name, so in that case the action is enabled iff the guard is enabled. Intuitively, once an action is selected, we randomly select a command per module in all modules containing this action name. Our approach resembles that for overlapping guards described above. See Fig. 8 for an intuitive example. To automate this, the updates require more care, cf. [38] for details.

Implementation. RUBICON is implemented on top of STORM's Python API and translates PRISM to `Dice` fully automatically. RUBICON supports all MCs in the PRISM benchmark suite and a large set of benchmarks from the PRISM website and the QVBS [35], with the note that we require a single initial state and ignore reward declarations. Furthermore, we currently do not support the hide/restrict process-algebraic compositions and some integer operations.

6 Empirical Comparisons

We compare and contrast the performance of STORM against RUBICON to empirically demonstrate the following strengths and weaknesses:[10]

Explicit Model Checking (STORM) represents the MC explicitly in a sparse matrix format. The approach suffers from the state space explosion, but has been engineered to scale to models with many states. Besides the state space, the sparseness of the transition matrix is essential for performance.

Symbolic Model Checking (STORM) represents the transition matrix and the reachability probability as an ADD. This method is strongest when the transition matrix and state vector have structure that enables a small ADD representation, like symmetry and sparsity.

RUBICON represents the set of paths through the MC as a (logical) BDD. This method excels when the state space has structure that enables a compact BDD representation, such as conditional independence, and hence scales well on examples with many (asymmetric) parallel processes or queries that admit a compact representation.

The sources, benchmarks and binaries are archived.[11]

There is no clear-cut model checking technique that is superior to others (see QCOMP [12]). We demonstrate that, while RUBICON is not competitive on some

[10] All experiments were conducted with STORM version 1.6.0 on the same server with 512 GB of RAM, using a single thread of execution. Time was reported using the built-in Unix `time` utility; the total wall-clock time is reported.

[11] http://doi.org/10.5281/zenodo.4726264 and http://github.com/sjunges/rubicon.

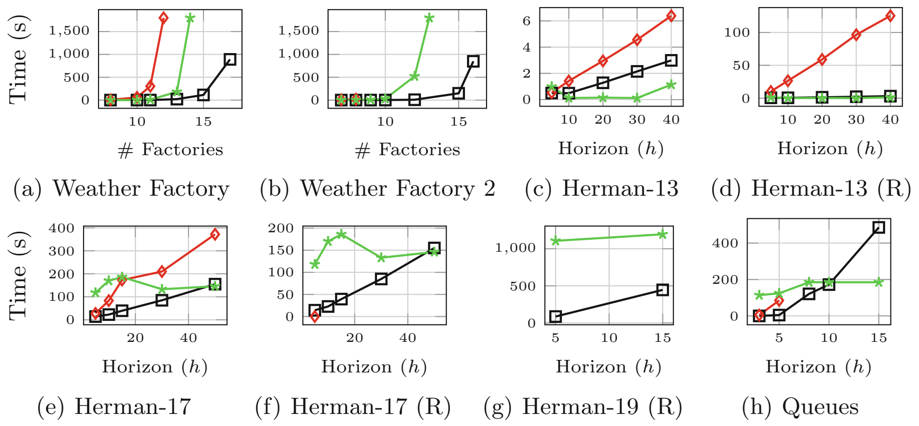


Fig. 9. Scaling plots comparing Rubicon (—◻—), Storm's symbolic engine (—◇—), and Storm's explicit engine (—*—). An "(R)" in the caption denotes random parameters.

commonly used benchmarks [52], it improves a modern model checking portfolio approach on a significant set of benchmarks. Below we provide several natural models on which Rubicon is superior to one or both competing methods. We also evaluated Rubicon on standard benchmarks, highlighting that Rubicon is applicable to models from the literature. We see that Rubicon is effective on Herman (elaborated below), has mixed results on BRP [38, Appendix], and is currently not competitive on some other standard benchmarks (NAND, EGL, LeaderSync). While not exhaustive, our selected benchmarks highlight specific strengths and weaknesses of Rubicon. Finally, a particular benefit of Rubicon is fast sampling of parametric chains, which we demonstrate on Herman and our factory example.

Scaling Experiments. In this section, we describe several scaling experiments (Fig. 9), each designed to highlight a specific strength or weakness.

Weather Factories. First, Fig. 9(a) describes a generalization of the motivating example from Sect. 1. In this model, the probability that each factory is on strike is dependent on a common random event: whether or not it is raining. The rain on each day is dependent on the previous day's weather. We plot runtime for an increasing number of factories for h=10. Both Storm engines eventually fail due to the state explosion and the number of distinct probabilities in the MC. Rubicon is orders of magnitude faster in comparison, highlighting that it does not depend on complete independence among the factories. Figure 9(b) shows a more challenging instance where the weather includes *wind* which, each day, affects whether or not the sun will shine, which in turn affects strike probability.

Herman. Herman is based on a distributed protocol [37] that has been well-studied [1,53] and which is one of the standard benchmarks in probabilistic model checking. Rather than computing the expected steps to 'stabilization', we consider the step-bounded probability of stabilization. Usually, all participants in the protocol flip a coin with the same bias. The model is then highly symmetric, and hence is amenable to symbolic representation with ADDs. Figures 9(c) and 9(e) show how the methods scale on Herman examples with 13 and 17 parallel processes. We observe that the explicit approach scales very efficiently in the number of iterations but has a much higher up-front model-construction cost, and hence can be slower for fewer iterations.

To study what happens when the coin biases vary over the protocol participants, we made a version of the Herman protocol where each participant's bias is randomly chosen, which ruins the symmetry and so causes the ADD-based approaches to scale significantly worse (Figs. 9(d) and 9(f), and 9(g)); we see that symbolic ADD-based approaches completely fail on Herman 17 and Herman 19 (the curve terminating denotes a memory error). RUBICON and the explicit approach are unaffected by varying parameters.

Queues. The Queues model has K queues of capacity Q where every step, tasks arrive with a particular probability. Three queues are of type 1, the others of type 2. We ask the probability that all queues of type 1 and at least one queue of type 2 is full within k steps. Contrary to the previous models, the ADD representation of the transition matrix is small. Figure 9(h) shows the relative scaling on this model with $K = 8$ and $Q = 3$. We observe that ADDs quickly fail due to inability to concisely represent the probability vector $\boldsymbol{x}$ from Sect. 3. RUBICON outperforms explicit model checking until $h = 10$.

Sampling Parametric Markov Chains. We evaluate performance for the pMC sampling problem outlined in Sect. 2. Table 1 gives for four models the time to construct the BDD and to perform WMC, as well as the time to construct an ADD in STORM and to perform model checking with this ADD. Finally, we show the time for STORM to compute the solution function of the pMC (with the explicit representation). The pMC sampling in STORM – symbolic and explicit – computes the reachability probabilities with concrete probabilities. RUBICON, in contrast, constructs a 'parametric' BDD once, amortizing the cost of repeated efficient evaluation. The 'parametric BDD' may be thought of as a solution function, as discussed in Sect. 4.1. STORM cannot compute these solution functions as efficiently. We observe in Table 1 that fast parametric sampling is realized in RUBICON: for instance, after a 40s up-front compilation of the factories example with 15 factories, we have a solution function in factorized form and it costs an order of magnitude less time to draw a sample. Hence, sampling and computation of solution functions of pMCs is a major strength of RUBICON.

7 Discussion, Related Work, and Conclusion

We have demonstrated that the probabilistic inference approach to probabilistic model checking can improve scalability on an important class of problems. Another benefit of the approach is for sampling pMCs. These are used to evaluate e.g., robustness of systems [1], or to synthesise POMDP controllers [41]. Many state-of-the-art approaches [17,19,24] require the evaluation of various instantiated MCs, and RUBICON is well-suited to this setting. More generally, support

Table 1. Sampling performance comparison and pMC model checking, time in seconds.

Model	Rubicon		Storm (w/ ADD)		Storm (explicit)
	Build	WMC	Build	Solve	pMC solving
Herman R 13 ($h = 10$)	3	<1	32	18	>1800
Herman R 17 ($h = 10$)	45	28	>1800	–	>1800
Factories 12 ($h = 15$)	2	<1	59	286	>1800
Factories 15 ($h = 15$)	40	4	>1800	–	>1800

of inference techniques opens the door to a variety of algorithms for additional queries, e.g., computing *conditional probabilities* [3,8].

An important limitation of probabilistic inference is that only finitely many paths can be stored. For infinite horizon properties in cyclic models, an infinite set of arbitrarily long paths would be required. However, as standard in probabilistic model checking, we may soundly approximate infinite horizons. Additionally, the inference algorithm in `Dice` does not support a notion of non-determinism. It thus can only be used to evaluate MCs, not Markov decision processes. However, [61] illustrates that this is not a conceptual limitation. Finally, we remark that Rubicon achieves its performance with a straightforward translation. We are optimistic that this is a first step towards supporting a larger class of models by improving the transpilation process for specific problems.

Related Work. The tight connection with inference has been recently investigated via the use of model checking for Bayesian networks, the prime model in probabilistic inference [56]. Bayesian networks can be described as probabilistic programs [10] and their operational semantics coincides with MCs [31]. Our work complements these insights by studying how symbolic model checking can be sped up by probabilistic inference.

The path-based perspective is tightly connected to *factored state spaces*. Factored state spaces are often represented as (bipartite) Dynamic Bayesian networks. ADD-based model checking for DBNs has been investigated in [25], with mixed results. Their investigation focuses on using ADDs for factored state space representations. We investigate using BDDs representing paths. Other approaches also investigated a path-based view: The symbolic encoding in [28] annotates propositional sub-formulae with probabilities, an idea closer to ours. The underlying process implicitly constructs an (uncompressed) CT leading to an exponential blow-up. Likewise, an explicit construction of a computation tree without factorization is considered in [62]. Compression by grouping paths has been investigated in two *approximate* approaches: [55] discretises probabilities and encodes into a satisfiability problem with quantifiers and bit-vectors. This idea has been extended [60] to a PAC algorithm by purely propositional encodings and (approximate) model counting [14]. Finally, factorisation exploits symmetries, which can be exploited using symmetry reduction [50]. We highlight that the latter is not applicable to the example in Fig. 1(d).

There are many techniques for exact probabilistic inference in various forms of probabilistic modeling, including probabilistic graphical models [20,54]. The semantics of graphical models make it difficult to transpile Prism programs, since commonly used operations are lacking. Recently, *probabilistic programming languages* have been developed which are more amenable to transpilation [13,23,29,30,59]. We target `Dice` due to the technical development that it enables in Sect. 4, which enabled us to design and explain our experiments. Closest related to `Dice` is ProbLog [27], which is also a PPL that performs inference via WMC; ProbLog has different semantics from `Dice` that make the translation less straightforward. The paper [61] uses an encoding similar to `Dice` for inferring specifications based on observed traces. ADDs and variants have been considered for probabilistic inference [15,18,58], which is similar to the process commonly used for probabilistic model checking. The planning community has developed their own disjoint sets of methods [45]. Some ideas from learning have been applied in a model checking context [11].

8 Conclusion

We present Rubicon, bringing probabilistic AI to the probabilistic model checking community. Our results show that Rubicon can outperform probabilistic model checkers on some interesting examples, and that this is not a coincidence but rather the result of a significantly different perspective.

References

1. Aflaki, S., Volk, M., Bonakdarpour, B., Katoen, J.P., Storjohann, A.: Automated fine tuning of probabilistic self-stabilizing algorithms. In: SRDS, pp. 94–103. IEEE (2017)
2. de Alfaro, L., Kwiatkowska, M., Norman, G., Parker, D., Segala, R.: Symbolic model checking of probabilistic processes using MTBDDs and the Kronecker representation. In: Graf, S., Schwartzbach, M. (eds.) TACAS 2000. LNCS, vol. 1785, pp. 395–410. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-46419-0_27
3. Andrés, M.E., van Rossum, P.: Conditional probabilities over probabilistic and nondeterministic systems. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 157–172. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_12
4. Baier, C., de Alfaro, L., Forejt, V., Kwiatkowska, M.: Model checking probabilistic systems. In: Clarke, E., Henzinger, T., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 963–999. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_28
5. Baier, C., Hensel, C., Hutschenreiter, L., Junges, S., Katoen, J.P., Klein, J.: Parametric Markov chains: PCTL complexity and fraction-free Gaussian elimination. Inf. Comput. **272**, 104504 (2020)
6. Baier, C., Hermanns, H.: Weak bisimulation for fully probabilistic processes. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 119–130. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-63166-6_14

7. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press, Cambridge (2008)
8. Baier, C., Klein, J., Klüppelholz, S., Märcker, S.: Computing conditional probabilities in Markovian models efficiently. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 515–530. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_43
9. Baluta, T., Shen, S., Shinde, S., Meel, K.S., Saxena, P.: Quantitative verification of neural networks and its security applications. In: CCS, pp. 1249–1264. ACM (2019)
10. Batz, K., Kaminski, B.L., Katoen, J.-P., Matheja, C.: How long, O Bayesian network, will I sample thee? - a program analysis perspective on expected sampling times. In: Ahmed, A. (ed.) ESOP 2018. LNCS, vol. 10801, pp. 186–213. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89884-1_7
11. Brázdil, T., et al.: Verification of Markov decision processes using learning algorithms. In: Cassez, F., Raskin, J.-F. (eds.) ATVA 2014. LNCS, vol. 8837, pp. 98–114. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11936-6_8
12. Budde, C.E., et al.: On correctness, precision, and performance in quantitative verification: QComp 2020 competition report. In: ISOLA. LNCS. Springer, Heidelberg (2020)
13. Carpenter, B., et al.: Stan: a probabilistic programming language. J. Stat. Soft. **VV**(Ii) (2016)
14. Chakraborty, S., Fried, D., Meel, K.S., Vardi, M.Y.: From weighted to unweighted model counting. In: IJCAI, pp. 689–695. AAAI Press (2015)
15. Chavira, M., Darwiche, A.: Compiling Bayesian networks using variable elimination. In: IJCAI, pp. 2443–2449 (2007)
16. Chavira, M., Darwiche, A.: On probabilistic inference by weighted model counting. Artif. Intell. **172**(6–7), 772–799 (2008)
17. Chen, T., Hahn, E.M., Han, T., Kwiatkowska, M.Z., Qu, H., Zhang, L.: Model repair for Markov decision processes. In: TASE, pp. 85–92. IEEE (2013)
18. Claret, G., Rajamani, S.K., Nori, A.V., Gordon, A.D., Borgström, J.: Bayesian inference using data flow analysis. In: FSE, pp. 92–102 (2013)
19. Cubuktepe, M., Jansen, N., Junges, S., Katoen, J.P., Topcu, U.: Scenario-based verification of uncertain MDPs. In: TACAS. LNCS, vol. 12078, pp. 287–305. Springer, Heidelberg (2020)
20. Darwiche, A.: SDD: a new canonical representation of propositional knowledge bases. In: IJCAI, pp. 819–826 (2011)
21. Darwiche, A., Marquis, P.: A knowledge compilation map. JAIR **17**, 229–264 (2002)
22. Daws, C.: Symbolic and parametric model checking of discrete-time Markov chains. In: Liu, Z., Araki, K. (eds.) ICTAC 2004. LNCS, vol. 3407, pp. 280–294. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31862-0_21
23. De Raedt, L., Kimmig, A., Toivonen, H.: ProbLog: a probabilistic prolog and its application in link discovery. In: IJCAI, vol. 7, pp. 2462–2467 (2007)
24. Dehnert, C., et al.: `PROPhESY`: a PRObabilistic ParamEter SYnthesis tool. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 214–231. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_13
25. Deininger, D., Dimitrova, R., Majumdar, R.: Symbolic model checking for factored probabilistic models. In: Artho, C., Legay, A., Peled, D. (eds.) ATVA 2016. LNCS, vol. 9938, pp. 444–460. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46520-3_28
26. van Dijk, T., van de Pol, J.: Multi-core symbolic bisimulation minimisation. STTT **20**(2), 157–177 (2018)

27. Fierens, D., et al.: Inference and learning in probabilistic logic programs using weighted Boolean formulas. Theory Pract. Log. Prog. **15**(3), 358–401 (2015)
28. Fränzle, M., Hermanns, H., Teige, T.: Stochastic satisfiability modulo theory: a novel technique for the analysis of probabilistic hybrid systems. In: Egerstedt, M., Mishra, B. (eds.) HSCC 2008. LNCS, vol. 4981, pp. 172–186. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78929-1_13
29. Gehr, T., Misailovic, S., Vechev, M.: PSI: exact symbolic inference for probabilistic programs. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 62–83. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_4
30. Gordon, A.D., Henzinger, T.A., Nori, A.V., Rajamani, S.K.: Probabilistic programming. In: FOSE, pp. 167–181. ACM (2014)
31. Gretz, F., Katoen, J.P., McIver, A.: Operational versus weakest pre-expectation semantics for the probabilistic guarded command language. Perform. Eval. **73**, 110–132 (2014)
32. Hahn, E.M., Hermanns, H., Wachter, B., Zhang, L.: PASS: abstraction refinement for infinite probabilistic models. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 353–357. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12002-2_30
33. Hahn, E.M., Hermanns, H., Zhang, L.: Probabilistic reachability for parametric Markov models. STTT **13**(1), 3–19 (2011)
34. Hartmanns, A., Hermanns, H.: The modest toolset: an integrated environment for quantitative modelling and verification. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 593–598. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_51
35. Hartmanns, A., Klauck, M., Parker, D., Quatmann, T., Ruijters, E.: The quantitative verification benchmark set. In: TACAS. LNCS, vol. 11427, pp. 344–350. Springer, Heidelberg (2019)
36. Hensel, C., Junges, S., Katoen, J.P., Quatmann, T., Volk, M.: The probabilistic model checker storm. STTT (2021, to appear)
37. Herman, T.: Probabilistic self-stabilization. Inf. Process. Lett. **35**(2), 63–67 (1990)
38. Holtzen, S., Junges, S., Vazquez-Chanlatte, M., Millstein, T., Seshia, S.A., Van den Broeck, G.: Model checking finite-horizon Markov chains with probabilistic inference. CoRR abs/2105.12326 (2021)
39. Holtzen, S., Van den Broeck, G., Millstein, T.: Scaling exact inference for discrete probabilistic programs. PACMPL OOPSLA, November 2020
40. Jansen, D.N., Groote, J.F., Timmers, F., Yang, P.: A near-linear-time algorithm for weak bisimilarity on Markov chains. In: CONCUR. LIPIcs, vol. 171, pp. 8:1–8:20. Schloss Dagstuhl - LZI (2020)
41. Junges, S., et al.: Finite-state controllers of POMDPs using parameter synthesis. In: UAI, pp. 519–529. AUAI Press (2018)
42. Katoen, J.-P., Gretz, F., Jansen, N., Kaminski, B.L., Olmedo, F.: Understanding probabilistic programs. In: Meyer, R., Platzer, A., Wehrheim, H. (eds.) Correct System Design. LNCS, vol. 9360, pp. 15–32. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23506-6_4
43. Katoen, J.-P., Kemna, T., Zapreev, I., Jansen, D.N.: Bisimulation minimisation mostly speeds up probabilistic model checking. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 87–101. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71209-1_9
44. Kattenbelt, M., Kwiatkowska, M.Z., Norman, G., Parker, D.: A game-based abstraction-refinement framework for Markov decision processes. FMSD **36**(3), 246–280 (2010)

45. Klauck, M., Steinmetz, M., Hoffmann, J., Hermanns, H.: Bridging the gap between probabilistic model checking and probabilistic planning: survey, compilations, and empirical comparison. JAIR **68**, 247–310 (2020)
46. Klein, J., et al.: Advances in probabilistic model checking with PRISM: variable reordering, quantiles and weak deterministic büchi automata. STTT **20**(2), 179–194 (2018)
47. Koller, D., Friedman, N.: Probabilistic Graphical Models: Principles and Techniques. MIT Press, Cambridge (2009)
48. Kozen, D.: Semantics of probabilistic programs. JCSS **22**(3), 328–350 (1981)
49. Kwiatkowska, M., Norman, G., Parker, D.: Probabilistic symbolic model checking with PRISM: a hybrid approach. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 52–66. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-46002-0_5
50. Kwiatkowska, M., Norman, G., Parker, D.: Symmetry reduction for probabilistic model checking. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 234–248. Springer, Heidelberg (2006). https://doi.org/10.1007/11817963_23
51. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_47
52. Kwiatkowska, M.Z., Norman, G., Parker, D.: The PRISM benchmark suite. In: QEST, pp. 203–204. IEEE (2012)
53. Kwiatkowska, M.Z., Norman, G., Parker, D.: Probabilistic verification of Herman's self-stabilisation algorithm. Formal Aspects Comput. **24**(4–6), 661–670 (2012)
54. Pearl, J.: Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference. Morgan Kaufmann (1988)
55. Rabe, M.N., Wintersteiger, C.M., Kugler, H., Yordanov, B., Hamadi, Y.: Symbolic approximation of the bounded reachability probability in large Markov chains. In: Norman, G., Sanders, W. (eds.) QEST 2014. LNCS, vol. 8657, pp. 388–403. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10696-0_30
56. Salmani, B., Katoen, J.-P.: Bayesian inference by symbolic model checking. In: Gribaudo, M., Jansen, D.N., Remke, A. (eds.) QEST 2020. LNCS, vol. 12289, pp. 115–133. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-59854-9_9
57. Sang, T., Beame, P., Kautz, H.A.: Performing Bayesian inference by weighted model counting. In: AAAI, vol. 5, pp. 475–481 (2005)
58. Smolka, S., et al.: Scalable verification of probabilistic networks. In: PLDI, pp. 190–203. ACM (2019)
59. van de Meent, J.W., Paige, B., Yang, H., Wood, F.: An Introduction to Probabilistic Programming. arXiv:1809.10756 (2018)
60. Vazquez-Chanlatte, M., Rabe, M.N., Seshia, S.A.: A model counter's guide to probabilistic systems. CoRR abs/1903.09354 (2019)
61. Vazquez-Chanlatte, M., Seshia, S.A.: Maximum causal entropy specification inference from demonstrations. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12225, pp. 255–278. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53291-8_15
62. Wimmer, R., Braitling, B., Becker, B.: Counterexample generation for discrete-time Markov chains using bounded model checking. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 366–380. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-93900-9_29

Enforcing Almost-Sure Reachability in POMDPs

Sebastian Junges[1(✉)], Nils Jansen[2], and Sanjit A. Seshia[1]

[1] University of California at Berkeley, Berkeley, USA
sjunges@berkeley.edu
[2] Radboud University Nijmegen, Nijmegen, The Netherlands

Abstract. Partially-Observable Markov Decision Processes (POMDPs) are a well-known stochastic model for sequential decision making under limited information. We consider the EXPTIME-hard problem of synthesising policies that almost-surely reach some goal state without ever visiting a bad state. In particular, we are interested in computing the winning region, that is, the set of system configurations from which a policy exists that satisfies the reachability specification. A direct application of such a winning region is the safe exploration of POMDPs by, for instance, restricting the behavior of a reinforcement learning agent to the region. We present two algorithms: A novel SAT-based iterative approach and a decision-diagram based alternative. The empirical evaluation demonstrates the feasibility and efficacy of the approaches.

1 Introduction

Partially observable Markov decision processes (POMDPs) constitute the standard model for agents acting under partial information in uncertain environments [34,52]. A common problem is to find a policy for the agent that maximizes a reward objective [36]. This problem is undecidable, yet, well-established approximate [27], point-based [43], or Monte-Carlo-based [49] methods exist. In safety-critical domains, however, one seeks a *safe* policy that exhibits strict behavioral guarantees, for instance in the form of temporal logic constraints [44]. The aforementioned methods are not suitable to deliver provably safe policies. In contrast, we employ almost-sure reach-avoid specifications, where the probability to reach a set of *avoid* states is zero, and the probability to *reach* a set of goal states is one. Our **Challenge 1** is to compute a policy that adheres to such specifications. Furthermore, we aim to ensure the *safe exploration of a POMDP*, with safe reinforcement learning [23] as direct application. **Challenge 2** is then

This work is partially supported by NSF grants 1545126 (VeHICaL), 1646208 and 1837132, by the DARPA contracts FA8750-18-C-0101 (AA) and FA8750-20-C-0156 (SDCPS), by Berkeley Deep Drive, and by Toyota under the iCyPhy center.
This research has been partially funded by NWO grant OCENW.KLEIN.187: “Provably Correct Policies for Uncertain Partially Observable Markov Decision Processes”.

to compute a large set of safe policies for the agent to choose from at any state of the POMDP. Such sets of policies are called *permissive policies* [21,31].

POMDP Almost-Sure Reachability Verification. Let us remark that in POMDPs, we cannot directly observe in which state we are, but we are in general able to track a *belief*, i.e., a distribution over states that describes where in the POMDP we may be. The belief allows us to formulate the following **verification task**:

For a POMDP, sets of target and avoid states, and a belief, does a policy exist such that we reach the target states without ever visiting a bad state?

The underlying EXPTIME-complete problem requires—in general—policies with access to memory of exponential size in the number of states [4,18]. For safe exploration and, e.g., to support nested temporal properties, the ability to solve this problem *for each belief in the POMDP* is essential.

We base our approaches on the concept of a *winning region*, also referred to as controllable or attractor regions. Such regions are sets of *winning beliefs* from which a policy exists that guarantees to satisfy an almost-sure specification. The verification task relates three concrete problems which we tackle in this paper: (1) *Decide* whether a belief is winning, (2) *compute* the *maximal* winning region, and (3) *compute* a *large* yet not necessarily maximal winning region. We now outline our two approaches. First, we directly exploit model checking for MDPs [5] using belief abstractions. The second, much faster approach iteratively exploits *satisfiability solving* (SAT) [8]. Finally, we define a scheme to enable safe reinforcement learning [23] for POMDPs, referred to as *shielding* [2,30].

MDP Model Checking. A prominent approach gives the semantics of a POMDP via an (infinite) belief MDP whose states are the beliefs in the POMDP [36]. For almost-sure specifications, it is sufficient to consider *belief-supports* rather than beliefs. In particular, two beliefs with the same support are either both in a winning region or not [47]. We abstract a belief MDP into a finite belief-support MDP, whose states are the support of beliefs. The (maximal) winning region are (all) states of the belief-support MDP from which one can almost surely reach a belief support that contains a goal state without visiting belief support states that contain an avoid state.

To find a winning region in the POMDP, we thus just have to solve almost-sure reachability in this finite MDP. The number of belief supports, however, is exponentially large in the number of POMDP states, threatening the efficient application of explicit state verification approaches. Symbolic state space representations are a natural option to mitigate this problem [7]. We construct a symbolic description of the belief support MDP and apply state-of-the-art symbolic model checking. Our experiments show that this approach (referred to as *MDP Model Checking*) does in general not alleviate the exponential blow-up.

Incremental SAT Solving. While the belief support model exploits the structure of the belief support MDP by using a symbolic state space representation, it does not exploit elementary properties of the structure of winning regions. To overcome the scalability challenge, we aim to exploit information from the original POMDP, rather than working purely on the belief-support MDP. In a nutshell, our approach computes the winning regions in a backward fashion by *optimistically* searching policies without memory on the POMDP level. Concretely, starting from the belief support states that shall be reached almost-surely, further states are added to the winning region if we quickly can find a policy that reaches these states without visiting those that are to avoid. We search for these policies by incrementaly employing an encoding based on SAT solving. This symbolic encoding avoids an expensive construction of the belief support MDP. The computed winning region directly translates to sufficient constraints on the set of safe policies, i.e., each policy satisfying these constraints satisfies, by construction, the specification. The key idea is to successively add short-cuts corresponding to already known safe policies. These changes to the structure of the POMDP are performed implicitly on the SAT encoding. The resulting scalable method is sound, but not complete by itself. However, it can be rendered complete by trading off a certain portion of the scalability; intuitively one would eventually search for policies with larger amounts of memory.

Shielding. An agent that stays within a winning region is guaranteed to adhere to the specification. In particular, we *shield* (or *mask*) any action of the agent that may lead out of the winning region [1,39,42]. We stress that the shape of the winning region is independent of the transition probabilities or rewards in the POMDP. This independence means that the only prior knowledge we need to assume is the topology, that is, the graph of the POMDP. A pre-computation of the winning region thus yields a shield and allows us to restrict an agent to safely explore environments, which is the essential requirement for safe reinforcement learning [22,23] of POMDPs. The shield can be used with any RL agent [2].

Comparison with the State-of-the-Art. Similar to our approach, [15] solves almost-sure specifications using SAT. Intuitively, the aim is to find a so-called *simple policy* that is Markovian (aka memoryless). Such a policy may not exist, yet, the method can be applied to a POMDP that has an extended state space to account for finite memory [33,37]. There are three shortcomings that our incremental SAT approach overcomes. First, one needs to pre-define the memory a policy has at its disposal, as well as a fixed lookahead on the exploration of the POMDP. Our encoding does not require to fix these hyperparameter a priori. Second, the approach is only feasible if small memory bounds suffice. Our approach scales to models that require policies with larger memory bounds. Third, the approach finds a single simple policy starting from a pre-defined initial state. Instead, we find a large winning region. For safe exploration, this means that we may exclude many policies and never explore important parts of the system, harming the final performance of the agent. Shielding MDPs is not new [2,9,10,30]. However, those methods do neither take partial observability into account, nor can they guarantee reaching desirable states. Nam and Alur [39] cover partial observability and reachability, but do not account for stochastic uncertainty.

Experiments. To showcase the feasibility of our method, we adopted a number of typical POMDP environments. We demonstrate that our method scales better than the state of the art. We evaluate the shield by letting an agent explore the

POMDP environment according to the permissive policy, thereby enforcing the satisfaction of the almost-sure specification. We visualize the resulting behavior of the agent in those environments with a set of videos.

Contributions. Our paper makes four contributions: (1) We present an incremental SAT-based approach to compute policies that satisfy almost-sure properties. The method scales to POMDPs whose belief-support states count billions; (2) The novel approach is able to find large winning regions that yield permissive policies. (3) We implement a straightforward approach that constructs the belief-support symbolically using state-of-the-art model checking. We show that its completeness comes at the cost of limited scalability. (4) We construct a shield for almost-sure specifications on POMDPs which enforces at runtime that *no unsafe states are visited* and that, under mild assumptions, *the agent almost-surely reaches the set of desirable states.*

Further Related Work. Chatterjee et al. compute winning regions for minimizing a reward objective via an explicit state representation [17], or consider almost-sure reachability using an explicit state space [16,51]. The problem of determining any winning policy can be cast as a strong cyclic planning problem, proposed earlier with decision diagrams [7]. Indeed, our BDD-based implementation on the belief-support MDP can be seen as a reimplementation of that approach.

Quantitative variants of reach-avoid specifications have gained attention in, e.g., [11,28,40]. Other approaches restrict themselves to simple policies [3,33,45,58]. Wang et al. [55] use an iterative Satisfiability Modulo Theories (SMT) [6] approach for quantitative finite-horizon specifications, which requires computing beliefs. Various general POMDP approaches exist, e.g., [26,27,29,48,49,54,56]. The underlying approaches depend on discounted reward maximization and can satisfy almost-sure specifications with high reliability. However, enforcing probabilities that are close to 0 or 1 requires a discount factor close to 1, drastically reducing the scalability of such approaches [28]. Moreover, probabilities in the underlying POMDP need to be precisely given, which is not always realistic [14].

Another line of work (for example [53]) uses an idea similar to winning regions with uncertain specifications, but in a fully observable setting. Finally, complementary to shielding, there are approaches that guide reinforcement learning (with full observability) via temporal logic constraints [24,25].

2 Preliminaries and Formal Problem

We briefly introduce POMDPs and their semantics in terms of belief MDPs, before formalising and studying the problem variants outlined in the introduction. We present belief-support MDPs as a finite abstraction of infinite belief MDPs.

We define the support $\mathit{supp}(\mu) = \{x \in X \mid \mu(x) > 0\}$ of a discrete probability distribution μ and denote the set of all distributions with $\mathit{Distr}(X)$.

Definition 1 (MDP). *A* Markov decision process *(MDP) is a tuple* $\mathcal{M} = \langle S, \mathrm{Act}, \mu_{\mathrm{init}}, \mathbf{P}\rangle$ *with a set* S *of states, an initial distribution* $\mu_{\mathrm{init}} \in \mathit{Distr}(S)$*, a finite set* Act *of actions, and a transition function* $\mathbf{P}\colon S \times \mathrm{Act} \to \mathit{Distr}(S)$*.*

Let $\text{post}_s(\alpha) = \mathit{supp}(\mathbf{P}(s,\alpha))$ denote the states that may be the successors of the state $s \in S$ for action $\alpha \in \text{Act}$ under the distribution $\mathbf{P}(s,\alpha)$. If $\text{post}_s(\alpha) = \{s\}$ for all actions α, s is called *absorbing*.

Definition 2 (POMDP). *A* partially observable MDP *(POMDP) is a tuple* $\mathcal{P} = \langle \mathcal{M}, \Omega, \text{obs} \rangle$ *with* $\mathcal{M} = \langle S, \text{Act}, \mu_{\text{init}}, \mathbf{P} \rangle$ *the underlying MDP with finite* S*,* Ω *a finite set of observations, and* $\text{obs}\colon S \to \Omega$ *an observation function. We assume that there is a unique initial observation, i.e., that* $|\{\text{obs}(s) \mid s \in \mathit{supp}(\mu_{\text{init}})\}| = 1$.

More general observation functions $\text{obs}\colon S \to \mathit{Distr}(\Omega)$ are possible via a (polynomial) reduction [17]. A path through an MDP is a sequence π, $\pi = (s_0,\alpha_0)(s_1,\alpha_1)\ldots s_n$ of states and actions. such that $s_{i+1} \in \text{post}_{s_i}(\alpha_i)$ for $\alpha_i \in \text{Act}$ and $0 \leq i < n$. The observation function obs applied to a path yields an observation(-action) sequence $\text{obs}(\pi)$ of observations and actions.

For modeling flexibility, we allow actions to be unavailable in a state (e.g., opening doors is only available when at a door), and it turned out to be crucial to handle this explicitly in the following algorithms. Technically, the transition function is a partial function, and the enabled actions are a set $\text{EnAct}(s) = \{\alpha \in \text{Act} \mid \text{post}_s(\alpha) \neq \emptyset\}$. To ease the presentation, we assume that states s, s' with the same observation share a set of enabled actions $\text{EnAct}(s) = \text{EnAct}(s')$.

Definition 3 (Policy). *A policy* $\sigma\colon (S \times \text{Act})^* \times S \to \mathit{Distr}(\text{Act})$ *maps a path* π *to a distribution over actions. A policy is* observation-based*, if for each two paths* π*,* π' *it holds that* $\text{obs}(\pi) = \text{obs}(\pi') \Rightarrow \sigma(\pi) = \sigma(\pi')$*. A policy is* memoryless*, if for each* π*,* π' *it holds that* $\mathsf{last}(\pi) = \mathsf{last}(\pi') \Rightarrow \sigma(\pi) = \sigma(\pi')$*. A policy is* deterministic*, if for each* π*,* $\sigma(\pi)$ *is a Dirac distribution, i.e., if* $|\mathit{supp}(\sigma(\pi))| = 1$.

Policies resolve nondeterminism and partial observability by turning a (PO)MDP into the *induced* infinite discrete-time Markov chain whose states are the finite paths of the (PO)MDP. Probability measures are defined on this Markov chain.

For POMDPs, a *belief* describes the probability of being in certain state based on an observation sequence. Formally, a belief $\mathfrak{b}$ is a distribution $\mathfrak{b} \in \mathit{Distr}(S)$ over the states. A state s with positive belief $\mathfrak{b}(s) > 0$ is in the *belief support*, $s \in \mathit{supp}(b)$. Let $\mathit{Pr}^{\sigma}_{\mathfrak{b}}(S')$ denote the probability to reach a set $S' \subseteq S$ of states from belief $\mathfrak{b}$ under the policy σ. More precisely, $\mathit{Pr}^{\sigma}_{\mathfrak{b}}(S')$ denotes the probability of all paths that reach S' from $\mathfrak{b}$ when nondeterminism is resolved by σ.

The policy synthesis problem usually consists in finding a policy that satisfies a certain specification for a POMDP. We consider *reach-avoid* specifications, a subclass of indefinite horizon properties [46]. For a POMDP $\mathcal{P}$ with states S, such a specification is $\varphi = \langle \mathit{REACH}, \mathit{AVOID} \rangle \subseteq S \times S$. We assume that states in AVOID and in REACH are (made) absorbing and $\mathit{REACH} \cap \mathit{AVOID} = \emptyset$.

Definition 4 (Winning). *A policy* σ *is* winning *for* φ *from belief* $\mathfrak{b}$ *in (PO)MDP* $\mathcal{P}$ *iff* $\mathit{Pr}^{\sigma}_{\mathfrak{b}}(\mathit{AVOID}) = 0$ *and* $\mathit{Pr}^{\sigma}_{\mathfrak{b}}(\mathit{REACH}) = 1$*, i.e., if it reaches AVOID with probability zero and REACH with probability one (almost-surely) when* $\mathfrak{b}$ *is the initial state. Belief* $\mathfrak{b}$ *is* winning *for* φ *in* $\mathcal{P}$ *if there exists a winning policy from* $\mathfrak{b}$.

We omit $\mathcal{P}$ and φ whenever it is clear from the context and simply call $\mathfrak{b}$ winning.

Problem 1: Given a POMDP, a belief $\mathfrak{b}$, and a specification φ, decide whether $\mathfrak{b}$ is winning and find a policy σ that is winning from $\mathfrak{b}$.

The problem is EXPTIME-complete [18]. Contrary to MDPs, it is not sufficient to consider memoryless policies.

Model checking queries for POMDPs often rely on the analysis of the *belief MDP*. Indeed, we may analyse this generally infinite model. Let us first recap a formal definition of the belief MDP, using the presentation from [11]. In the following, let $\mathbf{P}(s, \alpha, z) := \sum_{s' \in S} [\text{obs}(s') = z] \cdot \mathbf{P}(s, \alpha, s')$ denote the probability[1] to move to (a state with) observation z from state s using action α. Then, $\mathbf{P}(\mathfrak{b}, \alpha, z) := \sum_{s \in S} \mathfrak{b}(s) \cdot \mathbf{P}(s, \alpha, z)$ is the probability to observe z after taking α in $\mathfrak{b}$. We define the *belief obtained by taking α from $\mathfrak{b}$, conditioned on observing z*:

$$\mathit{update}(\mathfrak{b}|\alpha, z)(s') := \frac{[\text{obs}(s') = z] \cdot \sum_{s \in S} \mathfrak{b}(s) \cdot \mathbf{P}(s, \alpha, s')}{\mathbf{P}(\mathfrak{b}, \alpha, z)}. \tag{1}$$

Definition 5 (Belief MDP). *The* belief MDP *of POMDP $\mathcal{P} = \langle \mathcal{M}, \Omega, \text{obs} \rangle$ where $\mathcal{M} = \langle S, \text{Act}, \mu_{\text{init}}, \mathbf{P} \rangle$ is the MDP $\mathit{BelMDP}(\mathcal{P}) := \langle \mathcal{B}, \text{Act}, \mathbf{P}_{\mathcal{B}}, \mu_{\text{init}} \rangle$ with $\mathcal{B} = \mathit{Distr}(S)$, and transition function $\mathbf{P}_{\mathcal{B}}$ given by*

$$\mathbf{P}_{\mathcal{B}}(\mathfrak{b}, \alpha, \mathfrak{b}') := \begin{cases} \mathbf{P}(\mathfrak{b}, \alpha, \text{obs}(\mathfrak{b}')) & \text{if } \mathfrak{b}' = \mathit{update}(\mathfrak{b}|\alpha, \text{obs}(\mathfrak{b}')), \\ 0 & \text{otherwise.} \end{cases}$$

Due to (1) and the unique initial observation, we may restrict the beliefs to $\mathcal{B} = \bigcup_{z \in \Omega} \mathit{Distr}(\{s \mid \text{obs}(s) = z\})$, that is, each belief state has a unique associated observation. We can lift specifications to belief MDPs: *Avoid-beliefs* are the set of beliefs $\mathfrak{b}$ such that $\mathit{supp}(\mathfrak{b}) \cap \mathit{AVOID} \neq \emptyset$, and *reach-beliefs* are the set of beliefs $\mathfrak{b}$ such that $\mathit{supp}(\mathfrak{b}) \subseteq \mathit{REACH}$.

Towards obtaining a finite abstraction, the main algorithmic idea is the following. For the qualitative reach-avoid specifications we consider, the belief probabilities are irrelevant—*only the belief support is important* [47].

Lemma 1. *For winning belief $\mathfrak{b}$, belief $\mathfrak{b}'$ with $\mathit{supp}(\mathfrak{b}) = \mathit{supp}(\mathfrak{b}')$ is winning.*

Consequently, we can abstract the belief MDP into a finite belief support MDP.

Definition 6 (Belief-Support MDP). *For a POMDP $\mathcal{P} = \langle \mathcal{M}, \Omega, \text{obs} \rangle$ with $\mathcal{M} = \langle S, \text{Act}, \mu_{\text{init}}, \mathbf{P} \rangle$, the finite state space of a* belief-support MDP *$\mathcal{P}_B$ is $B = \{ b \subseteq S \mid \forall s, s' \in b \colon \text{obs}(s) = \text{obs}(s') \}$ where each state is the support of a belief state. Action α in state b leads (with an irrelevant positive probability $p > 0$) to a state b', if*

$$b' \in \Big\{ \bigcup_{s \in b} \text{post}_s(\alpha) \cap \{s \mid \text{obs}(s) = z\} \mid z \in \Omega \Big\}.$$

[1] We use Iverson brackets: $[x] = 1$ if x holds and 0 otherwise.

Thus, transitions between states within b and b' are mimicked in the POMDP. Equivalently, the following clarifies the belief-support MDP as an abstraction of the belief MDP: there are transitions with action α between b and b', if there exists beliefs $\mathfrak{b}, \mathfrak{b}'$ with $\mathit{supp}(\mathfrak{b}) = b$ and $\mathit{supp}(\mathfrak{b}') = b'$, such that $\mathfrak{b}' \in \mathrm{post}_{\mathfrak{b}}(\alpha)$. We lift the specification as before:

Definition 7 (Lifted specification). *For $\varphi = \langle \mathit{AVOID}, \mathit{REACH} \rangle$, we define $\varphi_B = \langle \mathit{AVOID}_B, \mathit{REACH}_B \rangle$ with $\mathit{AVOID}_B = \{ b \mid b \cap \mathit{AVOID} \neq \emptyset \}$, and $\mathit{REACH}_B = \{ b \mid b \subseteq \mathit{REACH} \}$.*

We obtain the following lemma, which follows from the fact that almost-sure reachability is a graph property[2].

Lemma 2. *If belief $\mathfrak{b}$ is winning in the POMDP $\mathcal{P}$ for φ, then the support $\mathit{supp}(\mathfrak{b})$ is winning in the belief-support MDP $\mathcal{P}_B$ for φ_B.*

Lemma 2 yields an equivalent reformulation of Problem 1 for belief supports:

Problem 1 (equivalent): Given a POMDP $\mathcal{P}$, belief $\mathfrak{b}$, and specification φ, decide whether $\mathit{supp}(\mathfrak{b})$ is winning for φ_B in the belief-support MDP $\mathcal{P}_B$.

3 Winning Regions

This section provides the observations on winning regions, a key concept for this paper. An important consequence of Lemma 2 and the reformulation of Problem 1 to the belief-support MDP is that the initial distribution of the POMDP is no longer relevant. Winning policies for individual beliefs may be composed to a policy that is winning for all of these beliefs, using the individual action choices.

Lemma 3. *If the policies σ and σ' are winning for the belief supports b and b', respectively, then there exists a policy σ'' that is winning for both b and b'.*

While this statement may seem trivial on the MDP (or equivalently on beliefs), we notice that it does not hold for POMDP states. As a natural consequence, we are able to consider winning beliefs without referring to a specific policy.

Definition 8 (Winning region). *Let σ be a policy. A set $W^{\sigma}_{\varphi} \subseteq B$ of belief supports is a* winning region for φ and σ, *if σ is winning from each $b \in W^{\sigma}_{\varphi}$. A set $W_{\varphi} \subseteq B$ is a winning region for φ, if every $b \in W_{\varphi}$ is winning. The region containing all winning beliefs is the* maximal winning region[3].

[2] Although the probabilities are not relevant to compute almost-sure reachability, it is important to notice that almost-sure reachability is different from sure-reachability [5]: For almost-sure reachability, there can be an infinite path that never reaches the target, as long as the probability mass over all those paths is 0. Almost-sure reachability can, however, be expressed as sure-reachability in a particular game-setting [47].

[3] In some literature, *winning region* always refers to a *maximal* winning region.

Observe that the maximal winning region in MDPs exists for qualitative reachability, but not for quantitative reachability, which we do not consider here.

Problem 2: Given a POMDP $\mathcal{P}$ and a specification φ, find the maximal winning region W_φ.

Using this definition of winning regions, we are able to reformulate **Problem 1** by asking whether the support of some belief $\mathfrak{b}$ is in the winning region.

Part of **Problem 1** was to compute a winning policy. Below, we study the connection between the winning region and winning policies. We are interested in subsets of the maximal winning region that exhibit two properties:

Definition 9 (Deadlock-free). *A set W of belief-supports $W \subseteq B$ is* deadlock-free*, if for every $b \in W$, an action $\alpha \in \mathrm{EnAct}(b)$ exists such that $\mathrm{post}_b(\alpha) \subseteq W$.*

Definition 10 (Productive). *A set of belief supports $W \subseteq B$ is* productive *(towards a set $REACH_B$), if from every $b \in W$, there exists a (finite) path $\pi = b_0\alpha_1 b_1 \dots b_n$ from b_0 to $b_n \in REACH_B$ with $b_i \in W$ and $\mathrm{post}_{b_i}(\alpha) \subseteq W$ for all $1 \leq i \leq n$.*

Every productive region is deadlock-free, as *REACH*-states are absorbing. The maximal winning region is productive towards $REACH_B$ (and thus deadlock-free) by definition. Intuitively, while a deadlock-free region ensures that one never has to leave the region, any productive winning region ensures that from every belief support within this region there is a policy to stay in the winning region and that can almost-surely reach a *REACH*-state. In particular, to find a winning policy (Challenge 1) or for the purpose of safe exploration (Challenge 2), it is sufficient to find a productive subset of the maximal winning region. We detail on this insight in Sect. 6.

Problem 3: Given a POMDP $\mathcal{P}$ and a specification φ, find a (large) productive winning region W_φ.

To allow a compact representation of winning regions, we exploit that for any belief support $b' \subseteq b$ it holds that $\mathrm{post}_{b'}(\alpha) \subseteq \mathrm{post}_b(\alpha)$ for all actions $\alpha \in \mathrm{Act}$, that is, the successors of b' are contained in the successors of b.

Lemma 4. *For winning belief support b, $b' \subseteq b$ is winning.*

4 Iterative SAT-Based Computation of Winning Regions

We devise an approach for iteratively computing an increasing sequence of productive winning regions. The approach delivers a compact symbolic encoding of winning regions: For a belief (or belief-support) state from a given winning region, we can efficiently decide whether the outcome of an action emanating from the state stays within the winning region.

Key ingredient is the computation of so-called memoryless winning policies. We start this section by briefly recapping how to compute such policies directly

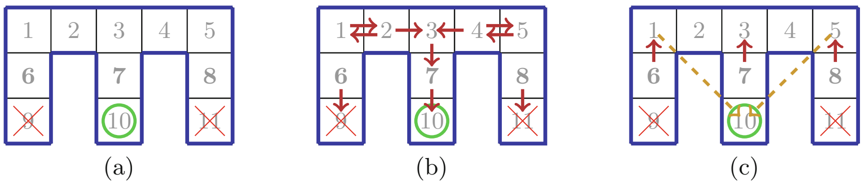


Fig. 1. Cheese-Maze example to explain memoryless policies and shortcuts

on the POMDP, before we build an efficient incremental approach on top of this base method. In particular, we first present a naive iterative algorithm based on the notion of *shortcuts*, then describe how to implicitly add shortcuts within the encoding, and then finally combine the ideas to an efficient algorithm.

4.1 One-Shot Approach to Find Small Policies from a Single Belief

We aim to solve **Problem 1** and determine a winning policy. The number of policies is exponential in the actions and the (exponentially many) belief support states. Searching among doubly exponentially many possibilities is intractable in general. However, Chatterjee et al. [15] observe that often much simpler winning policies exist and provides a *one-shot approach* to find them. The essential idea is to search only for memoryless observation-based policies $\sigma\colon \Omega \rightarrow \mathit{Distr}(\mathrm{Act})$ that are winning for the (initial) belief support b.

Example 1. Consider the small Cheese-POMDP [35] in Fig. 1(a). States are cells, actions are moving in the cardinal directions (if possible), and observations are the directions with adjacent cells, e.g., the boldface states 6, 7, 8 share an observation. We set $\mathit{REACH} = \{10\}$ and $\mathit{AVOID} = \{9, 11\}$. From belief support $b = \{6, 8\}$ there is no memoryless winning policy—In states $\{6, 8\}$ we have to go north, which prevents us from going south in state 7. However, we can find a memoryless winning policy for $\{1, 5\}$, see Fig. 1(b).

This problem is NP-complete, and it is thus natural to encode the problem as a satisfiability query in propositional logic. We mildly adapt the original encoding of winning policies [15]. We introduce three sets of Boolean variables: $A_{z,\alpha}$, C_s and $P_{s,j}$. If a policy takes action $\alpha \in \mathrm{Act}$ with positive probability upon observation $z \in \Omega$, then and only then, $A_{z,\alpha}$ is `true`. If under this policy a state $s \in S$ is reached from some initial belief support b_ι with positive probability, then and only then, C_s is `true`. We define a maximal rank k to ensure the productivity. For each state s and rank $0 \leq j \leq k$, variable $P_{s,j}$ indicates rank j for s, that is, a path from s leads to $s' \in \mathit{REACH}$ within j steps.[4] A winning policy is then obtained by finding a satisfiable solution (via a SAT solver) to the conjunction $\Psi_{\mathcal{P}}^{\varphi}(b_\iota, k)$ of the constraints (2a)–(5), where $S_? = S \setminus \big(\mathit{AVOID} \cup \mathit{REACH}\big)$.

[4] Notice that a state s can have multiple 'ranks' in this encoding. Its rank is the smallest j such that $P_{s,j}$ is `true`.

$$\bigwedge_{s \in b_\iota} C_s \qquad (2a) \qquad\qquad \bigwedge_{z \in \Omega} \Big(\bigvee_{\alpha \in \mathrm{EnAct}(z)} A_{z,\alpha} \Big) \qquad (2b)$$

The initial belief support is clearly reachable (2a). The conjunction in (2b) ensures that in every observation, at least one action is taken.

$$\bigwedge_{s \in \mathit{AVOID}} \neg C_s \quad \wedge \quad \bigwedge_{\substack{s \in S \\ \alpha \in \mathrm{EnAct}(s)}} \Big(C_s \wedge A_{\mathrm{obs}(s),\alpha} \rightarrow \bigwedge_{s' \in \mathrm{post}_s(\alpha)} C_{s'} \Big) \tag{3}$$

The conjunction (3) ensures that for any model for these formulas, the set of states $\{s \in S \mid C_s = \texttt{true}\}$ is reachable, does not overlap with *AVOID*, and is transitively closed under reachability (for the policy described by $A_{z,\alpha}$).

$$\bigwedge_{s \in S_?} C_s \rightarrow P_{s,k} \tag{4}$$

$$\bigwedge_{s \notin \mathit{REACH}} \neg P_{s,0} \quad \wedge \quad \bigwedge_{\substack{s \in S_? \\ 1 \le j \le k}} P_{s,j} \leftrightarrow \Big(\bigvee_{\alpha \in \mathrm{EnAct}(s)} \big(A_{\mathrm{obs}(s),\alpha} \wedge \big(\bigvee_{s' \in \mathrm{post}_s(\alpha)} P_{s',j-1} \big) \big) \Big) \tag{5}$$

Conjunction (4) states that any state that is reached almost-surely reaches a state in *REACH*, i.e., that there is a path (of length at most) k to the target. Conjunctions (5) describe a ranking function that ensures the existence of this path. Only states in *REACH* have rank zero, and a state with positive probability to reach a state with rank $j-1$ within a step has rank at most j.

By [15, Thm. 2], it holds that the conjunction $\Psi_{\mathcal{P}}^{\varphi}(b_\iota, k)$ of the constraints (2a)–(5) is satisfiable, if there is a memoryless observation-based policy such that φ is satisfied. If $k = |S|$, then the reverse direction also holds. If $k < |S|$, we may miss states with a higher rank. Large values for k are practically intractable [15], as the encoding grows significantly with k. Pandey and Rintanen [41] propose extending SAT-solvers with a dedicated handling of ranking constraints.

In order to apply this to small-memory policies, one can unfold $\log(m)$ bits of memory of such a policy into an m times larger POMDP [15,33], and then search for a memoryless policy in this larger POMDP. Chatterjee et al. [15] include a slight variation to this unfolding, allowing smaller-than-memoryless policies by enforcing the same action over various observations.

4.2 Iterative Shortcuts

We exploit the one-shot approach to create a naive iterative algorithm that constructs a productive winning region. The iterative algorithm avoids the following restrictions of the one-shot approach. (1) In order to increase the likelihood of finding winning policies, we do not restrict ourselves to small-memory policies, and (2) we do not have to fix a maximal rank k. These modifications allow us to find more winning policies, without guessing hyper-parameters. As we do not need to fix the belief-state, those parts of the winning region that are easy to find for the solver are encountered first.

The One-Shot Approach on Winning Regions. To understand the naive iterative algorithm, it is helpful to consider the previous encoding in the light of **Problem 3**, i.e., finding productive winning regions. Consider first the interpretation of the variables. Indeed, observe that we have found *the same* winning policy for all states s where C_s is `true`. Consequentially, any belief support $b_z = \{s \mid C_s\ \texttt{true} \wedge \text{obs}(s) = z\}$ is winning.

Lemma 5. *If σ is winning for b and b', then σ is also winning for $b \cup b'$.*

This lemma is somewhat dual to Lemma 4, but requires a fixed policy. The constraints (3) and ensure that a winning-region is deadlock-free. The constraints (4) and (5) ensure productivity of the winning region.

Adding Shortcuts Explicitly. The key idea is that we iteratively add *short-cuts* in the POMDP that represent known winning policies. We find a winning policy σ for some belief states in the first iteration, and then add a fresh action α_σ to all (original) POMDP states: This action leads – with probability one – to a *REACH* state, if the state is in the wining belief-support under policy σ. Otherwise, the action leads to an *AVOID* state.

Definition 11. *For POMDP $\mathcal{P} = \langle \mathcal{M}, \Omega, \text{obs}\rangle$ where $\mathcal{M} = \langle S, \text{Act}, \mu_{\text{init}}, \mathbf{P}\rangle$ and a policy σ with associated winning region W^σ_φ, and assuming w.l.o.g., $\top \in$ REACH and $\bot \in$ AVOID, we define the* shortcut POMDP $\mathcal{P}\{\sigma\} = \langle \mathcal{M}', \Omega, \text{obs}\rangle$ *with* $\mathcal{M}' = \langle S, \text{Act}', \mu_{\text{init}}, \mathbf{P}'\rangle$, $\text{Act}' = \text{Act} \cup \{\alpha_\sigma\}$, $\mathbf{P}'(s, \alpha) = \mathbf{P}(s, \alpha)$ *for all* $s \in S$ *and* $\alpha \in \text{Act}$, *and* $\mathbf{P}'(s, \alpha_\sigma) = \{\top \mapsto [\{s\} \in W^\sigma_\varphi], \bot \mapsto [\{s\} \notin W^\sigma_\varphi]\}$.

Lemma 6. *For a POMDP $\mathcal{P}$ and policy σ, the (maximal) winning regions for $\mathcal{P}\{\sigma\}$ and $\mathcal{P}$ coincide.*

First, adding more actions will not change a winning belief-support to be not winning. Furthermore, by construction, taking the novel action will only lead to a winning belief-support whenever following σ from that point onwards would be a winning policy. The *key* benefit is that adding shortcuts may extend the set of belief-support states that win via a memoryless policy. This observation also gives rise to the following extension to the one-shot approach.

Example 2. We continue with Example 1. If we add shortcuts, we can now find a memoryless winning policy for $b = \{6, 8\}$, depicted in Fig. 1(c).

Iterative Shortcuts to Extend a Winning Region. The idea is now to run the one-shot approach, extract the winning region, add the shortcuts to the POMDP, and rerun the one-shot approach. To make the one-shot approach applicable in this setting, it only needs one change: Rather than fixing an initial belief-support, we ask for an arbitrary new belief-support to be added to the states that we have previously covered. We use a data structure Win such that $\mathsf{Win}(z)$ encodes all winning belief supports with observation z. Internally, the data structure stores maximal winning belief supports (w.r.t. set inclusion, see also Lemma 4) as bit-vectors. By construction, for every $b \in \mathsf{Win}(z)$, a winning region exists, i.e., conceptually, there is a shortcut-action leading to *REACH*.

Algorithm 1 Naive construction of winning regions

```
**Input**: POMDP $\mathcal{P}$, reach-avoid specification $\varphi$
**Output**: Winning region encoded in $\mathsf{Win}$
$\mathsf{Win}(z) \leftarrow \{s \in \mathit{REACH} \mid \mathrm{obs}(s) = z\}$ for all $z \in \Omega$
$\Phi \leftarrow \mathit{Encode}(\mathcal{P}, \varphi, \mathsf{Win})$          ▷ Create encoding (2b),(3),(6),(7).
**while** $\exists \eta$ s.t. $\eta \models \Phi$ **do**          ▷ Call an SMT solver
    $\mathsf{Win}(z) \leftarrow \mathsf{Win}(z) \cup \{b \mid s \in b \text{ iff } \eta(C_s)\}$ for all $z \in \Omega$
    $\mathcal{P} \leftarrow \mathcal{P}\{\sigma_\eta\}$          ▷ Extend POMDP with Def. 11
                                                                   ▷ with $\sigma_\eta$ policy encoded by $\eta$.
    $\Phi \leftarrow \mathit{Encode}(\mathcal{P}, \varphi, \mathsf{Win})$
```

We extend the encoding (in partial preparation of the next subsection) and add a variable $U_z \in b$ that is `true` if the policy is winning in a belief support that is not yet in $\mathsf{Win}(z)$. We replace (2a) with:

$$\bigvee_{z\in\Omega} U_z \quad \wedge \bigwedge_{\substack{z\in\Omega\\ \mathsf{Win}(z)=\emptyset}} \Big(U_z \leftrightarrow \bigvee_{\substack{s\in S\\ \mathrm{obs}(s)=z}} C_s\Big) \quad \wedge \bigwedge_{\substack{z\in\Omega\\ \mathsf{Win}(z)\neq\emptyset}} \Big(U_z \leftrightarrow \bigwedge_{X\in\mathsf{Win}(z)} \bigvee_{\substack{s\in S\setminus X\\ \mathrm{obs}(s)=z}} C_s\Big) \tag{6}$$

For an observation z for which we have not found a winning belief support yet, finding a policy from any state s with obs(s) updates the winning region. Otherwise, it means finding a winning policy for a belief support that is not subsumed by a previous one (6).

Real-Valued Ranking. To avoid setting a maximal path length, we use unbounded (real) variables R_s rather than Boolean variables for the ranking [57]. This relaxation avoids the growth of the encoding and admits arbitrarily large ranks with a fixed-size encoding into difference logic. This logic is an extension to propositional logic that can be checked using an SMT solver [6].

$$\bigwedge_{s\in S_?} C_s \rightarrow \Big(\bigvee_{\alpha\in\mathrm{EnAct}(s)} \big(A_{\mathrm{obs}(s),\alpha} \wedge \big(\bigvee_{s'\in\mathrm{post}_s(\alpha)} R_s > R_{s'}\big)\big)\Big) \tag{7}$$

We replace (4) and (5): A state must have a successor state with a lower rank – as before, but with real-valued ranks (7).

Algorithm. Together, the algorithm is given in Algorithm 1. We initialize the winning region based on the specification, then encode the POMDP using the (modified) one-shot encoding. As long as the SMT solver finds policies that are winning for a new belief-support, we add those belief supports to the winning region. In each iteration, Win contains a winning region. Once we find no more policies that extend the winning region on the extended POMDP, we terminate.

The algorithm always terminates because the set of winning regions is finite, but in general does not solve **Problem 2**. Formally, the maximal winning region is a greatest fixpoint [5] and we iterate from below, i.e., the fixpoint that we find will be the smallest fixpoint (of the operation that we implement). However, iterating from above requires to reason that none of the doubly-exponentially many policies is winning for a particular belief support state; whereas our approach profits from finding simple strategies early on. Unfolding of memory as discussed earlier also makes this algorithm complete, yet, suffers from the same blow-up. A main advantage is that the algorithm often avoids the need for unfolding when searching for a winning policy or large winning regions.

Next, we address two weaknesses: First, the algorithm currently creates a new encoding in every iteration, yielding significant overhead. Second, the algorithm in many settings requires adding a bit of memory to realize behavior where in a particular observation, we *first* want to execute an action α and *then* follow a shortcut from the state (with the same observation) reached from there. We adapt the encoding to explicitly allow for these (non-memoryless) policies.

4.3 Incremental Encoding of Winning Regions

In this section, instead of naively adjusting the POMDP, we realize the idea of adding shortcuts directly on the encoding. This encoding is the essential step towards an efficacious approach for solving **Problem 3**. We find winning states based on a previous solution, and instead of adding actions, we allow the solver to decide following individual policies from each observation. In Sect. 4.4, we embed this encoding into an improved algorithm.

Our encoding represents an observation-based policy that can decide to take a shortcut, which means that it follows a previously computed winning policy from there (implicitly using Lemma 3). In addition to $A_{z,\alpha}$, C_s and R_s from the previous encoding, we use the following variables: The policy takes shortcuts in states s where D_s is `true`. For each observation, we must take the same shortcut, referred to by a positive integer-valued index I_z. More precisely, I_z refers to a shortcut from a previously computed (fragment of a) winning region stored in $\mathsf{Win}(z)_{I_z}$. The policy may decide to *switch*, that is, to follow a shortcut *after* taking an action starting in a state with observation z. If F_z is `true`, the policy takes some action from z-states and from the next state, we take a shortcut. The encoding thus implicitly represents policies that are not memoryless but rather allow for a particular type of memory.
The conjunction of (6) and (8)–(13) yields the encoding $\Phi_{\mathcal{P}}^{\varphi}(\mathsf{Win})$:

$$\bigwedge_{z\in\Omega}\Big(\bigvee_{\alpha\in\mathrm{EnAct}(z)} A_{z,\alpha}\Big) \quad\wedge\quad \bigwedge_{s\in AVOID} \neg C_s \wedge \neg D_s \tag{8}$$

$$\bigwedge_{\substack{s\in S\\ \alpha\in\mathrm{EnAct}(s)}} \Big(C_s \wedge A_{\mathrm{obs}(s),\alpha} \wedge \neg F_{\mathrm{obs}(s)} \quad\rightarrow \bigwedge_{s'\in\mathrm{post}_s(\alpha)} C_{s'}\Big) \tag{9}$$

$$\bigwedge_{\substack{s\in S\\ \alpha\in\mathrm{EnAct}(s)}} \Big(C_s \wedge A_{\mathrm{obs}(s),\alpha} \wedge F_{\mathrm{obs}(s)} \quad\rightarrow \bigwedge_{s'\in\mathrm{post}_s(\alpha)} D_{s'}\Big) \tag{10}$$

Similar to (2b), (3), we select at least one action and *AVOID*-states should not be reached (8). States reached are closed under the transitive closure, however,

Algorithm 2 Naive construction of winning regions with incremental encoding

```
Input: POMDP $\mathcal{P}$, reach-avoid specification $\varphi$
Output: Winning region encoded in $\mathsf{Win}$
$\mathsf{Win}(z) \leftarrow \{s \in \mathit{REACH} \mid \mathrm{obs}(s) = z\}$ for all $z \in \Omega$
$\Phi \leftarrow \mathit{Encode}(\mathcal{P}, \varphi, \mathsf{Win})$                ▷ Create encoding (6),(8)–(13).
while $\exists \eta$ s.t. $\eta \models \Phi$ do                                  ▷ Call an SMT solver
    $\mathsf{Win}(z) \leftarrow \mathsf{Win}(z) \cup \{b \mid s \in b \text{ iff } \eta(C_s)\}$ for all $z \in \Omega$
    $\Phi \leftarrow \mathit{Encode}(\mathcal{P}, \varphi, \mathsf{Win})$
```

only if we do not switch to taking a shortcut (9). Furthermore, we mark the states reached after switching (10) and need to select a shortcut for these states.

$$\bigwedge_{s \in S} \big(D_s \;\rightarrow\; I_{\mathrm{obs}(s)} > 0\big) \quad\wedge\quad \bigwedge_{z \in \Omega} I_z \leq |\mathsf{Win}(z)| \tag{11}$$

$$\bigwedge_{\substack{z \in \Omega \\ 0 < i \leq |\mathsf{Win}(z)|}} \quad \bigwedge_{\substack{s \in S \setminus \mathsf{Win}(z)_i \\ \mathrm{obs}(s) = z}} D_s \;\rightarrow\; I_z \neq i \tag{12}$$

If we reach a state s after switching, then we must pick a shortcut. We can only pick an index that reflects a found winning region (11). If we pick this shortcut reflecting a winning region (fragment) for observation z, then we are winning from the states in $\mathsf{Win}(z)_i$, but not from any other state s with that observation. Thus, for $s \notin \mathsf{Win}(z)_i$, if we are going to follow any shortcut (that is, D_s holds), we should not pick this particular shortcut encoded by I_z (because it will lead to an *AVOID*-state). In terms of the policy: Taking this previously computed policy from state s is not (known to) lead us to a *REACH*-state (12). Finally, we update the ranking to account for shortcuts.

$$\bigwedge_{s \in S_?} C_s \rightarrow \Big(\bigvee_{\alpha \in \mathrm{EnAct}(s)} \big(A_{\mathrm{obs}(s),\alpha} \wedge \big(\bigvee_{s' \in \mathrm{post}_s(\alpha)} R_s > R_{s'} \big)\big) \vee F_{\mathrm{obs}(s)} \Big) \tag{13}$$

We make a slight adaption to (7): Either we have a successor state with a lower rank (as before) or we follow a shortcut—which either leads to the target or to violating the specification (13). We formalize the correctness of the encoding:

Lemma 7. *If $\eta \models \Phi_{\mathcal{P}}^{\varphi}(\mathsf{Win})$, then for every observation z, the belief support $b_z = \{s \mid \eta(C_s) = \mathit{true}, \mathrm{obs}(s) = z\}$ is winning.*

Algorithm 2 is a straightforward adaption of Algorithm 1 that avoids adding shortcuts explicitly (and uses the updated encoding). As before, the algorithm terminates and solves **Problem 3**. We conclude:

Theorem 1. *In any iteration, Algorithm 2 computes a productive winning region.*

4.4 An Incremental Algorithm

We adapt the algorithm sketched above to exploit the incrementality of modern SMT solvers. Furthermore, we aim to reduce the invocations of the solver by finding some extensions to the winning region via a graph-based algorithm.

Algorithm 3 Incremental construction of winning regions

```
Input: POMDP $\mathcal{P}$, reach-avoid specification $\varphi$
Output: Winning region encoded in Win
Win$(z) \leftarrow \{s \in \mathit{REACH} \mid \mathrm{obs}(s) = z\}$ for all $z \in \Omega$
Win $\leftarrow$ $\mathit{GraphPreprocessing}$(Win)
$\Phi_{\text{fix}} \leftarrow \mathit{Encode}_{\text{fix}}(\mathcal{P}, \varphi, \mathsf{Win})$        ▷ Create encoding (8)–(13)
$\Phi_{\text{inc}} \leftarrow \mathit{Encode}_{\text{inc}}(\mathcal{P}, \varphi, \mathsf{Win})$        ▷ Encode (6)
while $\exists \eta$ s.t. $\eta \models \Phi_{\text{fix}} \wedge \Phi_{\text{inc}}$ do        ▷ Call an SMT solver, fix $\eta$
    do        ▷ Extend policy
        $\Phi_\eta \leftarrow \bigwedge\{A_{z,\alpha} \mid \eta(U_z) \wedge \eta(A_{z,\alpha})\}$        ▷ Part. fix policy
    while $\exists \eta$ s.t. $\eta \models \Phi_{\text{fix}} \wedge \Phi_{\text{var}} \wedge \Phi_\eta$        ▷ Call SMT, fix $\eta$
    Win$(z) \leftarrow$ Win$(z) \cup \{B \mid s \in B$ iff $\eta(C_s)\}$ for all $z \in \Omega$
    Win $\leftarrow$ $\mathit{GraphPreprocessing}$(Win)
    $\Phi_{\text{fix}} \leftarrow \Phi_{\text{fix}} \wedge \mathit{Encode}_{(11)(12)}(\mathcal{P}, \varphi, \mathsf{Win})$        ▷ Update: (11),(12)
    $\Phi_{\text{inc}} \leftarrow \mathit{Encode}_{\text{inc}}(\mathcal{P}, \varphi, \mathsf{Win})$        ▷ Encode (6)
```

Graph-Based Preprocessing. To reduce the number of SMT invocations, we employ polynomial-time graph-based heuristics. The first step is to use (fully observable) MDP model checking on the POMDP as follows: find all states that under each (not necessarily observation-based) policy reach an *AVOID*-state with positive probability, and make them absorbing. Then, we find all states that under *each* policy reach a *REACH*-state almost-surely. Then, we iteratively search for *winning observations* and use them to extend the *REACH*-states. An observation z is winning, if the belief-support $\{s \mid \mathrm{obs}(s) = z\}$ is winning. We start with a previously determined winning region W. We iteratively update W by adding states $b_z = \{s \mid \mathrm{obs}(s) = z\}$ for some observation z, if there is an action α such that from every $s \in b_z$, it holds $\mathrm{post}_s(\alpha) \subseteq W$. The iterative updates are interleaved with MDP model checking on the POMDP as described above until we find a fixpoint.

Optimized Algorithm. We improve Algorithm 2 along four dimensions to obtain Algorithm 3. First, we employ fewer updates of the winning region: We aim to extend the policy as much as possible, i.e., we want the SMT-solver to find more states with the same observation that are winning under the same policy. Therefore, we fix the variables for action choices that yield a new winning policy, and let the SMT solver search whether we can extend the corresponding winning region by finding more states and actions that are compatible with the partial policy. Second, we observe that between (outer) iterations, large parts of the encoding stay intact, and use an incremental approach in which we first push all the constraints from the POMDP onto the stack, then all the constraints from the winning region, and finally a constraint that asks for progress. After we found a new policy, we pop the last constraint from the stack, add new constraints regarding the winning region (notice that the old constraints remain intact), and push new constraints that ask for extending the winning region to the stack. We refresh the encoding periodically to avoid unnecessary cluttering. Third, further constraints (1) make the usage of shortcuts more flexible—we allow taking shortcuts either immediately or after the next action, and (2) enable an even more incremental encoding with some minor technical reformulations. Fourth, we add the graph-preprocessing discussed above during the outer iteration.

5 Symbolic Model Checking for the Belief-Support MDP

In this section, we briefly describe how we encode a given POMDP into a belief-support MDP to employ symbolic, off-the-shelf probabilistic model checking. In particular, we employ symbolic (decision-diagram, DD) representations of the belief-support MDP as we expect this MDP to be huge. Constructing that DD representation effectively is not entirely trivial. Instead, we advocate constructing a (modular) symbolic description of the belief support MDP. Concretely, we automatically generate a model description in the MDP modeling language JANI [13],[5] and then apply off-the-shelf model checking on the JANI description.

Conceptually, we create a belief-support MDP with auxiliary states to allow for a concise encoding.[6] We use this auxiliary state $\hat{b}$ to describe for any transition the conditioning on the observation. Concretely, a single transition $\mathbf{P}(b, \alpha, b')$ in the belief-support MDP is reflected by two transitions $\mathbf{P}(b, \alpha, \hat{b})$ and $\mathbf{P}(\hat{b}, \alpha_\perp, b')$ in our encoding, where $\alpha_\perp$ is a unique dummy action. We encode states using triples $\langle \texttt{belsup}, \texttt{newobs}, \texttt{lact} \rangle$. `belsup` is a bit vector with entries for every state s that we use to encode the belief support. Variables `newobs` and `lact` store an observation and an action and are relevant only for the auxiliary states. Technically, we now encode the first transition from b with the nondeterministic action α to $\hat{b}$. $\mathbf{P}(b, \alpha)$ then yields (with arbitrary positive) probability a new observation that will reflect the observation $\mathrm{obs}(b')$. We store α and $\mathrm{obs}(b')$ in `lact` and `newobs`, respectively. The second step is a single deterministic (dummy) action updating `belsup` while taking into account `newobs`. The step also resets `lact` and `newobs`.

The encoding of the transitions as follows: For the first step, we create nondeterministic choices for each action α and observation z. We guard these choices with z meaning that the edge is only applicable to states having observation z, i.e., the guard is $\bigvee_{s \in S, \mathrm{obs}(s)=z} \texttt{belsup}(s)$. With these guarded edges, we define the destinations: With an arbitrary[7] probability p, we go to an observation z_1 *if* there is at least one state in $s \in \texttt{belsup}$ which has a successor state $s' \in \mathrm{post}_s(\alpha)$ with $\mathrm{obs}(s') = z_1$.

[5] The description here works on a network of synchronized state machines as is also common in the PRISM language.

[6] The usage of message passing or *indexed assignments* in JANI would circumvent the need for intermediate states, but is to the best of our knowledge not supported by decision-diagram based model checkers.

[7] We leave this a parametric probability in model building to reduce the number of different probabilities, as this is beneficial for the size of the decision diagram that Storm constructs – it will only have leafs 0, p, 1. Technically, such MDPs are not necessarily well-defined but we can employ model checking on the graph structure.

The following pseudocode reflects the first step in the transition encoding. The syntax is as follows: **take** an action **if** a Boolean guard is satisfied, then updates are executed with probability **prob**. An example for a guard is an observation z.

$$\textbf{take}\,\alpha\,\textbf{if}\,z\,\textbf{then}\begin{cases}\textbf{prob}\ \left(\bigvee_{\substack{s\in S\\ \mathbf{P}(s,\alpha,z_1)>0}}\texttt{belsup}(s)\ ?\ p : 0\right): & \begin{matrix}\texttt{newobs}\leftarrow z_1\\ \texttt{lact}\leftarrow\alpha\end{matrix}\\ \dots & \dots\\ \textbf{prob}\ \left(\bigvee_{\substack{s\in S\\ \mathbf{P}(s,\alpha,z_n)>0}}\texttt{belsup}(s)\ ?\ p : 0\right): & \begin{matrix}\texttt{newobs}\leftarrow z_n\\ \texttt{lact}\leftarrow\alpha\end{matrix}\end{cases}$$

The second step synchronously updates each state s' in the POMDP independently: The entry $\texttt{belsup}(s')$ is set to $\texttt{true}$ if $\text{obs}(s) = \texttt{newobs}$ and if there is a state s currently $\texttt{true}$ in (the old) $\texttt{belsup}$ with $s' \in \text{post}_s(\texttt{lact})$. The step thus can be captured by the following pseudocode for each s':

$$\textbf{take}\,\alpha_\perp\,\textbf{if}\,\texttt{true}\,\textbf{then}\,\textbf{prob}\,1 : \texttt{belsup}(s') \leftarrow \left(\bigvee\nolimits_s \mathbf{P}(s,\texttt{lact},s') > 0\right) \wedge \text{obs}(s')$$

Finally, whenever the dummy action $\alpha_\perp$ is executed, we also reset the variables `newobs` and `lact`. The resulting encoding thus has transitions in the order of $|S| + |\Omega|^2 \cdot |\max_{z\in\Omega}\text{EnAct}(z)|$.

6 Almost-Sure Reachability Shields in POMDPs

In this section, we define a *shield* for POMDPs – towards the application of safe exploration (Challenge 2) – that blocks actions which would lead an agent out of a winning region. In particular, the shield imposes restrictions on policies to satisfy the reach-avoid specification. Technically, we adapt so-called *permissive* policies [21,31] for a belief-support MDP. To force an agent to stay within a productive winning region W_φ for specification φ, we define a *φ-shield* $\nu\colon b \to 2^{\text{Act}}$ such that for any winning b for φ we have $\nu(b) \subseteq \{\alpha \in \text{Act} \mid \text{post}_b(\alpha) \subseteq W_\varphi\}$, i.e., an action is part of the shield $\nu(b)$ if it exclusively leads to belief support states within the winning region.

A shield ν restricts the set of actions an arbitrary policy may take[8]. We call such restricted policies *admissible*. Specifically, let b_τ be the belief support after observing an observation sequence τ. Then policy σ is ν-admissible if $supp(\sigma(\tau)) \subseteq \nu(b_\tau)$ for every observation-sequence τ. Consequently, a policy is *not* admissible if for some observation sequence τ, the policy selects an action $\alpha \in \text{Act}$ which is not allowed by the shield.

Some admissible policies may choose to stay in the winning region without progressing towards the *REACH* states. Such a policy adheres to the avoid-part of the specification, but violates the reachability part. To enforce *progress*, we

[8] While memory policies based on the belief (support) are sufficient to ensure almost-sure reachability, the goal is to shield other policies that do not necessarily fall in this restricted class.

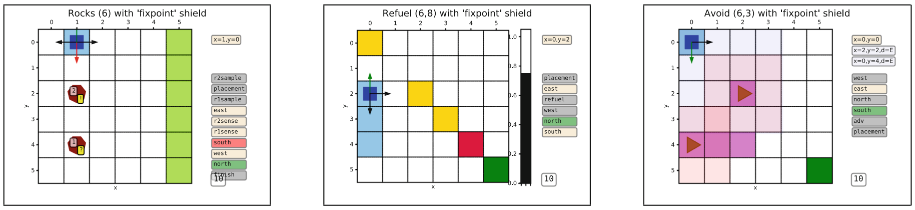


Fig. 2. Video stills from simulating a shielded agent on three different benchmarks.

adapt a notion of *fairness*. A policy is fair if it takes every action infinitely often at any belief support state that appears infinitely often along a trace [5]. For example, a policy that randomizes (arbitrarily) over all actions is fair–we notice that most reinforcement learning policies are therefore fair.

Theorem 2. *For a φ-shield ν and a winning belief support b, any* fair *ν-admissible policy satisfies φ from b.*

We give a proof (sketch) in [32, Appendix]. The main idea is to show that the induced Markov chain of any admissible policy has only bottom SCCs that contain *REACH*-states.

Remark 1. If φ is a safety specification (where $Pr^{\sigma}_{\mathfrak{b}}(AVOID) = 0$ suffices), we can rely on deadlock-free winning regions rather than productive winning regions and drop the fairness assumption.

7 Empirical Evaluation

We investigate the applicability of our incremental approach (Algorithm 3) to **Challenge 1** and **Challenge 2**, and compare with our adaption and implementation of the one-shot approach [15], see Sect. 4.1. We also employ the MDP model-checking approach from Sect. 5. Experiments, videos, source code are archived[9].

Setting. We implemented the one-shot algorithm, our incremental algorithm, and the generation of the JANI description of the belief support MDP into the model checker Storm [19] on top of the SMT solver z3 [38]. To compare with the one-shot algorithm for **Problem 1**, that is, for finding a policy from the initial state, we add a variant of Algorithm 3. Intuitively, any outer iteration starts with an SMT-check to see whether we find a policy covering the initial states. We realize the latter by fixing (temporarily) the C_s-variables. In the first iteration, this configuration and its resulting policy closely resemble the one-shot approach. For the MDP model-checking approach, we use Storm (from the C++ API) with the `dd` engine and default settings.

For the experiments, we use a MacBook Pro MV962LL/A, a single core, no randomization, and use a 6 GB memory limit. The time-out (TO) is 15 min.

[9] http://doi.org/10.5281/zenodo.4784940 or on http://github.com/sjunges/shielding-POMDPs.

Baseline. We compare with the one-shot algorithm including the graph-based preprocessing to identify more winning observations. We use two setups: (1) We (manually, a-priori) search for optimal hyper-parameters for each instance. We search for the smallest amount of memory possible, and for the smallest maximal rank k (being a multiplicative of five) that yields a result. Guessing parameters as an "oracle" is time-consuming and unrealistic. We investigate (2) the performance of the one-shot algorithm by fixing the hyper-parameters to two memory-states and $k = 30$. These parameters provide results for most benchmarks.

Benchmarks. Our benchmarks involve agents operating in $N \times N$ grids, inspired by, e.g., [12,15,20,50,51]. See Fig. 2 for video stills of simulating the following benchmarks. *Rocks* is a variant of *rock sample.* The grid contains two rocks which are either valuable or dangerous to collect. To find out with certainty, the rock has to be sampled from an adjacent field. The goal is to collect a valuable rock, bring it to the drop-off zone, and not collect dangerous rocks. *Refuel* concerns a rover that shall travel from one corner to the other, while avoiding an obstacle on the diagonal. Every movement costs energy and the rover may recharge at recharging stations to its full battery capacity E. It receives noisy information about its position and battery level. *Evade* is a scenario where a robot needs to reach a destination and evade a faster agent. The robot has a limited range of vision (R), but may scan the whole grid instead of moving. A certain safe area is only accessible by the robot. *Intercept* is inverse to *Evade* in the sense that the robot aims to meet an agent before it leaves the grid via one of two available exits. On top of the view radius, the agent observes a corridor in the center of the grid. *Avoid* is a related scenario where a robot shall keep distance to patrolling agents that move with uncertain speed, yielding partial information about their position The robot may exploit their predefined routes. *Obstacle* contains static obstacles where the robot needs to reach the exit. Its initial state and movement are uncertain, and it only observes whether the current position is a trap or exit.

Results for Challenge 1. Table 1 details the numerical benchmark results. For each benchmark instance (columns), we report the name and relevant characteristics: the number of states ($|S|$), the number of transitions (#Tr, the edges in the graph described by the POMDP), the number of observations ($|\Omega|$), and the number of belief support states ($|b|$). For the incremental method, we provide the run time (Time, in seconds), the number of outer iterations (#Iter.) in Algorithm 3, and the number of invocations of the SMT solver (#solve), and the approximate size of the winning region ($|W|$). We then report these numbers when searching for a policy that wins from the initial state. For the one-shot method, we provide the time for the optimal parameters (on the next line)–TOs reflect settings in which we did not find any suitable parameters, and the time for the preset parameters (2,30), or N/A if no policy can be found with these parameters. Finally, for (belief-support) MDP model checking, we give only the run times.

The incremental algorithm finds winning policies for the initial state *without guessing parameters* and is often *faster* versus the one-shot approach with an

Table 1. Numerical results towards solving **Problem 1** and **Problem 3**.

		Inst.	*Rocks* (*N*)		*Refuel* (*N*,*E*)		*Evade* (*N*,*R*)		*Avoid* (*N*,*R*)		*Intercept* (*N*,*R*)		*Obstacle* (*N*)	
			4	6	6,8	7,7	6,2	7,2	6,3	7,4	7,1	7,2	6	8
		$\lvert S\rvert$	331	816	270	302	4232	8108	5976	13021	4705	4705	37	65
		#Tr	3484	7292	1301	1545	28866	57570	14373	33949	18049	18049	224	421
		$\lvert\Omega\rvert$	65	74	36	35	2202	4172	3300	8584	2002	2598	4	4
		$\lvert b\rvert$	3.5E**5**	7.7E**25**	5.6E**14**	7.4E**19**	1.1E**8**	4.4E**11**	1.1E**15**	2.9E**17**	6.4E**10**	2.7E**9**	1.1E**9**	2.9E**17**
incremental	fixpoint	Time	19	753	6	3	142	613	167	745	116	86	2	30
		#Iter.	36	284	40	30	4	6	3	4	8	8	68	150
		#solve	1702	13650	1023	528	681	1129	629	1027	1171	976	839	4291
		$\lvert W\rvert$	3.5E**5**	7.7E**25**	1.2E**11**	2.1E**8**	1.0E**8**	4.2E**11**	1.1E**15**	2.9E**17**	9.2E**4**	2.9E**4**	4.1E**7**	3.8E**14**
	initial	Time	17	226	2	2	49	576	10	40	11	2	<1	<1
		#Iter.	29	65	2	4	1	1	1	1	2	1	10	12
		#solve	1215	2652	62	80	1	1	1	1	81	1	114	229
		$\lvert W\rvert$	4.4E**4**	1.8E**13**	8.4E**6**	3.7E**4**	5.0E**7**	1.0E**11**	3.7E**5**	6.9E**10**	6.2E**3**	2.1E**3**	4.1E**5**	4.5E**9**
1-shot	opt	Time	120	TO	2	<1	12	270	22	53	8	1	1	195
		Mem,k	2,10	?	2,15	2,15	1,20	1,30	1,30	1,25	2,10	1,10	6,10	5,50
	fix	Time	TO	TO	11	37	TO	TO	TO	TO	28	18	N/A	N/A
MDP		Time	400	TO	219	MO	TO	TO	TO	TO	TO	TO	6	MO

oracle providing optimal parameters, and significantly faster than the one-shot approach with reasonably fixed parameters. In detail, *Rocks* shows that we can handle large numbers of iterations, solver invocations, and winning regions. The incremental approach scales to larger models, see e.g., *Avoid*. *Refuel* shows a large sensitivity of the one-shot method on the lookahead (going from 15 to 30 increases the runtime), while *Evade* shows sensitivity to memory (from 1 to 2). In contrast, the incremental approach does not rely on user-input, yet delivers comparable performance on *Refuel* or *Avoid*. It suffers slightly on *Evade*, where the one-shot approach has reduced overhead. We furthermore conclude that off-the-shelf MDP model checking is not a fast alternative. Its advantage is the guarantee to find the maximal winning region, however, for our benchmarks, maximal winning regions (empirically) coincide with the results from the incremental fixpoint approach.

Results for Challenge 2. Winning regions obtained from running incrementally to a fixpoint are significantly larger than when running them only until an initial winning policy is found (cf. the table), but requires extra computational effort.

If a *shielded agent* moves randomly through the grid-worlds, the larger winning regions indeed induce more permissiveness, that is, freedom to move for the agent (cf. the videos, Fig. 2). This observation can also be quantified. In Table 2, we compare the two different types of shields. For both, we give average and standard deviation over permissiveness over 250 paths. We choose to approximate permissiveness along a path as the number of cumulative actions allowed by the permissive scheduler along a path, divided by the number of cumulative actions available in the POMDP along that path. As the shield is correct by construction, each run indeed never visits avoid states and eventually reaches the target (albeit after many steps). This statement is not true for the unshielded agents.

Table 2. Quantification of permissiveness using fraction of allowed actions.

Inst.		*Rocks* (N)		*Refuel* (N,E)		*Evade* (N,R)		*Avoid* (N,R)		*Intercept* (N,R)		*Obstacle* (N)	
		4	6	6,8	7,7	6,2	7,2	6,3	7,4	7,1	7,2	6	8
initial	avg	**0.85**	**0.81**	**0.43**	**0.36**	**0.62**	**0.50**	**0.51**	**0.56**	**0.45**	**0.47**	**0.68**	**0.74**
	stdev	0.066	0.070	0.046	0.014	0.046	0.043	0.013	0.019	0.037	0.047	0.040	0.047
fixpoint	avg	**0.88**	**0.89**	**0.77**	**0.73**	**0.86**	**0.87**	**0.78**	**0.80**	**0.78**	**0.84**	**0.73**	**0.73**
	stdev	0.060	0.037	0.037	0.024	0.015	0.016	0.015	0.017	0.078	0.070	0.036	0.059

8 Conclusion

We provided an incremental approach to find POMDP policies that satisfy almost-sure reachability specifications. The superior scalability is demonstrated on a string of benchmarks. Furthermore, this approach allows to shield agents in POMDPs and guarantees that any exploration of an environment satisfies the specification, without needlessly restricting the freedom of the agent. We plan to investigate a tight interaction with state-of-the-art reinforcement learning and quantitative verification of POMDPs. For the latter, we expect that an explicit approach to model checking the belief-support MDP can be feasible.

References

1. Akametalu, A.K., Kaynama, S., Fisac, J.F., Zeilinger, M.N., Gillula, J.H., Tomlin, C.J.: Reachability-based safe learning with Gaussian processes. In: CDC, pp. 1424–1431. IEEE (2014)
2. Alshiekh, M., Bloem, R., Ehlers, R., Könighofer, B., Niekum, S., Topcu, U.: Safe reinforcement learning via shielding. In: AAAI. AAAI Press (2018)
3. Amato, C., Bernstein, D.S., Zilberstein, S.: Optimizing fixed-size stochastic controllers for POMDPs and decentralized POMDPs. Auton. Agents Multi Agent Syst. **21**(3), 293–320 (2010). https://doi.org/10.1007/s10458-009-9103-z
4. Baier, C., Größer, M., Bertrand, N.: Probabilistic ω-automata. J. ACM **59**(1), 1:1–1:52 (2012)
5. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press, Cambridge (2008)
6. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Satisfiability, pp. 825–885. IOS Press (2009)
7. Bertoli, P., Cimatti, A., Pistore, M.: Towards strong cyclic planning under partial observability. In: ICAPS, pp. 354–357. AAAI (2006)
8. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability. IOS Press (2009)
9. Bloem, R., Jensen, P.G., Könighofer, B., Larsen, K.G., Lorber, F., Palmisano, A.: It's time to play safe: Shield synthesis for timed systems. CoRR abs/2006.16688 (2020)
10. Bloem, R., Könighofer, B., Könighofer, R., Wang, C.: Shield synthesis: runtime enforcement for reactive systems. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 533–548. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_51

11. Bork, A., Junges, S., Katoen, J.-P., Quatmann, T.: Verification of indefinite-horizon POMDPs. In: Hung, D.V., Sokolsky, O. (eds.) ATVA 2020. LNCS, vol. 12302, pp. 288–304. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-59152-6_16
12. Brockman, G., et al.: Open AI Gym. CoRR abs/1606.01540 (2016)
13. Budde, C.E., Dehnert, C., Hahn, E.M., Hartmanns, A., Junges, S., Turrini, A.: JANI: quantitative model and tool interaction. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10206, pp. 151–168. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54580-5_9
14. Burns, B., Brock, O.: Sampling-based motion planning with sensing uncertainty. In: ICRA, pp. 3313–3318. IEEE (2007)
15. Chatterjee, K., Chmelik, M., Davies, J.: A symbolic SAT-based algorithm for almost-sure reachability with small strategies in POMDPs. In: AAAI, pp. 3225–3232. AAAI Press (2016)
16. Chatterjee, K., Chmelik, M., Gupta, R., Kanodia, A.: Qualitative analysis of POMDPs with temporal logic specifications for robotics applications. In: ICRA, pp. 325–330. IEEE (2015)
17. Chatterjee, K., Chmelik, M., Gupta, R., Kanodia, A.: Optimal cost almost-sure reachability in POMDPs. Artif. Intell. **234**, 26–48 (2016)
18. Chatterjee, K., Doyen, L., Henzinger, T.A.: Qualitative analysis of partially-observable Markov decision processes. In: Hliněný, P., Kučera, A. (eds.) MFCS 2010. LNCS, vol. 6281, pp. 258–269. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15155-2_24
19. Dehnert, C., Junges, S., Katoen, J.-P., Volk, M.: A Storm is coming: a modern probabilistic model checker. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 592–600. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_31
20. Dietterich, T.G.: The MAXQ method for hierarchical reinforcement learning. In: ICML, pp. 118–126. Morgan Kaufmann (1998)
21. Dräger, K., Forejt, V., Kwiatkowska, M., Parker, D., Ujma, M.: Permissive controller synthesis for probabilistic systems. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 531–546. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_44
22. Fulton, N., Platzer, A.: Safe reinforcement learning via formal methods: toward safe control through proof and learning. In: AAAI, pp. 6485–6492. AAAI Press (2018)
23. García, J., Fernández, F.: A comprehensive survey on safe reinforcement learning. J. Mach. Learn. Res. **16**, 1437–1480 (2015)
24. Hahn, E.M., Perez, M., Schewe, S., Somenzi, F., Trivedi, A., Wojtczak, D.: Omega-regular objectives in model-free reinforcement learning. In: Vojnar, T., Zhang, L. (eds.) TACAS 2019. LNCS, vol. 11427, pp. 395–412. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17462-0_27
25. Hasanbeig, M., Abate, A., Kroening, D.: Cautious reinforcement learning with logical constraints. In: AAMAS, pp. 483–491. IFAAMAS (2020)
26. Hausknecht, M.J., Stone, P.: Deep recurrent Q-learning for partially observable MDPs. In: AAAI, pp. 29–37. AAAI Press (2015)
27. Hauskrecht, M.: Value-function approximations for partially observable Markov decision processes. J. Artif. Intell. Res. **13**, 33–94 (2000)
28. Horák, K., Bosanský, B., Chatterjee, K.: Goal-HSVI: heuristic search value iteration for goal POMDPs. In: IJCAI, pp. 4764–4770. ijcai.org (2018)
29. Jaakkola, T.S., Singh, S.P., Jordan, M.I.: Reinforcement learning algorithm for partially observable Markov decision problems. In: NIPS, pp. 345–352 (1994)

30. Jansen, N., Könighofer, B., Junges, S., Serban, A., Bloem, R.: Safe reinforcement learning using probabilistic shields (invited paper). In: CONCUR. LIPIcs, vol. 171, pp. 3:1–3:16. Schloss Dagstuhl - LZI (2020)
31. Junges, S., Jansen, N., Dehnert, C., Topcu, U., Katoen, J.-P.: Safety-constrained reinforcement learning for MDPs. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 130–146. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_8
32. Junges, S., Jansen, N., Seshia, S.A.: Enforcing almost-sure reachability in POMDPs. CoRR abs/2007.00085 (2020)
33. Junges, S., Jansen, N., Wimmer, R., Quatmann, T., Winterer, L., Katoen, J.P., Becker, B.: Finite-state controllers of POMDPs using parameter synthesis. In: UAI, pp. 519–529. AUAI Press (2018)
34. Kaelbling, L.P., Littman, M.L., Cassandra, A.R.: Planning and acting in partially observable stochastic domains. Artif. Intell. **101**(1–2), 99–134 (1998)
35. Littman, M.L., Cassandra, A.R., Kaelbling, L.P.: Learning policies for partially observable environments: Scaling up. In: ICML, pp. 362–370. Morgan Kaufmann (1995)
36. Madani, O., Hanks, S., Condon, A.: On the undecidability of probabilistic planning and infinite-horizon partially observable Markov decision problems. In: AAAI, pp. 541–548. AAAI Press (1999)
37. Meuleau, N., Kim, K.E., Kaelbling, L.P., Cassandra, A.R.: Solving POMDPs by searching the space of finite policies. In: UAI, pp. 417–426. Morgan Kaufmann (1999)
38. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
39. Nam, W., Alur, R.: Active learning of plans for safety and reachability goals with partial observability. IEEE Trans. Syst. Man Cybern. Part B **40**(2), 412–420 (2010)
40. Norman, G., Parker, D., Zou, X.: Verification and control of partially observable probabilistic systems. Real-Time Syst. **53**(3), 354–402 (2017). https://doi.org/10.1007/s11241-017-9269-4
41. Pandey, B., Rintanen, J.: Planning for partial observability by SAT and graph constraints. In: ICAPS, pp. 190–198. AAAI Press (2018)
42. Pecka, M., Svoboda, T.: Safe exploration techniques for reinforcement learning - an overview. In: Hodicky, J. (ed.) MESAS 2014. LNCS, vol. 8906, pp. 357–375. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-13823-7_31
43. Pineau, J., Gordon, G., Thrun, S.: Point-based value iteration: an anytime algorithm for POMDPs. In: IJCAI, pp. 1025–1032. Morgan Kaufmann (2003)
44. Pnueli, A.: The temporal logic of programs. In: FOCS, pp. 46–57. IEEE CS (1977)
45. Poupart, P., Boutilier, C.: Bounded finite state controllers. In: NIPS, pp. 823–830. MIT Press (2003)
46. Puterman, M.L.: Markov Decision Processes. Wiley, Hoboken (1994)
47. Raskin, J., Chatterjee, K., Doyen, L., Henzinger, T.A.: Algorithms for omega-regular games with imperfect information. Log. Methods Comput. Sci. **3**(3) (2007)
48. Shani, G., Pineau, J., Kaplow, R.: A survey of point-based POMDP solvers. Auton. Agent. Multi-Agent Syst. **27**(1), 1–51 (2013). https://doi.org/10.1007/s10458-012-9200-2
49. Silver, D., Veness, J.: Monte-Carlo planning in large POMDPs. In: NIPS, pp. 2164–2172 (2010)
50. Smith, T., Simmons, R.: Heuristic search value iteration for POMDPs (2004)

51. Svorenová, M., et al.: Temporal logic motion planning using POMDPs with parity objectives: case study paper. In: HSCC, pp. 233–238. ACM (2015)
52. Thrun, S., Burgard, W., Fox, D.: Probabilistic Robotics. The MIT Press, Cambridge (2005)
53. Turchetta, M., Berkenkamp, F., Krause, A.: Safe exploration for interactive machine learning. In: NeurIPS, pp. 2887–2897 (2019)
54. Walraven, E., Spaan, M.T.J.: Accelerated vector pruning for optimal POMDP solvers. In: AAAI, pp. 3672–3678. AAAI Press (2017)
55. Wang, Y., Chaudhuri, S., Kavraki, L.E.: Bounded policy synthesis for POMDPs with safe-reachability objectives. In: AAMAS, pp. 238–246. IFAAMAS (2018)
56. Wierstra, D., Foerster, A., Peters, J., Schmidhuber, J.: Solving deep memory POMDPs with recurrent policy gradients. In: de Sá, J.M., Alexandre, L.A., Duch, W., Mandic, D. (eds.) ICANN 2007. LNCS, vol. 4668, pp. 697–706. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74690-4_71
57. Wimmer, R., Jansen, N., Ábrahám, E., Katoen, J.P., Becker, B.: Minimal counterexamples for linear-time probabilistic verification. Theor. Comput. Sci. **549**, 61–100 (2014)
58. Winterer, L., Wimmer, R., Jansen, N., Becker, B.: Strengthening deterministic policies for POMDPs. In: Lee, R., Jha, S., Mavridou, A., Giannakopoulou, D. (eds.) NFM 2020. LNCS, vol. 12229, pp. 115–132. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-55754-6_7

Rigorous Roundoff Error Analysis of Probabilistic Floating-Point Computations

George Constantinides[1], Fredrik Dahlqvist[1,2], Zvonimir Rakamarić[3], and Rocco Salvia[3](✉)

[1] Imperial College London, London, UK
g.constantinides@ic.ac.uk
[2] University College London, London, UK
f.dahlqvist@ucl.ac.uk
[3] University of Utah, Salt Lake City, USA
{zvonimir,rocco}@cs.utah.edu

Abstract. We present a detailed study of roundoff errors in probabilistic floating-point computations. We derive closed-form expressions for the distribution of roundoff errors associated with a random variable, and we prove that roundoff errors are generally close to being uncorrelated with their generating distribution. Based on these theoretical advances, we propose a model of IEEE floating-point arithmetic for numerical expressions with probabilistic inputs and an algorithm for evaluating this model. Our algorithm provides rigorous bounds to the output and error distributions of arithmetic expressions over random variables, evaluated in the presence of roundoff errors. It keeps track of complex dependencies between random variables using an SMT solver, and is capable of providing sound but tight probabilistic bounds to roundoff errors using symbolic affine arithmetic. We implemented the algorithm in the PAF tool, and evaluated it on FPBench, a standard benchmark suite for the analysis of roundoff errors. Our evaluation shows that PAF computes tighter bounds than current state-of-the-art on almost all benchmarks.

1 Introduction

There are two common sources of randomness in a numerical computation (a straight-line program). First, the computation might be using inherently noisy data, for example from analog sensors in cyber-physical systems such as robots, autonomous vehicles, and drones. A prime example is data from GPS sensors, whose error distribution can be described very precisely [2] and which we study in some detail in Sect. 2. Second, the computation itself might sample from random number generators. Such probabilistic numerical routines, known as Monte-Carlo methods, are used in a wide variety of tasks, such as integration [34,42], optimization [43], finance [25], fluid dynamics [32], and computer graphics [30]. We

Supported in part by the National Science Foundation awards CCF 1552975, 1704715, the Engineering and Physical Sciences Research Council (EP/P010040/1), and the Leverhulme Project Grant "Verification of Machine Learning Algorithms".

A. Silva and K. R. M. Leino (Eds.): CAV 2021, LNCS 12760, pp. 626–650, 2021.
https://doi.org/10.1007/978-3-030-81688-9_29

call numerical computations whose input values are sampled from some probability distributions *probabilistic computations*.

Probabilistic computations are typically implemented using floating-point arithmetic, which leads to roundoff errors being introduced in the computation. To strike the right balance between the performance and energy consumption versus the quality of the computed result, expert programmers rely on either a manual or automated floating-point error analysis to guide their design decisions. However, the current state-of-the-art approaches in this space have primary focused on *worst-case* roundoff error analysis of *deterministic* computations. So what can we say about floating-point roundoff errors in a probabilistic context? Is it possible to probabilistically quantify them by computing confidence intervals? Can we, for example, say with 99% confidence that the roundoff error of the computed result is smaller than some chosen constant? What is the distribution of outputs when roundoff errors are taken into account? In this paper, we explore these and similar questions. To answer them, we propose a rigorous – that is to say *sound* – approach to quantifying roundoff errors in probabilistic computations. Based on this approach, we develop an automatic tool that efficiently computes an overapproximate probabilistic profile of roundoff errors.

As an example, consider the floating-point arithmetic expression $(X+Y)\div Y$, where X and Y are random inputs represented by independent random variables. In Sect. 4, we first show how the computation in *finite-precision* of a single arithmetic operation such as $X+Y$ can be modeled as $(X+Y)(1+\varepsilon)$, where ε is also a random variable. We then show how this random variable can be computed from first principles and why it makes sense to view $(X+Y)$ and $(1+\varepsilon)$ as independent expressions, which in turn allows us to easily compute the distribution of $(X+Y)(1+\varepsilon)$. The distribution of ε depends on that of $X+Y$, and we therefore need to evaluate arithmetic operations between random variables. When the operands are independent – as in $X+Y$ – this is standard [48], but when the operands are dependent – as in the case of the division in $(X+Y)\div Y$ – this is a hard problem. To solve it, we adopt and improve a technique for soundly bounding these distributions described in [3]. Our improvement comes from the use of an SMT solver to reason about the dependency between $(X+Y)$ and Y and remove regions of the state-space with zero probability. We describe this in Sect. 6.

We can thus soundly bound the output distribution of any probabilistic computation, such as $(X+Y)\div Y$, performed in floating-point arithmetic. This gives us the ability to perform *probabilistic range analysis* and prove rigorous assertions like: 99% of the outputs of a floating-point computation are smaller than a given constant bound. In order to perform *probabilistic roundoff error analysis* we develop *symbolic affine arithmetic* in Sect. 5. This technique is combined with probabilistic range analysis to compute *conditional roundoff errors*. Specifically, we over-approximate the maximal error conditioned on the output landing in the 99% range computed by the probabilistic range analysis, meaning conditioned on the computations not returning an outlier.

We implemented our model and algorithms in a tool called PAF (for Probabilistic Analysis of Floating-point errors). We evaluated PAF on the standard floating-point benchmark suite FPBench [11], and compared its range and error analysis with the worst-case roundoff error analyzer FPTaylor [46,47] and the probabilistic roundoff error analyzer PrAn [36]. We present the results in Sect. 7, and show that FPTaylor's worst-case analysis is often overly pessimistic in the probabilistic setting, while PAF also generates tighter probabilistic error bounds than PrAn on almost all benchmarks.

We summarize our contributions as follows:

(i) We derive a closed-form expression (6) for the distribution of roundoff errors associated with a random variable. We prove that roundoff errors are generally close to being uncorrelated with their input distribution.
(ii) Based on these results we propose a model of IEEE 754 floating-point arithmetic for numerical expressions with probabilistic inputs.
(iii) We evaluate this model by developing a new algorithm for rigorously bounding the output range and roundoff error distributions of floating-point arithmetic expressions with probabilistic inputs.
(iv) We implement this model in the PAF tool,[1] and perform probabilistic range and roundoff error analysis on a standard benchmark suite. Our comparison with the current state-of-the-art shows the advantages of our approach in terms of computing tighter, and yet still rigorous, probabilistic bounds.

2 Motivating Example

GPS sensors are inherently noisy. Bornholt [1] shows that the conditional probability of the true coordinates given a GPS reading is distributed according to a Rayleigh distribution. Interestingly, since the density of any Rayleigh distribution is always zero at $x = 0$, it is extremely unlikely that the true coordinates lie in a small neighborhood of those given by the GPS reading. This leads to errors, and hence the sensed coordinates should be corrected by adding a probabilistic error term which, on average, shifts the observed coordinates into an area of high probability for the true coordinates [1,2]. The latitude correction is given by:

$$\mathtt{TrueLat} = \mathtt{GPSLat} + ((\mathtt{radius} * \mathtt{sin}(\mathtt{angle})) * \mathtt{DPERM}), \tag{1}$$

where `radius` is Rayleigh distributed, `angle` uniformly distributed, `GPSLat` is the latitude, and `DPERM` a constant for converting meters into degrees.

A developer trying to strike the right balance between resources, such as energy consumption or execution time, versus the accuracy of the computation, might want to run a rigorous worst-case floating-point analysis tool to determine which floating-point format is accurate enough to process GPS signals. This is mandatory if the developer requires rigorous error bounds holding with 100% certainty. The problem when analyzing a piece of code involving (1) is that the Rayleigh distribution has $[0, \infty)$ as its support, and *any* worst-case roundoff error analysis will return an infinite error bound in this situation. To get a meaningful (numeric) error bound, we need to truncate the support of the distribution. The most conservative truncation is $[0, max]$, where max is the largest representable number (not causing an overflow) at the target floating-point precision format.

[1] PAF is open source and publicly available at https://github.com/soarlab/paf.

Table 1. Roundoff error analysis for the probabilistic latitude correction of (1).

Precision	Max	FPTaylor	PAF 100%	PAF 99.9999%	
				Absolute	Meters
Double	$\approx 10^{307}$	4.3e+286	4.3e+286	4.1e−15	4.5e−10
Single	$\approx 10^{38}$	2.1e+26	2.1e+26	3.7e−06	4.1e−1
Half	$\approx 10^{4}$	2.5e−2	2.5e−2	2.4e−2	2667

In Table 1, we report a detailed roundoff error analysis of (1) implemented in IEEE 754 double-, single-, and half-precision formats, with `GPSLat` set to the latitude of the Greenwich observatory. With each floating-point format, we associate the range $[0, max]$ of the truncated Rayleigh distribution. We compute worst-case roundoff error bounds for (1) with the state-of-the-art error analyzer FPTaylor [47] and with our tool PAF by setting the confidence interval to 100%. As expected, the error bounds from the two tools are identical. Finally, we compute the 99.9999% *conditional roundoff error* using PAF. This value is an upper bound to the roundoff error *conditioned* on the computation having landed in an interval capturing 99.9999% of all possible outputs. Column Absolute gives the error in degrees and Meters in meters ($1^{\circ} \approx$111km).

By looking at the results obtained without our *probabilistic error analysis* (columns FPTaylor and PAF 100%), the developer might *erroneously* conclude that half-precision format is the most appropriate to implement (1) because it results in the smallest error bound. However, with the information provided by the 99.9999% *conditional roundoff error*, the developer can see that the *average* error is many orders of magnitude smaller than the worst-case scenarios. Armed with this information, the developer can conclude that with a roundoff error of roughly 40 cm (4.1e−1 ms) when correcting 99.9999% of GPS latitude readings, working in single-precision is an adequate compromise between efficiency and accuracy of the computation.

This motivates the innovative concept of *probabilistic precision tuning*, evolved from standard worst-case precision tuning [5,12], to determine which floating-point format is the most appropriate for a given computation. As an example, let us do a probabilistic precision tuning exercise for the latitude correction computation of (1). We truncate the Rayleigh distribution in the interval $[0, 10^{307}]$, and assume we can tolerate up to 1e−5 roundoff error (roughly 1 m). First, we manually perform worst-case precision tuning using FPTaylor to determine that the minimal floating-point format not violating the given error bound needs 1022 mantissa and 11 exponent bits. Such large custom format is prohibitively expensive, in particular for devices performing frequent GPS readings, like smartphones or smartwatches. Conversely, when we manually perform probabilistic precision tuning using PAF with a confidence interval set to 99.9999%, we determine we need only 22 mantissa and 11 exponent bits. Thanks to PAF, the developer can provide a custom confidence interval of interest to the probabilistic precision tuning routine to adjust for the extremely unlikely corner cases like the ones we described for (1), and ultimately obtain more optimal tuning results.

3 Preliminaries

3.1 Floating-Point Arithmetic

Given a *precision* $p \in \mathbb{N}$ and an *exponent range* $[e_{min}, e_{max}] \triangleq \{n \mid n \in \mathbb{N} \wedge e_{min} \leq n \leq e_{max}\}$, we define $\mathbb{F}(p, e_{min}, e_{max})$, or simply $\mathbb{F}$ if there is no ambiguity, as the set of extended real numbers

$$\mathbb{F} \triangleq \left\{ (-1)^s 2^e \left(1 + \frac{k}{2^p}\right) \middle| s \in \{0,1\}, e \in [e_{min}, e_{max}], 0 \leq k < 2^p \right\} \cup \{-\infty, 0, \infty\}$$

Elements $z = z(s,e,k) \in \mathbb{F}$ will be called *floating-point representable numbers* (for the given precision p and exponent range $[e_{min}, e_{max}]$) and we will use the variable z to represent them. The variable s will be called the *sign*, the variable e the *exponent*, and the variable k the *significand* of $z(s,e,k)$.

Next, we introduce a *rounding map* $\text{Round} : \mathbb{R} \to \mathbb{F}$ that rounds to nearest (or to $-\infty/\infty$ for values smaller/greater than the smallest/largest finite element of $\mathbb{F}$) and follows any of the IEEE 754 rounding modes in case of a tie. We will not worry about which choice is made since the set of mid-points will always have probability zero for the distributions we will be working with. All choices are thus equivalent, probabilistically speaking, and what happens in a tie can therefore be left unspecified. We will denote the extended real line by $\overline{\mathbb{R}} \triangleq \mathbb{R} \cup \{-\infty, \infty\}$. The (signed) *absolute error function* $\text{err}_{\text{abs}} : \mathbb{R} \to \overline{\mathbb{R}}$ is defined as: $\text{err}_{\text{abs}}(x) = x - \text{Round}(x)$. We define the sets $\lfloor z, z \rceil \triangleq \{y \in \mathbb{R} \mid \text{Round}(y) = \text{Round}(z)\}$. Thus if $z \in \mathbb{F}$, then $\lfloor z, z \rceil$ is the collection of all reals rounding to z. As the reader will see, the basic result of Sect. 4 (Eq. (5)) is expressed entirely using the notation $\lfloor z, z \rceil$ which is parametric in the choice of the Round function. It follows that our results apply to rounding modes other that round-to-nearest with minimal changes. The *relative error function* $\text{err}_{\text{rel}} : \mathbb{R} \setminus \{0\} \to \overline{\mathbb{R}}$ is defined by

$$\text{err}_{\text{rel}}(x) = \frac{x - \text{Round}(x)}{x}.$$

Note that $\text{err}_{\text{rel}}(x) = 1$ on $\lfloor 0, 0 \rceil \setminus \{0\}$, $\text{err}_{\text{rel}}(x) = \infty$ on $\lfloor -\infty, -\infty \rceil$ and $\text{err}_{\text{rel}}(x) = -\infty$ on $\lfloor \infty, \infty \rceil$. Recall also the fact [26] that $-2^{-(p+1)} < \text{err}_{\text{rel}}(x) < 2^{-(p+1)}$ outside of $\lfloor 0, 0 \rceil \cup \lfloor -\infty, -\infty \rceil \cup \lfloor \infty, \infty \rceil$. The quantity $2^{-(p+1)}$ is usually called the *unit roundoff* and will be denoted by u.

For $z_1, z_2 \in \mathbb{F}$ and $\text{op} \in \{+, -, \times, \div\}$ an (infinite-precision) arithmetic operation, the traditional model of IEEE 754 floating-point arithmetic [26,39] states that the finite-precision implementation $\texttt{op}_\texttt{m}$ of op must satisfy

$$z_1 \ \texttt{op}_\texttt{m} \ z_2 = (z_1 \ \text{op} \ z_2)(1 + \delta) \qquad |\delta| \leq u, \tag{2}$$

We leave dealing with subnormal floating-point numbers to future work. The model given by Eq. (2) stipulates that the implementation of an arithmetic operation can induce a relative error of magnitude *at most* u. The exact size of the error is, however, not specified and Eq. (2) is therefore a *non-deterministic* *model of computation*. It follows that numerical analyses based on Eq. (2) must consider *all* possible relative errors δ and are fundamentally *worst-case* analyses. Since the output of such a program might be the input of another, one should also consider non-deterministic inputs, and this is indeed what happens with automated tools for roundoff error analysis, such as Daisy [12] or FPTaylor [46, 47], which require for each variable of the program a (bounded) range of possible values in order to perform a worst-case analysis (*cf.* GPS example in Sect. 2).

In this paper, we study a model formally similar to Eq. (2), namely

$$z_1 \ \texttt{op}_\texttt{m} \ z_2 = (z_1 \text{ op } z_2)(1+\delta) \qquad \delta \sim \mathit{dist}. \tag{3}$$

The difference is that δ is now *distributed according to dist*, a probability distribution whose support is $[-u, u]$. In other words, we move from a non-deterministic to a *probabilistic* model of roundoff errors. This is similar to the 'Monte Carlo arithmetic' of [41], but whilst *op. cit. postulates* that *dist* is the uniform distribution on $[-u, u]$, we compute *dist* from first principles in Sect. 4.

3.2 Probability Theory

To fix the notation and be self-contained, we present some basic notions of probability theory which are essential to what follows.

Cumulative Distribution Functions and Probability Density Functions. We assume that the reader is (at least intuitively) familiar with the notion of a (real) random variable. Given a random variable X we define its Cumulative Distribution Function (CDF) as the function $c(t) \triangleq \mathbb{P}[X \leq t]$. If there exists a non-negative integrable function $d : \mathbb{R} \to \mathbb{R}$ such that

$$c(t) \triangleq \mathbb{P}[X \leq t] = \int_{-\infty}^{t} d(t) \ dt$$

then we call $d(t)$ the Probability Density Function (PDF) of X. If it exists, then it can be recovered from the CDF by differentiation $d(t) = \frac{\partial}{\partial t} c(t)$ by the fundamental theorem of calculus.

Not all random variables have a PDF: consider the random variable which takes value 0 with probability $1/2$ and value 1 with probability $1/2$. For this random variable it is impossible to write $\mathbb{P}[X \leq t] = \int d(t) \ dt$. Instead, we will write the distribution of such a variable using the so-called Dirac delta measure at 0 and 1 as $1/2\delta_0 + 1/2\delta_1$. It is possible for a random variable to have a PDF covering part of its distribution – its *continuous part* – and a sum of Dirac deltas covering the rest of its distribution – its *discrete part*. We will encounter examples of such random variables in Sect. 4. Finally, if X is a random variable and $f : \mathbb{R} \to \mathbb{R}$ is a measurable function, then $f(X)$ is a random variable. In particular $\text{err}_{\text{rel}}(X)$ is a random variable which we will describe in Sect. 4.

Arithmetic on Random Variables. Suppose X, Y are *independent* random variables with PDFs f_X and f_Y, respectively. Using the arithmetic operations we can form new random variables $X+Y, X-Y, X \times Y, X \div Y$. The PDFs of these new random variables can be expressed as operations on f_X and f_Y, which can be found in [48]. It is important to note that these operations are only valid if X and Y are assumed to be independent. When an arithmetic expression containing variable repetitions is given a random variable interpretation, this independence can no longer be assumed. In the expression $(X+Y) \div Y$ the sub-term $(X+Y)$ can be interpreted by the formulas of [48] if X, Y are independent. However, the sub-terms $X+Y$ and Y cannot be interpreted in this way since $X+Y$ and Y are clearly not independent random variables.

Soundly Bounding Probabilities. The constraint that the distribution of a random variable must integrate to 1 makes it impossible to order random variables in the 'natural' way: if $\mathbb{P}[X \in A] \leq \mathbb{P}[Y \in A]$, then $\mathbb{P}[Y \in A^c] \leq \mathbb{P}[X \in A^c]$, i.e., we cannot say that $X \leq Y$ if $\mathbb{P}[X \in A] \leq \mathbb{P}[Y \in A]$. This means that we cannot quantify our probabilistic uncertainty about a random variable by sandwiching it between two other random variables as one would do with reals or real-valued functions. One solution is to restrict the sets used in the comparison, i.e., declare that $X \leq Y$ iff $\mathbb{P}[X \in A] \leq \mathbb{P}[Y \in A]$ for A ranging over a given set of 'test subsets'. Such an order can be defined by taking as 'test subsets' the intervals $(-\infty, x]$ [44]. This order is called the *stochastic order*. It follows from the definition of the CDF that this order can be defined by simply saying that $X \leq Y$ iff $c_X \leq c_Y$, where c_X and c_Y are the CDFs of X and Y, respectively. If it is possible to sandwich an unknown random variable X between known lower and upper bounds $X_{lower} \leq X \leq X_{upper}$ using the stochastic order then it becomes possible to give sound bounds to the quantities $\mathbb{P}[X \in [a,b]]$ via

$$\mathbb{P}[X \in [a,b]] = c_X(b) - c_X(a) \leq c_{X_{upper}}(b) - c_{X_{lower}}(a)$$

P-Boxes and DS-Structures. As mentioned above, giving a random variable interpretation to an arithmetic expression containing variable repetitions cannot be done using the arithmetic of [48]. In fact, these interpretations are in general analytically intractable. Hence, a common approach is to give up on soundness and approximate such distributions using Monte-Carlo simulations. We use this approach in our experiments to assess the quality of our sound results. However, we will also provide sound under- and over-approximations of the distribution of arithmetic expressions over random variables using the stochastic order discussed above. Since $X_{lower} \leq X \leq X_{upper}$ is equivalent to saying that $c_{X_{lower}}(x) \leq c_X(x) \leq c_{X_{upper}}(x)$, the fundamental approximating structure will be a pair of CDFs satisfying $c_1(x) \leq c_2(x)$. Such a structure is known in the literature as a *p-box* [19], and has already been used in the context of probabilistic roundoff errors in related work [3,36]. The data of a p-box is equivalent to a pair of sandwiching distributions for the stochastic order.

A *Dempster-Shafer structure* (DS-structure) of size N is a collection (i.e., set) of interval-probability pairs $\{([x_0, y_0], p_0), ([x_1, y_2], p_1), .., ([x_N, y_N], p_N)\}$ where $\sum_{i=0}^{N} p_i = 1$. The intervals in the collection might overlap. One can always convert a DS-structure to a p-box and back again [19], but arithmetic operations are much easier to perform on DS-structures than on p-boxes ([3]), which is why we will use DS-structures in the algorithm described in Sect. 6.

4 Distribution of Floating-Point Roundoff Errors

Our tool PAF computes *probabilistic* roundoff errors by conditioning the maximization of symbolic affine form (presented in Sect. 5) on the output of the computation landing in a confidence interval. The purpose of this section is to provide the necessary probabilistic tools to compute these intervals. In other words, this section provides the foundations of *probabilistic range analysis*. All proofs can be found in the extended version [7].

4.1 Derivation of the Distribution of Rounding Errors

Recall the probabilistic model of Eq. (3) where op is an infinite-precision arithmetic operation and $\texttt{op}_\texttt{m}$ its finite-precision implementation:

$$z_1 \ \texttt{op}_\texttt{m}\ z_2 = (z_1 \text{ op } z_2)(1+\delta) \qquad \delta \sim \mathit{dist}.$$

Let us also assume that z_1, z_2 are random variables with known distributions. Then z_1 op z_2 is also a random variable which can (in principle) be computed. Since the IEEE 754 standard states that $z_1 \ \texttt{op}_\texttt{m}\ z_2$ is computed by rounding the infinite precision operation z_1 op z_2, it is a completely natural consequence of the standard to require that δ is simply be given by

$$\delta = \text{err}_{\text{rel}}(z_1 \text{ op } z_2)$$

Thus, *dist* is the distribution of the random variable $\text{err}_{\text{rel}}(z_1 \text{ op } z_2)$. More generally, if X is a random variable with know distribution, we will show how to compute the distribution *dist* of the random variable

$$\text{err}_{\text{rel}}(X) = \frac{X - \text{Round}(X)}{X}.$$

We choose to express the distribution *dist* of relative errors *in multiples of the unit roundoff u*. This choice is arbitrary, but it allows us to work with a distribution on the conceptually and numerically convenient interval $[-1, 1]$, since the absolute value of the relative error is strictly bounded by u (see Sect. 3.1), rather than the interval $[-u, u]$.

To compute the density function of *dist*, we proceed as described in Sect. 3.2 by first computing the CDF $c(t)$ and then taking its derivative. Recall first from Sect. 3.1 that $\text{err}_{\text{rel}}(x) = 1$ if $x \in \lfloor 0, 0\rceil \setminus \{0\}$, $\text{err}_{\text{rel}}(x) = \infty$ if $x \in \lfloor -\infty, -\infty\rceil$, $\text{err}_{\text{rel}}(x) = -\infty$ if $x \in \lfloor \infty, \infty\rceil$, and $-u \leq \text{err}_{\text{rel}}(x) \leq u$ elsewhere. Thus:

$$\mathbb{P}\left[\text{err}_{\text{rel}}(X) = -\infty\right] = \mathbb{P}\left[X \in \lfloor \infty, \infty\rceil\right] \qquad \mathbb{P}\left[\text{err}_{\text{rel}}(X) = 1\right] = \mathbb{P}\left[X \in \lfloor 0, 0\rceil\right]$$
$$\mathbb{P}\left[\text{err}_{\text{rel}}(X) = \infty\right] = \mathbb{P}\left[X \in \lfloor -\infty, -\infty\rceil\right]$$

In other words, the probability measure corresponding to err_{rel} has three discrete components at $\{-\infty\}, \{1\}$, and $\{\infty\}$, which cannot be accounted for by a PDF (see Sect. 3.2). It follows that the probability measure *dist* is given by

$$\mathit{dist}_c + \mathbb{P}\left[X \in \lfloor 0, 0\rceil\right]\delta_1 + \mathbb{P}\left[X \in \lfloor -\infty, -\infty\rceil\right]\delta_\infty + \mathbb{P}\left[X \in \lfloor \infty, \infty\rceil\right]\delta_{-\infty} \quad (4)$$

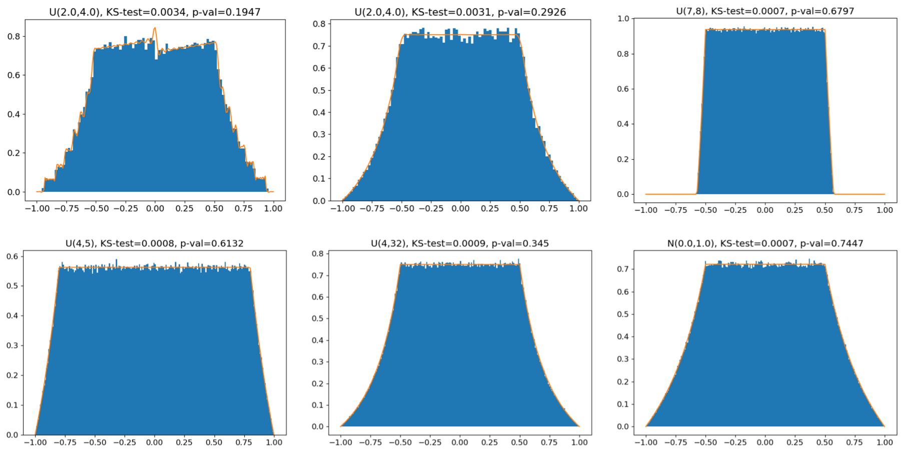


Fig. 1. Theoretical vs. empirical error distribution, clockwise from top-left: (i) Eq. (5) for $\mathsf{Unif}(2,4)$ 3 bit exponent, 4 bit significand, (ii) Eq. (5) for $\mathsf{Unif}(2,4)$ in half-precision, (iii) Eq. (6) for $\mathsf{Unif}(7,8)$ in single-precision, (iv) Eq. (6) for $\mathsf{Unif}(4,5)$ in single-precision, (v) Eq. (6) for $\mathsf{Unif}(4,32)$ in single-precision, (vi) Eq. (6) for $\mathsf{Norm}(0,1)$ in single-precision.

where $dist_c$ is a continuous measure that is not quite a probability measure since its total mass is $1 - \mathbb{P}[X \in \lfloor 0,0 \rceil] - \mathbb{P}[X \in \lfloor -\infty, -\infty \rceil] - \mathbb{P}[X \in \lfloor \infty, \infty \rceil]$. In general, $dist_c$ integrates to 1 in machine precision since $\mathbb{P}[X \in \lfloor 0,0 \rceil]$ is of the order of the smallest positive floating-point representable number, and the PDF of X rounds to 0 way before it reaches the smallest/largest floating-point representable number. However in order to be sound, we must in general include these three discrete components to our computations. The density $dist_c$ is given explicitly by the following result whose proof can already be found in [9].

Theorem 1. *Let X be a real random variable with PDF f. The continuous part $dist_c$ of the distribution of $\mathrm{err}_{\mathrm{rel}}(X)$ has a PDF given by*

$$d(t) = \sum_{z \in \mathbb{F} \setminus \{-\infty, 0, \infty\}} \mathbb{1}_{\lfloor z,z \rceil}\left(\frac{z}{1-tu}\right) f\left(\frac{z}{1-tu}\right) \frac{u\,|z|}{(1-tu)^2}, \tag{5}$$

where $\mathbb{1}_A(x)$ is the indicator function which returns 1 if $x \in A$ and 0 otherwise.

Figure 1 (i) and (ii) shows an implementation of Eq. (5) applied to the distribution $\mathsf{Unif}(2,4)$, first in very low precision (3 bit exponent, 4 bit significand) and then in half-precision. The theoretical density is plotted alongside a histogram of the relative error incurred when rounding 100,000 samples to low precision (computed in double-precision). The reported statistic is the K-S (Kolmogorov-Smirnov) test which measures the likelihood that a collection of samples were drawn from a given distribution. This test reports that we cannot reject the hypothesis that the samples are drawn from the corresponding density. Note how in low precision the term in $\frac{1}{(1-tu)^2}$ induces a visible asymmetry on the central section of the distribution. This effect is much less pronounced in half-precision.

For low precisions, say up to half-precision, it is computationally feasible to explicitly go through all floating-point numbers and compute the density of the roundoff error distribution *dist* directly from Eq. (5). However, this rapidly becomes prohibitively computationally expensive for higher precisions (since the number of floating-point representable numbers grows exponentially).

4.2 High-Precision Case

As the working precision increases, a regime changes occurs: on the one hand it becomes practically impossible to enumerate all floating-point representable numbers as done in Eq. (5), but on the other hand sufficiently well-behaved density functions are numerically close to being constant at the scale of an interval between two floating-point representable numbers. We exploit this smoothness to overcome the combinatorial limit imposed by Eq. (5).

Theorem 2. *Let X be a real random variable with PDF f. The continuous part $dist_c$ of the distribution of $\mathrm{err}_{\mathrm{rel}}(X)$ has a PDF given by $d_c(t) = d_{hp}(t) + R(t)$ where $d_{hp}(t)$ is the function on $[-1, 1]$ defined by*

$$d_{hp}(t) = \begin{cases} \frac{1}{1-tu} \sum\limits_{s,e=e_{min}+1}^{e_{max}-1} \int_{(-1)^s 2^e(1-u)}^{(-1)^s 2^e(2-u)} \frac{|x|}{2^{e+1}} f(x)\, dx & |t| \leq \frac{1}{2} \\ \\ \frac{1}{1-tu} \sum\limits_{s,e=e_{min}+1}^{e_{max}-1} \int_{(-1)^s 2^e(1-u)}^{(-1)^s 2^e(\frac{1}{|t|}-u)} \frac{|x|}{2^{e+1}} f(x)\, dx & \frac{1}{2} < |t| \leq 1 \end{cases} \tag{6}$$

and $R(t)$ is an error whose total contribution $|R| \triangleq \int_{-1}^{1} |R(t)| dt$ can be bounded by

$$|R| \leq \mathbb{P}\left[\mathrm{Round}(X) = z(s, e_{min}, k)\right] + \mathbb{P}\left[\mathrm{Round}(X) = z(s, e_{max}, k)\right] + \frac{3}{4}\left(\sum_{s, e_{min} < e < e_{max}} |f'(\xi_{e,s})\xi_{e,s} + f(\xi_{e,s})| \frac{2^{2e}}{2^p}\right)$$

where for each exponent e and sign s, $\xi_{e,s}$ is a point in $[z(s, e, 0), z(s, e, 2^p - 1)]$ if $s = 0$ and in $[z(s, e, 2^p - 1), z(s, e, 0)]$ if $s = 1$.

Note how Eq. (6) reduces the sum over *all* floating-point representable numbers in Eq. (5) to a sum over *the exponents* by exploiting the regularity of f. Note also that since f is a PDF, it usually decreases very quickly away from 0, and its derivative decreases even quicker and $|R|$ thus tends to be very small and $|R| \to 0$ as the precision $p \to \infty$.

Figure 1 shows Eq. (6) for: (i) the distribution $\mathsf{Unif}(7, 8)$ where large significands are more likely, (ii) the distribution $\mathsf{Unif}(4, 5)$ where small significands are more likely, (iii) the distribution $\mathsf{Unif}(4, 32)$ where significands are equally likely, and (iv) the distribution $\mathsf{Norm}(0, 1)$ with infinite support. The graphs show the density function given by Eq. (6) in single-precision versus a histogram of the relative error incurred when rounding 1,000,000 samples to single-precision (computed in double-precision). The K-S test reports that we cannot reject the hypothesis that the samples are drawn from the corresponding distributions.

4.3 Typical Distribution

The distributions depicted in graphs (ii), (v) and (vi) of Fig. 1 are very similar, despite being computed from very different input distributions. What they have in common is that their input distributions have the property that all significands in their supports are equally likely. We show that under this assumption, the distribution of roundoff errors given by Eq. (5) converges to a unique density as the precision increases, irrespective of the input distribution! Since significands are frequently equiprobable (it is the case for a third of our benchmarks), this density is of great practical importance. If one had to choose 'the' canonical distribution for roundoff errors, we claim that the density given below should be this distribution, and we therefore call it the *typical distribution*; we depict it in Fig. 2 and formalize it with the following theorem, which can mostly be found in [9].

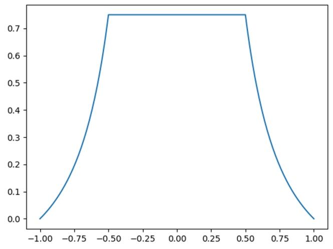


Fig. 2. Typical distribution.

Theorem 3. *If X is a random variable such that $\mathbb{P}[\mathrm{Round}(X) = z(s, e, k_0)] = \frac{1}{2^p}$ for any significand k_0, then*

$$d_{typ}(t) \triangleq \lim_{p\to\infty} d(t) = \begin{cases} \frac{3}{4} & |t| \le \frac{1}{2} \\ \frac{1}{2}\left(\frac{1}{t} - 1\right) + \frac{1}{4}\left(\frac{1}{t} - 1\right)^2 & |t| > \frac{1}{2} \end{cases} \tag{7}$$

where $d(t)$ is the exact density given by Eq. (5).

4.4 Covariance Structure

The result above can be interpreted as saying that if X is such that all mantissas are equiprobable, then X and $\mathrm{err}_{\mathrm{rel}}(X)$ are asymptotically independent (as $p \to \infty$). Much more generally, we now show that if a random variable X has a sufficiently regular PDF, it is close to being uncorrelated from $\mathrm{err}_{\mathrm{rel}}(X)$. Formally, we prove that the covariance

$$\mathrm{Cov}(X, \mathrm{err}_{\mathrm{rel}}(X)) = \mathbb{E}[X.\mathrm{err}_{\mathrm{rel}}(X)] - \mathbb{E}[X]\,\mathbb{E}[\mathrm{err}_{\mathrm{rel}}(X)] \tag{8}$$

is small, specifically of the order of u. Note that the expectation in the first summand above is taken w.r.t. the joint distribution of X and $\mathrm{err}_{\mathrm{rel}}(X)$.

The main technical obstacles to proving that the expression above is small are that $\mathbb{E}[\mathrm{err}_{\mathrm{rel}}(X)]$ turns out to be difficult to compute (we only manage to bound it) and that the joint distribution $\mathbb{P}[X \in A \wedge \mathrm{err}_{\mathrm{rel}}(X) \in B]$ does not have a PDF since it is not continuous w.r.t. the Lebesgue measure on $\mathbb{R}^2$. Indeed, it is supported by the graph of the function $\mathrm{err}_{\mathrm{rel}}$ which has a Lebesgue measure of 0. This does not mean that it is impossible to compute the expectation

$$\mathbb{E}[X.\mathrm{err}_{\mathrm{rel}}(X)] = \int_{\mathbb{R}^2} xut \, d\mathbb{P} \tag{9}$$

but it is necessary to use some more advanced probability theory. We will make the simplifying assumption that the density of X is constant on each interval $\lfloor z, z \rceil$ in order to keep the proof manageable. In practice this is an extremely good approximation. Without this assumption, we would need to add an error term similar to that of Theorem 2 to the expression below. This is not conceptually difficult, but it is messy, and would distract from the main aim of the following theorem which is to bound $\mathbb{E}[\mathrm{err}_{\mathrm{rel}}(X)]$, compute $\mathbb{E}[X.\mathrm{err}_{\mathrm{rel}}(X)]$, and show that the covariance between X and $\mathrm{err}_{\mathrm{rel}}(X)$ is typically of the order of u.

Theorem 4. *If the density of X is piecewise constant on intervals $\lfloor z, z \rceil$, then*

$$\left(L - \mathbb{E}[X]\, K\frac{u}{6}\right) \leq \mathrm{Cov}(X, \mathrm{err}_{\mathrm{rel}}(X)) \leq \left(L - \mathbb{E}[X]\, K\frac{4u}{3}\right)$$

where $L = \sum_{s,e} f((-1)^s 2^e)(-1)^s 2^{2e}\frac{3u^2}{2}$ *and* $K = \sum_{s,e=e_{min}+1}^{e_{max}-1} \int_{(-1)^s 2^e(1-u)}^{(-1)^s 2^e(2-u)} \frac{|x|}{2^{e+1}}$ $f(x)\, dx$.

If the distribution of X is centered (i.e., $\mathbb{E}[X] = 0$) then L is the exact value of the covariance, and it is worth noting that L is fundamentally an artifact of the floating-point representation and is due to the fact that the intervals $\lfloor 2^e, 2^e \rceil$ are not symmetric. More generally, for $\mathbb{E}[X]$ of the order of, say, 2, the covariance will be small (of the order of u) as $K \leq 1$ (since $|x| \leq 2^{e+1}$ in each summand). For very large values of $\mathbb{E}[X]$ it is worth noting that there is a high chance that L is also be very large, partially canceling $\mathbb{E}[X]$. An illustration of this is given by the *doppler* benchmark examined in Sect. 7, an outlier as it has an input variable with range [20, 20000]. Nevertheless, even for this benchmark the bounds of Theorem 4 still give a small covariance of the order of 0.001.

4.5 Error Terms and P-Boxes

In low-precision we can use the exact formula Eq. (5) to compute the error distribution. However, in high-precision, approximations (typically extremely good) like Eqs. (6) and (7) must be used. In order to remain sound in the implementation of our model (see Sect. 6) we must account for the error made by this approximation. We have not got the space to discuss the error made by Eq. (7), but taking the term $|R|$ of Theorem 2 as an illustration, we can use the notion of p-box described in Sect. 3.2 to create an object which soundly approximates the error distribution. We proceed as follows: since $|R|$ bounds the total error accumulated over all $t \in [-1, 1]$, we can soundly bound the CDF $c(t)$ of the error distribution given by Eq. (6) by using the p-box

$$c^{-}(t) = \max(0, c(t) - |R|) \qquad \text{and} \qquad c^{+}(t) = \min(1, c(t) + |R|)$$

5 Symbolic Affine Arithmetic

In this section, we introduce *symbolic affine arithmetic*, which we employ to generate the symbolic form for the roundoff error that we use in Sect. 6.3. Affine arithmetic [6] is a model for range analysis that extends classic interval arithmetic [40] with information about linear correlations between operands. Symbolic affine arithmetic extends standard affine arithmetic by keeping the coefficients of the noise terms *symbolic*. We define a *symbolic affine form* as

$$\hat{x} = x_0 + \sum_{i=1}^{n} x_i \epsilon_i, \qquad \text{where } \epsilon_i \in [-1, 1]. \tag{10}$$

We call x_0 the central symbol of the affine form, while x_i are the symbolic coefficients for the noise terms ϵ_i. We can always convert a symbolic affine form to its corresponding interval representation. This can be done using interval arithmetic or, to avoid precision loss, using a global optimizer.

Affine operations between symbolic forms follow the usual rules, such as

$$\alpha\hat{x} + \beta\hat{y} + \zeta = \alpha x_0 + \beta y_0 + \zeta + \sum_{i=1}^{n} (\alpha x_i + \beta y_i)\epsilon_i$$

Non-linear operations cannot be represented exactly using an affine form. Hence, we approximate them like in standard affine arithmetic [49].

Sound Error Analysis with Symbolic Affine Arithmetic. We now show how the roundoff errors get propagated through the four arithmetic operations. We apply these propagation rules to an arithmetic expression to accurately keep track of the roundoff errors. Since the (absolute) roundoff error directly depends on the range of a computation, we describe range and error together as a pair `(range: Symbol,` $\widehat{err}$`: Symbolic Affine Form)`. Here, `range` represents the infinite-precision range of the computation, while $\widehat{err}$ is the symbolic affine form for the roundoff error in floating-point precision. Unary operators (e.g., rounding) take as input a (range, error form) pair, and return a new output pair; binary operators take as input two pairs, one per operand. For linear operators, the ranges and errors get propagated using the standard rules of affine arithmetic.

For the multiplication, we distribute each term in the first operand to every term in the second operand:

$$(\mathtt{x},\ \widehat{err}_x) * (\mathtt{y},\ \widehat{err}_y) = (\mathtt{x*y},\ \mathtt{x} * \widehat{err}_y + \mathtt{y} * \widehat{err}_x + \widehat{err}_x * \widehat{err}_y)$$

The output range is the product of the input ranges and the remaining terms contribute to the error. Only the last (quadratic) expression cannot be represented exactly in symbolic affine arithmetic; we bound such non-linearities using a global optimizer. The division is computed as the term-wise multiplication of the numerator with the inverse of the denominator. Hence, we need the inverse of the denominator error form, and then we can proceed as for multiplication. To compute the inverse, we leverage the symbolic expansion used in FPTaylor [46].

Finally, after every operation we apply the unary rounding operator from Eq. (2). The infinite-precision range is not affected by rounding. The rounding operator appends a fresh noise term to the symbolic error form. The coefficient for the new noise term is the (symbolic) floating-point range given by the sum of the input range with the input error form.

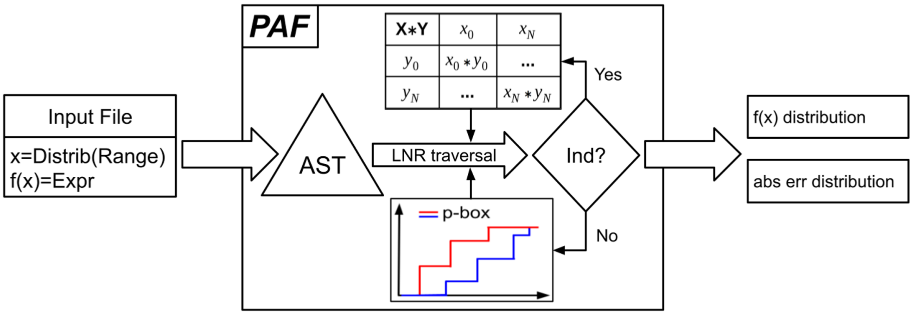


Fig. 3. Toolflow of PAF.

6 Algorithm and Implementation

In this section, we describe our probabilistic model of floating-point arithmetic and how we implement it in a prototype named PAF (for Probabilistic Analysis of Floating-point errors). Figure 3 shows the toolflow of PAF.

6.1 Probabilistic Model

PAF takes as input a text file describing a probabilistic floating-point computation and its input distributions. The kinds of computations we support are captured with this simple grammar:

$$\mathtt{t} ::= \mathtt{z} \mid \mathtt{x_i} \mid \mathtt{t}\ \mathtt{op_m}\ \mathtt{t} \qquad \mathtt{z} \in \mathbb{F}, \mathtt{i} \in \mathbb{N},\ \mathtt{op_m}\ \in \{+, -, \times, \div\}$$

Following [8,31], we interpret each computation $\mathtt{t}$ given by the grammar as a random variable. We define the interpretation map $[\![-]\!]$ over the computation tree inductively. The base case is given by $[\![\mathtt{z}(s, e, k)]\!] \triangleq (-1)^s 2^e(1 + k2^{-p})$ and $[\![\mathtt{x_i}]\!] \triangleq X_i$, where the real numbers $[\![\mathtt{z}(s, e, k)]\!]$ are understood as constant random variables and each X_i is a random input variable with a user-specified distribution. Currently, PAF supports several well-known distributions out-of-the-box (e.g., uniform, normal, exponential), and the user can also define custom distributions as piecewise functions. For the inductive case $[\![\mathtt{t_1}\ \mathtt{op_m}\ \mathtt{t_2}]\!]$, we put the lessons from Sect. 4 to work. Recall first the probabilistic model from Eq. (3):

$$x \; \texttt{op}_\texttt{m} \; y = (x \text{ op } y)(1+\delta), \qquad \delta \sim \mathit{dist}$$

In Sect. 4.1, we showed that *dist* should be taken as the distribution of the actual roundoff errors of the random elements (x op y). We therefore define:

$$[\![\texttt{t}_1 \; \texttt{op}_\texttt{m} \; \texttt{t}_2]\!] \triangleq ([\![\texttt{t}_1]\!] \text{ op } [\![\texttt{t}_2]\!]) \times (1 + \text{err}_{\text{rel}}([\![\texttt{t}_1]\!] \text{ op } [\![\texttt{t}_2]\!])) \tag{11}$$

To evaluate the model of Eq. (11), we first use the appropriate closed-form expression Eqs. (5) to (7) derived in Sect. 4 to evaluate the distribution of the random variable $\text{err}_{\text{rel}}([\![\texttt{t}_1]\!] \text{ op } [\![\texttt{t}_2]\!])$—or the corresponding p-box as described in Sect. 4.5. We then use Theorem 4 to justify evaluating the multiplication operation in Eq. (11) *independently*—that is to say by using [48]—since the roundoff process is very close to being uncorrelated to the process generating it. The validity of this assumption is also confirmed experimentally by the remarkable agreement of Monte-Carlo simulations with this analytical model.

We now introduce the algorithm for evaluating the model given in Eq. (11). The evaluation performs an in-order (LNR) traversal of the *Abstract Syntax Tree* (AST) of a computation given by our grammar, and it feeds the results to the parent level along the way. At each node, it computes the probabilistic range of the intermediate result using the probabilistic ranges computed for its children nodes (i.e., operands). We first determine whether the operands are independent or not (Ind? branch in the toolflow), and we either apply a cheaper (i.e., no SMT solver invocations) algorithm if they are independent (see below) or a more involved one (see Sect. 6.2) if they are not. We describe our methodology at a generic intermediate computation in the AST of the expression.

We consider two distributions X and Y discretized into DS-structures DS_X and DS_Y (Sect. 3.2), and we want to derive the DS-structure DS_Z for $Z = X \text{ op } Y$, op $\in \{+, -, \times, \div\}$. Together with the DS-structures of the operands, we also need the traces $trace_X$ and $trace_Y$ containing the history of the operations performed so far, one for each operand. A trace is constructed at each leaf of the AST with the input distributions and their range. It is then propagated to the parent level and populated at each node with the current operation. Such history traces are critical when dealing with dependent operations since they allow us to interrogate an SMT solver about the feasibility of the current operation, as we describe in the next section. When the operands are independent, we simply use the arithmetic operations on independent DS-structures [3].

6.2 Computing Probabilistic Ranges for Dependent Operands

When the operands are dependent, we start by assuming that the dependency is unknown. This assumption is sound because the dependency of the operation is included in the set of unknown dependencies, while the result of the operation is no longer a single distribution but a p-box. Due to this "unknown assumption", the CDFs of the output p-box are a very pessimistic over-approximation of the operation, i.e., they are far from each other. Our key insight is to use an

Algorithm 1. Dependent Operation $Z = X$ op Y

```
1: function DEP_OP($DS_X$, op , $DS_Y$, $trace_X$, $trace_Y$)
2:     $DS_Z = list()$
3:     for all $([x_1, x_2], p_x) \in DS_X$ do
4:         for all $([y_1, y_2], p_y) \in DS_Y$ do
5:             $[z_1, z_2] = [x_1, x_2]$ op $[y_1, y_2]$        ▷ operation between intervals
6:             $[z'_1, z'_2] = SMT.prune([z_1, z_2])$
7:             if $SMT.check(trace_X \wedge trace_Y \wedge [x_1, x_2] \wedge [y_1, y_2])$ is $SAT$ then
8:                 $p_Z$ = unknown-probability
9:             else
10:                $p_Z = 0$
11:            $DS_Z.append(([z'_1, z'_2], p_Z))$
12:    $trace_Z = trace_X \cup trace_Y \cup \{Z = X$ op $Y\}$
13:    return $DS_Z, trace_Z$
```

SMT solver to prune infeasible combinations of intervals from the input DS-structures, which prunes regions of zero probability from the output p-box. This probabilistic pruning using a solver squeezes together the CDFs of the output p-box, often resulting in a much more accurate over-approximation. With the solver, we move from an unknown to a *partially known* dependency between the operands. Currently, PAF supports the Z3 [17] and dReal [23] SMT solvers.

Algorithm 1 shows the pseudocode of our algorithm for computing the probabilistic output range (i.e., DS-structure) for dependent operands. When dealing with dependent operands, interval arithmetic (line 5) might not be as precise as in the independent case. Hence, we use an SMT solver to prune away any over-approximations introduced by interval arithmetic when computing with dependent ranges (line 6); this use of the solver is orthogonal to the one dealing with probabilities. On line 7, we check with an SMT solver whether the current combination of ranges $[x_1, x_2]$ and $[y_1, y_2]$ is compatible with the traces of the operands. If the query is satisfiable, the probability is strictly greater than zero but currently unknown (line 8). If the query is unsatisfiable, we assign a probability of zero to the range in DS_Z (line 10). Finally, we append a new range to the DS-structure DS_Z (line 11). Note that the loops are independent, and hence in our prototype implementation we run them in parallel.

After this algorithm terminates, we still need to assign probability values to all the unknown-probability ranges in DS_Z. Since we cannot assign an exact value, we compute a range of potential values $[p_{z_{min}}, p_{z_{max}}]$ instead. This computation is encoded as a *linear programming* routine exactly as in [3].

6.3 Computing Conditional Roundoff Error

The final step of our toolflow computes the conditional roundoff error by combining the symbolic affine arithmetic error form of the computation (see Sect. 5) with the probabilistic range analysis described above. The symbolic error form gets maximized conditioned on the results of all the intermediate operations

Algorithm 2. Conditional Roundoff Error Computation

```
1:  function COND_ERR($DSS, errorForm, confidence$)
2:      $allRanges = list()$
3:      for all $DS_i \in DSS$ do
4:          $focals = sorted(DS_i,\ key = prob,\ order = descending)$
5:          $accumulator = 0$
6:          $ranges = \emptyset$
7:          for all $([x_1, x_2], p_x) \in focals$ do
8:              $accumulator = accumulator + p_x$
9:              $ranges = ranges \cup [x_1, x_2]$
10:             if $accumulator \geq confidence$ then
11:                 $allRanges.append(ranges)$
12:                 break
13:     $error = maximize(errorForm, allRanges)$
14:     return $error$
```

landing in the given confidence interval (e.g., 99%) of their respective ranges (computed as described in the previous section). Note that conditioning only on the last operation of the computation tree (i.e., the AST root) would lead to extremely pessimistic over-approximation since all the outliers in the intermediate operations would be part of the maximization routine. This would lead to our tool PAF computing pessimistic error bounds typical of worst-case analyzers.

Algorithm 2 shows the pseudocode of the roundoff error computation algorithm. The algorithm takes as input a list DSS of DS-structures (one for each intermediate result range in the computation), the generated symbolic error form, and a confidence interval. It iterates over all intermediate DS-structures (line 3), and for each it determines the ranges needed to support the chosen confidence intervals (lines 4–12). In each iteration, it sorts the list of range-probability pairs (i.e., focal elements) of the current DS-structure by their probability value in a descending order (line 4). This is a heuristic that prioritizes the focal elements with most of the probability mass and avoids the unlikely outliers that cause large roundoff errors into the final error computation. With the help of an accumulator (line 8), we keep collecting focal elements (line 9) until the accumulated probability satisfies the confidence interval (line 10). Finally, we maximize the error form conditioned to the collected ranges of intermediate operations (line 13). The maximization is done using the rigorous global optimizer Gelpia [24].

7 Experimental Evaluation

We evaluate PAF (version 1.0.0) on the standard FPBench benchmark suite [11, 20] that uses the four basic operations we currently support $\{+, -, \times, \div\}$. Many of these benchmarks were also used in recent related work [36] that we compare against. The benchmarks come from a variety of domains: embedded software (*bsplines*), linear classifications (*classids*), physics computations (*dopplers*), filters (*filters*), controllers (*traincars*, *rigidBody*), polynomial approximations of functions (*sine*, *sqrt*), solving equations (*solvecubic*), and global optimizations (*trids*). Since FPBench has been primarily used for worst-case roundoff error analysis, the benchmarks come with ranges for input variables, but they do not specify input distributions. We instantiate the benchmarks with three well-known distributions for all the inputs: uniform, standard normal distribution, and double exponential (Laplace) distribution with $\sigma = 0.01$ which we will call 'exp'. The normal and exp distributions get truncated to the given range. We assume single-precision floating-point format for all operands and operations.

To assess the accuracy and performance of PAF, we compare it with PrAn (commit 7611679 [10]), the current state-of-the-art tool for automated analysis of probabilistic roundoff errors [36]. PrAn currently supports only uniform and normal distributions. We run all 6 tool configurations and report the best result for each benchmark. We fix the number of intervals in each discretization to 50 to match PrAn. We choose 99% as the confidence interval for the computation of our conditional roundoff error (Sect. 6.3) and of PrAn's probabilistic error. We also compare our probabilistic error bounds against FPTaylor (commit efbbc83 [21]), which performs worst-case roundoff error analysis, and hence it does not take into account the distributions of the input variables. We ran our experiments in parallel on a 4-socket 2.2 GHz 8-core Intel Xeon E5-4620 machine.

Table 2 compares roundoff errors reported by PAF, PrAn, and FPTaylor. PAF outperforms PrAn by computing tighter probabilistic error bounds on almost all benchmarks, occasionally by orders of magnitude. In the case of uniform input distributions, PAF provides tighter bounds for 24 out of 27 benchmarks, for 2 benchmarks the bounds from PrAn are tighter, while for *sqrt* they are the same. In the case of normal input distributions, PAF provides tighter bounds for all the benchmarks. Unlike PrAn, PAF supports probabilistic output range analysis as well. We present these results in the extended version [7].

In Table 2, of particular interest are benchmarks (10 for normal and 18 for exp) where the error bounds generated by PAF for the 99% confidence interval are at least an order of magnitude tighter than the worst-case bounds generated by FPTaylor. For such a benchmark and input distribution, PAF's results inform a user that there is an opportunity to optimize the benchmark (e.g., by reducing precision of floating-point operations) if their use-case can handle at most 1% of inputs generating roundoff errors that exceed a user-provided bound. FPTaylor's results, on the other hand, do not allow for a user to explore such fine-grained trade-offs since they are worst-case and do not take probabilities into account.

In general, we see a gradual reduction of the errors transitioning from uniform to normal to exp. When the input distributions are uniform, there is a significant chance of generating a roundoff error of the same order of magnitude as the worst-case error, since all inputs are equally likely. The standard normal distribution concentrates more than 99% of probability mass in the interval $[-3, 3]$, resulting in the *long tail* phenomenon, where less than 0.5% of mass spreads in the interval $[3, \infty]$. When the normal distribution gets truncated in a neighborhood of zero (e.g., $[0, 1]$ for *bsplines* and *filters*) nothing changes with respect to the uniform case—there is still a high chance of committing errors close to the worst-case.

Table 2. Roundoff error bounds reported by PAF, PrAn, and FPTaylor given uniform (uni), normal (norm), and Laplace (exp) input distributions. We set the confidence interval to 99% for PAF and PrAn, and mark the smallest reported roundoff errors for each benchmark in bold. Asterisk (*) highlights a difference of more than one order of magnitude between PAF and FPTaylor.

Benchmark	Uniform		Normal		Exp	FpTaylor
	PAF	PrAn	PAF	PrAn	PAF	
bspline0	**5.71e−08**	6.12e−08	**5.71e−08**	6.12e−08	**5.71e−08**	5.72e−08
bspline1	**1.86e−07**	2.08e−07	**1.86e−07**	2.08e−07	**6.95e−08**	1.93e−07
bspline2	**1.94e−07**	2.13e−07	**1.94e−07**	2.13e−07	**2.11e−08**	2.10e−07
bspline3	**4.22e−08**	4.65e−08	**4.22e−08**	4.65e−08	**7.62e−12***	4.22e−08
classids0	**6.93e−06**	8.65e−06	**4.45e−06**	8.64e−06	**1.70e−06**	6.85e−06
classids1	**3.71e−06**	4.63e−06	**2.68e−06**	4.62e−06	**7.62e−07**	3.62e−06
classids2	**5.23e−06**	7.32e−06	**3.85e−06**	7.32e−06	**1.46e−06**	5.15e−06
doppler1	**7.95e−05**	1.17e−04	**5.08e−07***	1.17e−04	**4.87e−07***	6.10e−05
doppler2	**1.43e−04**	2.45e−04	**6.61e−07***	2.45e−04	**6.28e−07***	1.11e−04
doppler3	**4.55e−05**	5.12e−05	**9.11e−07***	5.12e−05	**8.95e−07***	3.41e−05
filter1	**1.25e−07**	2.03e−07	**1.25e−07**	2.03e−07	**5.43e−09***	1.25e−07
filter2	**7.93e−07**	1.01e−06	**6.13e−07**	1.01e−06	**2.90e−08***	7.93e−07
filter3	**2.34e−06**	2.86e−06	**2.05e−06**	2.87e−06	**1.09e−07***	2.23e−06
filter4	**4.15e−06**	5.20e−06	**4.15e−06**	5.20e−06	**4.61e−07**	3.81e−06
rigidbody1	1.74e−04	**1.58e−04**	**6.14e-06***	1.58e−04	**4.80e−07***	1.58e−04
rigidbody2	1.96e−02	**9.70e−03**	**5.99e-05***	9.70e−03	**9.55e−07***	1.94e−02
sine	**2.37e−07**	2.40e−07	**2.37e−07**	2.40e−07	**1.49e−08***	2.38e−07
solvecubic	**1.78e−05**	1.83e−05	**6.84e−06**	1.83e−05	**2.76e−06**	1.60e−05
sqrt	**1.54e−04**	**1.54e-04**	**1.10e−06***	1.54e−04	**2.46e−07***	1.51e−04
traincars1	**1.76e−03**	1.96e−03	**8.26e−04**	1.96e−03	**4.50e−04**	1.74e−03
traincars2	**1.04e−03**	1.36e−03	**3.61e−04**	1.36e−03	**2.83e−05***	9.46e−04
traincars3	**1.75e−02**	2.29e−02	**9.56e−03**	2.29e−02	**8.95e−04***	1.80e−02
traincars4	**1.81e−01**	2.30e−01	**8.87e−02**	2.30e−01	**7.33e−03***	1.81e−01
trid1	**6.01e−03**	6.03e−03	**1.58e−05***	6.03e−03	**1.58e−05***	6.06e−03
trid2	**1.03e−02**	1.17e−02	**2.42e−05***	1.17e−02	**2.43e−05***	1.03e−02
trid3	**1.75e−02**	1.95e−02	**6.80e−05***	1.95e−02	**6.77e−05***	1.75e−02
trid4	**2.69e−02**	2.88e−02	**2.64e−04***	3.03e−02	**2.64e−04***	2.66e−02

However, when the normal distribution gets truncated to a wider range (e.g., $[-100, 100]$ for *trids*), then the outliers causing large errors are very rare events, not included in the 99% confidence interval. The exponential distribution further compresses the 99% probability mass in the tiny interval $[-0.01, 0.01]$, so the long tails effect is common among all the benchmarks.

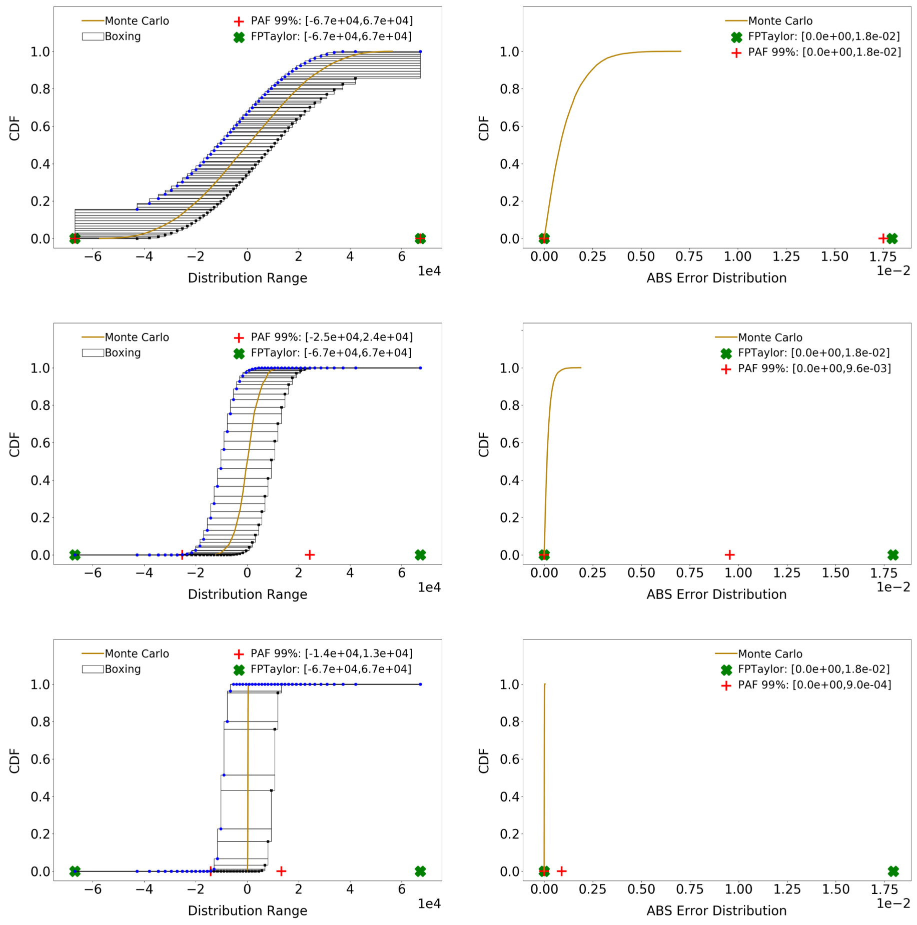


Fig. 4. CDFs of the range (left) and error (right) distributions for the benchmark *traincars3* for uniform (top), normal (center), and exp (bottom).

The runtimes of PAF vary between 10 min for small benchmarks, such as *bsplines*, to several hours for benchmarks with more than 30 operations, such as *trid4*; they are always less than two hours, except for *trids* with 11 h and *filters* with 6 h. The runtime of PAF is usually dominated by Z3 invocations, and the long runtimes are caused by numerous Z3 timeouts that the respective benchmarks induce. The runtimes of PrAn are comparable to PAF since they are always less than two hours, except for *trids* with 3 h, *sqrt* with 3 h, and *sine* with 11 h. Note that neither PAF nor PrAn are memory intensive.

To assess the quality of our rigorous (i.e., sound) results, we implement Monte Carlo sampling to generate both roundoff error and output range distributions. The procedure consists of randomly sampling from the provided input distributions, evaluating the floating-point computation in both the specified and high-precision (e.g., double-precision) floating-point regimes to measure the roundoff error, and finally partitioning the computed errors into bins to get an approximation (i.e., histogram) of the PDF. Of course, Monte Carlo sampling does not provide rigorous bounds, but is a useful tool to assess how far the rigorous bounds computed statically by PAF are from an empirical measure of the error.

Figure 4 shows the effects of the input distributions on the output and roundoff error ranges of the *traincars3* benchmark. In the error graphs (right column), we show the Monte Carlo sampling evaluation (yellow line) together with the error bounds from PAF with 99% confidence interval (red plus symbol) and FPTaylor's worst-case bounds (green crossmark). In the range graphs (left column), we also plot PAF's p-box over-approximations. We can observe that in the case of uniform inputs the computed p-boxes overlap at the extrema of the output range. This phenomenon makes it impossible to distinguish between 99% and 100% confidence intervals, and hence as expected the bound reported by PAF is almost identical to FPTaylor's. This is not the case for normal and exponential distributions, where PAF can significantly improve both the output range and error bounds over FPTaylor. This again illustrates how pessimistic the bounds from worst-case tools can be when the information about the input distributions is not taken into account. Finally, the graphs illustrate how the p-boxes and error bounds from PAF follow their respective empirical estimations.

8 Related Work

Our work draws inspiration from *probabilistic affine arithmetic* [3,4], which aims to bound probabilistic uncertainty propagated through a computation; a similar goal to our probabilistic range analysis. This was recently extended to polynomial dependencies [45]. On the other hand, PAF detects any non-linear dependency supported by the SMT solver. While these approaches show how to bound moments, we do not consider moments but instead compute conditional roundoff error bounds, a concern specific to the analysis of floating-point computations. Finally, the concentration of measure inequalities [4,45] provides bounds for (possibly very large) problems that can be expressed as sums of random variables, for example multiple increments of a noisy dynamical system, but are unsuitable for typical floating-point computations (such as FPBench benchmarks).

The most similar approach to our work is the recent static probabilistic roundoff error analysis called PrAn [36]. PrAn also builds on [3], and inherits the same limitations in dealing with dependent operations. Like us, PrAn hinges on a discretization scheme that builds p-boxes for both the input and error distributions and propagates them through the computation. The question of how these p-boxes are chosen is left open in the PrAn approach. In contrast, we take the input variables to be user-specified random variables, and show how the distribution of each error term can be computed directly and exactly from the random variables generating it (Sect. 4). Furthermore, unlike PrAn, PAF leverages the non-correlation between random variables and the corresponding error distribution (Sect. 4.4). Thus, PAF performs the rounding in Eq. (3) as an *independent* operation. Putting these together leads to PAF computing tighter probabilistic roundoff error bounds than PrAn, as our experiments show (Sect. 7).

The idea of using a probabilistic model of rounding errors to analyze *deterministic* computations can be traced back to Von Neumann and Goldstine [51]. Parker's so-called 'Monte Carlo arithmetic' [41] is probably the most detailed description of this approach. We, however, consider *probabilistic* computations. For this reason, the famous critique of the probabilistic approach to roundoff errors [29] does not apply to this work. Our preliminary report [9] presents some early ideas behind this work, including Eqs. (5) and (7) and a very rudimentary range analysis. However, this early work manipulated distributions *unsoundly*, could not handle any repeated variables, and did not provide any roundoff error analysis. Recently, probabilistic roundoff error models have also been investigated using the concentration of measure inequalities [27,28]. Interestingly, this means that the distribution of errors in Eq. (3) can be left almost completely unspecified. However, as in the case of related work from the beginning of this section [4,45], concentration inequalities are very ill-suited to the applications captured by the FPBench benchmark suite.

Worst-case analysis of roundoff errors has been an active research area with numerous published approaches [12–16,18,22,33,35,37,38,46,47,50]. Our symbolic affine arithmetic used in PAF (Sect. 5) evolved from rigorous affine arithmetic [14] by keeping the coefficients of the noise terms symbolic, which often leads to improved precision. These symbolic terms are very similar to the first-order Taylor approximations of the roundoff error expressions used in FPTaylor [46,47]. Hence, PAF with the 100% confidence interval leads to the same worst-case roundoff error bounds as computed by FPTaylor (Sect. 7).

Acknowledgments. We thank Ian Briggs and Mark Baranowski for their generous and prompt support with Gelpia. We also thank Alexey Solovyev for his detailed feedback and suggestions for improvements. Finally, we thank the anonymous reviewers for their insightful reviews that improved our final version, and program chairs for carefully guiding the review process.

References

1. Bornholt, J.: Abstractions and techniques for programming with uncertain data. Undergraduate honours thesis, Australian National University (2013)
2. Bornholt, J., Mytkowicz, T., McKinley, K.S.: Uncertain <T>: a first-order type for uncertain data. In: ASPLOS (2014)
3. Bouissou, O., Goubault, E., Goubault-Larrecq, J., Putot, S.: A generalization of p-boxes to affine arithmetic. Computing **89**, 189–201 (2012). https://doi.org/10.1007/s00607-011-0182-8
4. Bouissou, O., Goubault, E., Putot, S., Chakarov, A., Sankaranarayanan, S.: Uncertainty propagation using probabilistic affine forms and concentration of measure inequalities. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 225–243. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_13

5. Chiang, W.F., Baranowski, M., Briggs, I., Solovyev, A., Gopalakrishnan, G., Rakamarić, Z.: Rigorous floating-point mixed-precision tuning. In: POPL (2017)
6. Comba, J.L.D., Stolfi, J.: Affine arithmetic and its applications to computer graphics. In: SIBGRAPI (1993)
7. Constantinides, G., Dahlqvist, F., Rakamarić, Z., Salvia, R.: Rigorous roundoff error analysis of probabilistic floating-point computations (2021). arXiv:2105.13217
8. Dahlqvist, F., Kozen, D.: Semantics of higher-order probabilistic programs with conditioning. In: POPL (2019)
9. Dahlqvist, F., Salvia, R., Constantinides, G.A.: A probabilistic approach to floating-point arithmetic. In: ASILOMAR (2019). Non-peer-reviewed extended abstract
10. Daisy. https://github.com/malyzajko/daisy
11. Damouche, N., Martel, M., Panchekha, P., Qiu, C., Sanchez-Stern, A., Tatlock, Z.: Toward a standard benchmark format and suite for floating-point analysis. In: Bogomolov, S., Martel, M., Prabhakar, P. (eds.) NSV 2016. LNCS, vol. 10152, pp. 63–77. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-54292-8_6
12. Darulova, E., Izycheva, A., Nasir, F., Ritter, F., Becker, H., Bastian, R.: Daisy - framework for analysis and optimization of numerical programs (tool paper). In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10805, pp. 270–287. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89960-2_15
13. Darulova, E., Kuncak, V.: Trustworthy numerical computation in Scala. In: OOPSLA (2011)
14. Darulova, E., Kuncak, V.: Sound compilation of reals. In: POPL (2014)
15. Das, A., Briggs, I., Gopalakrishnan, G., Krishnamoorthy, S., Panchekha, P.: Scalable yet rigorous floating-point error analysis. In: SC (2020)
16. Daumas, M., Melquiond, G.: Certification of bounds on expressions involving rounded operators. ACM Trans. Math. Softw. **37**, 1–20 (2010)
17. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
18. Delmas, D., Goubault, E., Putot, S., Souyris, J., Tekkal, K., Védrine, F.: Towards an industrial use of FLUCTUAT on safety-critical avionics software. In: Alpuente, M., Cook, B., Joubert, C. (eds.) FMICS 2009. LNCS, vol. 5825, pp. 53–69. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04570-7_6
19. Ferson, S., Kreinovich, V., Grinzburg, L., Myers, D., Sentz, K.: Constructing probability boxes and dempster-shafer structures. Technical report, Sandia National Lab (2015)
20. FPBench: standards and benchmarks for floating-point research. https://fpbench.org
21. FPTaylor. https://github.com/soarlab/fptaylor
22. Fu, Z., Bai, Z., Su, Z.: Automated backward error analysis for numerical code. In: OOPSLA (2015)
23. Gao, S., Kong, S., Clarke, E.M.: dReal: an SMT solver for nonlinear theories over the reals. In: Bonacina, M.P. (ed.) CADE 2013. LNCS (LNAI), vol. 7898, pp. 208–214. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38574-2_14
24. Gelpia: a global optimizer for real functions. https://github.com/soarlab/gelpia
25. Glasserman, P.: Monte Carlo Methods in Financial Engineering. Springer, Heidelberg (2013). https://doi.org/10.1007/978-0-387-21617-1
26. Higham, N.J.: Accuracy and Stability of Numerical Algorithms. SIAM (2002)
27. Higham, N.J., Mary, T.: A New Approach to Probabilistic Rounding Error Analysis. SISC (2019)

28. Ipsen, I.C.F., Zhou, H.: Probabilistic error analysis for inner products. SIMAX (2019)
29. Kahan, W.: The improbability of probabilistic error analyses for numerical computations (1996)
30. Kajiya, J.T.: The rendering equation. In: SIGGRAPH (1986)
31. Kozen, D.: Semantics of probabilistic programs. In: JCSS (1981)
32. Landau, D.P., Binder, K.: A Guide to Monte Carlo Simulations in Statistical Physics. Cambridge University Press, Cambridge (2014)
33. Lee, W., Sharma, R., Aiken, A.: Verifying bit-manipulations of floating-point. In: PLDI (2016)
34. Lepage, G.P.: VEGAS – an adaptive multi-dimensional integration program. Technical report, Cornell (1980)
35. Linderman, M.D., Ho, M., Dill, D.L., Meng, T.H., Nolan, G.P.: Towards program optimization through automated analysis of numerical precision. In: CGO (2010)
36. Lohar, D., Prokop, M., Darulova, E.: Sound probabilistic numerical error analysis. In: IFM (2019)
37. Magron, V., Constantinides, G., Donaldson, A.: Certified roundoff error bounds using semidefinite programming. TOMS **43**, 1–31 (2017)
38. Martel, M.: RangeLab: a static-analyzer to bound the accuracy of finite-precision computations. In: SYNASC (2011)
39. Microprocessor Standards Committee of the IEEE Computer Society: IEEE Standard for Floating-Point Arithmetic (2019)
40. Moore, R.E.: Interval Analysis. Prentice-Hall, Hoboken (1966)
41. Parker, D.S., Pierce, B., Eggert, P.R.: Monte Carlo arithmetic: how to gamble with floating point and win. Comput. Sci. Eng. **2**, 58–68 (2000)
42. Press, W.H., Farrar, G.R.: Recursive stratified sampling for multidimensional Monte Carlo integration. Comput. Phys. **4**, 190–195 (1990)
43. Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P.: Numerical Recipes in C. Cambridge University Press, Cambridge (1988)
44. Rothschild, M., Stiglitz, J.E.: Increasing risk: I. A definition. J. Econ. Theory **2**, 225–243 (1970)
45. Sankaranarayanan, S., Chou, Y., Goubault, E., Putot, S.: Reasoning about uncertainties in discrete-time dynamical systems using polynomial forms. In: NeurIPS (2020)
46. Solovyev, A., Baranowski, M.S., Briggs, I., Jacobsen, C., Rakamarić, Z., Gopalakrishnan, G.: Rigorous estimation of floating-point round-off errors with Symbolic Taylor Expansions. TOPLAS **41**, 1–39 (2018)
47. Solovyev, A., Jacobsen, C., Rakamarić, Z., Gopalakrishnan, G.: Rigorous estimation of floating-point round-off errors with Symbolic Taylor Expansions. In: FM (2015)
48. Malik, H.J.: The Algebra of Random Variables (Springer, MD). Wiley (1979)
49. Stolfi, J., Figueiredo, L.H.D.: Self-validated numerical methods and applications. In: IMPA (1997)
50. Titolo, L., Feliú, M.A., Moscato, M., Muñoz, C.A.: An abstract interpretation framework for the round-off error analysis of floating-point programs. In: VMCAI (2018)
51. Von Neumann, J., Goldstine, H.H.: Numerical inverting of matrices of high order. Bull. Am. Math. Soc. **53**, 1021–1099 (1947)

Model-Free Reinforcement Learning for Branching Markov Decision Processes

Ernst Moritz Hahn[1], Mateo Perez[2], Sven Schewe[3], Fabio Somenzi[2], Ashutosh Trivedi[2](✉), and Dominik Wojtczak[3]

[1] University of Twente, Enschede, The Netherlands
[2] University of Colorado Boulder, Boulder, USA
ashutosh.trivedi@colorado.edu
[3] University of Liverpool, Liverpool, UK

Abstract. We study reinforcement learning for the optimal control of Branching Markov Decision Processes (BMDPs), a natural extension of (multitype) Branching Markov Chains (BMCs). The state of a (discrete-time) BMCs is a collection of entities of various types that, while spawning other entities, generate a payoff. In comparison with BMCs, where the evolution of a each entity of the same type follows the same probabilistic pattern, BMDPs allow an external controller to pick from a range of options. This permits us to study the best/worst behaviour of the system. We generalise model-free reinforcement learning techniques to compute an optimal control strategy of an unknown BMDP in the limit. We present results of an implementation that demonstrate the practicality of the approach.

1 Introduction

Branching Markov Chains (BMCs), also known as Branching Processes, are natural models of population dynamics and parallel processes. The state of a BMC consists of entities of various types, and many entities of the same type may coexist. Each entity can branch in a single step into a (possibly empty) set of entities of various types while disappearing itself. This assumption is natural, for instance, for annual plants that reproduce only at a specific time of the year, or for bacteria, which either split or die. An entity may spawn a copy of itself, thereby simulating the continuation of its existence.

The offspring of an entity is chosen at random among options according to a distribution that depends on the type of the entity. The type captures significant differences between entities. For example, stem cells are very different from

This work was supported by the Engineering and Physical Sciences Research Council through grant EP/P020909/1 and by the National Science Foundation through grant 2009022. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 101032464 (SyGaST) and 864075 (CAESAR).

A. Silva and K. R. M. Leino (Eds.): CAV 2021, LNCS 12760, pp. 651–673, 2021.
https://doi.org/10.1007/978-3-030-81688-9_30

regular cells; parallel processes may be interruptible or have different privileges. The type may reflect characteristics of the entities such as their age or size.

Although entities coexist, the BMC model assumes that there is no interaction between them. Thus, how an entity reproduces and for how long it lives is the same as if it were the only entity in the system. This assumption greatly improves the computational complexity of the analysis of such models and is appropriate when the population exists in an environment that has virtually unlimited resources to sustain its growth. This is a common situation that holds when a species has just been introduced into an environment, in an early stage of an epidemic outbreak, or when running jobs in cloud computing.

BMCs have a wide range of applications in modelling various physical phenomena, such as nuclear chain reactions, red blood cell formation, population genetics, population migration, epidemic outbreaks, and molecular biology. Many examples of BMC models used in biological systems are discussed in [12].

Branching Markov Decision Processes (BMDPs) extend BMCs by allowing a controller to choose the branching dynamics for each entity. This choice is modelled as nondeterministic, instead of random. This extension is analogous to how Markov Decision Processes (MDPs) generalise Markov chains (MCs) [24]. Allowing an external controller to select a mode of branching allows us to study the best/worst behaviour of the examined model.

As a motivating example, let us discuss a simple model of cloud computing. A computation may be divided into tasks in order to finish it faster, as each server may have different computational power. Since the computation of each task depends on the previous one, the total running time is the sum of the running times of each spawned task as well as the time needed to split and merge the result of each computation into the final solution. As we shall see, the execution of each task is not guaranteed to be successful and is subject to random delays. Specifically, let us consider the following model with two different types (T and S), and two actions (a_1 and a_2). This BMDP consists of the main task, T, that may be split (action a_1) into three smaller tasks, for simplicity assumed to be of the same type S, and this split and merger of the intermediate results takes 1 hour (1h). Alternatively (action a_2), we can execute the whole task T on the main server, but it will be slow (8 h). Task S can (action a_1) be run on a reliable server in 1.6 h or (action a_2) an unreliable one that finishes after 1 h (irrespective of whether or not the computation is completed successfully), but with a 40% chance we need to rerun this task due to the server crashing. We can represent this model formally as:

$$\begin{aligned} &T \xrightarrow{a_1} SSS \qquad [1\text{h}] \qquad && S \xrightarrow{a_1} \epsilon && [1.6\text{h}] \\ &T \xrightarrow{a_2} \epsilon \qquad [8\text{h}] \qquad && S \xrightarrow{a_2} 40\% : S \text{ or } 60\% : \epsilon && [1\text{h}] \end{aligned}$$

We would like to know the infimum of the expected running time (i.e. the expected running time when optimal decisions are made) of task T. In this case the optimal control is to pick action a_1 first and then actions a_1 for all tasks S with a total running time of 5.8 h. The expected running time when picking actions a_2 for S instead would be $1 + 3 \cdot 1/0.6 = 6$ [hours].

Let us now assume that the execution of tasks S for action a_1 may be interrupted with probability 30% by a task of higher priority (type H). Moreover, these H tasks may be further interrupted by tasks with even higher priority (to simplify matters, again modelled by type H). The computation time of T is prolonged by 0.1 h for each H spawned. Our model then becomes:

$$\begin{array}{llllllll} T \xrightarrow{a_1} SSS & [1\text{h}] & S \xrightarrow{a_1} 30\% : H \text{ or } 70\% : \epsilon & [1.6\text{h}] & H \xrightarrow{*} & 30\% : HH \text{ or} & \\ T \xrightarrow{a_2} \epsilon & [8\text{h}] & S \xrightarrow{a_2} 40\% : S \text{ or } 60\% : \epsilon & [1\text{h}] & & 70\% : \epsilon & [0.1\text{h}] \end{array}$$

As we shall see, the expected total running time of H can be calculated by solving the equation $x = 0.3(x + x) + 0.1$, which gives $x = 0.25$ [hour]. So the expected running time of S using action a_1 increases by $0.3 \cdot 0.25 = 0.075$ [hour]. This is enough for the optimal strategy of running S to become a_2. Note that if the probability of H being interrupted is at least 50% then the expected running time of H becomes ∞.

When dealing with a real-life process, it is hard to come up with a (probabilistic and controlled) model that approximates it well. This requires experts to analyse all possible scenarios and estimate the probability of outcomes in response to actions based on either complex calculations or the statistical analysis of sufficient observational data. For instance, it is hard to estimate the probability of an interrupt H occurring in the model above without knowing which server will run the task, its usual workload and statistics regarding the priorities of the tasks it executes. Even if we do this estimation well, unexpected or rare events may happen that would require us to recalibrate the model as we observe the system under our control.

Instead of building such a model explicitly first and fixing the probabilities of possible transitions in the system based on our knowledge of the system or its statistics, we advocate the use of reinforcement learning (RL) techniques [27] that were successfully applied to finding optimal control for finite-state Markov Decision Processes (MDPs). Q-learning [30] is a well-studied model-free RL approach to compute an optimal control strategy without knowing about the model apart from its initial state and the set of actions available in each of its states. It also has the advantage that the learning process converges to the optimal control while exploiting along the way what it already knows. While the formulation of the Q-learning algorithm for BMDPs is straightforward, the proof that it works is not. This is because, unlike the MDPs with discounted rewards for which the original Q-learning algorithm was defined, our model does not have an explicit contraction in each step, nor does boundedness of the optimal values or one-step updates hold. Similarly, one cannot generalise the result from [11] that estimates the time needed for the Q-learning algorithm to converge within ϵ of the optimal values with high probability for finite-state MDPs.

1.1 Related Work

The simplest model of BMCs are Galton-Watson processes [31], discrete-time models where all entities are of the same type. They date as far back as 1845 [14] and were used to explain why some aristocratic family surnames became extinct. The generalisation of this model to multiple types of entities was first studied in 1940s by Kolmogorov and Sevast'yanov [17]. For an overview of the results known for BMCs, see e.g. [13] and [12]. The precise computational complexity of decision problems about the probabilities of extinction of an arbitrary BMC was first established in [9]. The problem of checking if a given BMC terminates almost surely was shown in [5] to be strongly polynomial. The probability of acceptance of a run of a BMC by a deterministic parity tree automaton was studied in [4] and shown to be computable in PSPACE and in polynomial time for probabilities 0 or 1. In [16] a generalisation of the BMCs was considered that allowed for limited synchronisation of different tasks.

BMDPs, a natural generalisation of BMCs to a controlled setting, have been studied in the OR literature e.g., [23,26]. Hierarchical MDPs (HMDPs) [10] are a special case of BMDPs where there are no cycles in the offspring graph (equivalently, no cyclic dependency between types). BMDPs and HMDPs have found applications in manpower planning [29], controlled queuing networks [2, 15], management of livestock [20], and epidemic control [1,25], among others. The focus of these works was on optimising the expected average, or the discounted reward over a run of the process, or optimising the population growth rate. In [10] the decision problem whether the optimal probability of termination exceeds a threshold was studied: it was shown to be solvable in PSPACE and at least as hard as the square-root sum problem, but one can determine if the optimal probability is 0 or 1 in polynomial time. In [7], it was shown that the approximation of the optimal probability of extinction for BMDPs can be done in polynomial time. The computational complexity of computing the optimal expected total cost before extinction for BMDPs follows from [8] and was shown there to be computable in polynomial time via a linear program formulation. The problem of maximising the probability of reaching a state with an entity of a given type for BMDPs was studied in [6]. In [28] an extension of BMDPs with real-valued clocks and timing constraints on productions was studied.

1.2 Summary of the Results

We show that an adaptation of the Q-learning algorithm converges almost surely to the optimal values for BMDPs under mild conditions: all costs are positive and each Q-value is selected for update independently at random. We have implemented the proposed algorithm in the tool Mungojerrie [21] and tested its performance on small examples to demonstrate its efficiency in practice. To the best of our knowledge, this is the first time model-free RL has been used for the analysis of BMDPs.

2 Problem Definitions

2.1 Preliminaries

We denote by $\mathbb{N}$ the set of non-negative integers, by $\mathbb{R}$ the set of reals, by $\mathbb{R}_+$ the set of positive reals, and by $\mathbb{R}_{\geq 0}$ the set of non-negative reals. We let

$\widetilde{\mathbb{R}}_+ = \mathbb{R}_+ \cup \{\infty\}$, and $\widetilde{\mathbb{R}}_{\geq 0} = \mathbb{R}_{\geq 0} \cup \{\infty\}$. We denote by $|X|$ the cardinality of a set X and by X^* (X^ω) the set of all possible finite (infinite) sequences of elements of X. Finite sequences are also called lists.

Vectors and Lists. We use $\bar{x}, \bar{y}, \bar{c}$ to denote vectors and $\bar{x}_i$ or $\bar{x}(i)$ to denote its i-th entry. We let $\bar{0}$ denote a vector with all entries equal to 0; its size may vary depending on the context. Likewise $\bar{1}$ is a vector with all entries equal to 1. For vectors $\bar{x}, \bar{y} \in \widetilde{\mathbb{R}}^n_{\geq 0}$, $\bar{x} \leq \bar{y}$ means $x_i \leq y_i$ for every i, and $\bar{x} < \bar{y}$ means $\bar{x} \leq \bar{y}$ and $x_i \neq y_i$ for some i. We also make use of the infinity norm $\|\bar{x}\|_\infty = \max_i |\bar{x}(i)|$.

We use α, β, γ to denote finite lists of elements. For a list $\alpha = a_1, a_2, \ldots, a_k$ we write α_i for the i-th element a_i of list α and $|\alpha|$ for its length. For two lists α and β we write $\alpha \cdot \beta$ for their concatenation. The empty list is denoted by ϵ.

Probability Distributions. A *finite discrete probability distribution* over a countable set Q is a function $\mu : Q \to [0,1]$ such that $\sum_{q \in Q} \mu(q) = 1$ and its support set $supp(\mu) = \{q \in Q \mid \mu(q) > 0\}$ is finite. We say that $\mu \in \mathcal{D}(Q)$ is a *point distribution* if $\mu(q) = 1$ for some $q \in Q$.

Markov Decision Processes. Markov decision processes [24], are a well-studied formalism for systems exhibiting nondeterministic and probabilistic behaviour.

Definition 1. *A* Markov decision process *(MDP) is a tuple $\mathcal{M} = (S, A, p, c)$ where:*

- *S is the set of* states*;*
- *A is the set of* actions*;*
- *$p : S \times A \to \mathcal{D}(S)$ is a partial function called the* probabilistic transition function*; and*
- *$c : S \times A \to \mathbb{R}$ is the* cost function*.*

We say that an MDP $\mathcal{M}$ is *finite* (*discrete*) if both S and A are finite (countable). We write $A(s)$ for the set of actions available at s, i.e., the set of actions a for which $p(s,a)$ is defined. In an MDP $\mathcal{M}$, if the current state is s, then one of the actions in $A(s)$ is chosen nondeterministically. If the chosen action is a then the probability of reaching state $s' \in S$ in the next step is $p(s,a)(s')$ and the cost incurred is $c(s,a)$.

2.2 Branching Markov Decision Processes

We are now ready to define (multitype) BMDPs.

Definition 2. *A* branching Markov decision process *(BMDP) is a tuple $\mathcal{B} = (P, A, p, c)$ where:*

- *P is a finite set of* types*;*
- *A is a finite set of* actions*;*
- *$p : P \times A \to \mathcal{D}(P^*)$ is a partial function called the* probabilistic transition function *where every $\mathcal{D}(\cdot)$ is a finite discrete probability distribution; and*

– $c : P \times A \to \mathbb{R}_+$ *is the* cost function.

We write $A(q)$ for the set of actions available to an entity of type $q \in P$, i.e., the set of actions a for which $p(q, a)$ is defined. A *Branching Markov Chain (BMC)* is simply a BMDP with just one action available for each type.

Let us first describe informally how BMDPs evolve. A state of a BMDP $\mathcal{B}$ is a list of elements of P that we call *entities.* A BMDP starts at some initial configuration, $\alpha^0 \in P^*$, and the controller picks for one of the entities one of the actions available to an entity of its type. In the new configuration α^1, this one entity is replaced by the list of new entities that it spawned. This list is picked according to the probability distribution $p(q, a)$ that depends both on the type of the entity, q, and the action, a, performed on it by the controller. The process proceeds in the same manner from α^1, moving to α^2, and from there to α^3, etc. Once the state ϵ is reached, i.e., when no entities are present in the system, the process stays in that state forever.

Definition 3 (Semantics of BMDP). *The semantics of a BMDP* $\mathcal{B} = (P, A, p, c)$ *is an MDP* $\mathcal{M}_\mathcal{B} = (States_\mathcal{B}, Actions_\mathcal{B}, Prob_\mathcal{B}, Cost_\mathcal{B})$ *where:*

– $States_\mathcal{B} = P^*$ *is the set of states;*
– $Actions_\mathcal{B} = \mathbb{N} \times A$ *is the set of actions;*
– $Prob_\mathcal{B} : States_\mathcal{B} \times Actions_\mathcal{B} \to \mathcal{D}(States_\mathcal{B})$ *is the probabilistic transition function such that, for* $\alpha \in States_\mathcal{B}$ *and* $(i, a) \in Actions_\mathcal{B}$*, we have that* $Prob_\mathcal{B}(\alpha, (i, a))$ *is defined when* $i \le |\alpha|$ *and* $a \in A(\alpha_i)$*; moreover*

$$Prob_\mathcal{B}(\alpha, (i, a))(\alpha_1 \ldots \alpha_{i-1} \cdot \beta \cdot \alpha_{i+1} \ldots) = p(\alpha_i, a)(\beta),$$

for every $\beta \in P^*$ *and* 0 *in all other cases.*
– $Cost_\mathcal{B} : States_\mathcal{B} \times Actions_\mathcal{B} \to \mathbb{R}_+$ *is the cost function such that*

$$Cost_\mathcal{B}(\alpha, (i, a)) = c(\alpha_i, a).$$

For a given BMDP $\mathcal{B}$ and states $\alpha \in States_\mathcal{B}$, we denote by $Actions_\mathcal{B}(\alpha)$ the set of actions $(i, a) \in Actions_\mathcal{B}$, for which $Prob_\mathcal{B}(\alpha, (i, a))$ is defined.

Note that our semantics of BMDPs assumes an explicit listing of all the entities in a particular order similar to [10]. One could, instead, define this as a multi-set or simply a vector just counting the number of occurrences of each entity as in [23]. As argued in [10], all these models are equivalent to each other. Furthermore, we assume that the controller expands a single entity of his choice at the time rather all of them being expanded simultaneously. As argued in [32], that makes no difference for the optimal values of the expected total cost that we study in this paper, provided that all transitions' costs are positive.

2.3 Strategies

A *path* of a BMDP $\mathcal{B}$ is a finite or infinite sequence

$$\pi = \alpha^0, ((i_1, a_1), \alpha^1), ((i_2, a_2), \alpha^2), ((i_3, a_3), \alpha^3), \ldots \\ \in States_\mathcal{B} \times ((Actions_\mathcal{B} \times States_\mathcal{B})^* \cup (Actions_\mathcal{B} \times States_\mathcal{B})^\omega),$$

consisting of the initial state and a finite or infinite sequence of action and state pairs, such that $Prob_{\mathcal{B}}(\alpha^j, (i_j, a_j))(\alpha^{j+1}) > 0$ for any $0 \leq j \leq |\pi|$, where $|\pi|$ is the number of actions taken during path π. ($|\pi| = \infty$ if the path is infinite.) For a path π, we denote by $\pi_{A(j)} = (i_j, a_j)$ the j-th action taken along path π, by $\pi_{S(j)} (= \alpha^j)$ the j-th state visited, where $\pi_{S(0)} (= \alpha^0)$ is the initial state, and by $\pi(j) (= \alpha^0, ((i_1, a_1), \alpha^1), \ldots, ((i_j, a_j), \alpha^j))$ the first j action-state pairs of π.

We call a path of infinite (finite) length a *run* (*finite path*). We write $Runs_{\mathcal{B}}$ ($FPath_{\mathcal{B}}$) for the sets of all runs (finite paths) and $Runs_{\mathcal{B},\alpha}$ ($FPath_{\mathcal{B},\alpha}$) for the sets of all runs (finite paths) that start at a given initial state $\alpha \in States_{\mathcal{B}}$, i.e., paths π with $\pi_{S(0)} = \alpha$. We write $last(\pi)$ for the last state of a finite path π.

A *strategy* in BMDP $\mathcal{B}$ is a function $\sigma : FPath_{\mathcal{B}} \to \mathcal{D}(Actions_{\mathcal{B}})$ such that, for all $\pi \in FPath_{\mathcal{B}}$, $supp(\sigma(\pi)) \subseteq Actions_{\mathcal{B}}(last(\pi))$. We write $\Sigma_{\mathcal{B}}$ for the set of all strategies. A strategy is called *static*, if it always applies an action to the first entity in any state and for all entities of the same type in any state it picks the same action. A static strategy τ is essentially a function of the form $\sigma : P \to A$, i.e., for an arbitrary $\pi \in FPath_{\mathcal{B}}$, we have $\tau(\pi) = (1, \sigma(last(\pi)_1))$ whenever $last(\pi) \neq \epsilon$.

A strategy $\sigma \in \Sigma_{\mathcal{B}}$ and an initial state α induce a probability measure over the set of runs of BMDP $\mathcal{B}$ in the following way: the basic open sets of $Runs_{\mathcal{B}}$ are of the form $\pi \cdot (Actions_{\mathcal{B}} \times States_{\mathcal{B}})^\omega$, where $\pi \in FPath_{\mathcal{B}}$, and the measure of this open set is equal to $\prod_{i=0}^{|\pi|-1} \sigma(\pi(i))(\pi_{A(i+1)}) \cdot Prob_{\mathcal{B}}(\pi_{S(i)}, \pi_{A(i+1)})(\pi_{S(i+1)})$ if $\pi_{S(0)} = \alpha$ and equal to 0 otherwise. It is a classical result of measure theory that this extends to a unique measure over all Borel subsets of $Runs_{\mathcal{B}}$ and we will denote this measure by $P^\sigma_{\mathcal{B},\alpha}$.

Let $f : Runs_{\mathcal{B}} \to \widetilde{\mathbb{R}}_+$ be a function measurable with respect to $P^\sigma_{\mathcal{B},\alpha}$. The expected value of f under strategy σ when starting at α is defined as $\mathbb{E}^\sigma_{\mathcal{B},\alpha}\{f\} = \int_{Runs_{\mathcal{B}}} f \; \mathrm{d}P^\sigma_{\mathcal{B},\alpha}$ (which can be ∞ even if the probability that the value of f is infinite is 0). The infimum expected value of f in $\mathcal{B}$ when starting at α is defined as $\mathcal{V}_*(\alpha)(f) = \inf_{\sigma \in \Sigma_{\mathcal{B}}} \mathbb{E}^\sigma_{\mathcal{B},\alpha}\{f\}$. A strategy, $\widehat{\sigma}$, is said to be optimal if $\mathbb{E}^{\widehat{\sigma}}_{\mathcal{B},\alpha}\{f\} = \mathcal{V}_*(\alpha)(X)$ and ε-optimal if $\mathbb{E}^{\widehat{\sigma}}_{\mathcal{B},\alpha}\{f\} \leq \mathcal{V}_*(\alpha)(f) + \varepsilon$. Note that ε-optimal strategies always exists by definition. We omit the subscript $\mathcal{B}$, e.g., in $States_{\mathcal{B}}$, $\Sigma_{\mathcal{B}}$, etc., when the intended BMDP is clear from the context.

For a given BMDP $\mathcal{B}$ and $N \geq 0$ we define $\mathrm{Total}_N(\pi)$, the cumulative cost of a run π after N steps, as $\mathrm{Total}_N(\pi) = \sum_{i=0}^{N-1} Cost(\pi_{S(i)}, \pi_{A(i+1)})$. For a configuration $\alpha \in States$ and a strategy $\sigma \in \Sigma$, let $\mathrm{ETotal}_N(\mathcal{B}, \alpha, \sigma)$ be the *N-step expected total cost* defined as $\mathrm{ETotal}_N(\mathcal{B}, \alpha, \sigma) = \mathbb{E}^\sigma_{\mathcal{B},\alpha}\big\{\mathrm{Total}_N\big\}$ and the *expected total cost* be $\mathrm{ETotal}_*(\mathcal{B}, \alpha, \sigma) = \lim_{N \to \infty} \mathrm{ETotal}_N(\mathcal{B}, \alpha, \sigma)$. This last value can potentially be ∞. For each starting state α, we compute the *optimal expected cost* over all strategies of a BMDP starting at α, denoted by $\mathrm{ETotal}_*(\mathcal{B}, \alpha)$, i.e.,

$$\mathrm{ETotal}_*(\mathcal{B}, \alpha) = \inf_{\sigma \in \Sigma_{\mathcal{B}}} \mathrm{ETotal}(\mathcal{B}, \alpha, \sigma).$$

As we are going to prove in Theorem 4.b that, for any $\alpha \in \mathit{States}$, we have

$$\mathrm{ETotal}_*(\mathcal{B}, \alpha) = \sum_{i=1}^{|\alpha|} \mathrm{ETotal}_*(\mathcal{B}, \alpha_i).$$

This justifies focusing on this value for initial states that consist of a single entity only, as we will do in the following section.

3 Fixed Point Equations

Following [8], we define here a linear equation system with a minimum operator whose *Least Fixed Point* solution yields the desired optimal values for each type of a BMDP with non-negative costs. This system generalises the Bellman's equations for finite-state MDPs. We use a variable x_q for each unknown $\mathrm{ETotal}_*(\mathcal{B}, q)$ where $q \in P$. Let $\bar{x}$ be the vector of all x_q, where$q \in P$. The system has one equation of the form $x_q = F_q(\bar{x})$ for each type $q \in P$, defined as

$$x_q = \min_{a \in A(q)} \Big(c(q,a) + \sum_{\alpha \in P^*} p(q,a)(\alpha) \sum_{i \le |\alpha|} x_{\alpha_i} \Big) \ . \tag{$\spadesuit$}$$

We denote the system in vector form by $\bar{x} = F(\bar{x})$. Given a BMDP, we can easily construct its associated system in linear time. Let $\bar{c}^* \in \widetilde{\mathbb{R}}^n_{\ge 0}$ denote the n-dimensional vector of $\mathrm{ETotal}_*(\mathcal{B}, q)$'s where $n = |P|$. Let us define $\bar{x}^0 = \bar{0}$, $\bar{x}^{k+1} = F^{k+1}(\bar{0}) = F(\bar{x}^k)$, for $k \ge 0$.

Theorem 4. *The following hold:*

(a) The map $F : \widetilde{\mathbb{R}}^n_{\ge 0} \to \widetilde{\mathbb{R}}^n_{\ge 0}$ is monotone and continuous (and so $\bar{0} \le \bar{x}^k \le \bar{x}^{k+1}$ for all $k \ge 0$).
(b) $\bar{c}^* = F(\bar{c}^*)$.
(c) For all $k \ge 0$, $\bar{x}^k \le \bar{c}^*$.
(d) For all $\bar{c}' \in \widetilde{\mathbb{R}}^n_{\ge 0}$, if $\bar{c}' = F(\bar{c}')$, then $\bar{c}^* \le \bar{c}'$.
(e) $\bar{c}^* = \lim_{k\to\infty} \bar{x}^k$.

Proof.

(a) All equations in the system $F(x)$ are minimum of linear functions with non-negative coefficients and constants, and hence monotonicity and continuity are preserved.
(b) It suffices to show that once action a is taken when starting with a single entity q and, as a result, q is replaced by α with probability $p(q,a)(\alpha)$, then the expected total cost is equal to:

$$c(q,a) + \sum_{i \le |\alpha|} \mathrm{ETotal}_*(\mathcal{B}, \alpha_i) \ . \tag{$\clubsuit$}$$

This is because then the expected total cost of picking action a when at q is just a weighted sum of these expressions with weights $p(q,a)(\alpha)$ for offspring α. And finally, to optimise the cost, one would pick an action a with the smallest such expected total cost showing that

$$\mathrm{ETotal}_*(\mathcal{B},q) = \min_{a \in A(q)} \Big(c(q,a) + \sum_{\alpha \in P^*} p(q,a)(\alpha) \sum_{i \leq |\alpha|} \mathrm{ETotal}_*(\mathcal{B},\alpha_i) \Big)$$

indeed holds.

Now, to show (♣), consider an ϵ-optimal strategy σ_i for a BMDP that starts at α_i. It can easily be composed into a strategy σ that starts at α just by executing σ_1 first until all descendants of α_1 die out, before moving on to σ_2, etc. If one of these strategies, σ_i, never stops executing then, due to the assumption that all costs are positive, the expected total cost when starting with α_i has to be infinite and so has to be the overall cost when starting with α (as all descendants of α_i have to die out before the overall process terminates), so (♣) holds. This shows that $c(q,a) + \sum_{i \leq |\alpha|} \mathrm{ETotal}_*(\mathcal{B}, x_{\alpha_i})$ can be achieved when starting at α. At the same time, we cannot do better because that would imply the existence of a strategy σ' for one of the entities σ_j with a better cost than its optimal cost $\mathrm{ETotal}_*(\mathcal{B},\alpha_j)$.

(c) Since $\bar{x}^0 = \bar{0} \leq \bar{c}^*$ and due to (b), it follows by repeated application of F to both sides of this inequality that $\bar{x}^k \leq F(\bar{c}^*) = \bar{c}^*$, for all $k \geq 0$.

(d) Consider any fixed point $\bar{c}'$ of the equation system $F(\bar{x})$. We will prove that $\bar{c}^* \leq \bar{c}'$. Let us denote by σ' a static strategy that picks for each type an action with the minimum value of operator F in $\bar{c}'$, i.e., for each entity q we choose $\sigma'(q) = \arg\min_{a \in A(q)} \Big(c(q,a) + \sum_{\alpha \in P^*} p(q,a)(\alpha) \sum_{i \leq |\alpha|} \bar{c}'_{\alpha_i} \Big)$, where we break ties lexicographically.

We now claim that, for all $k \geq 0$, $\mathrm{ETotal}_k(\mathcal{B},q,\sigma') \leq \bar{c}'_q$ holds. For $k=0$, this is trivial as $\mathrm{ETotal}_k(\mathcal{B},q,\sigma') = 0 \leq \bar{c}'_q$. For $k > 0$, we have that

$$\begin{aligned}
\mathrm{ETotal}_k(\mathcal{B},q,\sigma') &\overset{(1)}{\leq} c(q,\sigma'(q)) + \sum_{\alpha \in P^*} p(q,\sigma'(q))(\alpha) \sum_{i \leq |\alpha|} \mathrm{ETotal}_{k-1}(\mathcal{B},\alpha_i,\sigma') \\
&\overset{(2)}{\leq} c(q,\sigma'(q)) + \sum_{\alpha \in P^*} p(q,\sigma'(q))(\alpha) \sum_{i \leq |\alpha|} \bar{c}'_{\alpha_i} \\
&\overset{(3)}{=} \min_{a \in A(q)} \Big(c(q,a) + \sum_{\alpha \in P^*} p(q,a)(\alpha) \sum_{i \leq |\alpha|} \bar{c}'_{\alpha_i} \Big) \overset{(4)}{=} \bar{c}'_q
\end{aligned}$$

where (1) follows from the fact that after taking action $\sigma'(q)$ first, there are only $k-1$ steps left of the BMDP $\mathcal{B}$ that would need to be distributed among the offspring α of q somehow. Allowing for $k-1$ steps for each of the entities α_i is clearly an overestimate of the actual cost. (2) follows from the inductive assumption. (3) follows from the definition of σ'. The last equality, (4), follows from the fact that $\bar{c}'$ is a fixed point of F.

Finally, for every $q \in P$, from the definition we have $\bar{c}^*_q = \mathrm{ETotal}_*(\mathcal{B},q) \leq$

ETotal$_*(\mathcal{B}, q, \sigma') = \lim_{k\to\infty}$ ETotal$_k(\mathcal{B}, q, \sigma')$ and each element of the last sequence was just shown to be $\leq \bar{c}'_q$.

(e) We know that $\bar{x}^* = \lim_{k\to\infty} \bar{x}^k$ exists in $\widetilde{\mathbb{R}}^n_{\geq 0}$ because it is a monotonically non-decreasing sequence (note that some entries may be infinite). In fact we have $\bar{x}^* = \lim_{k\to\infty} F^{k+1}(\bar{0}) = F(\lim_{k\to\infty} F^k(\bar{0}))$, and thus $\bar{x}^*$ is a fixed point of F. So from (d) we have $\bar{c}^* \leq \bar{x}^*$. At the same time, due to (c), we have $\bar{x}^k \leq \bar{c}^*$ for all $k \geq 0$, so $\bar{x}^* = \lim_{k\to\infty} \bar{x}^k \leq \bar{c}^*$ and thus $\lim_{k\to\infty} \bar{x}^k = \bar{c}^*$.

□

The following is a simple corollary of Theorem 4.

Corollary 5. *In BMDPs, there exists an optimal static control strategy σ^*.*

Proof. It is enough to pick as σ^*, the strategy σ' from Theorem 4.d, for $\bar{c}' = \bar{c}^*$. We showed there that for all $k \geq 0$ and $q \in P$ we have ETotal$_k(\mathcal{B}, q, \sigma^*) \leq \bar{c}'_q$. So ETotal$_*(\mathcal{B}, q, \sigma^*) = \lim_{k\to\infty}$ ETotal$_k(\mathcal{B}, q, \sigma^*) \leq \bar{c}^*_q =$ ETotal$_*(\mathcal{B}, q)$, so in fact ETotal$_*(\mathcal{B}, q, \sigma^*) =$ ETotal$_*(\mathcal{B}, q)$ has to hold as clearly ETotal$_*(\mathcal{B}, q, \sigma^*) \geq$ ETotal$_*(\mathcal{B}, q)$. □

Note that for a BMDPs with a fixed static strategy σ (or equivalently BMCs), we have that $F(\bar{x}) = B_\sigma \bar{x} + \bar{c}_\sigma$, for some non-negative matrix $B_\sigma \in \mathbb{R}^{n\times n}_{\geq 0}$, and a positive vector $\bar{c}_\sigma > 0$ consisting of all one step costs $c(q, \sigma(q))$. We will refer to F as F_σ in such a case and exploit this fact later in various proofs.

We now show that $\bar{c}^*$ is in fact essentially a unique fixed point of F.

Theorem 6. *If $F(\bar{x}) = \bar{x}$ and $\bar{x}_q < \infty$ for some $q \in P$ then $\bar{x}_q = \bar{c}^*_q$.*

Proof. By Corollary 5, there exists an optimal static strategy, denoted by σ^*, which yields the finite optimal reward vector $\bar{c}^*$.

We clearly have that $\bar{x} = F(\bar{x}) \leq F_{\sigma^*}(\bar{x})$, because σ^* is just one possible pick of actions for each type rather than the minimal one as in (♠). Furthermore,

$$\begin{aligned}
F_{\sigma^*}(\bar{x}) &= B_{\sigma^*}\bar{x} + b_{\sigma^*} \\
&\leq B_{\sigma^*}(B_{\sigma^*}\bar{x} + b_{\sigma^*}) + b_{\sigma^*} \\
&= B^2_\sigma \bar{x} + (B_{\sigma^*} + 1)b^*_\sigma \\
&\leq \ldots \leq \lim_{k\to\infty} B^k_{\sigma^*}\bar{x} + \Big(\sum_{k=0}^{\infty} B^k_{\sigma^*}\Big) b_{\sigma^*}.
\end{aligned}$$

Note that $\bar{c}^* = (\sum_{k=0}^\infty B^k_{\sigma^*}) b_{\sigma^*}$, because

$$\bar{c}^* = \lim_{k\to\infty} F^k(\bar{0}) = \lim_{k\to\infty} F^k_{\sigma^*}(\bar{0}) = \lim_{k\to\infty} \sum_{i=0}^{k} B^i_{\sigma^*} b_{\sigma^*}.$$

Due to Theorem 4.d, we know that $\bar{c}^*_q \leq \bar{x}_q < \infty$, so all entries in the q-th row of $B^k_{\sigma^*}$ have to converge to 0 as $k \to \infty$, because otherwise the q-th row of $\sum_{k=0}^{\infty} B_{\sigma^*}^k$ would have at least one infinite value and, as a result, the q-th position of $\bar{c}^* = (\sum_{k=0}^{\infty} B_{\sigma^*}^k) b_{\sigma^*}$ would also be infinite as all entries of b_{σ^*} are positive. Therefore, $\lim_{k\to\infty}(B_{\sigma^*}^k \bar{x})_q = 0$ and so

$$\bar{x}_q \leq (\lim_{k\to\infty} B_{\sigma^*}^k \bar{x})_q + ((\sum_{k=0}^{\infty} B_{\sigma^*}^k) b_{\sigma^*})_q = \bar{c}_q^*.$$

The proof is now complete. □

4 Q-learning

We next discuss the applicability of Q-learning to the computation of the fixed point defined in the previous section.

Q-learning [30] is a well-studied model-free RL approach to compute an optimal strategy for discounted rewards. Q-learning computes so-called Q-values for every state-action pair. Intuitively, once Q-learning has converged to the fixed point, $Q(s,a)$ is the optimal reward the agent can get while performing action a after starting at s. The Q-values can be initialised arbitrarily, but ideally they should be close to the actual values. Q-learning learns over a number of episodes, each consisting of a sequence of actions with bounded length. An episode can terminate early if a sink-state or another non-productive state is reached. Each episode starts at the designated initial state s_0. The Q-learning process moves from state to state of the MDP using one of its available actions and accumulates rewards along the way. Suppose that in the i-th step, the process has reached state s_i. It then either performs the currently (believed to be) optimal action (so-called *exploitation* option) or, with probability ϵ, picks uniformly at random one of the actions available at s_i (so-called *exploration* option). Either way, if a_i, r_i, and s_{i+1} are the action picked, reward observed and the state the process moved to, respectively, then the Q-value is updated as follows:

$$Q_{i+1}(s_i, a_i) = (1 - \lambda_i) Q_i(s_i, a_i) + \lambda_i (r_i + \gamma \cdot \max_a Q_i(s_{i+1}, a)) \ ,$$

where $\lambda_i \in]0,1[$ is the learning rate and $\gamma \in]0,1]$ is the discount factor. Note the model-freeness: this update does not depend on the set of transitions nor their probabilities. For all other pairs s,a we have $Q_{i+1}(s,a) = Q_i(s,a)$, i.e., they are left unchanged. Watkins and Dayan showed the convergence of Q-learning [30].

Theorem 7 (Convergence [30]). *For $\gamma < 1$, bounded rewards r_i and learning rates $0 \leq \lambda_i < 1$ satisfying:*

$$\sum_{i=0}^{\infty} \lambda_i = \infty \ \textit{and} \sum_{i=0}^{\infty} \lambda_i^2 < \infty,$$

we have that $Q_i(s,a) \to Q(s,a)$ as $i \to \infty$ for all $s, a \in S \times A$ almost surely if all (s,a) pairs are visited infinitely often.

However, in the total reward setting that corresponds to Q-learning with discount factor $\gamma = 1$, Q-learning may not converge, or converge to incorrect values. However, it is guaranteed to work for finite-state MDPs in the setting of undiscounted total reward with a target sink-state under the assumption that all strategies reach that sink-state almost surely. The assumption that we make instead is that every transition of BMDP incurs a positive cost. This guarantees that a process that does not terminate almost surely generates an expected infinite reward in which case the Q-learning will coverage (or rather diverge) to ∞, so our results generalise these existing results for Q-learning.

We adopt the Q-learning algorithm to minimise cost as follows. Each episode starts at the designated initial state $q_0 \in P$. The Q-learning process moves from state to state of the BMDP using one of its available actions and accumulates costs along the way. Suppose that, in the i-th step, the process has reached state α. It then selects uniformly at random one of the entities of α, e.g., the j-th one, α_j and either performs the currently (believed to be) optimal action or, with probability ϵ, picks an action uniformly at random among all the actions available for α_j. If c and β denote the observed cost and entities spawned by this action, respectively, then the Q-value of the pair α_j, a_i are updated as follows:

$$Q_{i+1}(\alpha_j, a_i) = (1 - \lambda_i)Q_i(\alpha_j, a_i) + \lambda_i\big(c + \sum_{i=1}^{|\beta|} \min_{a \in A(\beta_i)} Q_i(\beta_i, a)\big).$$

and all other Q-values are left unchanged. In the next section we show that Q-learning almost surely converges (diverges) to the optimal finite (respectively, infinite) value of $\bar{c}^*$ almost surely under rather mild conditions.

5 Convergence of Q-Learning for BMDPs

We show almost sure convergence of the Q-learning to the optimal values $\bar{c}^*$ in a number of stages. We first focus on the case when all optimal values in $\bar{c}^*$ are finite. In such a case, we show a weak convergence of the expected optimal values for BMCs to the unique fixed-point $\bar{c}^*$, as defined in Sect. 3. To establish this, we show that the expected Q-values are monotonically decreasing (increasing) if we start with Q-values $\kappa\bar{c}^*$ for $\kappa > 1$ ($\kappa < 1$). This convergence from above and below gives us convergence in expectation using the squeeze theorem.

We then establish almost sure convergence to $\bar{c}^*$ by proving a contraction argument, with the extra assumption that the selection of the Q-value to update is done independently at random in each step.

In the next step, we extend this result to BMDPs, first establishing that Q-learning will almost surely converge to the *region* of the Q-values less than or equal to $\bar{c}^*$. We then show that, when considering the pointwise limes inferior values of the sequences of Q-values, there is no point in that region such that every ε-ball around it has a non-zero probability to be represented in the limes inferior. This establishes that $\bar{c}^*$ is the fixed point the Q-values converge against.

Only at the very end, we show that Q-learning also converges (or rather diverges) to the optimal value even if that value happens to be infinite. We then turn to a type with non-finite optimal value and provide an argument for the divergence to ∞ of its corresponding Q-value.

We assume that all the Q-values are stored in a vector Q of size $(|P| \cdot |A|)$. We also use $Q(q, a)$ to refer to the entry for type $q \in P$ and action $a \in A(q)$. We introduce the *target for Q operator*, T, that maps a Q-values vector Q to:

$$T(Q)(q,a) = c(q,a) + \sum_{\alpha \in Q^*} p(q,a)(\alpha) \sum_{i=1}^{|\alpha|} \min_{a_i \in A(\alpha_i)} Q(\alpha_i, a_i) \ .$$

We call T the 'target', because, when the $Q(q, a)$ value is updated, then

$$\mathbb{E}(Q_{i+1}(q,a)) = (1-\lambda_i) Q_i(q,a) + \lambda_i T(Q_i)(q,a)$$

holds, whereas otherwise $Q_{i+1}(q, a) = Q_i(q, a)$.

Thus, when $Q(q, a)$ is selected for update with a chance of $p_{q,a}$, we have that

$$\mathbb{E}(Q_{i+1}(q,a)) = (1-\lambda_i p_{q,a}) Q_i(q,a) + \lambda_i p_{q,a} T(Q_i)(q,a) \ . \qquad (\heartsuit)$$

5.1 Convergence for BMCs with Finite $\bar{c}^*$

Since BMCs have only one action, we omit mentioning it for ease of notation.

Note that for BMCs, the target for the Q-values is a simple affine function:

$$T(Q)(q) = c(q) + \sum_{\alpha \in P^*} p(q)(\alpha) \sum_{i=1}^{|\alpha|} Q(\alpha_i).$$

And it coincides with operator F as defined in Sect. 3. Therefore, due to Theorem 6, $T(Q)$ has a unique fixed point which is $\bar{c}^*$. Moreover, $T(Q) = BQ + \bar{c}$, where B is a non-negative matrix and $\bar{c}$ is a vector of one step costs $c(q)$, which are all positive.

Naturally, applying T to a non-negative vector Q or multiplying it by B are monotone: $Q \geq Q' \rightarrow T(Q) \geq T(Q')$ and $BQ \geq BQ'$. Also, due to the linearity of T, $\mathbb{E}(T(Q)) = T(\mathbb{E}(Q))$ holds, where Q is a random vector.

We now start with a lemma describing the behaviour of Q-learning for initial Q-values when they happen to be equal to $\kappa\bar{c}^*$ for some $\kappa \geq 1$.

Lemma 8. *Let $Q_0 = \kappa\bar{c}^*$ for a scalar factor $\kappa \geq 1$. Then the following holds for all $i \in \mathbb{N}$,*

$$\bar{c}^* \leq T(\mathbb{E}(Q_i)) \leq \mathbb{E}(Q_{i+1}) \leq \mathbb{E}(Q_i),$$

assuming that Q-value to be updated in each step is selected independently at random.

Proof. We show this by induction. For the induction basis ($i = 0$), we have that $\bar{c}^* \leq Q_0$ by definition.

As $\bar{c}^*$ is the fixed-point of T, we have $T(\bar{c}^*) = \bar{c}^*$, and the monotonicity of T provides $T(\bar{c}^*) \leq T(Q_0)$. At the same time

$$\begin{aligned} T(Q_0) = T(\kappa\bar{c}^*) &= B\kappa\bar{c}^* + \bar{c} \\ &= \kappa(B\bar{c}^* + \bar{c}) - \kappa\bar{c} + \bar{c} \\ &= \kappa\bar{c}^* - (\kappa - 1)\bar{c} \\ &= Q_0 - (\kappa - 1)\bar{c} \leq Q_0. \end{aligned}$$

This provides $\bar{c}^* \leq T(\mathbb{E}(Q_0)) \leq \mathbb{E}(Q_0)$. Finally, $T(\mathbb{E}(Q_0)) \leq \mathbb{E}(Q_0)$ entails for a learning rate $\lambda_0 \in [0, 1]$ that $T(\mathbb{E}(Q_0)) \leq \mathbb{E}(Q_1) \leq \mathbb{E}(Q_0)$ due to ($\heartsuit$).

For the induction step ($i \mapsto i + 1$), we use the induction hypothesis

$$\bar{c}^* \leq T(\mathbb{E}(Q_i)) \leq \mathbb{E}(Q_{i+1}) \leq \mathbb{E}(Q_i).$$

The monotonicity of T and $\bar{c}^* \leq \mathbb{E}(Q_{i+1}) \leq \mathbb{E}(Q_i)$ imply that $T(\bar{c}^*) \leq T(\mathbb{E}(Q_{i+1})) \leq T(\mathbb{E}(Q_i))$ holds. With $T(\bar{c}^*) = \bar{c}^*$ (from the fixed point equations) and the induction hypothesis, $\bar{c}^* \leq T(\mathbb{E}(Q_{i+1})) \leq \mathbb{E}(Q_{i+1})$ follows.

Using $T(\mathbb{E}(Q_{i+1})) = \mathbb{E}(T(Q_{i+1}))$, this provides $\mathbb{E}(T(Q_{i+1})) \leq \mathbb{E}(Q_{i+1})$, which implies with $\lambda_{i+1} \in [0, 1]$ that

$$T(\mathbb{E}(Q_{i+1})) = \mathbb{E}(T(Q_{i+1})) \leq \mathbb{E}(Q_{i+2}) \leq \mathbb{E}(Q_{i+1})$$

holds, completing the induction step. □

By simply replacing all $\leq$ with $\geq$ in the above proof, we can get the following for all initial Q-values that happen to be $\kappa\bar{c}^*$ where $\kappa \leq 1$:

Lemma 9. *Let $Q_0 = \kappa\bar{c}^*$ for a scalar factor $\kappa \in [0, 1]$. Then the following holds for all $i \in \mathbb{N}$, assuming that the Q-value to update in each step is selected independently at random: $\bar{c}^* \geq T(\mathbb{E}(Q_i)) \geq \mathbb{E}(Q_{i+1}) \geq \mathbb{E}(Q_i)$.* □

We now first establish that the distance between Q and $\bar{c}^*$ can be upper bounded by the distance between Q and $T(Q)$ with a fixed linear factor $\mu > 0$.

Lemma 10. *There exists a constant $\mu > 0$ such that*

$$\sum_{q \in P} |(Q - T(Q)(q)| \geq \mu \sum_{q \in P} |(Q - \bar{c}^*(q)|$$

when $Q_0 = \kappa\bar{c}^*$.

Proof. We show this for $\kappa > 1$. The proof for $\kappa < 1$ is similar, and there is nothing to show for $\kappa = 1$.

We first consider the linear programme with a variable for each type with the following constraints for some fixed $\delta > 0$:

$$Q \geq \bar{c}^*, T(Q) \leq Q, \text{and} \sum_{q \in P} Q(q) = \sum_{q \in P} \bar{c}^*(q) + \delta.$$

An example solution to this constraint system is $Q = (1 + \frac{\delta}{\sum_{q\in P} \bar{c}^*(q)})\bar{c}^*$.

We then find a solution minimising the objective $\sum_{q\in P} |(Q - T(Q)(p)|$, noting that all entries are non-negative due to the first constraint. This is expressed by adding $2|P|$ constraints

$$x_q \geq Q(q) - T(Q)(q)$$
$$x_q \geq T(Q)(q) - Q(q)$$

and minimising $\sum_{q\in P} x_q$.

As $\bar{c}^*$ is the only fixed-point of T, and $\sum_{q\in P} Q(q) = \sum_{q\in P} \bar{c}^*(q) + \delta$ implies that, for an optimal solution Q^*, $Q^* \neq \bar{c}^*$, we have that

$$\sum_{q\in P} |(Q^* - T(Q^*)(q)| > 0.$$

Due to the constraint $Q \geq \bar{c}^*$, we always have $Q = \bar{c}^* + Q_\Delta$ for some $Q_\Delta > \bar{0}$. We can now re-formulate this linear programme to look for Q_Δ instead of Q:

$$Q_\Delta \geq \bar{0},$$
$$BQ_\Delta \leq Q_\Delta, \text{and}$$
$$\sum_{q\in P} Q_\Delta(q) = \delta,$$

with the objective to minimise $\sum_{q\in P} |(Q_\Delta - BQ_\Delta)(q)|$.

The optimal solution Q^*_Δ to this linear programme gives an optimal value $Q^* = \bar{c}^* + Q^*_\Delta$ for the former and, vice versa, the value Q^* for the former provides an optimal solution $Q^*_\Delta - \bar{c}^*$ for the latter, and these two solutions have the same value in their respective objective function.

Thus, while the former constraint system is convenient to show that the value of the objective function is positive, the latter constraint system is, except for $\sum_{q\in P} Q_\Delta(q) = \delta$, linear. This means that any optimal solution for $\delta = \delta_1$ can be obtained from the optimal solution for $\delta = \delta_2$ just by rescaling it by δ_1/δ_2. It follows that the optimal value of the objective function is linear in δ, e.g., there exists $\mu > 0$ such that its value is $\mu\delta$. $\square$

We now show that the sequence of Q-values updates converges in expectation to $\bar{c}^*$ when $Q_0 = \kappa\bar{c}^*$.

Lemma 11. *Let $Q_0 = \kappa\bar{c}^*$ where $\kappa \geq 0$. Then, assuming that each type-action pair is selected for update with a minimal probability $p_{\min}$ in each step, and that $\sum_{i=0}^{\infty} \lambda_i = \infty$, then $\lim_{i\to\infty} \mathbb{E}(Q_i) = \bar{c}^*$ holds.*

Proof. We proof this for $\kappa \geq 1$. A similar proof shows this for any $\kappa \in [0, 1]$. Lemma 8 provides that all $\mathbb{E}(Q_i)$ satisfy the constraints $\mathbb{E}(Q_i) \geq \bar{c}^*$ and $T(\mathbb{E}(Q_i)) \leq \mathbb{E}(Q_i)$.

Let $p_{\min}$ be the smallest probability any Q-value is selected with in each update step. Due to Lemma 10, there is a fixed constant $\mu > 0$ such that

$$\sum_{q \in P} |Q_i(q) - T(Q_i)(q)| \geq \mu \sum_{q \in P} |Q_i(q) - \bar{c}^*(q)| \ .$$

By taking the expected value of both sides and the fact that $\bar{c}^* \leq T(\mathbb{E}(Q_i)) \leq \mathbb{E}(Q_{i+1}) \leq \mathbb{E}(Q_i)$ due to Lemma 8, we get

$$\sum_{q \in P} \mathbb{E}(Q_i)(q) - T(\mathbb{E}(Q_i))(q) \geq \mu \sum_{q \in P} \mathbb{E}(Q_i)(q) - \bar{c}^*(q),$$

then due to (♡) we have

$$\sum_{q \in P} \mathbb{E}(Q_i)(q) - \mathbb{E}(Q_{i+1})(q) \geq \mu p_{\min} \lambda_i \sum_{q \in P} \mathbb{E}(Q_i)(q) - \bar{c}^*(q),$$

and finally just by rearranging these terms we get

$$\sum_{q \in P} \mathbb{E}(Q_{i+1})(q) - \bar{c}^*(q) \leq (1 - \mu p_{\min} \lambda_i) \sum_{q \in P} \mathbb{E}(Q_i)(q) - \bar{c}^*(q) \ .$$

Note that all summands are positive by Lemma 8.

With $\sum_{i=0}^{\infty} \lambda_i = \infty$, we get that $\sum_{i=0}^{\infty} \mu p_{\min} \lambda_i = \infty$, because $p_{\min}$ and μ are fixed positive values. This implies that $\prod_{i=0}^{\infty}(1 - \mu p_{\min} \lambda_i) = 0$ and so the distance between $\mathbb{E}(Q_i)$ and $\bar{c}^*$ converges to 0. $\square$

Lemma 11 suffices to show convergence of Q-values in expectation.

Theorem 12. *When each Q-value is selected for an update with a minimal probability $p_{\min}$ in each step, and $\sum_{i=0}^{\infty} \lambda_i = \infty$, then $\lim_{i \to \infty} \mathbb{E}(Q_i) = \bar{c}^*$ holds for every starting Q-values $Q_0 \geq \bar{0}$.*

Proof. We first note that none of the entries of $\bar{c}^*$ can be 0. This implies that there is a scalar factor $\kappa \geq 0$ such that $\bar{0} \leq Q_0 \leq \kappa \bar{c}^*$. As the Q_i are monotone in the entries of Q_0, and as the property holds for $Q_0' = \bar{0} = 0 \cdot \bar{c}^*$ and $Q_0'' = \kappa \bar{c}^*$ by Lemma 11, the squeeze theorem implies that it also holds for Q_0. $\square$

Convergence of the expected value is a weaker property than expected convergence, which also explains why our assumptions are weaker than in Theorem 7. With the common assumption of sufficiently fast falling learning rates, $\sum_{i=0}^{\infty} {\lambda_i}^2 < \infty$, we will now argue that the pointwise limes inferior of the sequence of Q-values almost surely converges to $\bar{c}^*$. This will later allow us to infer convergence of the actual sequence of Q-values to $\bar{c}^*$.

Theorem 13. *When each Q-value is selected for update with a minimal probability $p_{\min}$ in each step,*

$$\sum_{i=0}^{\infty} \lambda_i = \infty \ \textit{and} \sum_{i=0}^{\infty} \lambda_i^2 < \infty,$$

then $\lim_{i \to \infty} Q_i = \bar{c}^*$ holds almost surely for every starting Q-values $Q_0 \geq \bar{0}$.

Proof. We assume for contradiction that, for some $\widehat{Q} \neq \bar{c}^*$, there is a non-zero chance of a sequence $\{Q_i\}_{i \in \mathbb{N}_0}$ such that

- $\|\widehat{Q} - \liminf_{i\to\infty} Q_i\|_\infty < \varepsilon'$ for all $\varepsilon' > 0$, and
- there is a type q such that $\widehat{Q}(q) < T(\widehat{Q})(q)$.

Then there must be an $\varepsilon > 0$ such that $\widehat{Q}(q) + 3\varepsilon < T(\widehat{Q} - 2\varepsilon \cdot \bar{1})(q)$. We fix such an $\varepsilon > 0$.

Now we have the assumption that the probability of $\|\widehat{Q} - \liminf_{n\to\infty} Q_i\|_\infty < \varepsilon$ is positive. Then, in particular, the chance that, at the same time, $\liminf_{i\to\infty} Q_i > \widehat{Q} - \varepsilon \cdot \bar{1}$ *and* $\liminf_{i\to\infty} Q_i < \widehat{Q} + \varepsilon \cdot \bar{1}$, is positive.

Thus, there is a positive chance that the following holds: there exists an n_ε such that, for all $i > n_\varepsilon$, $Q_i \geq \widehat{Q} - 2\varepsilon \cdot \bar{1}$. This implies

$$T(Q_i)(q) \geq T(\widehat{Q} - 2\varepsilon \cdot \bar{1})(q) > \widehat{Q}(q) + 3\varepsilon.$$

Thus, the expected limit value of $Q_i(q)$ is at least $\widehat{Q}(q) + 3\varepsilon$, for every tail of the update sequence. Now, we can use $\widehat{Q} - 2\varepsilon$ as a bound on the estimation of the updates in Q-learning as $Q_i \geq \widehat{Q} - 2\varepsilon \cdot \bar{1}$ holds. At the same time, the variation of the sum of the updates goes to 0 when $\sum i = 0^\infty \lambda_i^2$ is bounded. Therefore, it cannot be that $\liminf_{i\to\infty} Q_i < \widehat{Q} + \varepsilon \cdot \bar{1}$ holds; a contradiction.

We note that if, for a Q-values $Q \geq \bar{0}$, there is a $q \in P$ with $Q(q') < \bar{c}^*(q')$, then there is a $q \in P$ with $Q(q) < T(Q)(q)$ and $Q(q) < \bar{c}^*(q)$. This is because, for the Q-values Q' with $Q'(q) = \min\{Q(q), \bar{c}^*(q)\}$ for all $q \in Q$, $Q' < \bar{c}^*$. Thus, there must be a type $q \in P$ such that $\kappa = \frac{Q'(q)}{\bar{c}^*(q)} < 1$ is minimal, and $Q' \geq \kappa \bar{c}^*$. As we have shown before, $T(\kappa\bar{c}^*) = \kappa\bar{c}^* - (\kappa - 1)\bar{c}$, such that the following holds:

$$T(Q)(q) \geq T(Q')(q) \geq T(\kappa\bar{c}^*)(q) = \kappa\bar{c}^*(q) + (1-\kappa)c(q) > \bar{c}^*(q) = Q(q).$$

Thus, we have that $\liminf_{i\to\infty} Q_i \geq \bar{c}^*$ holds almost surely. With $\lim_{i\to\infty} \mathbb{E}(Q_i) = \bar{c}^*$, it follows that $\lim_{i\to\infty} Q_i = \bar{c}^*$. □

5.2 Convergence for BMDPs and Finite $\bar{c}^*$

We start with showing that, for BMDPs, the pointwise limes superior of each sequence is almost surely less than or equal to $\bar{c}^*$. We then proceed to show that the limes inferior of a sequence is almost surely $\bar{c}^*$, which together implies almost sure convergence.

Lemma 14. *When each Q-value of BMDP is selected for update with a minimal probability* $p_{\min}$ *in each step,* $\sum_{i=0}^\infty \lambda_i = \infty$*,* $\sum_{i=0}^\infty \lambda_i^2 < \infty$*, then* $\limsup_{i\to\infty} Q_i \leq \bar{c}^*$ *holds almost surely for every starting Q-values* $Q_0 \geq \bar{0}$.

Proof. To show the property for the limes superior, we fix an optimal static strategy σ^* that exists due to Corollary 5.

We define an BMC obtained by replacing each type q in the BMDP with $A(q) = \{a_1, \ldots, a_k\}$, by k types $(q, a_1), \ldots, (q, a_k)$ with one action, where each type q' is replaced by the type-action pair $(q', \sigma^*(q'))$.

It is easy to see that a type $(q, \sigma^*(q))$ for the resulting BMC has the same value as the type q and the type-action pair $(q, \sigma^*(q))$ in the BMDP that we started with.

When identifying these corresponding type-action pairs, we can look at the same sampling for the BMDP and the BMC, leading to sequences $Q_0, Q_1, Q_2, \ldots$ and $Q'_0, Q'_1, Q'_2, \ldots$, respectively, where $Q_0 = Q'_0$.

It is easy to see by induction that $Q_i \leq Q'_i$. Considering that $\{Q'_i\}_{i \in \mathbb{N}}$ almost surely converges to $\bar{c}^*$ by Theorem 13, we obtain our result. $\square$

Theorem 15. *When each Q-value of an BMDP is selected for update with a minimal probability* $p_{\min}$, $\sum_{i=0}^{\infty} \lambda_i = \infty$, $\sum_{i=0}^{\infty} \lambda_i^2 < \infty$, *then* $\lim_{i \to \infty} Q_i = \bar{c}^*$ *holds almost surely for every starting Q-values* $Q_0 \geq \bar{0}$.

Proof. As a first simple corollary from Lemma 14, we get the same result for the limes inferior (as $\liminf \leq \limsup$ must hold).

We now assume for contradiction that, for some vector $\widehat{Q} < \bar{c}^*$, there is a non-zero chance of a sequence $\{Q_i\}_{i \in \mathbb{N}}$ such that $\|\widehat{Q} - \liminf_{n \to \infty} Q_i\|_\infty < \varepsilon'$ for all $\varepsilon' > 0$.

As $\widehat{Q}$ is below the fixed point of T, there must be one type-action pair $(q, \sigma^*(q))$ such that $\widehat{Q}(q, \sigma^*(q)) < T(\widehat{Q})(q, \sigma^*(q))$ (cf. the proof of Theorem 13). Moreover, there must be an $\varepsilon > 0$ such that

$$\widehat{Q}(q, \sigma^*(q)) + 3\varepsilon < T(\widehat{Q} + 2\varepsilon \cdot \bar{1})(q, \sigma^*(q)).$$

We fix such an $\varepsilon > 0$.

Now we assume that the probability of $\|\widehat{Q} - \liminf_{n \to \infty} Q_i\|_\infty < \varepsilon$ is positive. Then the chance that, simultaneously, $\liminf_{i \to \infty} Q_i(q, \sigma^*(q)) > \widehat{Q}(q, \sigma^*(q)) - \varepsilon$ and $\liminf_{i \to \infty} Q_i(q, \sigma^*(q)) < \widehat{Q}(q, \sigma^*(q)) + \varepsilon$, is positive.

Thus, there is a positive chance that the following holds: there exists an n_ε such that, for all $i > n_\varepsilon$ we have $Q_i \geq \widehat{Q} - 2\varepsilon \cdot \bar{1}$. This entails

$$T(Q_i)(q, \sigma^*(q)) \geq T(\widehat{Q} - 2\varepsilon \cdot \bar{1})(q, \sigma^*(q)) > \widehat{Q}(q, \sigma^*(q)) + 3\varepsilon.$$

Thus, the expected limit value of $Q_i(q, \sigma^*(a))$ is at least $\widehat{Q}(q, \sigma^*(a)) + 3\varepsilon$, for every tail of the update sequence. Now, we can use $T(\widehat{Q} - 2\varepsilon \cdot \bar{1})(q, \sigma^*(a))$ as a bound on the estimation of $T(Q)(q, \sigma^*(q))$ during the update of the Q-value of the type-action pair $(q, \sigma^*(q))$. At the same time, the variation of the sum of the updates goes to 0 when $\sum_{i=0}^{\infty} \lambda_i^2$ is bounded. Therefore, it cannot be that $\liminf_{i \to \infty} Q_i(q, \sigma^*(a)) < \widehat{Q}(q, \sigma^*(a)) + \varepsilon$ holds; a contradiction. $\square$

5.3 Divergence

We now show divergence of $Q(q)$ to ∞ when at least one of the entries of $\bar{c}^*(q)$ is infinite. First due to Theorem 6 and its proof we have that $\bar{c}^* = \sum_{i=0}^{\infty} B^i \bar{c}$ for some non-negative B and positive $\bar{c}$. Therefore $\bar{c}^*$ is monotonic in B for BMCs. Likewise, the value of $\bar{c}^*$ for a BMDP depends only on the cost function and the expected number of successors of each type spawned: Two BMDPs with same cost functions and the expected numbers of successors have the same fixed point $\bar{c}^*$. Thus, if a type q with one action spawns either exactly one q or exactly one q' with a chance of 50% each, or if it spawns 10 successors of type q and another 10 or type q' with a chance of 5%, while dying without offspring with a chance of 95%, both lead to identical matrices B and so the same $\bar{c}^*$ (though this difference may impact the performance of Q-learning).

Naturally, raising the number of expected number of successors of any type for any type-action pair strictly raises $\bar{c}^*$, while lowering it reduces $\bar{c}^*$, and for every set of expected numbers, the value of $\bar{c}^*$ is either finite or infinite.

Let us consider a set of parameters at the fringe of finite vs. infinite $\bar{c}^*$, and let us choose them pointwise not larger than the parameters from the BMC or BMDP under consideration. As the fixed point from Sect. 3 is clearly growing continuously in the parameter values, this set of expected successors leads to a $\bar{c}^*$ which is not finite.

We now look at the family of parameter values that lead to $\alpha \in [0,1[$ times the expected successors from our chosen parameter at the fringe between finite and infinite values, and refer to it as the α-BMDP. Let also $\bar{c}^*_\alpha$ denote the fixed point for the reduced parameters. As the solution to the fixed point grows continuously, so does $\bar{c}^*_\alpha$. Moreover, if $\bar{c}^*_1 = \lim_{\alpha\to 1} \bar{c}^*_\alpha$ was finite, then $\bar{c}^*$ would be finite as well, because then $\bar{c}^*_1 = \bar{c}^*$.

Clearly, for all parameters $\alpha \in [0,1[$, the Q-values of an α-BMC or α-BMDP converge against $\bar{c}^*_\alpha$. Thus, the Q-values for the BMC or BMDP we have started with converges against a value, which is at least $\sup_{\alpha\in[0,1[} \bar{c}^*_\alpha$. As this is not a finite value, Q-learning diverges to ∞.

6 Experimental Results

We implemented the algorithm described in the previous section in the formal reinforcement learning tool Mungojerrie [21], a C++-based tool which reads BMDPs described in an extension of the PRISM language [18]. The tool provides an interface for RL algorithms akin to that of [3] and invokes a linear programming tool (GLOP) [22] to compute the optimal expected total cost based on the optimality equations (♠).

6.1 Benchmark Suite

The BMDPs on which we tested Q-learning are listed in Table 1. For each model, the numbers of types in the BMDP, are given. Table 1 also shows the total cost (as computed by the LP solver), which has full access to the BMDP. This is followed by the estimate of the total cost computed by Q-learning and the time taken by learning. The learner has several hyperparameters: ϵ is the exploration rate, α is the learning rate, and tol is the tolerance for Q-values to be considered different when selecting an optimal strategy. Finally, ep-l is the maximum episode length and ep-n is the number of episodes. The last two columns of Table 1 report the values of ep-l and ep-n when they deviate from the default values. All performance data are the averages of three trials with Q-learning. Since costs are undiscounted, the value of a state-action pair computed by Q-learning is a direct estimate of the optimal total cost from that state when taking that action.

Table 1. Q-learning results. The default values of the learner hyperparameters are: $\epsilon = 0.1$, $\alpha = 0.1$, tol= 0.01, ep-l= 30, and ep-n= 20000. Times are in seconds.

Name	Types	Optimal cost	Estimated cost	Time (avg.)	ep-l	ep-n
`cloud1`	3	5	5.026	0.369		
`cloud2`	4	5	5.016	0.369		
`bacteria1`	3	2.5	2.514	0.374		
`bacteria2`	3	1.34831	1.413	0.387		
`protein`	3	6	5.067	0.372		
`frozenSmall`	16	1.84615	1.740	2.834	100	
`rand68`	10	150.432	154.400	0.402		
`rand283`	9	4	4	0.075		1000
`rand945`	19	212	208.177	10.756	200	40000
`rand3242`	43	4	4.372	5.960	100	
`rand6417`	62	10	10	12.498	50	

Models `cloud1` and `cloud2` are based on the motivating example given in the introduction. Examples `bacteria1` and `bacteria2` model the population dynamics of a family of two bacteria [28] subject to two treatments. The objective is to determine which treatment results in the minimum expected cost to extinction of the bacteria population. The `protein` example models a stochastic Petri net description [19] corresponding to a protein synthesis example with entities corresponding to active and inactive genes and proteins. The example `frozenSmall` [3] is similar to classical frozen lake example, except that one of the holes result in branching the process in two entities. Entities that fall in the target cell become extinct. The objective is to determine a strategy that results in a minimum number of steps before extinction. Finally, the remaining 5 examples are randomly created BMDP instances.

7 Conclusion

We study the total reward optimisation problem for branching decision processes with unknown probability distributions, and give the first reinforcement learning algorithm to compute an optimal policy. Extending Q-learning is hard, even for branching processes, because they lack a central property of the standard convergence proof: as the value range of the Q-table is not a priori bounded for a given starting table Q_0, the variation of the disturbance is not bounded.

This looks like a more substantial obstacle than the one Q-learning faces when maximising undiscounted rewards for finite-state MDPs, and it is well known that this defeats Q-learning. So it is quite surprising that we could not only show that Q-learning works for branching processes, but extend these results to branching decision processes, too. Finally, in the previous section, we have demonstrated that our Q-learning algorithm works well on examples of reasonable size even with default hyperparameters, so it is ready to be applied in practice without the need for excessive hyperparameter tuning.

References

1. Becker, N.: Estimation for discrete time branching processes with application to epidemics. In: Biometrics, pp. 515–522 (1977)
2. Brázdil, T., Kiefer, S.: Stabilization of branching queueing networks. In: 29th International Symposium on Theoretical Aspects of Computer Science (STACS 2012), vol. 14, pp. 507–518 (2012). https://doi.org/10.4230/LIPIcs.STACS.2012.507
3. Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., Zaremba, W.: OpenAI Gym. CoRR abs/1606.01540 (2016)
4. Chen, T., Dräger, K., Kiefer, S.: Model checking stochastic branching processes. In: Rovan, B., Sassone, V., Widmayer, P. (eds.) MFCS 2012. LNCS, vol. 7464, pp. 271–282. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32589-2_26
5. Esparza, J., Gaiser, A., Kiefer, S.: A strongly polynomial algorithm for criticality of branching processes and consistency of stochastic context-free grammars. Inf. Process. Lett. **113**(10–11), 381–385 (2013)
6. Etessami, K., Stewart, A., Yannakakis, M.: Greatest fixed points of probabilistic min/max polynomial equations, and reachability for branching Markov decision processes. Inf. Comput. **261**, 355–382 (2018). https://doi.org/10.1016/j.ic.2018.02.013
7. Etessami, K., Stewart, A., Yannakakis, M.: Polynomial time algorithms for branching Markov decision processes and probabilistic min(max) polynomial bellman equations. Math. Oper. Res. **45**(1), 34–62 (2020). https://doi.org/10.1287/moor.2018.0970
8. Etessami, K., Wojtczak, D., Yannakakis, M.: Recursive stochastic games with positive rewards. Theor. Comput. Sci. **777**, 308–328 (2019). https://doi.org/10.1016/j.tcs.2018.12.018
9. Etessami, K., Yannakakis, M.: Recursive Markov chains, stochastic grammars, and monotone systems of nonlinear equations. J. ACM **56**(1), 1–66 (2009)
10. Etessami, K., Yannakakis, M.: Recursive Markov decision processes and recursive stochastic games. J. ACM **62**(2), 11:1–11:69 (2015). https://doi.org/10.1145/2699431
11. Even-Dar, E., Mansour, Y., Bartlett, P.: Learning rates for q-learning. J. Mach. Learn. Res. **5**(1) (2003)
12. Haccou, P., Haccou, P., Jagers, P., Vatutin, V.: Branching processes: variation, growth, and extinction of populations. No. 5 in Cambridge Studies in Adaptive Dynamics, Cambridge University Press (2005)
13. Harris, T.E.: The Theory of Branching Processes. Springer, Berlin (1963)

14. Heyde, C.C., Seneta, E.: I. J. Bienaymé: Statistical Theory Anticipated. Springer, Heidelberg (1977). https://doi.org/10.1007/978-1-4684-9469-3
15. Jo, K.Y.: Optimal control of service in branching exponential queueing networks. In: 26th IEEE Conference on Decision and Control, vol. 26, pp. 1092–1097. IEEE (1987)
16. Kiefer, S., Wojtczak, D.: On probabilistic parallel programs with process creation and synchronisation. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 296–310. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19835-9_28
17. Kolmogorov, A.N., Sevastyanov, B.A.: The calculation of final probabilities for branching random processes. Doklady Akad. Nauk. U.S.S.R. (N.S.) **56**, 783–786 (1947)
18. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_47
19. Munsky, B., Khammash, M.: The finite state projection algorithm for the solution of the chemical master equation. J. Chem. Phys. **124**(4), 044104+ (2006)
20. Nielsen, L.R., Kristensen, A.R.: Markov decision processes to model livestock systems. In: Plà-Aragonés, L.M. (ed.) Handbook of Operations Research in Agriculture and the Agri-Food Industry. ISORMS, vol. 224, pp. 419–454. Springer, New York (2015). https://doi.org/10.1007/978-1-4939-2483-7_19
21. Perez, M., Somenzi, F., Trivedi, A.: Mungojerrie: formal reinforcement learning (2021). https://plv.colorado.edu/mungojerrie/. University of Colorado Boulder
22. Perron, L., Furnon, V.: Or-tools (version 7.2) (2019). https://developers.google.com/optimization. Google
23. Pliska, S.R.: Optimization of multitype branching processes. Manag. Sci. **23**(2), 117–124 (1976)
24. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley, Hoboken (1994)
25. Rao, A., Bauch, C.T.: Classical Galton-Watson branching process and vaccination. Int. J. Pure Appl. Math. **44**(4), 595 (2008)
26. Rothblum, U.G., Whittle, P.: Growth optimality for branching Markov decision chains. Math. Oper. Res. **7**(4), 582–601 (1982)
27. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction, 2nd edn. MIT Press, Cambridge (2018)
28. Trivedi, A., Wojtczak, D.: Timed branching processes. In: 2010 Seventh International Conference on the Quantitative Evaluation of Systems, pp. 219–228. IEEE (2010)
29. Udom, A.U.: A Markov decision process approach to optimal control of a multi-level hierarchical manpower system. CBN J. Appl. Stat. **4**(2), 31–49 (2013)
30. Watkins, C.J., Dayan, P.: Q-learning. Mach. Learn. **8**(3–4), 279–292 (1992). https://doi.org/10.1007/BF00992698
31. Watson, H.W., Galton, F.: On the probability of the extinction of families. J. Anthrop. Inst. **4**, 138–144 (1874)
32. Wojtczak, D.: Recursive probabilistic models : efficient analysis and implementation. Ph.D. thesis, University of Edinburgh, UK (2009). http://hdl.handle.net/1842/3217

Software Verification

Cameleer: A Deductive Verification Tool for OCaml

Mário Pereira(✉) and António Ravara

NOVA LINCS, Nova School of Science and Technology, Lisbon, Portugal
{mjp.pereira,aravara}@fct.unl.pt

Abstract. We present Cameleer, an automated deductive verification tool for OCaml. We leverage on the recently proposed GOSPEL (Generic OCaml SPEcification Language) to attach rigorous, yet readable, behavioral specification to OCaml code. The formally-specified program is fed to our toolchain, which translates it into an equivalent one in WhyML, the programming and specification language of the Why3 verification framework. We report on successful case studies conducted in Cameleer.

Keywords: Deductive software verification · OCaml · Why3 · GOSPEL

1 Introduction

Over the past decades, we have witnessed a tremendous development in the field of deductive software verification [11], the practice of turning the correctness of code into a mathematical statement and then prove it. Interactive proof assistants have evolved from obscure and mysterious tools into *de facto* standards for proving industrial-size projects. On the other end of the spectrum, the so-called *SMT revolution* and the development of reusable intermediate verification infrastructures contributed decisively to the development of practical automated deductive verifiers.

Despite all the advances in deductive verification and proof automation, little attention has been given to the family of *functional languages* [27]. Let us consider, for instance, the OCaml language. It is well suited for verification, given its well-defined semantics, clear syntax, and state-of-the-art type system. Yet, the community still lacks an easy to use framework for the specification and verification of OCaml code. The working programmers must either re-implement their code in a proof-aware language (and then rely on code extraction), or they must turn themselves into interactive frameworks. Cameleer fills the gap, being a tool for the deductive verification of programs written in OCaml, with a clear

This work is partly supported by the HORIZON 2020 Cameleer project (Marie Skłodowska-Curie grant agreement ID:897873) and NOVA LINCS (Ref. UIDB/04516/2020).

A. Silva and K. R. M. Leino (Eds.): CAV 2021, LNCS 12760, pp. 677–689, 2021.
https://doi.org/10.1007/978-3-030-81688-9_31

focus on proof automation. Cameleer uses the recently proposed GOSPEL [5], a specification language for OCaml. We advocate here the vision of the *specifying programmer*: the person who writes the code should also be able to naturally provide suitable specification. GOSPEL terms are written in a subset of the OCaml language, which makes them more appealing to the regular programmer. Moreover, we believe specification and implementation should co-exist and evolve together, which is exactly the approach followed in Cameleer.

Cameleer takes as input a GOSPEL-annotated OCaml program and translates it into an equivalent counterpart in WhyML, the programming and specification language of the Why3 framework [16]. Why3 is a toolset for the deductive verification of software, clearly oriented towards automated proof. A distinctive feature of Why3 is that it interfaces with several different off-the-shelf theorem provers, namely SMT solvers.

Contributions. To the best of our knowledge, Cameleer is the first deductive verification tool for annotated OCaml programs. It handles a realistic subset of the language, and its interaction with the Why3 verification framework greatly increases proof automation. Our set of case studies successfully verified with the Cameleer tool constitutes, by itself, an important contribution towards building a comprehensive body of verified OCaml codebases. Finally, it is worth noting that the original presentation of GOSPEL was limited to the specification of interface files. In the scope of this work, we have extended it to include implementation primitives, such as loop invariants and ghost code (*i.e.*, code that has no computational purpose and is used only to ease specification and proof effort) evolving GOSPEL from an interface specification language into a more mature proof language.

2 Illustrative Example – Binary Search

Higher-Order Implementation. Fig. 1 presents an implementation of binary search, where the comparison function, `cmp`, is given as an argument to the main function. For the sake of readability, we give the type of arguments and return value of function `binary_search`, but these can be inferred by the OCaml compiler.

The function contract is given after its definition as a GOSPEL annotation, written within comments of the form `(*@ ... *)`. The first line names the returned value. Next, the first precondition establishes that the `cmp` is a total pre order following the OCaml convention: if `x` is smaller than `y`, then `cmp x y < 0`; if `x` is greater than `y`, then `cmp x y > 0`; finally, `cmp x y = 0` if `x` and `y` are equal values[1]. It is worth noting that GOSPEL, hence Cameleer, assumes `cmp` to be a pure function (*i.e.*, a function without any form of side-effects). The second precondition requires the array to be sorted according to the `cmp` relation. Finally, the last two clauses capture the possible outcomes of execution: the regular postcondition (`ensures` clause) states the returned index is within the bounds of `a` and its value is equal to `v`; the exceptional postcondition (`raises`)

[1] For the sake of space, we omit the definition of predicate `is_total_pre_order`.

```
let binary_search (cmp: 'a -> 'a -> int) (a: 'a array) (v: 'a) : int =
  let l = ref 0 in
  let u = ref (length a - 1) in
  let exception Found of int in
  try while !l <= !u do
      (*@ variant   !u - !l *)
      (*@ invariant 0 <= !l && !u < length a *)
      (*@ invariant forall i. 0 <= i < length a -> cmp a.(i) v = 0 ->
            !l <= i <= !u *)
      let m = !l + (!u - !l) / 2 in
      let c = cmp a.(m) v in
      if c < 0 then l := m + 1
      else if c > 0 then u := m - 1
      else raise (Found m)
    done;
    raise Not_found
  with Found i -> i
(*@ i = binary_search cmp a v
      requires is_total_pre_order cmp
      requires forall i j. 0 <= i <= j < length a -> cmp a.(i) a.(j) <= 0
      ensures  0 <= i < length a && compare a.(i) v = 0
      raises   Not_found -> forall i. 0 <= i < length a -> cmp a.(i) v <> 0 *)
```

Fig. 1. Binary search implemented as a functor.

states that whenever exception `Not_found` is raised, there is no such index within bounds whose value is equal to `v`. As usual in deductive verification, the presence of the `while` loop requires one to supply a loop invariant. Here, it boils down to the two `invariant` clauses, which state the limits of the search space are always within the bounds of `a` and that for every index `i` for which `a.(i)` is equal to `v`, then `i` must be within the limits of the current search space. We also provide a decreasing measure (`variant`) in order to prove loop termination.

Assuming file `binary_search.ml` contains the program of Fig. 1, starting a proof with Cameleer is as easy as typing `cameleer binary_search.ml` in a terminal. Users are immediately presented with the Why3 IDE, where they can conduct the proof. Twelve verification conditions are generated for `binary_search`: two for loop invariant initialization, four loop invariant preservation (two for each branch of `if..then..else`), two for safety (check division by zero and index in array bounds), two for loop termination (one for each branch), and finally one for each postcondition. All of these are easily discharged by SMT solvers.

Functor-Based Implementation. The implementation in Fig. 2 depicts (the skeleton of) an alternative implementation of the binary search routine. Instead of passing the comparison function as an argument of `binary_search`, here the functor `Make` takes as argument a module of type `OrderedType`, which provides a monomorphic comparison function over a type `t`. This is the same approach found in the OCaml standard library, namely in the `Set` and `Map` modules. The

`@logic` attribute instructs Cameleer that `cmp` is both a programming and logical function. This is what allows us to provide the axiom about the behavior of `cmp`.

Other than the call to `Ord.cmp`, the implementation and specification of `binary_search` does not change, hence we omit it here. When fed into Cameleer, the functorial implementation generates the *exact same* twelve verification conditions as the higher-order counterpart, all of them easily discharged as well. Thus, the use of a functor does not impose any verification burden, showing the flexibility of Cameleer to handle different idiomatic OCaml programming styles.

```
module type OrderedType = sig
  type t

  val[@logic] cmp: t -> t -> int
  (*@ axiom total_pre_order: is_total_pre_order cmp *)
end

module Make (Ord: OrderedType) = struct
  let binary_search a v =
    ...
    try while !l <= !u do
        ...
        let c = Ord.cmp a.(m) in
        ...
    (*@ i = binary_search a v ... *)
end
```

Fig. 2. Binary search implemented as a functor.

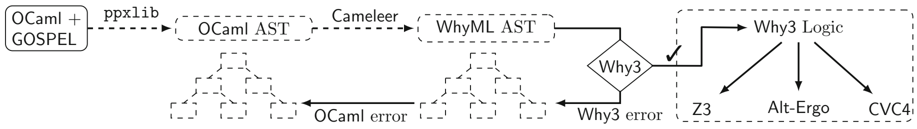


Fig. 3. Cameleer verification workflow.

3 Implementation

Cameleer Workflow. Figure 3 depicts the verification workflow of the Cameleer tool. We use the GOSPEL toolchain[2], in order to parse and manipulate (via the `ppxlib` library) the abstract syntax tree of the GOSPEL-annotated OCaml program. A dedicated parser and type-checker (extended to handle implementation features) treat GOSPEL special comments and attach the generated specification to nodes in the OCaml AST. Cameleer translates the decorated AST into an

[2] https://github.com/ocaml-gospel/gospel.

equivalent WhyML representation, which is then fed to Why3. The Why3 type-and-effect system might reject the input program, in which case the reported error is propagated back to the level of the original OCaml code. Otherwise, if the translated program fits Why3 requirements, the underlying VCGen computes a set of verification conditions that can then be discharged by different solvers. Throughout all this pipeline, the user only has to write the OCaml code and GOSPEL specification (represented in Fig. 3 as a full-lined box), while every other element is automatically generated (dash-lined boxes). The user never needs to manipulate or even care about the generated WhyML program. In short, the Cameleer user intervenes in the beginning and in the end of the process, *i.e.*, in the initial specifying phase and in the last step, helping Why3 to close the proof. Our development effort currently amounts to 1.8K non-blank lines of OCaml code.

Translation into WhyML. The core of Cameleer is a translation from GOSPEL-annotated OCaml code into WhyML. In order to guide our implementation effort, we have defined such a translation as a set of inductive inference rules between the source and target languages [26]. Here, rather than focusing on more fundamental aspects, we give a brief overview of how the translation works in practice.

OCaml and WhyML are both dialects of the ML-family, sharing many syntactic and semantics traits. Hence, translation of OCaml expressions and declarations into WhyML is rather straightforward: GOSPEL annotations are readily translated into WhyML specification, while supported OCaml programming constructions (including ghost code) are easily mapped into semantically-equivalent WhyML constructions. Consider, for instance the following piece of OCaml code:

```
type 'a non_empty_list = { self: 'a list }
(*@ invariant self <> [] *)

let[@ghost] hd (l: 'a non_empty_list) = match l with
  | [] -> assert false
  | x :: _ -> x
(*@ r = hd l
      ensures match l with
              | [] -> false
              | x :: _ -> r = x *)
```

For such case, Cameleer generates the following WhyML program:

```
type non_empty_list 'a = { self: list 'a }
invariant { self <> Nil }

let ghost hd (l: non_empty_list 'a)
  returns { r -> match l with
                 | Nil -> false
                 | Cons x _ -> x = r end }
= match l with
  | Nil -> absurd
  | Cons x _ -> x end
```

Other than the small syntactic differences, the generated WhyML program is identically to the original OCaml one. In particular, the @ghost annotation generates a ghost function in WhyML, while the assert false expression (which is treated in a special way by the OCaml type-checker) is translated into the absurd construction, with the same semantics. Supplied annotations, in this case postcondition and type invariant, are readily mapped into equivalent specification.

The translation of the OCaml module language is more interesting and involved. A WhyML program is a list of modules, a module is a list of top-level declarations, and declarations can be organized within *scopes*, the WhyML unit for namespaces management. However, there is no dedicated syntax for functors on the Why3 side. These are represented, instead, as modules containing only abstract symbols [17]. Thus, when translating OCaml functors into WhyML, we need to be more creative. If we consider, for instance, the Make functor from Fig. 2, Cameleer will generate the following WhyML program:

```
scope Make
  scope Ord
    type t

    val function cmp t t : int
    axiom total_pre_order: is_total_pre_order cmp
  end

  let binary_search a v = ...
end
```

The functor argument Ord is encoded as a nested scope inside Make. This means the binary_search implementation can access any symbol from the Ord namespace, via name qualification (*e.g.*, Ord.t and Ord.cmp).

Interaction with Why3. One distinguishing feature of the Why3 architecture is that it can be extended to accommodate new front-end languages [32, Chap. 4]. Building on the devised OCaml to WhyML translation scheme, we use the Why3 API to build an in-memory representation of the WhyML program. We also register OCaml as an admissible input language for Why3, which amounts to instructing Why3 to recognize .ml files as a valid input format and triggering our translation in such case. Following this integration, we can use any Why3 tool, out of the box, to process a .ml file. We are currently using the extract and session tools: the latter to gather statistics about number of generated verification conditions and proof time; the former to erase ghost code.

Limitations of Using Why3. The WhyML specification sub-language and GOSPEL are similar. Moreover, they share some fundamental principles, namely the arguments of functions are not aliased by construction and each data structure carries an implicit representation predicate. However, one can use GOSPEL to formally specify OCaml programs which cannot be translated into WhyML. This is evident when it comes to recursive mutable data structures. Consider, for instance, the `cell` type from the `Queue` module of the OCaml standard library[3]:

```
type 'a cell = Nil | Cons of { content: 'a; mutable next: 'a cell }
```

As we attempt to translate such data type, Why3 emits the following error:

```
This field has non-pure type, it cannot be used in a recursive
type definition
```

Recursive mutable data types are beyond the scope of Why3's type-and-effect discipline [14], since these can introduce arbitrary memory aliasing which breaks the *bounded-mutability* principle of Why3 (*i.e.*, all aliases must be statically-known). The solution would be to resort to an axiomatic memory model of OCaml in Why3, or to employ a richer program logic, *e.g.*, Separation Logic [28] or Implicit Dynamic Frames [31]. We describe such an extension as future work (Sect. 6).

4 Evaluation

In order to assess the usability and performance of Cameleer, we have put together a test suite of over 1000 lines of OCaml code. The reported case studies are all automatically verified. To build our gallery of verified programs we used a combination of Alt-Ergo 2.4.0, CVC4 1.8, and Z3 4.8.6. Figure 4 summarizes important metrics about our verified case studies: the number of generated verification conditions for each example; the total lines of OCaml code, GOSPEL specification, and lines of ghost (these are also included in the number of OCaml LOC), respectively; the time it takes (in seconds) to replay a proof; and finally, if the proof is immediately discharged, *i.e.*, no extra user effort is required other than writing down suitable specification.

Our test bed includes OCaml implementations issued from realistic and massively used programming libraries: the `List.fold_left` iterator and `Stack` module from the OCaml standard library; the Leftist Heap implementation from `ocaml-containers`[4]; finally, the applicative `Queue` module from `OCamlgraph`[5]. We have used Cameleer to verify programs of different nature. These include: numerical programs (*e.g.*, binary multiplication and fast exponentiation); sorting and searching (*e.g.*, binary search and insertion sort); logical algorithms (conversion of a propositional formula into conjunctive normal form); array scanning (finding duplicate values in an array of integers); small-step iterators; data structures implemented as functors (*e.g.*, Pairing Heaps and Binary Search Trees); historical algorithms (checking a large routine by Turing, Boyer-Moore's majority algorithm, FIND by Hoare, and binary tree same fringe); examples in Rustain Leino's forthcoming textbook "Program Proofs"; and higher-order implementations (height of a binary tree computed in CPS). Both small-step iterators and

[3] https://caml.inria.fr/pub/docs/manual-ocaml/libref/Queue.html.
[4] https://github.com/c-cube/ocaml-containers/blob/master/src/core/CCHeap.ml
[5] https://github.com/backtracking/ocamlgraph/blob/master/src/lib/persistentQueue.ml

Case study	# VCs	LOC / Spec. / Ghost	Proof time	Immediate
Applicative Queue	23	25 / 17 / 4	1.26	✓
Arithmetic Compiler	258	235 / 44 / 155	16.31	✗
Binary Multiplication	12	10 / 6 / 0	0.69	✓
Binary Search	37	62 / 40 / 0	1.23	✓
Binary Search Trees	31	20 / 26 / 0	1.45	✗
Checking a Large Routine	16	25 / 15 / 0	0.75	✓
CNF Conversion	93	113 / 47 / 14	2.92	✓
Duplicates in an Array	11	10 / 9 / 0	0.63	✓
Ephemeral Queue	44	40 / 29 / 7	1.34	✓
Even-odd Test	6	6 / 8 / 0	0.55	✓
Factorial	8	10 / 9 / 0	0.64	✓
Fast Exponentiation	5	4 / 5 / 0	0.62	✓
Fibonacci	15	16 / 15 / 2	0.64	✓
FIND Algorithm	6	13 / 7 / 0	0.57	✓
Insertion Sort	17	13 / 34 / 0	1.28	✓
Integer Square Root	11	8 / 15 / 0	0.63	✓
Leftist Heap	161	99 / 178 / 11	4.33	✓
Mjrty	25	33 / 12 / 0	2.56	✓
OCaml List.fold_left	28	5 / 21 / 0	0.79	✗
OCaml Stack	22	25 / 27 / 1	0.89	✓
Pairing Heap	70	65 / 101 / 29	2.30	✗
Program Proofs	63	93 / 54 / 24	1.60	✗
Same Fringe	23	22 / 16 / 0	0.78	✓
Small-step Iterators	46	42 / 52 / 2	2.01	✗
Tree Height CPS	4	8 / 8 / 0	0.80	✓
Union Find	67	36 / 29 / 7	6.19	✓

Fig. 4. Summary of the case studies verified with the Cameleer tool.

the `list_fold` function use a modular approach to reason about iteration [18]. Our largest case study to date is a toy compiler from arithmetic expressions to a stack machine, while Union Find features the most involved, but very elegant, specification. The former is inspired by the presentation in Nielsons' textbook [25]; the latter follows recently proposed specification techniques [7,12] to achieve fully automatic proofs of correctness and termination.

The runtimes shown in Fig. 4 were measured by averaging over ten runs on a Lenovo Thinkpad X1 Carbon 8th Generation, running Linux Mint 20.1, OCaml 4.11.1, and Why3 1.3.3 (developer version). They show that Cameleer can effectively verify realistic OCaml code in a decent amount of time. Following good practices in deductive verification, Cameleer allows the user to write *ghost code* in order to ease proof and specification. The number of lines of ghost code in Fig. 4 stands for ghost fields in record types, ghost functions, and lemma functions. In particular, the arithmetic compiler example uses lemma functions to prove, by induction, results about semantics preservation. Finally, case studies marked with ✗ required some form of manual interaction in the Why3 IDE [9]. These are very simple proofs by induction (of auxiliary lemmas) and case analysis, in order to better guide SMT solvers.

From our experience developing this gallery of verified programs, we believe the required annotation effort is reasonable, although non-negligible. Some case studies, namely the Heap implementations, feature a considerable amount of lines of GOSPEL specification. However, these are classic definitions (*e.g.*, minimum element) and results (*e.g.*, the root of the Heap is the minimum element), which are easily adapted to any variant of Heap implementation.

5 Related Work

Automated Deductive Verification. One can cite Why3, F* [1], Dafny [23], and Viper [24] as successful automated deductive verification tools. Formal proofs are conducted in the proof-aware language of these frameworks, and then executable reliable code can be automatically extracted. In the Cameleer project, we chose to develop a verification tool that accepts as input a program written directly in OCaml, instead of a dedicated proof language. This obviates the need to re-write entire OCaml codebases (*e.g.*, libraries), just for the sake of verification.

Regarding tools that tackle the verification of programs written in mainstream languages, one can cite Frama-C [21] (for the C language), VeriFast [20] (C and Java), Nagini [10] (Python), Leon [22] (Scala), Spec# [3] (C#), and Prusti [2] (Rust). Despite the remarkable case studies verified with these tools, programs written in the these languages can quickly degenerate into a nightmare of pointer manipulation and tricky semantics issues. We argue the OCaml language presents a number of features that make it a better target for formal verification.

Finally, language-based approaches offer an alternative path to the verification of software. Liquid Haskell [34] extends standard Haskell types with Liquid Types [29], a form of refinement types [30], in order to prove properties about realistic Haskell programs [33]. In this approach, verification conditions are generated and discharged during type-checking. This is also its major weakness: in order to remain decidable, the expressiveness of the refinement language is hindered. In Cameleer, the use of GOSPEL allows us to provide rich specification to relevant case studies, while still achieving good proof automation results.

Deductive Verification of OCaml Programs. Prior to our work, CFML [4] and `coq-of-ocaml` [8] were the only available tools for the deductive verification of OCaml-written code, via translation into the Coq proof language. On one hand, CFML features an embedding of a higher-order Separation Logic in Coq, together with a *characteristic formulae* generator. On the other hand, `coq-of-ocaml` compiles non-mutable OCaml programs to pure Gallina code. These two tools have been successfully applied to the verification of non-trivial case studies, such as the correctness and worst-case amortized complexity bound of cycle detection algorithm [19], as well as part of the Tezos' blockchain protocol[6]. However, they

[6] https://clarus.github.io/coq-of-ocaml/examples/tezos/.

still require a tremendous level of expertise and manual effort from users. Also, no behavioral specification is provided with the OCaml implementation. The user must write specification at the level of the generated code, which breaks our vision that implementation and specification must coexist and evolve together.

The VOCaL project aims at developing a mechanically verified OCaml library [6]. One of the main novelties of this project is the combined use of three different verification tools: Why3, CFML, and Coq. The GOSPEL specification language was developed in the scope of this project, as a tool-agnostic language that could be manipulated by any of the three mentioned frameworks. Up to this point, the three mentioned tools were only using GOSPEL for interface specification, and not as a proof language. We believe the Cameleer approach nicely complements the existing toolchains [13] in the VOCaL ecosystem.

6 Conclusions and Future Work

In this paper we presented Cameleer, a tool for automated deductive verification of OCaml programs, with bounded mutability. We use the recently proposed GOSPEL language, which we also extended in the scope of this work, in order to attach formal specification to an OCaml program. Cameleer fulfills a gap in the OCaml community, by providing programmers with a tool to directly specify and verify their implementations. By departing from the interactive-based approach, we believe Cameleer can be an important step towards bringing more OCaml programmers to include formal methods techniques in their daily routines.

The core of Cameleer is a translation from OCaml annotated code into WhyML. The two languages share many common traits (both in their syntax and semantics), so it makes sense to target this intermediate verification language in the first major iteration of Cameleer. We have successfully applied our tool and approach to the verification of several case studies. These include implementations issued from existing libraries, and scale up to data structures implemented as functors and tricky effectful computations. In the future, we intend to apply Cameleer to the verification of even larger case studies.

What We Do Not Support. Currently, we target a subset of the OCaml language which roughly corresponds to `caml-light`, with basic support for the module language (including functors). Also, WhyML limits effectful computations to the cases where alias is information statically known, which limits our support for higher-order functions and mutable recursive data structures. Adding support for the objective layer of the OCaml language would require a major extension to the GOSPEL language and a redesign of our translation into WhyML. Nonetheless, Why3 has been used in the past to verify Java-written programs [15], so in principle an encoding of OCaml objects in WhyML is possible.

We do not support some of the more advanced type features in OCaml, namely Generalized Algebraic Data Types (GADTs) and polymorphic variants. One way to support such constructions would to be extend the type system of Why3 itself, which would likely mean a considerable redesign of the WhyML language.

Another possible route is to extend the core of Cameleer with the ability to translate OCaml code into other, richer, verification frameworks.

Interface with Viper and CFML. In order to augment the class of OCaml programs we can treat, we plan on extending Cameleer to target the Viper infrastructure and the CFML tool. On one hand, Viper is an intermediate verification language based on Separation Logic but oriented towards SMT-based software verification, allowing one to automatically verify heap-dependent programs. On the other hand, the CFML tool allows one to verify effectful higher-order programs. We plan on extending the CFML translation engine, in order to take source-code level GOSPEL annotations into account. Since it targets the rich proof language and type system of Coq, it can in principle be extended to reason about GADTs and other advanced OCaml features. Even if it relies on an interactive proof assistant, CFML provides a comprehensive tactics library that eases proof effort.

Our ultimate goal is to grow Cameleer to a verification tool that can simultaneously benefit from the best features of different intermediate verification frameworks. Our motto: we want Cameleer to be able to verify parts of OCaml code using Why3, others with Viper, and some very specific functions with CFML.

References

1. Ahman, D., et al.: Dijkstra monads for free. In: 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL), pp. 515–529. ACM (2017). https://doi.org/10.1145/3009837.3009878
2. Astrauskas, V., Müller, P., Poli, F., Summers, A.J.: Leveraging Rust Types for Modular Specification and Verification. Proc. ACM Program. Lang. **3**(OOPSLA) (2019). https://doi.org/10.1145/3360573
3. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: an overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30569-9_3
4. Charguéraud, A.: Characteristic formulae for the verification of imperative programs. In: Chakravarty, M.M.T., Hu, Z., Danvy, O. (eds.) ACM SIGPLAN International Conference on Functional Programming (ICFP), pp. 418–430 (2011). https://doi.org/10.1145/2034773.2034828
5. Charguéraud, A., Filliâtre, J.-C., Lourenço, C., Pereira, M.: GOSPEL—providing OCaml with a formal specification language. In: ter Beek, M.H., McIver, A., Oliveira, J.N. (eds.) FM 2019. LNCS, vol. 11800, pp. 484–501. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30942-8_29
6. Charguéraud, A., Filliâtre, J.C., Pereira, M., Pottier, F.: VOCAL - A Verified OCAml Library. In: ML Family Workshop (2017). https://hal.inria.fr/hal-01561094
7. Charguéraud, A., Pottier, F.: Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits. J. Autom. Reason. **62**(3), 331–365 (2017). https://doi.org/10.1007/s10817-017-9431-7
8. Claret, G.: Program in Coq. (Programmer en Coq). Ph.D. thesis, Paris Diderot University, France (2018). https://tel.archives-ouvertes.fr/tel-01890983

9. Dailler, S., Marché, C., Moy, Y.: Lightweight interactive proving inside an automatic program verifier. In: 4th Workshop on Formal Integrated Development Environment (F-IDE) (2018)
10. Eilers, M., Müller, P.: Nagini: a static verifier for python. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 596–603. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_33
11. Filliâtre, J.C.: Deductive software verification. Int. J. Softw. Tools Technol. Transf. (STTT) **13**(5), 397–403 (2011). https://doi.org/10.1007/s10009-011-0211-0
12. Filliâtre, J.C.: Simpler proofs with decentralized invariants. J. Log. Algebraic Methods Program. (2020, to appear). https://hal.inria.fr/hal-02518570
13. Filliâtre, J.C., et al.: A toolchain to produce verified OCaml libraries. Research report, Université Paris-Saclay (2020). https://hal.archives-ouvertes.fr/hal-01783851
14. Filliâtre, J.C., Gondelman, L., Paskevich, A.: A pragmatic type system for deductive verification. Research report, Université Paris Sud (2016). https://hal.archives-ouvertes.fr/hal-01256434v3
15. Filliâtre, J.-C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 173–177. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73368-3_21
16. Filliâtre, J.-C., Paskevich, A.: Why3—where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 125–128. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_8
17. Filliâtre, J.-C., Paskevich, A.: Abstraction and genericity in Why3. In: Margaria, T., Steffen, B. (eds.) ISoLA 2020. LNCS, vol. 12476, pp. 122–142. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-61362-4_7
18. Filliâtre, J.-C., Pereira, M.: A modular way to reason about iteration. In: Rayadurgam, S., Tkachuk, O. (eds.) NFM 2016. LNCS, vol. 9690, pp. 322–336. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40648-0_24
19. Guéneau, A., Jourdan, J., Charguéraud, A., Pottier, F.: Formal proof and analysis of an incremental cycle detection algorithm. In: Harrison, J., O'Leary, J., Tolmach, A. (eds.) 10th International Conference on Interactive Theorem Proving, (ITP). LIPIcs, vol. 141, pp. 18:1–18:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019). https://doi.org/10.4230/LIPIcs.ITP.2019.18
20. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: a powerful, sound, predictable, fast verifier for C and Java. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 41–55. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_4
21. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: a software analysis perspective. Formal Aspects Comput. **27**(3), 573–609 (2015). https://doi.org/10.1007/s00165-014-0326-7
22. Kuncak, V.: Developing verified software using leon. In: Havelund, K., Holzmann, G., Joshi, R. (eds.) NFM 2015. LNCS, vol. 9058, pp. 12–15. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-17524-9_2
23. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20

24. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: a verification infrastructure for permission-based reasoning. In: Jobstmann, B., Leino, K.R.M. (eds.) VMCAI 2016. LNCS, vol. 9583, pp. 41–62. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49122-5_2
25. Nielson, H.R., Nielson, F.: Semantics with Applications: An Appetizer. Undergraduate Topics in Computer Science. Springer, Heidelberg (2007). https://doi.org/10.1007/978-1-84628-692-6
26. Pereira, M., Ravara, A.: Cameleer: a deductive verification tool for OCaml. CoRR (2021). https://arxiv.org/abs/2104.11050
27. Régis-Gianas, Y., Pottier, F.: A hoare logic for call-by-value functional programs. In: Audebaud, P., Paulin-Mohring, C. (eds.) MPC 2008. LNCS, vol. 5133, pp. 305–335. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70594-9_17
28. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science, LICS 2002, pp. 55–74. IEEE Computer Society, USA (2002)
29. Rondon, P.M., Kawaguchi, M., Jhala, R.: Liquid types. In: Gupta, R., Amarasinghe, S.P. (eds.) 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 159–169. ACM (2008). https://doi.org/10.1145/1375581.1375602
30. Rushby, J., Owre, S., Shankar, N.: Subtypes for specifications: predicate subtyping in PVS. IEEE Trans. Softw. Eng. **24**(9), 709–720 (1998). https://doi.org/10.1109/32.713327
31. Smans, J., Jacobs, B., Piessens, F.: Implicit dynamic frames. ACM Trans. Program. Lang. Syst. **34**(1), 2:1–2:58 (2012). https://doi.org/10.1145/2160910.2160911
32. The Why3 Development Team: The Why3 platform, version 1.3.3. LRI, CNRS & Univ. Paris-Sud & INRIA Saclay (2020). http://why3.lri.fr/manual.pdf
33. Vazou, N., Breitner, J., Kunkel, R., Horn, D.V., Hutton, G.: Theorem proving for all: equational reasoning in liquid haskell (functional pearl). In: Wu, N. (ed.) 11th ACM SIGPLAN International Symposium on Haskell, pp. 132–144. ACM (2018). https://doi.org/10.1145/3242744.3242756
34. Vazou, N., Seidel, E.L., Jhala, R., Vytiniotis, D., Jones, S.L.P.: Refinement types for haskell. In: Jeuring, J., Chakravarty, M.M.T. (eds.) 19th ACM SIGPLAN International Conference on Functional Programming (ICFP), pp. 269–282. ACM (2014). https://doi.org/10.1145/2628136.2628161

LLMC: Verifying High-Performance Software

Freark I. van der Berg(✉)

Formal Methods and Tools, University of Twente,
Enschede, The Netherlands
f.i.vanderberg@utwente.nl

Abstract. Multi-threaded unit tests for high-performance thread-safe data structures typically do not test all behaviour, because only a single scheduling of threads is witnessed per invocation of the unit tests. Model checking such unit tests allows to verify all interleavings of threads. These tests could be written in or compiled to LLVM IR. Existing LLVM IR model checkers like DIVINE and Nidhugg, use an LLVM IR interpreter to determine the next state. This paper introduces LLMC, a multi-core explicit-state model checker of multi-threaded LLVM IR that translates LLVM IR to LLVM IR that is *executed* instead of interpreted. A test suite of 24 tests, stressing data structures, shows that on average LLMC clearly outperforms the state-of-the-art tools DIVINE and Nidhugg.

1 Introduction

High-performance software often uses thread-safe data structures to allow multiple threads access to the data, without corrupting it. Unit tests for such data structures typically do not test all behaviour, because the thread scheduler of the run-time environment non-deterministically chooses only a single interleaving. Thus, only a single trace is witnessed each time the unit test is invoked. If we would *model check* [1] these unit tests, we can witness all possible traces by exploring all thread schedules. Because it does not depend on the run-time environment, model checking can become part of a continuous integration pipeline, enabling push-button verification of multi-threaded software.

These thread-safe data structures can be written in or compiled to LLVM IR, the intermediate representation of the LLVM Project [2]. The LLVM Project is a collection of modular and reusable compiler and toolchain technologies. Many front-ends for LLVM IR exist, for example for C, C++, Java, Ruby, and Rust, potentially allowing an LLVM IR model checker to be usable for many languages.

1.1 Related Work

Model checkers that operate on LLVM IR already exist, for example DIVINE, Nidhugg, RCMC and LLBMC. DIVINE [3] is a stateful multi-core model checker of multi-threaded LLVM IR. It has many features such as capturing I/O during model checking, SC and TSO memory models, library support such as `libc` and `libpthread`. Input programs are linked with DIVINE's operating system layer, DiOS, and are interpreted as a whole on the DiVM virtual machine.

DIVINE detects memory operations to thread-private memory, by traversing the heap on-the-fly and recognizing if a memory-object is either known only to one thread or to multiple [4]. In the former case, memory operations to that memory-object can be *collapsed*, i.e. joined with the previous instruction.

Nidhugg [5] is a stateless multi-core model checker of multi-threaded LLVM IR that uses an LLVM IR interpreter. It features a sophisticated partial-order reduction, *rfsc* [6], that categorizes traces according to which read reads from which write and traverses only one trace in each category. In practice this reduction is quite powerful. However, Nidhugg comes with a caveat: because Nidhugg is stateless, common prefixes of traces are traversed once per trace instead of once in total. This down-side of a stateless approach becomes more pronounced with longer and more often occurring common traces. Moreover, Nidhugg might not terminate in the presence of infinite loops.

RCMC [7] is also a stateless LLVM IR model checker. During execution within its LLVM IR interpreter, it keeps track of a happens-before graph of all observed memory operations. Using this, RCMC can determine the possible values a read can observe, without simply executing all interleavings of all threads. Unlike Nidhugg, it does not support heap memory and is only released in binary form.

CBMC [8] is a bounded model checker for C and C++ programs, using SMT solving to check for memory safety, exceptions, undefined behaviour and assertions. Loops and recursion are a problem for CBMC when their bound cannot be determined: one needs to set an upper bound on the number of unwindings.

LLBMC [9] is similar to CBMC, using SMT-solving to find bugs, but only for single-threaded C/C++ programs and it operates on LLVM IR.

Other, less related tools include SMACK [10], SeaHorn [11] and KLEE [12].

1.2 Contribution

This paper introduces LLMC 0.2, a stateful multi-core model checker of multi-threaded LLVM IR. Instead of using an LLVM IR interpreter like DIVINE, Nidhugg and LLMC 0.1 [13], it transforms input LLVM IR to LLVM IR that implements the DMC API, the next-state interface to the model checker DMC [14]. We call this transformation process LL2DMC and combined with DMC (Fig. 1), it allows for up to three orders of magnitude higher throughput (states/s) than DIVINE. At present, LLMC lacks sophisticated state space reductions, causing state space sizes of roughly two orders of magnitude larger than DIVINE. We compared LLMC to DIVINE and Nidhugg using a test suite covering various data structures. Overall, despite the lack of sophisticated reductions, LLMC is on average an order of magnitude faster than DIVINE and ∼3.8x faster than Nidhugg. Additionally, LLMC is able to compute the state spaces of the tests where DIVINE or Nidhugg fail.

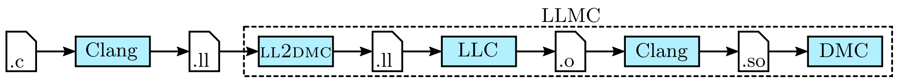


Fig. 1. The flow of how an LLVM IR input program is verified in LLMC.

2 LLMC: Low-Level Model Checker

This section explains how the transformation process (LL2DMC) transforms the input LLVM IR of a program to LLVM IR that implements the DMC API. LLMC supports LLVM IR compiled from C and C++, by handling a number of builtins (e.g. `__atomic_*` for atomic memory operations), part of `libpthread` (for thread support), `libc` (e.g. for memory allocation) and global constructors.

2.1 DMC Model Checker

The model created by LL2DMC is given to DMC to explore. DMC interacts with the model via the DMC API (NEXTSTATE API and DTREE API combined) as illustrated in Fig. 2: after requesting the initial state from the model, DMC continues to request successor states, until the state space has been generated. A state is a vector of 32-bit integers; two states need not be of the same length.

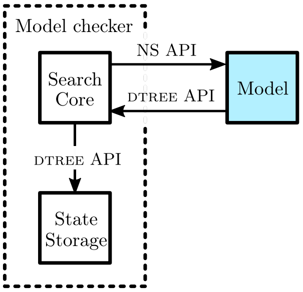


Fig. 2. DMC model checker

The states are stored in the concurrent compression tree DTREE [14], allowing lossless compression, fast insertion and duplicate detection of states. When inserted, states are given a unique `StateID`. A `StateID` can be stored in states as well, thus allowing the creation of a DAG of states: a *root-state* and *sub-states*. Additionally, DTREE allows incremental updates to a state, without having the actual contents of the state and it allows partial reconstruction of states. This *delta interface* uses the `StateID` to identify states and can avoid needless copying of entire states, increasing performance. DMC exposes these DTREE features as part of the DMC API [14].

2.2 Input Language to LL2DMC: LLVM IR

To understand how LLMC handles input LLVM IR [2], we briefly explain it here. LLVM IR supports control flow by way of basic blocks. Basic blocks are a list of instructions that execute sequentially. The last instruction of a basic block is a terminator instruction, such as a branch (jump) instruction or `return` statement.

LLVM IR uses single static assignment form for register values. To support data flow depending on control flow, ϕ-nodes exist. These nodes are instructions at the beginning of a basic block that take a value depending on the basic block from which was jumped to the basic block containing the ϕ-nodes.

2.3 Output of LL2DMC: Model Implementing DMC API

The output of LL2DMC is a model that implements the NEXTSTATE API part of the DMC API of the model checker DMC [14]. The NEXTSTATE API requires two interfaces from a model: one to communicate the initial state and one to generate next states, given a state.

The *initial state* of a model generated by LL2DMC is as if one just started the program: registers are unused, global memory is initialized to 0 and a call to the global constructor (`@llvm.global_ctors`) is set up. Global constructors are functions that are called before `main`, which are used to initialize memory and miscellaneous initialization, such that the executable is set up properly before `main` is invoked. Having the initial state in this manner, allows the global constructor to be part of the state space and thus be checked as well.

Starting with the initial state, DMC will keep asking the model to generate the next states for a given state, by invoking the *next-state interface* of the model, until there are no more new states of which to request next states. Given a state, the next-state interface determines the states reachable from that state. In the case of a model generated by LL2DMC, first the global constructors of the modelled program are explored, thus faults in global constructors are detected. When the global constructors are completed, a call to `main` is set up. At this point, the exploration is performed until no new states are visited.

2.4 State Space Exploration

This section describes the next-state function and how it is generated from LLVM IR. Figure 3 describes what a state looks like. A state contains information not unlike what an operating system keeps track of [15]. All instructions are mapped to a unique index, such that the `PC` (program counter) uniquely identifies the current position in code. The field `Thread Results` holds the return values of finished threads; the field `#threads` specifies the number of threads in the current state. The remainder of the state constitutes a list of per-thread data.

Each thread has its own `PC` and can independently manipulate it by function calls or branching. `Status` fields are used to indicate whether the thread/program is running, done or failed. Each thread has its own set of `Registers`, the current state of LLVM IR registers. The size of `Registers` is determined by the function requiring the largest number of LLVM IR registers. Function calls manipulate these registers and the list of stack frames described by `Previous frame`.

A `Field` is a `StateID` to a sub-state, as described in Sect. 2.1. The separation into a root-state and sub-states allows sub-states to grow and the state storage component of DMC, DTREE, to compress them using tree compression [14]. It also allows the use of the delta interface: a write to memory can be simply translated

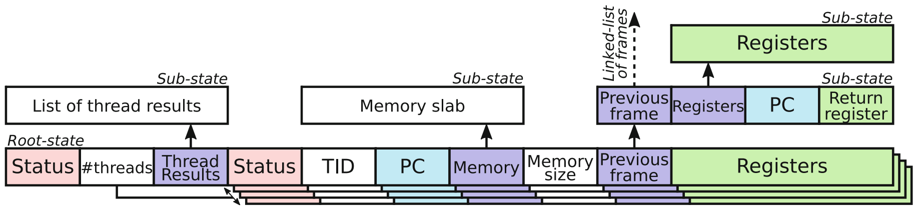


Fig. 3. A description of the state used by LLMC.

to a single, efficient call, taking the current `Memory` index, the offset to write to and the new data. The resulting index can be written to `Memory`.

A single LLVM IR instruction in the program is translated to many LLVM IR instructions in the model. We will distinguish LLVM IR registers in the model from registers in the source program by calling the former *model-registers*. In general, a single LLVM IR instruction is translated to a single step with three phases: In the *Preamble* phase, operands to the source LLVM IR instruction are remapped to model-registers and loaded from `Registers` or `Memory`. In the *Action* phase, the source LLVM IR instruction is cloned, with the operands remapped to the LLVM IR model-registers set up during the Preamble phase. In the *Epilogue* phase, if the source LLVM IR instruction assigns a value to a register, the value returned by the cloned instruction is written to `Registers`.

Listing 1 illustrates how a step is performed as part of the next-state function. Multiple steps can be performed as part of the same transition (line 8), as long as the changes are local to the thread (line 4). This is explained in more detail in Sect. 2.5. The `step` function is called for every thread in the state vector.

2.4.1 Register Manipulation

Note that the `Registers` are not separated into a sub-state, like `Memory`. We chose this such that simple register manipulating LLVM IR instructions would have no need for an indirection and directly translate to an identical instruction, with its operands mapped such that they are loaded from the `Registers` and the return value of the instruction written back to the corresponding register. This allows us to trivially collapse such instructions, combining the Preamble phases, requiring dependencies only to be loaded once.

2.4.2 Memory Instructions

Memory instructions such as loads and stores can be directly mapped to the delta interface, reading or writing only a part of the `Memory` sub-state. There is no distinction between memory allocated on the stack (`alloca`) and on the heap

Listing 1 In the next-state function, the `step` function is called for each thread.

```
 1 void step(StateVector sv, int threadID)
 2   bool onlyLocal = true; # true while handling commutative instructions
 3   bool emit = false;     # set to true when new state is to be emitted
 4   while(sv.threads[threadID].pc > 0 and onlyLocal)
 5     switch(sv.threads[threadID].pc)
 6       case 0: break; # not running, do nothing
 7       case SomePC: # PC of first instruction of group
 8         # statically collapsed instructions: preamble, action, epilogue
 9         # sv.threads[threadID].pc, onlyLocal and emit may change
10       ...
11   if(emit) MC.insert(sv); # emit new state if needed
```

(`malloc`): both allocate memory by growing the `Memory` sub-state. The returned pointer describes which thread created the memory and the offset within the sub-state. Any thread can write to and read from any such memory location. At present, memory cannot be freed, so `free` has no effect. Because of the tree compression, this has no detrimental effect on memory usage, but does mean LLMC currently does not detect `free`-related bugs.

2.4.3 Branching, Function Calls and Threading

To support control flow in LLMC, the `PC` can be changed to the index assigned to the first instruction in the target basic block. If the target basic block contains ϕ-nodes, those registers are updated to the value corresponding to the basic block we are branching from.

Function calls set up a new stack frame with the current `Registers`, `PC` and where to write the return value, then pushes it to the linked list of frames pointed to by `Previous frame`. A return from a function pops the top frame from the list of frames, copies the `Registers` into the state vector, updates the `PC` and writes the return value into the right register. There is no bound on the number of frames; the last frame has `Previous frame` set to 0, indicating no next frame.

Threads are created (`pthread_create`) by enlarging the root state with enough space to fit another thread and incrementing `#threads`. When a thread is done, it is marked as such, but not removed from the state vector. This is to retain the memory allocated by a thread. Due to the compression of DTREE, it has little impact on the memory foot print of the state space. The return value from the thread is added to `Thread results`, where it can be read (`pthread_join`).

2.5 State Space Reduction

Instructions that only have an effect local to a thread do not change the behaviour of another thread. Such instructions are commutative; their respective ordering is not relevant. Thus, such instructions can be collapsed with the previous or next instruction. For example, instructions that read and write only to registers of a thread are local instructions and do not influence another thread. Branching and function calls are other such commutative instructions.

LLMC collapses commutative instructions statically as well as dynamically. The latter is needed to collapse instructions after conditional control flow, because statically the condition is unknown. On-the-fly, the condition is evaluated, the branch taken and it is determined if the next instruction can be collapsed.

2.5.1 Thread-Private Memory LLMC collapses all such commutative instructions, with the important exception of memory operations on memory only accessible to the current thread (memory operations to memory accessible to other threads are never collapsed). This requires knowledge on what memory each thread can access, which LLMC currently does not track. DIVINE implements [4] this by traversing the memory graph in every state, using a run-time type system to identify pointers and how to follow them (edges); each allocation yields a node.

Nidhugg uses a partial-order-reduction [6] that takes into account from which write a value read by a read originates. In this process, memory operations to thread-private memory are indeed collapsed, because a read can read only a single value: the last value written by the thread itself. The current version of LLMC does not feature an on-the-fly state space reduction for memory operations. Instead, we preprocess the input LLVM IR and statically annotate memory operations that cannot be proven to be local to a thread. While this does reduce the state space, because many operations are to stack variables that remain thread-private, it can only approach the on-the-fly reductions of DIVINE and Nidhugg.

3 Evaluation

Table 1 shows a feature comparison between the tools mentioned in Sect. 1.1. The table shows that RCMC and CBMC do not support dynamic memory in the presence of multiple threads. This limits their usability for our use case, model checking multi-threaded tests of data structures, since numerous thread-safe data structures use dynamic memory. Furthermore, RCMC, CBMC and LLBMC do not support infinite loops and only have limited support for spin-locks. More complex infinite loops like appending a new node in the Michael-Scott queue [17] using `compare-and-swap` are not supported. Thus, we focus on an experimental comparison between LLMC, DIVINE and Nidhugg on execution time, memory footprint of the state space and scalability across multiple threads, since all three tools support using multiple threads for model checking.

Table 1. A feature comparison between the tools mentioned in Sect. 1.1.

			Supported Features									Checks					
Tool	Version	Source	Memory models	Threads	Heap memory	Infinite loops	Global constructors	I/O	Filesystem emulation	__atomic_* intrinsics	atomic {load,store}	Memory access	assert()	Out of Memory	Invalid free	Double free	Memory leaks
DIVINE [3]	4.4.2 (5494190)	LLVM IR	ST[a]	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓
Nidhugg [5]	0.2 (45664bc)	LLVM IR	STPWA[a]	✓	✓	✓	∼[d]				✓	✓	✓	✓	✓	✓	
RCMC [7]	n/a	LLVM IR	(W)RC11	✓	∼[b]	∼[c]	∼[d]			✓	✓					✓	
CBMC [8]	5.10 (ef00f47)	C/C++	STP[a]	✓	∼[b]	∼[c]	∼[d]						✓		✓	✓	✓
LLBMC [9]	2013.1	LLVM IR	n/a		✓	∼[c]	∼[d]						✓		✓	✓	
LLMC [13]	0.1	LLVM IR	STP[a]	✓	✓	✓	✓			✓	✓	✓	✓				
LLMC	0.2 (a732c63)	LLVM IR	S[a]	✓	✓	✓	✓			✓	✓	✓	✓				

[a] Models [16]: **S**) Sequentially consistent; **T**) TSO; **P**) PSO; **W**) POWER; **A**) ARM.
[b] Not supported in combination with threads.
[c] Only trivial spin-locks are supported.
[d] Threads within global constructors not supported.

We ran our experiments on a Dell R930 with 4 E7-8890-v4 CPUs totaling 96 cores and 2 TiB RAM. All sources were compiled using GCC 9.3.0.

3.1 Test Suite

We tested the tools using four real-world concurrent LLVM IR data structures, one concurrent algorithm and one protocol. Sources for all tests are available online[1]. We instantiate the tests with various combinations of threads and number of elements inserted, processed or dequeued. All combinations are listed later, in Table 2. These six tests cover different classes of problem types, different shapes of state spaces, and serve to illustrate the strengths and weaknesses of the tools:

- **SortedLinkedList** ● illustrates a concurrency problem where a number of elements are inserted by a number of threads, with a single outcome: all paths converge to one state. Elements can be inserted throughout the chain.
- **LinkedList** ■, similar to SortedLinkedList ●, but with various outcomes, because the list is not sorted. It has high contention on the head of the chain.
- **Prefixsum** ⬢ is a concurrent approach to determine all sums up to any index in an array. It highlights the ability of the model checker to determine thread-private memory, because the two-pass prefixsum algorithm actually partitions the problem into separate per-thread problems that require no communication and one single-threaded part.
- **Hashmap** ♦ illustrates a concurrency problem where a key is inserted using `compare-and-swap`, followed by either atomically storing the value or busy-waiting on the value, if the key already exists (`findOrPut` [18]). The latter involves atomically loading the value until a non-empty value is loaded.
- **MSQ** ▲ is the well-known Michael-Scott queue [17]. It is similar to LinkedList ■, with the addition of dequeue operations, which may return nothing when the queue is empty. The dequeuer can be made *blocking* by calling `dequeue` until it successfully dequeues an element; this is done in ▲2 and ▲4.
- **Philosophers** ▼ is the Dining Philosophers Problem [19], a commonly used protocol to illustrate issues in concurrent resource management. It involves P philosophers and P forks; each philosopher grabs their left fork, then the right, then puts the right fork back, then the left. This is repeated R times. The crux is that each fork is a shared resource for two philosophers. For our tests suite, this illustrates contention on multiple elements in a single array.

These tests highlight the strengths and weaknesses of each tool using real-world data structures and algorithms. The well-known Michael-Scott queue ▲ for example is used in many software packages. They reflect different *kinds* of state spaces: LinkedList ■ focuses on "wide" state spaces, with many end states; SortedLinkedList ● examples state spaces that go wide, but converge into a single end state; Prefixsum ⬢ highlights the model-checker's ability to detect thread-local memory: model checkers that can detect this have a narrow state space, otherwise a model checker will explore all interleavings.

[1] https://github.com/bergfi/llmc/tree/cav2021/tests/performance.

3.2 Observations and Considerations

For each model, we verified that all expected end states were reachable. For example for ■1, we manually verified that all $8!/(4!4!) = 70$ possible outcomes of the linked list were generated.

We witnessed DIVINE returning varying state space sizes across different runs on the same test when using multiple threads, indicating a concurrency problem. It also occasionally crashed, most often when using 192 threads. Even though this indicates the answers DIVINE gives might not be correct, we opted to include the results, assuming they would at least provide an indication of the performance.

Furthermore, we did run RCMC on a number of tests. RCMC often runs out of memory before crashing; likely the result of an infinite loop. For even some small tests, it could not finish within 100x the time other tools needed.

3.3 Experimental Results

Figure 4 shows the results of LLMC compared to DIVINE on state space exploration time (4a) and Nidhugg on wall-clock time (4b) when applied to the models from Table 2. These graphs indicate relative performance: the uppermost (blue) line for example indicates the line where LLMC is 100x faster. Figure 4c compares LLMC (lower data points) and DIVINE (upper data points) on the memory compression of the state spaces they generate. Figure 4d compares LLMC (upper data points) and DIVINE (lower data points) on the throughput of states per second.

3.3.1 LLMC vs DIVINE

Looking at the results in Fig. 4a, we see that LLMC outperforms DIVINE by at least 5x in all test cases except Prefixsum ⬢ and two SortedLinkedList ● tests. LLMC suffers in the Prefixsum ⬢ tests because of the lack of dynamic thread-private memory detection. This results in significantly larger state spaces, up to three orders of magnitude for ⬢4, as seen in Fig. 4c.

Comparing the sorted ● and non-sorted ■ linked list cases, we notice LLMC is able to outperform DIVINE in the non-sorted cases by higher factors than the sorted cases. This difference can be explained by that the two tools generate more similarly sized state spaces for non-sorted ■ cases, but not for sorted ● cases. For example, LLMC generates ∼14.4x more states than DIVINE for ●4, but only ∼2.2x more for ■4. This highlights LLMC is lacking a reduction technique, which works for DIVINE in the sorted cases, but not as well for the non-sorted cases.

For the two Hashmap ♦ cases that both tools completed, LLMC outperforms DIVINE by 8.4x and 157x. Since the hash map is a single global memory object all threads can access, LLMC does not have the disadvantage of lacking a dynamic thread-private memory reduction. DIVINE crashed for the two other ♦ test cases.

DIVINE is unable to complete two of the four Michael-Scott queue ▲ tests, crashing out, the others are verified 86x and 272x faster by LLMC than by DIVINE.

As the complexity of the Philosopher ▼ test cases increases, LLMC increasingly outperforms DIVINE. The two tools generate similarly sized state state spaces,

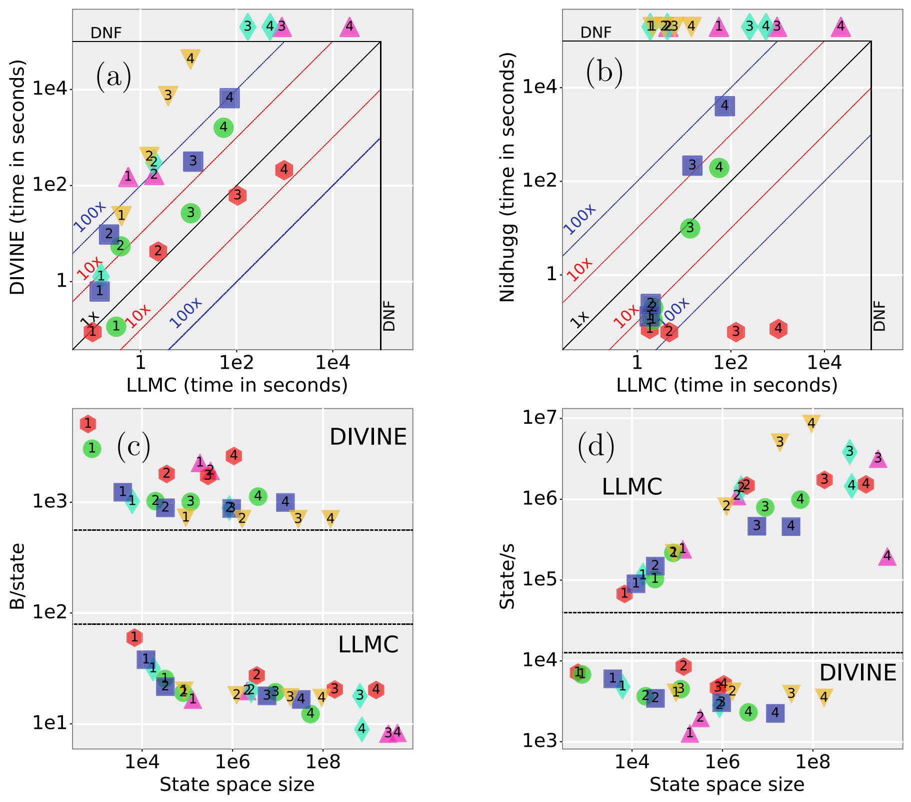


Fig. 4. All experimental results, see Table 2 for a legend. Results above the DNF line mean the tool on the y-axis Did Not Finish, not supporting the test.

because the high contention leaves relatively few memory instructions to be collapsed by DIVINE's reduction, thus levelling the playing field.

In summary, LLMC is able to outperform DIVINE in most of the test cases, mostly between 10x–100x faster, with an outlier as high as 2450x faster (▼4). This highlights the performance difference, as on average LLMC visits ∼1.4M

Table 2. The six tests with various combinations of number of threads and elements, totaling 24 input programs. MSQ ▲ configurations describe a combination of **E**nqueuers and (**[B]**locking) **D**equeuers in parallel (‖) and sequential (;).

SortedLinkedList			LinkedList			Prefixsum			Hashmap			MSQ		Philosophers		
	Threads	Elements		Thrds	Elems		Thrds	Elems		Thrds	KV-pairs		Configuration		*P*	*R*
● 1	2	8	■ 1	2	8	⬢ 1	2	80	◆ 1	3	9	▲ 1	E‖E‖D‖D	▼ 1	4	2
● 2	3	6	■ 2	3	6	⬢ 2	4	80	◆ 2	4	12	▲ 2	(E‖E‖E);(D‖D‖D) [B]	▼ 2	4	4
● 3	3	9	■ 3	3	9	⬢ 3	6	60	◆ 3	4	16	▲ 3	E‖E‖E‖D‖D‖D	▼ 3	4	8
● 4	4	8	■ 4	4	8	⬢ 4	6	90	◆ 4	6	12	▲ 4	E‖E‖E‖D‖D [B]	▼ 4	4	12

states per second (∼8.5M states/s for ▼4), where DIVINE visits ∼4k states per second (Fig. 4d).

3.3.2 LLMC vs Nidhugg

Moving on to Fig. 4b, we notice Nidhugg is unable to complete any of the Michael-Scott queue ▲, Hashmap ♦ or Philosopher ▼ test cases. This is because Nidhugg supports neither the `__atomic_*` instructions needed for the Michael-Scott queue ▲ nor the spin-lock used in the Hashmap ♦ and Philosopher ▼ tests. We tried Nidhugg's transformation capabilities to transform the spin-lock to an assume statement, thus limiting the traces traversed to the ones where the condition of the spin-lock holds, but the generated LLVM IR was invalid and could not be used. Additionally, we tried an experimental version (7b8be8a) with a changelog containing potential fixes to no avail.

We see that Nidhugg outperforms LLMC in the Prefixsum ⬢ test cases consistently by multiple orders of magnitude: Nidhugg traverses only a *single* trace for each of these test cases. This highlights the strength of Nidhugg in its ability to conclude that each read can only read a single value. Without this technique, LLMC needs to exhaustively go through all interleavings of the threads.

For the linked list, sorted ● and non-sorted ■, we see that as the cases get bigger, LLMC is able to outperform Nidhugg. This highlights the disadvantage of stateless model checking: bigger state spaces tend to cause more common prefixes of paths, which causes more work for stateless model checking.

3.3.3 Scalability

Figure 5 shows the results for various number of threads for SortedLinkedList3.9 ③, chosen for the performance similarity of the three tools. The graph shown is typical: other test expose similar patterns as the one we highlight here. DIVINE does not scale well in the number of threads: its peak performance lies typically around 4 or 8 threads, confirmed by the DIVINE developers[2]. Nidhugg expectedly does scale very well, as threads just execute a specific trace, with hardly and communication. LLMC shows some scalability, but a ∼4x improvement using 192 threads leaves a lot of room for improvement[3].

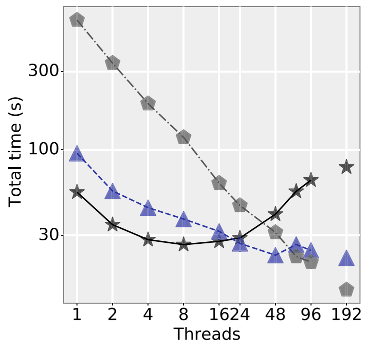


Fig. 5. Scalability comparison of DIVINE ★, LLMC ▲, Nidhugg ⬟.

[2] https://divine.fi.muni.cz/trac/ticket/44.
[3] https://github.com/bergfi/dmc/issues/1.

3.3.4 DMC and Dtree

We highlight one aspect of the performance of LLMC: the underlying model checker DMC and its storage component DTREE [14]. In Figure 4c, we notice that although LLMC on average generates state spaces of an order of magnitude larger compared to DIVINE, it uses two orders of magnitude less memory per state, due to DTREE. Furthermore, DTREE allows to apply a delta to a state without reconstructing the entire state. Since states are typically ∼2kiB in these tests, this significantly avoids copying memory and increases performance.

4 Conclusion

We have introduced LLMC 0.2[4], the multi-threaded low-level model checker that model checks software via LLVM IR. It translates the input LLVM IR into a model LLVM IR that implements the DMC API, the API of the high-performance model checker DMC. This allows LLMC to *execute* the model's next-state function, instead of *interpreting* the input LLVM IR, like DIVINE and Nidhugg. We compared LLMC to these tools using a test suite of 24 tests, covering various data structures. LLMC outperforms DIVINE and Nidhugg up to three orders of magnitude, while other tests have shown areas for improvement. Averaging the results of all completed tests, LLMC is an order of magnitude faster than DIVINE and ∼3.4x faster than Nidhugg. DIVINE and Nidhugg are unable to complete 4 and 12 tests, respectively, due to crashing or not supporting infinite loops or `__atomic_*` library calls.

Future Work. LLMC will benefit most from a state space reduction technique that collapses memory instructions to thread-private memory. We aim to integrate this as part of a memory emulation layer that also adds support for relaxed memory models. Even without the dynamic reduction technique, the results show that LLMC in its current form is a high performing tool to model check software.

References

1. Baier, C., Katoen, J.: Principles of Model Checking. MIT Press, Cambridge (2008)
2. Lattner, C.: LLVM: an infrastructure for multi-stage optimization. Master's thesis, Computer Science Department, University of Illinois at Urbana-Champaign, Urbana, IL, December 2002. http://llvm.cs.uiuc.edu
3. Baranová, Z., et al.: Model checking of C and C++ with DIVINE 4. In: D'Souza, D., Narayan Kumar, K. (eds.) ATVA 2017. LNCS, vol. 10482, pp. 201–207. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68167-2_14
4. Rockai, P., Still, V., Cerná, I., Barnat, J.: DiVM: model checking with LLVM and graph memory. J. Syst. Softw. **143**, 1–13 (2018). https://doi.org/10.1016/j.jss.2018.04.026

[4] https://github.com/bergfi/llmc.

5. Abdulla, P.A., Aronis, S., Atig, M.F., Jonsson, B., Leonardsson, C., Sagonas, K.: Stateless model checking for TSO and PSO. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 353–367. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_28
6. Abdulla, P.A., Atig, M.F., Jonsson, B., Lång, M., Ngo, T.P., Sagonas, K.: Optimal stateless model checking for reads-from equivalence under sequential consistency. Proceedings of ACM Program. Lang. **3**(dOOPSLA), 150:1–150:29 (2019). https://doi.org/10.1145/3360576
7. Kokologiannakis, M., Lahav, O., Sagonas, K., Vafeiadis, V.: Effective stateless model checking for C/C++ concurrency. Proc. ACM Program. Lang. **2**(POPL), 17:1–17:32 (2018). https://doi.org/10.1145/3158105
8. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_15
9. Falke, S., Merz, F., Sinz, C.: The bounded model checker LLBMC. In: Denney, E., Bultan, T., Zeller, A. (eds.) 2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, 11–15 November 2013, pp. 706–709. IEEE (2013). https://doi.org/10.1109/ASE.2013.6693138
10. Carter, M., He, S., Whitaker, J., Rakamaric, Z., Emmi, M.: SMACK software verification toolchain. In: Visser, W., Williams, L. (eds.) Proceedings of the 38th IEEE/ACM International Conference on Software Engineering (ICSE) Companion. ACM, pp. 589–592 (2016)
11. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 343–361. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_20
12. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Draves, R., van Renesse, R. (eds.) OSDI, pp. 209–224. USENIX Association (2008)
13. van der Berg, F.I.: Model checking LLVM IR using LTSmin: using relaxed memory model semantics, December 2013. http://essay.utwente.nl/65059/
14. van der Berg, F.I.: Recursive variable-length state compression for multi-core software model checking. In: Dutle, A., Moscato, M.M., Titolo, L., Muñoz, C.A., Perez, I. (eds.) NFM 2021. LNCS, vol. 12673, pp. 340–357. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-76384-8_21
15. Silberschatz, A., Galvin, P.B., Gagne, G.: Operating System Concepts, 10th edn. Wiley (2018). http://os-book.com/OS10/index.html
16. Gharachorloo, K.: Memory consistency models for shared-memory multiprocessors. Stanford, CA, USA, Technical report (1995). https://doi.org/10.5555/891506
17. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Burns, J.E., Moses, Y. (eds) PODC, pp. 267–275. ACM (1996)
18. van der Berg, F.I., van de Pol, J.: Concurrent chaining hash maps for software model checking. In: Barrett, C., Yang, J. (eds.) 2019 Formal Methods in Computer Aided Design (FMCAD), ser. Proceedings of the Conference on Formal Methods in Computer-Aided Design (FMCAD), pp. 46–54. IEEE, USA, October 2019. https://doi.org/10.23919/FMCAD.2019.8894279
19. Dijkstra, E.W.: Cooperating Sequential Processes, pp. 65–138. Springer, New York (2002). https://doi.org/10.1007/978-1-4757-3472-0_2

Formally Validating a Practical Verification Condition Generator

Gaurav Parthasarathy[1](✉), Peter Müller[1], and Alexander J. Summers[2]

[1] Department of Computer Science, ETH Zurich, Zurich, Switzerland
{gaurav.parthasarathy, peter.mueller}@inf.ethz.ch

[2] University of British Columbia, Vancouver, Canada
alex.summers@ubc.ca

Abstract. A program verifier produces reliable results only if both the *logic* used to justify the program's correctness is sound, and the *implementation* of the program verifier is itself correct. Whereas it is common to formally prove soundness of the logic, the implementation of a verifier typically remains unverified. Bugs in verifier implementations may compromise the trustworthiness of successful verification results. Since program verifiers used in practice are complex, evolving software systems, it is generally not feasible to formally verify their implementation.

In this paper, we present an alternative approach: we *validate successful runs* of the widely-used Boogie verifier by producing a *certificate* which proves correctness of the obtained verification result. Boogie performs a complex series of program translations before ultimately generating a verification condition whose validity should imply the correctness of the input program. We show how to certify three of Boogie's core transformation phases: the elimination of cyclic control flow paths, the (SSA-like) replacement of assignments by assumptions using fresh variables (passification), and the final generation of verification conditions. Similar translations are employed by other verifiers. Our implementation produces certificates in Isabelle, based on a novel formalisation of the Boogie language.

1 Introduction

Program verifiers are tools which attempt to prove the correctness of an implementation with respect to its specification. A successful verification attempt is, however, only meaningful if both the *logic* used to justify the program's correctness is sound, and the *implementation* of the program verifier is itself correct. It is common to formally prove soundness of the logic, but the implementations of program verifiers typically remain unverified. As is standard for complex software systems, bugs in verifier implementations can and do arise, potentially raising doubts as to the trustworthiness of successful verification results.

A. Silva and K. R. M. Leino (Eds.): CAV 2021, LNCS 12760, pp. 704–727, 2021.
https://doi.org/10.1007/978-3-030-81688-9_33

One way to close this gap is to prove a verifier's implementation correct. However, such a *once-and-for-all* approach faces serious challenges. Verifying an existing implementation bottom-up is not practically feasible because such implementations tend to be large and complex (for instance, the Boogie verifier [29] consists of over 30K lines of imperative C# code), use a variety of libraries, and are typically written in efficient mainstream programming languages which themselves lack a formalisation. Alternatively, one could develop a verifier that is correct by construction. However, this approach requires the verifier to be (re-)implemented in an interactive theorem prover (ITP) such as Coq [14] or Isabelle [24]. This precludes the free choice of implementation language and paradigm, exploitation of concurrency, and possibility of tight integration with standard compilers and IDEs, which is often desirable for program verifiers [4,5,13,26]. Both verification approaches substantially impede software maintenance, which is problematic since verifiers are often rapidly-evolving software projects (for instance, the Boogie repository [1] contains more than 5000 commits).

To address these challenges, in this work we employ a different approach. Instead of verifying the implementation once and for all, we *validate specific runs* of the verifier by automatically producing a *certificate* which proves the correctness of the obtained verification result. Our certificate generation formally relates the input and output of the verifier, but does so largely independently of its implementation, which can freely employ complex languages, algorithms, or optimisations. Our certificates are formal proofs in Isabelle, and so checkable by an independent trusted tool; their guarantees for a certified run of the verifier are as strong as those provided by a (hypothetical) verified verifier.

We apply our novel verifier validation approach to the widely-used Boogie verifier, which verifies programs written in the intermediate verification language Boogie. The Boogie verifier is a *verification condition generator*: it verifies programs by generating a verification condition (VC), whose validity is then discharged by an SMT solver. Certifying a verifier run requires proving that validity of the VC implies the correctness of the input program. Certification of the validity-checking of the VC is an orthogonal concern; our results can be combined with work in that area [11,15,19] to obtain end-to-end guarantees.

Like many automatic verifiers, Boogie is a *translational verifier*: it performs a sequence of substantial Boogie-to-Boogie translations (*phases*), simplifying the task and output of the final efficient VC computation [6,18]. The key challenges in certifying runs of the Boogie tool are to certify each of these phases, including final VC generation. In particular, we present novel techniques for making the following three key phases (and many smaller ones) of Boogie's tool chain certifying:

1. The elimination of loops (more precisely, cycles in the CFG) by reducing the correctness of loops to checking loop invariants *(CFG-to-DAG phase)*
2. The replacement of assignments by (SSA-style) introduction of fresh variables and suitable **assume** statements *(passification phase)*

3. The final generation of the VC, which includes the erasure and logical encoding of Boogie's polymorphic type system [33] *(VC phase)*.

The certification of such verifier phases is related to existing work on compiler verification [34] and validation [8,41,42]. However, the translations and the certified property we tackle here are fundamentally different from those in compilers. Compilers typically require that each execution of the target program corresponds to an execution of the source program. In contrast, the encoding of a program in a translational verifier typically has intentionally more executions (for instance, allows more non-determinism). Moreover, translational verifiers need to handle features not present in standard programming languages such as **`assume`** statements and background theories. Prior work on validating such verifier phases has been limited in the supported language and extent of the formal guarantee; we discuss comparisons in detail in Sect. 8.

Contributions. Our paper makes the following technical contributions.

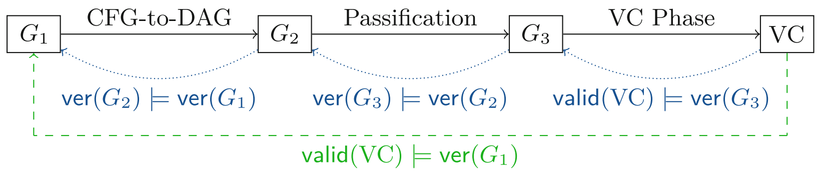
1. The first formal semantics for a significant subset of Boogie (including axioms, polymorphism, type constructors), mechanised in Isabelle.
2. A validation technique for two core program-to-program translations occurring in verifiers (CFG-to-DAG and passification).
3. A validation technique for the VC phase, handling polymorphism erasure and Boogie's type system encoding [31], for which no prior formal proof exists.
4. A version of the Boogie implementation that produces certificates for a significant subset of Boogie.

Making the Boogie verifier certifying is an important result, reducing the trusted code base for a wide variety of verification tools implemented via encodings into Boogie, e.g. Dafny [31], VCC [13], Corral [28], and Viper [35]. Moreover, the technical approach we present here can in future be applied to the certification of the translations performed by these tools, and those based on comparable intermediate verification languages such as Frama-C [26] and Krakatoa [17] based on Why3 [16] and Prusti [4] and VerCors [10] based on Viper [35].

Outline. Section 2 explains at a high-level, how our validation approach is structured for the different phases. Section 3 introduces a formal semantics for Boogie. Sections 4, 5 and 6 present our validation of the CFG-to-DAG, passification, and VC phases, respectively. Section 7 evaluates our certificate-producing version of Boogie. Section 8 discusses related work. Section 9 concludes. Further details are available in our accompanying technical report (hereafter, TR) [37].

2 Approach

A Boogie program consists of a set of procedures, each with a specification and a procedure body in the form of a (reducible) control-flow-graph (CFG), whose blocks contain basic commands; we present the formal details in the next section. Boogie verifies each procedure modularly, desugaring procedure calls according



G_1
CFG-to-DAG
G_2
Passification
G_3
VC Phase
VC
$\mathsf{ver}(G_2) \models \mathsf{ver}(G_1)$
$\mathsf{ver}(G_3) \models \mathsf{ver}(G_2)$
$\mathsf{valid}(\mathrm{VC}) \models \mathsf{ver}(G_3)$
$\mathsf{valid}(\mathrm{VC}) \models \mathsf{ver}(G_1)$

Fig. 1. Key phases of verification in Boogie and their certification. The solid edges show Boogie's transformations on a procedure body; each node G_i represents a control-flow-graph. Our final certificate (dashed edge) is constructed by formally linking the three phase certificates represented by the dotted edges. Each of the three phase certificates also incorporate extra smaller transformations that we do not show here.

to their specifications. Verification is implemented via a series of phases: program-to-program translations and a final computation of a VC to be checked by an SMT solver. Our goal is to formally certify (per run of Boogie) that validity of this VC implies the correctness of the original procedure.

To keep the complexity of certificates manageable, our technical approach is *modular* in three dimensions: decomposing our formal goal per *procedure* in the Boogie program, per *phase* of the Boogie verification, and per *block* in the CFG of each procedure. This modularity makes the full automation of our certification proofs in Isabelle practical. In the following, we give a high-level overview of this modular structure; the details are presented in subsequent sections.

Procedure Decomposition. Boogie has no notion of a main program or an overall program execution. A Boogie program is correct if each of its procedures is individually correct (that is, the procedure body has no failing traces, as we make precise in the next section). Boogie computes a separate VC for each procedure, and we correspondingly validate the verification of each procedure separately.

Phase Decomposition. We break our overall validation efforts down into per-phase sub-problems. In this paper, we focus on the following three most substantial and technically-challenging of these sequential phases, illustrated in Fig. 1. (1) The *CFG-to-DAG phase* translates a (possibly-cyclic) CFG to an acyclic CFG (*cf.* Sect. 4). This phase substantially alters the CFG structure, cutting loops using annotated loop invariants to over-approximate their executions. (2) The *passification phase* eliminates imperative updates by transforming the code into static single assignment (SSA) form and then replacing assignments with *constraints* on variable versions (*cf.* Sect. 5). Both of these phases introduce extra non-determinism and **assume** statements (which, if implemented incorrectly could make verification unsound by masking errors in the program). (3) The final *VC phase* translates the acyclic, passified CFG to a verification condition that, in addition to capturing the weakest precondition, encodes away Boogie's polymorphic type system [33].

We construct certificates for each of these key phases separately (depicted by the blue dotted lines in Fig. 1). For each phase, we certify that *if* the target of the translation phase is correct (a correct Boogie program for the first two phases; a valid VC for the VC phase) then the source (program) of the phase is correct. This modular approach lets us focus the proof strategy for each phase on its conceptually-relevant concerns, and provides robustness against *changes* to the verifier since at most the certification of the changed phases may need adjustment. Logically, our per-phase certificates are finally glued together to guarantee the analogous end-to-end property for the entire pipeline, depicted by the green dashed edge in Fig. 1. For our certificates, we import the input and output programs (and VC) of each key phase from Boogie into Isabelle; we do not reimplement any of Boogie's phases inside Isabelle.

The certificates of the key phases also incorporate various smaller transformations between the key phases, such as peephole optimisation. Our work also validates these smaller transformations, but we focus the presentation on the key phases in this paper. Boogie also performs several smaller translation steps *prior* to the CFG-to-DAG phase. These include transforming ASTs to corresponding CFGs, optimisations such as dead variable elimination, and desugaring procedure calls using their specifications (via explicit **assert**, **assume**, and **havoc** statements). Our approach applies analogously to these initial smaller phases, but our current implementation certifies only the pipeline of all phases from the (input to the) CFG-to-DAG phase onwards. Thus, our certificate relates Boogie's VC to the original source AST program so long as these prior translation steps are correct.

CFG Decomposition. When tackling the certification of *each* phase, we further break down validation of a procedure's CFG in the source program of the phase into sub-problems for each block in the CFG. We prove two results for each block in the source CFG:

1. *Local block lemmas:* We prove an independent lemma for each source CFG block in isolation, relating the executions through the block with the corresponding block in the target program (or the VC generated for that block, in the case of the VC phase). In particular, this lemma implies that if the target block has no failing executions (or the VC generated for that block holds, for the VC phase), neither does the source block for corresponding input states.
2. *Global block theorems:* We show analogous per-block results concerning all executions *from this block onwards* extending to the end of the procedure in question; we build these compositionally by reverse-topological traversal of either the source or target CFGs, as appropriate. The global block theorem for the entry block establishes correctness of the phase.

This decomposition separates command-level reasoning (local block lemmas) from CFG-level reasoning (global block theorems). It enables concise lemmas and proofs in Isabelle and makes each comprehensible to a human.

3 A Formal Semantics for Boogie

Our certificates prove that the validity of a VC generated by Boogie formally implies correctness of the Boogie CFG-to-DAG source program. This proof relies crucially on a formal semantics for Boogie itself. Our first contribution is the first such formal semantics for a significant subset of Boogie, mechanised in Isabelle. Our semantics uses the Boogie reference manual [29], the presentation of its type system [33], and the Boogie implementation for reference; none of those provide a formal account of the language. For space reasons, we explain only the key concepts of our detailed formalisation here; more details are provided in App. A of the TR [37] and the full Isabelle mechanisation is available as part of our accompanying artifact [36].

3.1 The Boogie Language

Boogie programs consist of a set of top-level declarations of global variables and constants (the *global data*), axioms, uninterpreted (polymorphic) functions, type constructors, and procedures. A procedure declaration includes parameter, local-variable, and result-variable declarations (the *local data*), a pre- and post-condition, and a procedure body given as a CFG.[1] CFGs are formalised as usual in terms of basic blocks (containing a possibly-empty list of *basic commands*), and edges; semantically, execution after a basic block continues via any of its successors non-deterministically.

$$\begin{aligned}
e &::= x \mid \texttt{false} \mid \texttt{true} \mid i \mid e_1\ bop\ e_2 \mid uop(e) \mid f[\vec{\tau}](\vec{e}) \mid \texttt{old}(e) \mid \\
&\qquad \forall x : \tau.\ e \mid \exists x : \tau.\ e \mid \forall_{ty}\, t.\ e \mid \exists_{ty}\, \tau.\ e \\
\tau &::= Int \mid Bool \mid C(\vec{\tau}) \mid t \quad c ::= \texttt{assume}\ e \mid \texttt{assert}\ e \mid x := e \mid \texttt{havoc}\ x
\end{aligned}$$

Fig. 2. The syntax of our formalised Boogie subset, where τ, e, and c, denote the types, expressions, and basic commands respectively; control-flow is handled via CFGs over the basic commands. *bop* and *uop* denote binary and unary operations, respectively.

The types, expressions, and basic commands in our Boogie subset are shown in Fig. 2. We support the primitive types *Int* and *Bool*; types obtained via declared type constructors are *uninterpreted types*; the sets of values such types denote are constrained only via Boogie axioms and **assume** commands. Moreover, types can contain type variables (for instance, to specify polymorphic functions).

Boogie expression syntax is largely standard (e.g. including typical arithmetic and boolean operations). Old-expressions **old**(e) evaluate the expression e w.r.t. the current local data and the global data as it *was* in the pre-state of the

[1] Source-level procedure specifications also include *modifies clauses*, declaring a set of global variables the procedure may modify. As we tackle Boogie programs after procedure calls have been desugared, there are no modifies clauses in our formalisation.

procedure execution. Boogie expressions also include universal and existential *value* quantification (written $\forall x : \tau.\ e$ and $\exists x : \tau.\ e$), as well as universal and existential *type* quantification (written $\forall_{ty}\, t.\ e$ and $\exists_{ty}\, t.\ e$). In the latter, t is bound in e and quantifies over *closed* Boogie types (i.e. types that do not contain any type variables).

Basic commands form the single-steps of traces through a Boogie CFG; sequential composition is implicit in the list of basic commands in a CFG basic block and further control flow (including loops) is prescribed by CFG edges. Boogie's basic commands are assumes, asserts, assignments, and havocs; **havoc** x non-deterministically assigns a value matching the type of variable x to x.

The main Boogie features *not* supported by our subset are maps and other primitive types such as bitvectors. Boogie maps are polymorphic and impredicative, i.e. one can define maps that contain themselves in their domain. Giving a semantic model for such maps in a proof assistant such as Isabelle or Coq is non-trivial; we aim to tackle this issue in the future. Modelling bitvectors will be simpler, although maintaining full automation may require some additional work.

3.2 Operational Semantics

Values and State Model. Our formalisation embeds integer and boolean values shallowly as their Isabelle counterparts; an Isabelle carrier type for all *abstract values* (those of uninterpreted types) is a parameter of our formalisation. Each uninterpreted type is (indirectly) associated with a *non-empty* subset of abstract values via a *type interpretation* map $\mathcal{T}$ from abstract values to (single) types; particular interpretations of uninterpreted types can be obtained via different choices of type interpretation $\mathcal{T}$.

One can understand Boogie programs in terms of the sets of possible *traces* through each procedure body. Traces are (as usual) composed of sequences of steps according to the semantics of basic commands and paths through the CFG; these can be finite or infinite (representing a non-terminating execution). A trace may halt in three cases: (1) an exit block of the procedure is reached in a state satisfying the procedure's postcondition (a *complete* trace),[2] (2) an **assert** A command is reached in a state not satisfying assertion A (a *failing* trace), or (3) an **assume** A command is reached in a state not satisfying A (a trace which *goes to magic* and stops). Our formalisation correspondingly includes three kinds of Boogie program states: a distinguished *failure state* F, a distinguished *magic state* M, and *normal states* $\mathsf{N}((os, gs, ls))$. A normal state is a triple of partial mappings from variables to values for the old global state (for the evaluation of old-expressions), the (current) global state, and the local state, respectively.

Expression Evaluation. An expression e evaluates to value v if the (big-step) judgement $\mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle e, \mathsf{N}(ns)\rangle \Downarrow v$ holds in the context $(\mathcal{T}, \Lambda, \Gamma, \Omega)$. Here, $\mathcal{T}$

[2] The case of the postcondition *not* holding is subsumed under point (2), since Boogie checks postconditions by generating extra **assert** statements.

```
assume i != 0
j := 0
while(i != 0)
inv j >= 0 ∧ (i = 0 ⇒ j > 0)
{
  if(i < 5) {
    j := j+1
  }
  i := i-1
}
assert j > 0
```

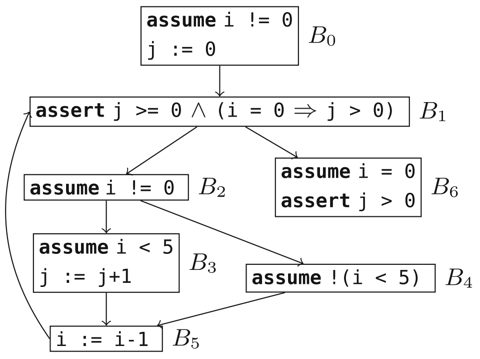


Fig. 3. Running example in source code and CFG representation, respectively.

is a *type interpretation* (as above), Λ is a *variable context*: a pair (G, L) of type declarations for the global (G) and local (L) data. Γ is a *function interpretation*, which maps each function name to a semantic function mapping a list of types and a list of values to a return value. The type substitution Ω maps type variables to types.

The rules defining this judgement can be found in App. A.2 of the TR [37]. For example, the following rule expresses when a universal type quantification evaluates to **true** (t is bound to the quantified type and may occur in e):

$$\frac{\forall \tau.\ \mathit{closed}(\tau) \Longrightarrow \mathcal{T}, \Lambda, \Gamma, \Omega(t \mapsto \tau) \vdash \langle e, ns \rangle \Downarrow \texttt{true}}{\mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle \forall_{ty}\, t.\ e, ns \rangle \Downarrow \texttt{true}}$$

The premise requires one to show that the expression e reduces to **true** for every possible type τ that is closed. In general, expression evaluation is possible only for well-typed expressions; we also formalise Boogie's type system and (for the first time) prove its type safety for expressions in Isabelle.

Command and CFG Reduction. The (big-step) judgement $\mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle c, s \rangle \rightarrow s'$ defines when a command c reduces in state s to state s'; the rules are in App. A.3 of the TR [37]. This reduction is lifted to lists of commands cs to model the semantics of a single trace through a CFG block (the judgement $\mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle cs, s \rangle\ [\rightarrow]\ s'$). The operational semantics of CFGs is modelled by the (small-step) judgement $\mathcal{T}, \Lambda, \Gamma, \Omega, G \vdash \delta \rightarrow_{\mathsf{CFG}} \delta'$, expressing that the CFG configuration δ reduces to configuration δ' in the CFG G. A CFG configuration is either *active* or *final*. An active configuration is given by a tuple $(\mathsf{inl}(b_n), s)$, where b_n is the block identifier indicating the current position of the execution and s is the current state. A final configuration consists of a tuple $(\mathsf{inr}(()), s)$ for state s (and unit value $()$) and is reached at the end of a block that has either no successors, or is in a magic or failure state.

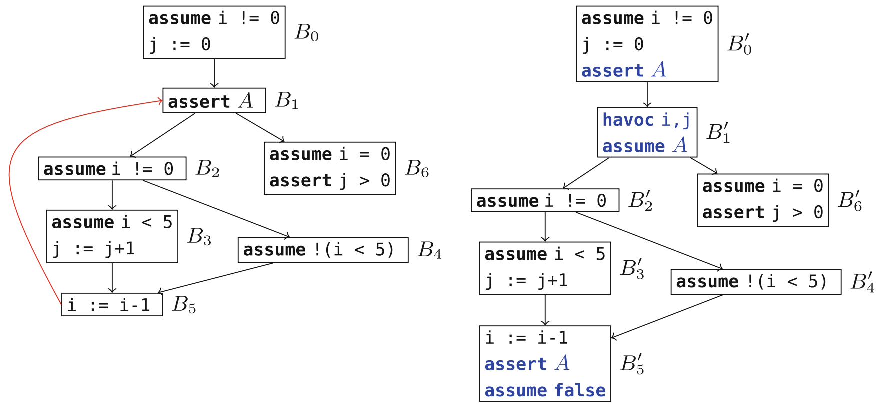


Fig. 4. The CFG-to-DAG phase applied to the running example (source is left, target is right). The back-edge (the red edge from B_5 to B_1 in the left CFG) is eliminated. The blue commands are new. A is given by `j >= 0` $\wedge$ `(i = 0` $\Rightarrow$ `j > 0)`.

3.3 Correctness

A procedure is *correct* if it has *no failing traces*. This is a *partial correctness* semantics; a procedure body whose traces never leave a loop is trivially correct provided that no intermediate **assert** commands fail. Procedure correctness relies on CFG correctness. A CFG G is correct w.r.t. a postcondition Q and a context $(\mathcal{T}, \Lambda, \Gamma, \Omega)$ in an initial normal state $\mathsf{N}(ns)$ if the following holds for all configurations (r, s'):

$$\mathcal{T}, \Lambda, \Gamma, \Omega, G \vdash (\mathsf{inl}(\mathsf{entry}(G)), \mathsf{N}(ns)) \rightarrow^*_{\mathsf{CFG}} (r, s') \Longrightarrow [s' \neq \mathsf{F} \wedge \\ (r = \mathsf{inr}(())) \Longrightarrow (\forall ns'.\ s' = \mathsf{N}(ns') \Longrightarrow \mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle Q, \mathsf{N}(ns')\rangle \Downarrow \mathbf{true}))]$$

where $\mathsf{entry}(G)$ is the entry block of G and $\rightarrow^*_{\mathsf{CFG}}$ is the reflexive-transitive closure of the CFG reduction. The postcondition is needed only if a final configuration is reached in a normal state, while failing states must be unreachable. Whenever we omit Q, we implicitly mean the postcondition to be simply **true**. In our tool, we consider only empty initial mappings Ω, since we do not support procedure type parameters (lifting our work to this feature will be straightforward).

For a procedure p to be correct w.r.t. a context, its body CFG must be correct w.r.t. the same context and p's postcondition, *for all* initial normal states $\mathsf{N}(ns)$ that satisfy p's precondition and which respect the context. For ns to *respect* a context, it must be well-typed and must satisfy the axioms when restricted to its constants. We say that p is *correct*, if it is correct w.r.t. *all well-formed contexts*, which must have a well-typed function interpretation and a type interpretation that inhabits every uninterpreted closed type (and only those).

Running Example. We will use the simple CFG of Fig. 3 as a running example, intended as body of a procedure with trivial (**true**) pre- and post-conditions.

The code includes a simple loop with a declared loop invariant, which functions as a classical Floyd/Hoare-style inductive invariant, and for the moment can be considered as an implicit **`assert`** statement at the loop head. The CFG has infinite traces: those which start from any state in which `i` is negative. Traces starting from a state in which `i` is zero go to magic; they do not reach the loop. The program is correct (has no failing traces): all other initial states will result in traces that satisfy the loop invariant and the final **`assert`** statement. If we removed the initial **`assume`** statement, however, there *would* be failing traces: the loop invariant check would fail if `i` were initially zero.

4 The CFG-to-DAG Phase

In this section, we present the validation for the CFG-to-DAG phase in the Boogie verifier. This phase is challenging as it changes the CFG structure, inserts additional non-deterministic assignments and **`assume`** statements, and must do so correctly for arbitrary (reducible) nested loop structures, which can include unstructured control flow (e.g. jumps out of loops).

4.1 CFG-to-DAG Phase Overview

The CFG-to-DAG phase applies to every *loop head* block identified by Boogie's implementation and any *back-edges* from a block reachable from the loop head block back to the loop head (following standard definitions for reducible CFGs [21]). Figure 4 illustrates the phase's effect on our running example. Block B_1 is the (only) loop head here, and the edge from B_5 to it the only back-edge (completing looping paths via B_2 and B_3 or B_2 and B_4). An **`assert`** A statement starting a loop head (like B_1) is interpreted as declaring A to be the loop invariant.[3] The CFG-to-DAG phase performs the following steps:

1. Accumulate a set X_H of all (local and global) variables *assigned-to* on *any looping path* from the loop head back to itself. In our example, X_H is $\{i, j\}$.
2. Move the **`assert`** A statement declaring a loop invariant (if any) from the loop head to the end of *each preceding* block (in our example: B_0 and B_5).
3. Insert **`havoc`** statements at the start of the loop head block per variable in X_H, followed by a single **`assume`** A statement (preceding any further statements).
4. For each block with a back-edge to the loop head, delete the back-edge; if this leaves the block with no successors, append **`assume false`** to its commands.[4]

The havoc-then-assume sequence introduced in step 3 can be understood as generating traces for *arbitrary values of* X_H satisfying the loop invariant A,

[3] In general, multiple asserts at the beginning of a loop head may form the invariant.

[4] Omitting **`assume false`** if there are no successors would be incomplete, since otherwise the postcondition would have to be satisfied.

effectively over-approximating the set of states reachable at the loop head in the original program. In particular, the remnants of any originally looping path (e.g. B'_1, B'_2, B'_3, B'_5) enforce that any non-failing trace starting from any such state must (due to the **`assert`** added to block B'_5 in step 2) result in a state which re-establishes the loop invariant. Such paths exist only to enforce this inductive step (analogously to the premise of a Hoare logic while rule); so long as the **`assert`** succeeds, we can discard these traces via step 4.

While we illustrate this step on a simple CFG, in general a loop head may have multiple back-edges, looping structures may nest, and edges may exit multiple loops. For the above translation to be correct, the CFG must be reducible and loop heads and corresponding back-edges identified accurately, which is complex in general. Importantly (but perhaps surprisingly), our work makes this phase of Boogie certifying *without* explicitly verifying (or even defining) these notions.

4.2 CFG-to-DAG Certification: Local Block Lemmas

We define first our local block lemmas for this phase. Recall that these prove that if executing the statements of a target block yields no failing executions, the same holds for the corresponding source block; this result is trivial for source blocks other than loop heads and their immediate predecessors, since these are unchanged in this phase. To enable eventual composition of our block lemmas, we need to also reflect the role of the **`assume`** and **`assert`** statements employed in this phase. The formal statement of our local block lemmas is as follows[5]:

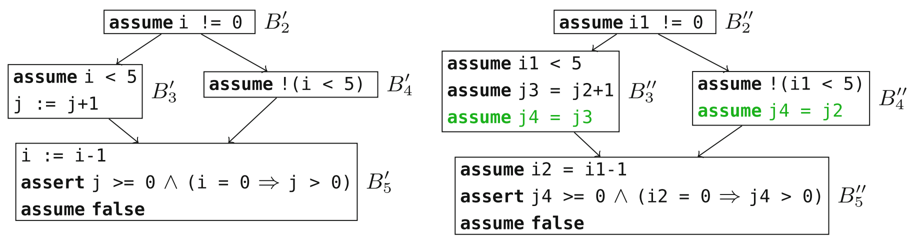
Theorem 1 (CFG-to-DAG Local Block Lemma). *Let B be a source block with commands cs_S, whose corresponding target block has commands cs_T. If B is a loop head, let X_H be as defined in CFG-to-DAG step 1 (and empty otherwise) and let A_{pre} be its loop invariant (or* **`true`** *otherwise). If B is a* predecessor *of a loop head, let A_{post} be the loop invariant of its successor (and* **`true`** *otherwise). Then,* if*:*

1. $\mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle cs_S, \mathsf{N}(ns_1)\rangle\ [\rightarrow]\ s'_1$
2. $\forall s'_2.\ \mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle cs_T, \mathsf{N}(ns_2)\rangle\ [\rightarrow]\ s'_2 \Longrightarrow s'_2 \neq \mathsf{F}$
3. *A_{pre} is satisfied in ns_1, and ns_2 differs from ns_1 only on variables in X_H and variables not defined in Λ*

then*: $s'_1 \neq \mathsf{F}$ and if s'_1 is a normal state, then (1) A_{post} is satisfied in s'_1, and (2) if no* **`assume false`** *was added at the end of cs_T, then there is a target execution in cs_T from $\mathsf{N}(ns_2)$ that reaches a normal state that differs from s'_1 only on variables not defined in Λ.*

The gist of this lemma is to capture *locally* the ideas behind the four steps of the phase. For example, consequence (1) reflects that *after* the transformation, any blocks that *were previously* predecessors of a loop head (B'_0 and B'_5 in our running example) will have an **`assert`** statement checking for the corresponding invariant (and so if the target program has no failing traces, in each trace this invariant will be true at that point).

[5] We omit some details regarding well-typedness, handled fully in our formalisation.



```
assume i != 0
assume i < 5
j := j+1
assume !(i < 5)
i := i-1
assert j >= 0 ∧ (i = 0 ⇒ j > 0)
assume false
```

```
assume i1 != 0
assume i1 < 5
assume j3 = j2+1
assume j4 = j3
assume !(i1 < 5)
assume j4 = j2
assume i2 = i1-1
assert j4 >= 0 ∧ (i2 = 0 ⇒ j4 > 0)
assume false
```

Fig. 5. The passification phase applied to the branch in the running example with the result on the right. The final (green) commands in B''_3 and B''_4 are the synchronisation commands. At the uppermost blocks shown here, the current versions of `i` and `j` are `i1` and `j2`, respectively. The full CFGs are shown in App. B of the TR [37].

4.3 CFG-to-DAG Certification: Global Block Theorems

We lift our certification to *all* traces through the source and target CFGs; the statement of the corresponding global block theorems is similar to that of local block theorems lifted to CFG executions, and for space reasons we do not present it here, but it is included in our Isabelle formalisation. In particular, we prove for each block (working in reverse topological order through the target CFG blocks) that if executions starting in the target CFG block never fail, neither do any executions starting from the corresponding source CFG block, and looping paths modify at most the variables havoced according to step 3 of the phase.

The major challenge in these proofs is reasoning about looping paths in the source CFG, since these revisit blocks. To solve this challenge, we perform inductive arguments per loop head in terms of the number of steps remaining in the trace in question.[6] Our global block theorem for a block B then carries as an assumption an induction hypothesis for each loop that contains B. Proving a global block theorem for the origin of a back-edge is taken care of by applying the corresponding induction hypothesis.

This proof strategy works only if we have obtained the induction hypothesis for the loop head *before* we use the global block theorem of the origin of a back-edge (otherwise we cannot discharge the block theorem's hypothesis). In other words, our proof implicitly shows the necessary requirement that loop heads (as identified by Boogie) dominate all back-edges reaching them *without us formalising any notion of domination, CFG reducibility, or any other advanced graph-theoretic concept.* This shows a major benefit of our validation approach over a once-and-for-all verification of Boogie itself: our proofs indirectly check that the identification of loop heads and back-edges guarantees the necessary *semantic properties* without being concerned with *how* Boogie's implementation computes this information.

[6] This may seem insufficient since traces can be infinite, but importantly a *failing* trace is always finite, and our theorems need only eliminate the chance of failing traces.

Our approach applies equally to nested loops and more-generally to reducible CFG structures; *all* corresponding induction hypotheses are carried through from the visited loop heads. The requirement that no more than the havoced variables X_H are modified in the source program is easily handled by showing that variables modified in an inner loop are a subset of those in outer loops. As for all of our results, our global block lemmas are proven automatically in Isabelle per Boogie procedure, providing per-run certificates for this phase.

5 The Passification Phase

In this section, we describe the validation of the passification phase in the Boogie verifier. Unlike the previous phase, passification makes no changes to the CFG structure, but makes substantial changes to the program states (via SSA-like renamings), substantially increases non-determinism, and employs **`assume`** statements to re-tame the sets of possible traces.

5.1 Passification Phase Overview

The main goal of passification is to eliminate assignments such that a more efficient VC can be ultimately generated [6,18,30]. In the Boogie verifier, this is implemented as a single transformation phase that can be thought of as two independent steps. Firstly, the source CFG is transformed into *static single assignment* (SSA) form, introducing *versions* (fresh variables) for each original program variable such that each version is assigned at most once in any program trace. In a second step, variable assignments are *completely eliminated*: each assignment command $x := e$ is replaced by **`assume`** $x = e$. Havoc statements are simply removed; their effect is implicit in the fact that a new variable version is used (via the SSA step) *after* such a statement.

Figure 5 shows the effect of this phase on four blocks of our running example (the full figure of the target CFG is shown in App. B of the TR [37]). The commands inserted just before the join block (here, B''_5) introduce a consistent variable version (here, `j4`) for use in the join block. It is convenient to speak of target variables in terms of their source program counterparts: we say e.g. that `j` *has version 4* on entry to block B'_5.

Compared to traces through the source program, the space of variable values in a trace through the target program is initially much larger; each version may, on entry to the CFG, have an arbitrary value. For example, `j4` may have any value on entry to B''_2; traces in which its value does not correspond to the constraint of the **`assume`** statements in B''_3 or B''_4 will go to magic and not reach B''_5. Importantly, however, not *all* traces go to magic; enough are preserved to simulate the executions of the original program: each **`assume`** statement constrains the value of exactly one variable version, and the same version is never constrained more than once. Capturing this delicate argument formally is the main challenge in certifying this step.

As extra parts of the passification phase, the Boogie verifier performs constant propagation and desugars old-expressions (using variable versions appropriate to the entry point of the CFG). We omit their descriptions here for brevity, but our implementation certifies them.

5.2 Passification Certification: Local Block Lemmas

To validate the passification phase, it is sufficient to show that each source execution is simulated by a corresponding target execution, made precise by constructing a relation between the states in these executions. Such *forward simulation* arguments are standard for proving correctness of compilers for deterministic languages. However, the situation here is more complex due to the fact that the target CFG has a much wider space of traces: the values of each versioned variable in the target program are initially unconstrained, meaning traces exist for all of their combinations. On the other hand, many of these traces do not survive the **`assume`** statements encountered in the target program. Picking the correct *single* trace or state to simulate a particular source execution would require knowledge of all variable assignments that are *going* to happen, which is not possible due to non-determinism and would preclude the block-modular proof strategies that our validation approach employs.

Instead, we generalise this idea to relating each single source state s with a *set* T of corresponding target program states. We define variable relations $\mathcal{V}_R$ at each point in a trace, making explicit the mappings used in the SSA step between source program variables and their corresponding versions. For example, on entry to block B'_2 in the source version of our running example (correspondingly B''_2 in the target), the $\mathcal{V}_R$ relation relates `i` to `i1` and `j` to `j2`. All states $t \in T$ must precisely agree with s w.r.t. $\mathcal{V}_R$ (e.g., $s(\texttt{i}) = t(\texttt{i1})$, $s(\texttt{j}) = t(\texttt{j2})$). On the other hand, our sets of states T are defined to be completely unconstrained (besides typing) for *future* variable versions. For example, for every $t \in T$ at the same point in our example, there will be states in T assigning each possible value (of the same type) to `i2` (and otherwise agreeing with t).

More precisely, for a set of variables X, we say that a set of states T *constrains at most* X *w.r.t. variable context* Λ if, for every $t \in T$, $z \notin X$, z is in Λ, and value v of z's type, we have $t[z \mapsto v] \in T$. In other words, the set T is closed under arbitrary changes to values of all variables in Λ but *not* in X. We construct our sets T such that they constrain at most *current and past versions* of program variables. It is this fact that enables us to handle subsequent **`assume`** statements in the target program and, in particular, to show that the set of possible traces in the target program never becomes empty while there are possible traces in the source program. For example, when relating the source command `j := j+1` in B'_3 with the target command **`assume`** `j3 = j2 + 1` in block B''_3, we use the fact that our set of states does not constrain `j3` to prove that, although many traces go to magic at this point, for a non-empty set of states $T' \subseteq T$ (those in which `j3` has the "right" value equal to `j2 + 1`), execution continues in the target.

We now make these notions more precise by showing the definition of our local block lemmas for the passification phase (See footnote 5).

Theorem 2 (Passification Local Block Lemma). *Let B be a source block with commands cs, whose corresponding target block has commands cs'; let $\mathcal{V}_R$ and $\mathcal{V}'_R$ be the variable relations at the beginning and end of B, respectively. Let X be a set of variable versions, and $\mathsf{N}(ns)$ be a normal state. Let T be a non-empty set of normal states such that $\mathsf{N}(ns)$ agrees with T according to $\mathcal{V}_R$, and T constrains at most X w.r.t. Λ_2. Furthermore, let Y be the variable versions corresponding to the targets of assignment and havoc statements in cs. If both*

1. $A, \Lambda_1, \Gamma, \Omega \vdash \langle cs, \mathsf{N}(ns)\rangle\ [\rightarrow]\ s' \wedge s' \neq \mathsf{M}$
2. $X \cap Y = \emptyset$

then there exists a non-empty set of normal states $T' \subseteq T$ s.t. T' constrains at most $X \uplus Y$ w.r.t. Λ_2 and for each $t' \in T'$, there exists a state t'^* s.t.

1. $A, \Lambda_2, \Gamma, \Omega \vdash \langle cs_2, t'\rangle\ [\rightarrow]\ t'^* \wedge (s' = \mathsf{F} \Longrightarrow t'^* = \mathsf{F})$
2. *If s' is a normal state, then s' and t' are related w.r.t. $\mathcal{V}'_R$ (and $t'^* = t'$).*

This lemma captures our generalised notion of forward simulation appropriately. The first conclusion expresses that the target does not get stuck and that failures are preserved, while the second shows that if the source execution neither fails nor stops then the resulting states are related. Note that premise 2 is essential in the proof to guarantee that the **assume** statements introduced by passification do not eliminate the chance to simulate source executions; the condition expresses that the variable versions newly constrained do not intersect with those previously constrained. To prove these lemmas over the commands in a single block, we are forced to check that the same version is not constrained twice.

5.3 Passification Certification: Global Block Theorems

As for all phases, we lift our local block lemmas to theorems certifying all executions *starting* from a particular block, and thus, ultimately, to entire CFGs. For the passification phase, most of the conceptual challenges are analogous to those of the local block lemmas; we similarly employ $\mathcal{V}_R$ relations between source variables and their corresponding target versions. To connect with our local block lemmas (and build up our global block theorems, which we do backwards through the CFG structure), we repeatedly require the key property that the set of variable versions constrained in our executions so far is disjoint from those which may be constrained by a subsequent **assume** statement (*cf.* premise 2 of our local block lemma above). Concretely tracking and checking disjointness of these concrete sets of variables is simple, but turns out to get expensive in Isabelle when the sets are large.

We circumvent this issue with our own *global versioning scheme* (as opposed to the versions used by Boogie, which are *independent* for different source variables): according to the CFG structure, we assign a *global* version number $\mathsf{ver}_{\mathcal{G}}(x)$

to each variable x in the target program such that, if x is constrained in a target block B' and y is constrained in another target block B'' reachable from B', then $\mathsf{ver}_{\mathcal{G}}(x) < \mathsf{ver}_{\mathcal{G}}(y)$. Such a consistent global versioning always exists in the target programs generated by Boogie because the only variables not constrained exactly once *in the program* are those used to synchronise executions (i.e. `j4` in Fig. 5), which always appear right before branches are merged. We can now encode our disjointness properties much more cheaply: we simply compare the *maximal* global version of all already-constrained variables with the *minimal* global version of those (potentially) to be constrained. Since we represent variables as integers in the mechanisation, we directly use our global version *as* the variable name for the target program; there is no need for an extra lookup table. Note that (readability aside) it makes no difference which variables names are used in intermediate CFGs; we ultimately care only about validating the original CFG.

6 The VC Phase

In this section, we present the validation of the VC phase in the Boogie verifier. This phase has two main aspects: (1) it encodes and desugars all aspects of the Boogie type system, employing additional uninterpreted functions and axioms to express its properties [33]; program expression elements such as Boogie functions are analogously desugared in terms of these additional uninterpreted functions, creating a non-trivial logical gap between expressions as represented in the VC and those from the input program. (2) It performs an efficient (block-by-block) calculation of a weakest precondition for the (acyclic, passified) CFG, resulting in a formula characterising its verification requirements, subject to background axioms and other hypotheses.

6.1 VC Structure

The generated VC has the following overall structure (represented as a shallow embedding in our certificates)[7]:

$$\forall \underbrace{\textit{VC quantifiers}}_{\substack{\text{type encoding parameters,}\\ \text{functions, variable values}}} . \; (\underbrace{\textit{VC assumptions}}_{\substack{\text{type encoding,}\\ \text{func./var./prog. axioms}}} \Longrightarrow \textit{CFG WP})$$

The VC quantifies over parameters required for the type encoding, as well as VC counterparts representing the variable values and functions in the Boogie program. The VC body is an implication, whose premise contains: (1) assumptions that axiomatise the type encoding parameters, (2) axioms expressing the typing of Boogie variables and functions, and (3) assumptions directly relating

[7] Note that top-level quantification over functions is implicit in the (first-order) SMT problem generated by Boogie; we quantify explicitly in our Isabelle representation.

to axioms explicitly declared in the Boogie program. The conclusion of the implication is an optimised version of the weakest (liberal) precondition (WP) of the CFG.[8]

6.2 Boogie's Logical Encoding of the Boogie Type System

We first briefly explain Boogie's logical encoding of its own type system. Values and types are represented at the VC level by two uninterpreted carrier sorts V and T. An uninterpreted function typ from V to T maps each value to the representation of its type. Boogie type constructors are each modelled with an (injective) uninterpreted function C with return sort T and taking arguments (per constructor parameter) of sort T. For example, a type constructor $\mathit{List}(t)$ is represented by a VC function from T to T. Projection functions are also generated for each type constructor (C_i^π for each type argument at position i), e.g. mapping the representation of a type $\mathit{List}(t)$ to the representation of type t.

This encoding is then used in the VC to recover Boogie typing constraints for the untyped VC terms. Recovering the constraints is not always straightforward due to optimisations performed by Boogie. For example, the VC translation of the Boogie expression $\forall_{ty}\, t.\ \forall x : \mathit{List}(t).\ e$ no longer quantifies over types; all original occurrences of t in e having been translated to $\mathit{List}_1^\pi(\mathit{typ}(x))$. This optimisation reflects that this particular type quantification is redundant, since t can be recovered from the type of x.[9]

6.3 Working from VC Validity

Our certificates assume that the generated VC is valid (certifying the validity-checking of the VC by an SMT solver is an orthogonal concern). However, connecting VC validity back to block-level properties about the specific program requires a number of technical steps. We need to construct Isabelle-level semantic values to *instantiate* the top-level quantifiers in the VC such that the corresponding VC assumptions (left-hand side of the VC) can be proved and, thus, validity of the corresponding WP can be deduced. Moreover, we must ensure that our instantiation yields a WP whose validity implies correctness of the Boogie program. For example, a top-level VC quantifier modelling a Boogie function f must be instantiated with a mathematical function that behaves in the same way as f for arguments of the correct type.

We instantiate the carrier sort V for values in the VC with the corresponding type denoting Boogie values in our formalisation; the carrier sort T for *types* is instantiated to be all Boogie types that do not contain free variables (i.e. closed types). Constructing explicit models for the quantified functions used to

[8] One difference in our version of the Boogie verifier is that we switched off the generation of extra variables introduced to report error traces [32]; these are redundant for programs that do not fail and further complicate the VC structure.

[9] Note that in the VC the quantification over x ranges over all values of sort V. An implication is used to consider only those x for which $\mathit{typ}(x) = \mathit{List}(\mathit{List}_1^\pi(\mathit{typ}(x)))$.

model Boogie's type system (satisfying, e.g., suitable inverse properties for the projection functions) is straightforward. For the VC-level variable values, we can directly instantiate the corresponding values in the initial Boogie program state.

VC-level functions representing those declared in the Boogie program are instantiated as (total) functions which, *for input values of appropriate type* (the arguments and output are untyped values of sort V), are defined simply to return the same values as the corresponding function in our model. However, perhaps surprisingly, Boogie's VC embedding of functions logically requires functions to return values of the specified return type even if the input values do not have the types specified by the function. In such cases, we define the instantiated function to return some value of the specified type, which is possible since in well-formed contexts every closed type has at least one value in our model.

After our instantiation, we need to prove the hypotheses of the VC's implication; in particular that all axioms (both those generated by the type system encoding and those coming from the program itself) are satisfied. The former are standard and simple to prove (given the work above), while the latter largely follow from the assumption that each declared axiom must be satisfied in the initial state restricted to the constants. The only remaining challenge is to relate VC expressions with the evaluation of corresponding Boogie expressions; an issue which also arises (and is explained) below, where we show how to connect validity of the instantiated WP to the program.

6.4 Certifying the VC Phase

Boogie's weakest precondition calculation is made size-efficient by the usage of explicit named constants for the weakest preconditions $wp(B, \mathbf{true})$ for each block B, which is defined in terms of the named constants for its successor blocks. For example, in Fig. 5, $wp(B_2'', \mathbf{true})$ is given by $i_1^{vc} \neq 0 \implies wp(B_3'', \mathbf{true}) \wedge wp(B_4'', \mathbf{true})$. Here i_1^{vc} is the value that we instantiated for the variable `i1`.

We exploit this modular construction of the generated weakest precondition for the local and global block theorems. We prove for each block B with commands cs the following local block lemma:

Theorem 3 (VC Phase Local Block Lemma).
If $A, \Lambda, \Gamma, \Omega \vdash \langle cs, \mathsf{N}(ns)\rangle \, [\rightarrow] \, s'$ *and* $wp(B, \mathbf{true})$ *holds, then* $s' \neq \mathsf{F}$ *and if* s' *is a normal state, then* $\forall B_{suc} \in successors(B).\ wp(B_{suc}, \mathbf{true})$.

Once one has proved this lemma for all blocks in the CFG, combining them to obtain the corresponding global block theorems (via our usual reverse walk of the CFG) is straightforward. The main challenge is in decomposing the proof for the local block lemma itself for a block B, for which we outline our approach next.

By this phase, the first command in B must be either an **`assume`** e or an **`assert`** e command. In the former case, we rewrite $wp(B, \mathbf{true})$ into the form $e^{vc} \implies H$, where e^{vc} is the VC counterpart of e and where H corresponds to the weakest precondition of the remaining commands. This rewriting may involve undoing certain optimisations Boogie's implementation performed on the formula structure. Next, we need to prove that e evaluates to e^{vc} (see below). Hence, if e evaluates to **true** (the execution does not go to magic) then H must be true, and we can continue inductively. The argument for **assert** e is similar but where we rewrite the VC to $e^{vc} \wedge H$ (i.e. e^{vc} and H must both hold); if e evaluates to e^{vc}, we know that the execution does not fail.

Proving that e evaluates to e^{vc} arises in both cases and also in our previous discharging of VC hypotheses. Note that, in contrast to e, e^{vc} is not a Boogie expression, but a shallowly embedded formula that includes the instantiations of quantified variables we constructed above. Showing this property works largely on syntax-driven rules that relate a Boogie expression with its VC counterpart, except for extra work due to mismatching function signatures and optimisations that Boogie made either to the formula structure or via the type system encoding (*cf.* Sect. 6.2). We handle some of these cases by showing that we can rewrite the formula back into the unoptimised standard form we require for our syntax-driven rules and in other cases we directly work with the optimised form. Both cases are automated using Isabelle tactics.

This concludes our discussion of the certification of Boogie's three key phases. Combining the three certificates yields an end-to-end proof that the validity of the generated verification conditions implies the correctness of the input program, that is, that the given verification run is sound.

7 Implementation and Evaluation

In this section, we evaluate our certifying version of the Boogie verifier [36], which produces Isabelle certificates proving the correctness of Boogie's pipeline for programs it verifies.

We have implemented our validation tool as a new C# module compiled with Boogie. We instrumented Boogie's codebase to call out to our module, which allows us to obtain information that we can use to validate the key phases, and extended parts of the codebase to extract information more easily. Moreover, we disabled counter-example related VC features and the generation of VC axioms for any built-in types and operators that we do not support. We added or changed fewer than 250 non-empty, uncommented lines of code across 11 files in the existing Boogie implementation.

Given an input file verified by Boogie, our work produces an Isabelle certificate per procedure p that certifies the correctness of the corresponding CFG-to-DAG source CFG as represented internally in Boogie. The generation and checking of the certificate is fully automatic, without any user input. We use a combination of custom and built-in Isabelle tactics. In addition to the three key phases we describe in detail, our implementation also handles several smaller transformations made by Boogie, such as constant propagation. Our tool currently supports the default options of Boogie (only) and does not support advanced source-level *attributes* (for instance, to selectively force procedures to be inlined).

Table 1. Selection of algorithmic examples with the lines of code (LOC), the number of procedures (#P), the time it takes for Isabelle to check the certficate in seconds (the average of 5 runs on a Lenovo T480 with 32 GB, i7-8550U 1.8 GhZ, Ubuntu 18.04 on the Windows Subsystem for Linux), and the certificate size expressed as the number of non-empty lines of Isabelle.

Name	LOC	#P	Time [s]	Size
TuringFactorial	29	1	19.4	1986
Find	27	2	27.3	2100
DivMod	69	2	28.4	4753
Summax [27]	23	1	19.1	1953
MaxOfArray [12]	22	1	19.9	1944
SumOfArray [12]	22	1	18.7	1534
Plateau [12]	50	1	22.9	2019
WelfareCrook [12]	52	1	39.4	2528
ArrayPartitioning [12]	57	2	27.6	3514
DutchFlag [12]	76	2	52.8	3994

We evaluated our work in two ways. Firstly, to evaluate the applicability of our certificate generation, we automatically collected all input files with at least one procedure from Boogie's test suite [1] which verify successfully and which either use no unsupported features or are easily desugared (by hand) into versions without them. This includes programs with procedure calls since Boogie simply desugars these in an early stage. For programs employing attributes, we checked whether the program still verifies *without* attributes, and if so we also kept these. In total, this yields 100 programs from Boogie's test suite. Secondly, we collected a corpus of ten Boogie programs which verify interesting algorithms with non-trivial specifications: three from Boogie's test suite and seven from the literature [12,27]. Where needed we manually desugared usages of Boogie maps (which we do not yet support) using type declarations, functions, and axioms.

Of the 100 programs from Boogie's test suite, we successfully generate certificates in 96 cases. The remaining 4 cases involve special cases that we do not handle yet. For 2 of them, extending our work is straightforward: one special case includes a naming clash and the other case can be amended by using a more specific version of a helper lemma. The remaining two fail because of our incomplete handling of function calls in the VC phase when combined with coercions between VC integers or booleans and their Boogie counterparts. Handling this is more challenging but is not a fundamental issue.

For the corpus of 10 examples, Table 1 shows the generated certificate size and the time for Isabelle to check their validity.[10] The ratio of certificate size to code size ranges from 41 to 89; this rather large ratio emphasises the substantial work in formally validating the substantial work which Boogie's implementation

[10] The time to generate the certificate is not included, but is negligible here.

performs. Optimisations to further reduce the ratio are possible. The validation of certificates takes usually under one second per line of code. While these times are not short, they are acceptable since certificate generation needs to run only for (verified) release versions of the program in question.

8 Related Work

Several works explore the validation of program verifiers. Garchery et al. [20] validate VC rewritings in the Why3 VC generator [16]. Unlike our work, they do not connect VCs with programs and do not handle the erasure of polymorphic types. Strub et al. [39] validate part of a previous version of the F* verifier [40] by generating a certificate for the F* type checker itself, which type checks programs by generating VCs. Like us, they assume the validity of the generated VC itself, but they do not consider program-to-program transformations such as ours. Another approach is taken by Aguirre [2] who shows how one can map proofs of the VC back to correctness of an F* program. They prove a once-and-for-all result, but the approach could be lifted to a validation approach using the proof-producing capability of SMT solvers [7]. Lifting the approach would require extending the work to handle classical instead of constructive VC proofs.

There is some work on proving VC generator implementations correct once and for all, although none of the proven tools are used in practice. Homeier and Martin [23] prove a VC generator correct in HOL for an executable language and a simpler VC phase than Boogie's. Herms et al. [22] prove a VC generator inspired by Why3 correct in Coq. However, some more-challenging aspects of Why3's VC transformation and polymorphic type system are not handled. Vogels et al. [44] prove a toolchain for a Boogie-like language correct in Coq, including passification and VC phases. However, the language is quite limited: without unstructured control flow, loops (i.e. no need for a CFG-to-DAG phase), functions, or polymorphism (i.e. no type encoding). Verifiers other than VC generators, include the verified Verasco static analyzer [25], which supports a realistic subset of C, but whose performance is not yet on par with unverified, industrial analyzers.

Validation has also been explored in other settings. Alkassar et al. [3] adjust graph algorithms to produce witnesses that can be then used by verified validators to check whether the result is correct. In the context of compiler correctness, many validation techniques express a per-run validator in Coq, prove it correct once-and-for-all [8,41,43], and then extract executable code (the extraction must be trusted). In the verified CompCert compiler [34], such validators have been used in combination with the once-and-for-all approach. Validators are used for phases that can be more easily validated than proved correct once and for all. One such example related to our certification of the passification phase is the validation of the SSA phase [8], dealing also with versioned variables in the target (but not with **assume** statements that prune executions). In contrast to our work, they require an explicit notion of CFG domination and they do not use a global versioning scheme to efficiently check that two parts of the CFG constrain disjoint versions. Our versioning idea is similar to a technique used for the validation of a dominator relation in a CFG [9], which assigns intervals to basic blocks (as opposed to assigning versions to variables) to efficiently determine whether a block dominates another one. The validation of the Cogent compiler [38] follows a similar approach to ours in that it generates proofs in Isabelle.

9 Conclusion

We have presented a novel verifier validation approach, and applied it successfully to three key phases of the Boogie verifier, providing formal underpinnings for both the language and its verifier for the first time. Our work demonstrates that it is feasible to provide strong formal guarantees regarding the verification results of practical VC generators written in modern mainstream languages.

In the future, we plan to extend our supported subset of Boogie, e.g. to include procedure calls and bitvectors. Supporting Boogie's potentially-impredicative maps is the main open challenge: maps can take other maps as input, potentially including themselves. The challenge with this feature is to still be able to express a type in Isabelle capturing all Boogie values despite the potentially-cyclic nature of map types. In practice, however, this may not be required in full generality: we have observed that Boogie front-ends rarely use maps that contain maps of the same type as input. Therefore, we plan to extend our technique to support a suitably-expressive restricted form of Boogie maps.

Acknowledgements. We thank Alain Delaët–Tixeuil for his earlier work on this topic, Thibault Dardinier for improving our artifact, Martin Clochard for helpful discussions and the anonymous reviewers for their valuable comments. This work was partially funded by the Swiss National Science Foundation (SNSF) under Grant No. 197065.

References

1. Boogie verifier repository. https://github.com/boogie-org/boogie
2. Aguirre, A.: Towards a provably correct encoding from F* to SMT. Technical report, INRIA (2016)
3. Alkassar, E., Böhme, S., Mehlhorn, K., Rizkallah, C.: A framework for the verification of certifying computations. JAR **52**(3), 241–273 (2014)
4. Astrauskas, V., Müller, P., Poli, F., Summers, A.J.: Leveraging Rust types for modular specification and verification. In: OOPSLA (2019)
5. Barnett, M., Fähndrich, M., Leino, K.R.M., Müller, P., Schulte, W., Venter, H.: Specification and verification: the Spec# experience. CACM **54**(6), 81–91 (2011)
6. Barnett, M., Leino, K.R.M.: Weakest-precondition of unstructured programs. In: PASTE (2005)
7. Barrett, C., de Moura, L., Fontaine, P.: Proofs in satisfiability modulo theories. In: All about Proofs, Proofs for All, Mathematical Logic and Foundations, vol. 55, pp. 23–44. College Publications (2015)
8. Barthe, G., Demange, D., Pichardie, D.: Formal verification of an SSA-based middle-end for compcert. TOPLAS **36**(1), 1–35 (2014)

9. Blazy, S., Demange, D., Pichardie, D.: Validating dominator trees for a fast, verified dominance test. In: ITP (2015)
10. Blom, S., Darabi, S., Huisman, M., Oortwijn, W.: The VerCors tool set: verification of parallel and concurrent software. In: iFM (2007)
11. Böhme, S., Weber, T.: Fast LCF-style proof reconstruction for Z3. In: ITP (2010)
12. Chen, Y., Furia, C.A.: Triggerless happy - intermediate verification with a first-order prover. In: iFM (2017)
13. Cohen, E., et al.: VCC: a practical system for verifying concurrent C. In: TPHOLs (2009)
14. Coq Development Team, T.: The Coq Reference Manual, version 8.10, available electronically at (2019). http://coq.inria.fr/documentation
15. Ekici, B., et al.: SMTCoq: a plug-in for integrating SMT solvers into Coq. In: CAV (2017)
16. Filliâtre, J.C., Paskevich, A.: Why3 – where programs meet provers. In: ESOP (2013)
17. Filliâtre, J.C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: CAV (2007)
18. Flanagan, C., Saxe, J.B.: Avoiding exponential explosion: generating compact verification conditions. In: POPL (2001)
19. Fleury, M., Schurr, H.: Reconstructing veriT proofs in Isabelle/HOL. In: PxTP (2019)
20. Garchery, Q., Keller, C., Marché, C., Paskevich, A.: Des transformations logiques passent leur certificat. In: JFLA (2020)
21. Hecht, M.S., Ullman, J.D.: Flow graph reducibility. SIAM J. Comput. **1**(2), 188–202 (1972)
22. Herms, P., Marché, C., Monate, B.: A certified multi-prover verification condition generator. In: VSTTE (2012)
23. Homeier, P.V., Martin, D.F.: A mechanically verified verification condition generator. Comput. J. **38**(2), 131–141 (1995)
24. Isabelle Development Team, T.: The Isabelle Documentation, version June 2019, available electronically at (2019). https://isabelle.in.tum.de/documentation.html
25. Jourdan, J.H., Laporte, V., Blazy, S., Leroy, X., Pichardie, D.: A formally-verified C static analyzer. In: POPL (2015)
26. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-c: a software analysis perspective. Formal Aspects Comput. **27**(3), 573–609 (2015)
27. Klebanov, V., et al.: The 1st verified software competition: Experience report. In: FM (2011)
28. Lal, A., Qadeer, S., Lahiri, S.K.: A solver for reachability modulo theories. In: CAV (2012)
29. Leino, K.R.M.: This is Boogie 2 (June 2008). https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/
30. Leino, K.R.M.: Efficient weakest preconditions. Inf. Process. Lett. **93**(6), 281–288 (2005)
31. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: LPAR (2010)
32. Leino, K.R.M., Millstein, T.D., Saxe, J.B.: Generating error traces from verification-condition counterexamples. Sci. Comput. Program. **55**(1–3), 209–226 (2005)
33. Leino, K.R.M., Rümmer, P.: A polymorphic intermediate verification language: design and logical encoding. In: TACAS (2010)

34. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: POPL (2006)
35. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: a verification infrastructure for permission-based reasoning. In: VMCAI (2016)
36. Parthasarathy, G., Müller, P., Summers, A.J.: Formally validating a practical verification condition generator - artifact (2021). https://doi.org/10.5281/zenodo.4726554
37. Parthasarathy, G., Müller, P., Summers, A.J.: Formally validating a practical verification condition generator (extended version) (2021). arXiv:2105.14381
38. Rizkallah, C., et al.: A framework for the automatic formal verification of refinement from Cogent to C. In: ITP (2016)
39. Strub, P.Y., Swamy, N., Fournet, C., Chen, J.: Self-certification: Bootstrapping certified typecheckers in F* with Coq. In: POPL (2012)
40. Swamy, N., et al.: Dependent types and multi-monadic effects in F*. In: POPL (2016)
41. Tristan, J.B., Leroy, X.: Formal verification of translation validators: a case study on instruction scheduling optimizations. In: POPL (2008)
42. Tristan, J.B., Leroy, X.: Verified validation of lazy code motion. In: PLDI (2009)
43. Tristan, J.B., Leroy, X.: A simple, verified validator for software pipelining. In: POPL (2010)
44. Vogels, F., Jacobs, B., Piessens, F.: A machine-checked soundness proof for an efficient verification condition generator. In: SAC (2010)

Automatic Generation and Validation of Instruction Encoders and Decoders

Xiangzhe Xu, Jinhua Wu, Yuting Wang(✉), Zhenguo Yin, and Pengfei Li

Shanghai Jiao Tong University, Shanghai 200240, China
yuting.wang@sjtu.edu.cn

Abstract. Verification of instruction encoders and decoders is essential for formalizing manipulation of machine code. The existing approaches cannot guarantee the critical *consistency* property, i.e., that an encoder and its corresponding decoder are mutual inverses of each other. We observe that consistent encoder-decoder pairs can be automatically derived from bijections inherently embedded in instruction formats. Based on this observation, we develop a framework for writing specifications that capture these bijections, for automatically generating encoders and decoders from these specifications, and for formally validating the consistency and soundness of the generated encoders and decoders by synthesizing proofs in Coq and discharging verification conditions using SMT solvers. We apply this framework to a subset of X86-32 instructions to illustrate its effectiveness in these regards. We also demonstrate that the generated encoders and decoders have reasonable performance.

Keywords: Formalized instruction formats · Verified parsing · Program synthesis · Proof synthesis · Translation validation

1 Introduction

Software that manipulates machine code such as compilers, OS kernels and binary analysis tools, relies on *instruction encoders and decoders* for extracting structural information of instructions from machine code and for translating such information back into binary forms. Because of the sheer amount of instructions provided by any instruction set architecture (ISA) and the complexity of instruction formats, it is extremely tedious and error-prone to implement instruction encoders and decoders by hand. Therefore, the literature contains abundant work on automatic generation of instruction encoders and decoders, often from specifications written in a formal language capable of concisely and accurately characterizing instruction formats on various ISAs [7,12,15].

Unfortunately, the above approaches generate little formal guarantee, therefore not suitable for rigorous analysis or verification of machine code. In those settings, instruction encoders and decoders are expected to be *consistent*, i.e., any encoder and its corresponding decoder are inverses of each other, and *sound*, i.e., they meet formal specifications of instruction formats that human could easily understand and check.

A. Silva and K. R. M. Leino (Eds.): CAV 2021, LNCS 12760, pp. 728–751, 2021.
https://doi.org/10.1007/978-3-030-81688-9_34

Consistency is essential for verification of machine code because it guarantees that manipulation and reasoning over the abstract syntax of instructions can be mirrored precisely onto their binary forms. For example, verification of assemblers requires that instruction decoding reverts the assembling (encoding) process [20]. However, the previously proposed approaches to verifying instruction encoders and decoders all fail to establish consistency: to handle the complexity of instruction formats (especially that of CISC architectures), they employ expressive but ambiguous specifications such as context-free grammars or variants of regular expressions, from which it is impossible to derive consistent encoders and decoders. A representative example is the bidirectional grammar proposed by Tan and Morrisett [18]. It is an extension of regular expressions for writing instruction specifications from which verified encoders and decoders can be generated. However, because of the ambiguity of such specifications, two different abstract instructions may be encoded into the same *bit string* (i.e., a sequence of bits). When the decoder is deterministic, not all encoded instructions can be decoded back to the original instructions.

In this paper, we present an approach to automatic construction of instruction encoders and decoders that are verified to be consistent and sound. It is based on the observation that an instruction format inherently implies a bijection between abstract instructions and their binary forms that manifests as the determinacy of instruction decoding in actual hardware. This is true even for the most complicated CISC architectures. From a well-designed instruction specification that *precisely* captures this bijection, we are able to extract an appropriate representation of instructions, a pair of instruction encoder and decoder between this representation and the binary forms of instructions, and the consistency and soundness proofs of the encoder and decoder.

Based on the above ideas, we develop a framework for automatically generating consistent and sound instruction encoders and decoders. It extends the approach to specifying and generating instruction encoders and decoders proposed by Ramsey and Fernández [15] with mechanisms for *validating* their soundness and consistency by using theorem provers and SMT solvers. The framework consists of the following components (which are also our technical contributions):

– *A specification language for describing instruction formats.* This language is deliberately weaker in expressiveness than regular expressions while strong enough for describing instruction formats on common ISAs. Different from the existing ISA specification languages, it is rich enough for precisely capturing the syntactical structures of instructions and their operands, which implicitly encode a bijection between the abstract and the binary representations of instructions.
– *The algorithms for automatically generating encoders and decoders from instruction specifications.* Given any instruction specification, they generate an abstract syntax of instructions, a partial function from the abstract syntax to bit strings (i.e., an encoder) and a partial function from bit strings to the abstract syntax (i.e., a decoder). The generated definitions are formalized in

the Coq theorem prover so that the encoder and decoder can be formally validated later.
– *The algorithms for automatically validating the consistency and soundness of the generated encoders and decoders.* Given any instruction specification, they synthesize the consistency and soundness proofs for the generated encoder and decoder in Coq. This is possible because the bijection implied by the original specification guarantees that the encoder and decoder are inverses of each other, under the requirement that the binary "shapes" of different instructions or operands do not overlap with each other. This requirement is inherently satisfied by any instruction format, and can be easily proved with SMT solvers.

To demonstrate the effectiveness of our framework, we have applied it to a subset of 32-bit X86 instructions. In the rest of this paper, we first introduce relevant background information for this work and discuss the inadequacy of the existing work in Sect. 2. We then give an overview of our framework in Sect. 3 by further elaborating on the points above. After that, we discuss the definition of our specification language and the ideas supporting its design in Sect. 4. In the two subsequent sections Sect. 5 and Sect. 6, we discuss the algorithms for automatically generating and validating encoders and decoders. In Sect. 7, we present the evaluation of our framework. Finally, we discuss related work and conclude in Sect. 8.

2 Background

For our approach to work, the specification language we use must support the instruction formats on contemporary RISC and CISC architectures. In this section, we first introduce the key characteristics of these formats and then present a running example. We conclude this section by exposing the inadequacy of the existing approaches in capturing the bijections between the abstract and binary forms of instructions.

2.1 The Characteristics of Instruction Formats

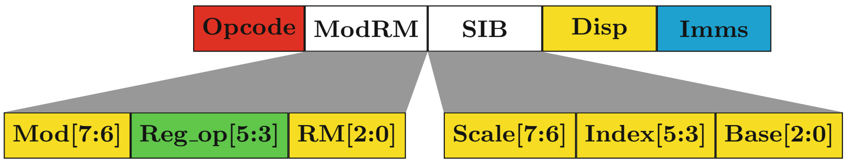


Fig. 1. The format of 32-bit X86 instructions

Instruction formats on CISC architectures may vary in length and structure even for the same type of instructions and may contain complex dependencies between their operands. In contrast, instructions on RISC architectures usually have fixed formats which are largely subsumed by CISC formats. Therefore, we focus on handling CSIC formats in this paper.

We use the format of 32-bit X86 instructions as an example to illustrate the complex characteristics of CISC instructions. It is depicted in Fig. 1. An instruction is divided into a sequence of *tokens* where each token is one or more bytes playing a particular role. The first token **Opcode** partially or fully determines the basic type of the instruction; it may be one to three bytes long. Following **Opcode** is an one-byte token **ModRM**. **ModRM** is further divided into a sequence of *fields* where a field $f[n_1 : n_2]$ represents a segment of the token named f that occupies the n_2-th to n_1-th bits in that token. Depending on the value of **Opcode**, **ModRM** may or may not exist. When it exists, the value of **Reg_op[5:3]** may contain the encoded representation of a register operand. Another operand of the instruction may be an *addressing mode*. It is collectively determined by the values of **Mod[7:6]**, **RM[2:0]**, the token **SIB** (scaled index byte) and the displacement **Disp** following **ModRM**. Finally, the instruction may have an operand of immediate values in the token **Imms**.

For simplicity of our discussion, we have omitted some details such as the optional prefixes of instructions in Fig. 1. However, this simplified form is already enough to expose the key characteristics and complexity of CISC instruction formats (some of which also manifest in RISC). We summarize them below:

1. *Instructions as Composition of Components*: At the abstract level, an instruction consists of a collection of *components*. Each component serves a specific purpose and concretely corresponds to certain fields or tokens in the instruction format. For example, the constituents of 32-bit X86 instructions can be classified into four different kinds of components (marked with different colors in Fig. 1): the component determining the types of instructions (**Opcode**), the component denoting register operands (**Reg_op[5:3]**), the component denoting addressing modes (**Mod[7:6]**, **RM[2:0]**, **SIB** and **Disp**) and the component denoting immediate values (**Imms**).
2. *Variance of Components*: The concrete forms of components vary in different ways. A component may correspond to a single token (e.g., **Opcode** and **Imms**), a single field (e.g., **Reg_op[5:3]**), a mixing of fields and tokens (e.g., addressing modes), or other forms not shown here. Moreover, the abstract and concrete forms of a *single* type of components can also vary significantly such as the different addressing modes supported by X86 (as we shall see in detail in the following section).
3. *Interleaving of Components*. In most cases, there are clear sequential orders between the concrete representations of components. For example, the component of addressing modes immediately follows that of opcode and precedes that of immediate values. In the other cases, components may be interleaved with each other. For example, the component of register operands is interleaved with the component of addressing modes.
4. *Dependencies between and in Components*: The existence and forms of components are affected by the dependencies between each other and between their

own fields or tokens. For example, if an instruction does not take any argument, then the value of its **Opcode** determines that there is no token following **Opcode**. For another example, when **Mod**[**7:6**] contains the value `0b11`, the addressing mode is simply a register operand. Otherwise, the addressing mode may further depends on the values in **SIB** and **Disp**.

Note that, despite the above complexity, an instruction format is designed to inherently embed a (partial) bijection between the binary forms of instructions and their abstract representation as the composition of components. This is to ensure the determinacy of instruction decoding in hardware. This bijection is the central property to be investigated in this work.

2.2 A Running Example

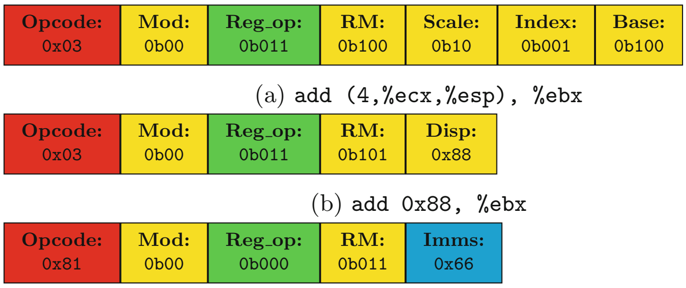
Table 1. The different forms of addressing modes

AddrMode	Mod	RM	Scale	Index	Base	Disp
$\mathbf{r}$	`0b11`	$\mathbf{r}$	–	–	–	–
$(\mathbf{r})$	`0b00`	$\mathbf{r} \neq$ `0b100` $\wedge \mathbf{r} \neq$ `0b101`	–	–	–	–
$(\mathbf{d})$	`0b00`	`0b101`	–	–	–	$\mathbf{d}$
$(\mathbf{s} * \mathbf{i} + \mathbf{b})$	`0b00`	`0b100`	$\mathbf{s}$	$\mathbf{i} \neq$ `0b100`	$\mathbf{b} \neq$ `0b101`	–
...	...	...	...	...	...	...

We present an example of encoding the `add` instruction to concretely illustrate the characteristics of the X86 instruction format. It will be used as a running example for the rest of the paper. The operands of `add` may have many forms. For simplicity, we only consider two cases: *1)* the first operand is a register while the second one is an addressing mode, and *2)* the first operand is an addressing mode while the second one is an immediate value.

In the first case, **Opcode** is `0x03`, indicating that **ModRM** exists and the first operand is encoded in its **Reg_op** field. The addressing mode has over 23 combinations because of the dependencies and constraints over their fields. We list only some of the combinations in Table 1, where - indicates that this field or token does not exist. The first row shows the direct addressing mode **r** where **Mod** is `0b11` and **RM** contains the encoded register operand **r**. The following three rows shows different kinds of indirect addressing modes. They are valid only if **Mod** is `0b00` and further constraints are satisfied. For example, the second row shows the indirect addressing mode **(r)** where **r** is encoded in **RM**. In this case, **r** must neither be **ESP** (encoded as `0b100`) nor be **EBP** (encoded as `0b101`). Similarly, the addressing mode $(\mathbf{s} * \mathbf{i} + \mathbf{b})$ requires that **RM** must be `0b100`, **Index** must not be `0b100` and **Base** must not be `0b101`.

In the second case, **Opcode** is `0x81`, indicating that **ModRM** exists, the first operand is an addressing mode, and the second operand is an immediate value following it. Here, **Reg_Op** must be `0b000`.



(a) `add (4,%ecx,%esp), %ebx`

(b) `add 0x88, %ebx`

(c) `add $0x66, (%ebx)`

Fig. 2. Some concrete examples of instruction encoding

We demonstrate the concrete examples of encoding `add (4,%ecx,%esp), %ebx`, `add 0x88, %ebx` and `add $0x66, (%ebx)` in Fig. 2 where `%ebx` and `%ecx` are encoded into `0b011` and `0b001`, respectively (the order of operands is *reversed* because we use the AT&T assembly syntax). Note how the forms of operands change significantly depending on the different values in the related fields. Note also, despite such complex dependencies, a bit string representing a valid `add` instruction corresponds to a *unique* combination of components.

2.3 Inadequacy of the Existing Approaches

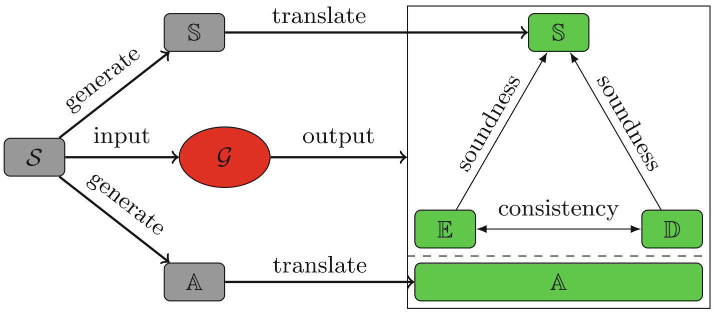
The existing approaches to specifying instructions are either *1)* too general and allow ambiguity or *2)* too low-level and break the component-based abstraction we just described. Either way, they fail to capture the inherent bijection embedded in an instruction format.

The bidirectional grammars [18] demonstrate the first kind of inadequacy. They contain the alternation grammar `Alt` g_1 g_2 for matching a bit string s when either the sub-grammar g_1 or g_2 matches s. The ambiguity arises when both g_1 and g_2 match s: in this case, the same s corresponds to two different internal representations. Therefore, bidirectional grammars cannot encode bijections in general. The same can be said for other work on verified parsing based on ambiguous grammars. We shall discuss them in detail in Sect. 8.

The Specification Language for Encoding and Decoding (or SLED) demonstrates the second kind of inadequacy [15]. It is a language for describing translations between symbolic and binary representations of machine instructions. On the surface, SLED takes the component-based view in specifying instructions. However, SLED specifications are interpreted through a normalization process by which every component is flattened into a sequence of tokens. After that, the structural information of components is completely lost. As a result, users can only derive encoders from the normalized specifications. They need to write decoders by using completely different specifications called "matching statements." This inability to generate matching encoders and decoders from a single specification is a common phenomenon in other approaches to ISA specifications.

In summary, no existing approach can precisely capture the bijections inherently embedded in instruction formats. This is the main intellectual problem we try to tackle in this paper. We shall elaborate on our solution to this problem in the remaining sections.

3 An Overview of the Framework



$\mathcal{S}$:	instruction specifications	(in CSLED)
$\mathcal{G}$:	algorithms for generating formal definitions and proofs	(in C++)
$\mathbb{A}$:	abstract syntax of instructions	(on paper and in Coq)
$\mathbb{S}$:	relational specifications of instructions	(on paper and in Coq)
$\mathbb{E}$ and $\mathbb{D}$:	encoders and decoders	(in Coq)

Fig. 3. The framework

We develop a framework for automatic generation of verified encoders and decoders that are consistent and sound. It is depicted in Fig. 3. To generate formally verified encoders and decoders, users first need to write down a specification of instructions $\mathcal{S}$ in a language called CSLED (or CoreSLED). CSLED is an enhancement to SLED for characterizing the bijection between the binary forms and the abstract syntax of instructions. Roughly speaking, $\mathcal{S}$ consists of a collection of *class* definitions, each of which defines a unique type of components that form instructions or their operands; the "top-most" class defines the type of instructions. Each class is associated with a set of *patterns* to uniquely determine a bijection between the binary and abstract forms of components in that class. Note that this bijection exists only when certain *well-formedness conditions* for patterns are satisfied. We shall elaborate on these ideas in Sect. 4.

From $\mathcal{S}$, the following definitions are generated and translated into Coq:

– The abstract syntax of instructions $\mathbb{A}$. It is a collection of algebraic data types corresponding to the classes defined in $\mathcal{S}$.
– A relational specification of $\mathcal{S}$ called $\mathbb{S}$. For each class, $\mathbb{S}$ contains a binary predicate that precisely captures the relation between components of that class and their binary forms. We write $\mathbb{R}[\![\mathcal{K}]\!]\ k\ l$ to denote that the component k of class $\mathcal{K}$ has the binary form l.

Then, $\mathcal{S}$ is fed into a collection of algorithms $\mathcal{G}$ to generate the following definitions and proofs in Coq:

- An encoder $\mathbb{E}$ and a decoder $\mathbb{D}$. The encoder is a set of partial functions—one for each class—from the abstract syntax of that class to bit strings. We write $\mathbb{E}_{\mathcal{K}}(k) = \lfloor l \rfloor$ to denote that l is the result of encoding a component k of class $\mathcal{K}$ where $\lfloor \rfloor$ denotes the `some` constructor of the option type. Conversely, the decoder is a set of partial functions from bit strings to the abstract syntax. We write $\mathbb{D}_{\mathcal{K}}(l{+\!+}l') = \lfloor (k, l') \rfloor$ to denote the decoding of the bit string l into a component k of class $\mathcal{K}$ where $++$ is the append operation of bit strings. Here, the tailing bit string l' represents the remaining bits after decoding the first component.
- The proof of consistency between the encoder and decoder. The consistency theorems are stated as the mutual inversion between the encoder and decoder:

$$\forall\, \mathcal{K}\ k\ l\ l', \mathbb{E}_{\mathcal{K}}(k) = \lfloor l \rfloor \Longrightarrow \mathbb{D}_{\mathcal{K}}(l{+\!+}l') = \lfloor (k, l') \rfloor.$$
$$\forall\, \mathcal{K}\ k\ l\ l', \mathbb{D}_{\mathcal{K}}(l{+\!+}l') = \lfloor (k, l') \rfloor \Longrightarrow \mathbb{E}_{\mathcal{K}}(k) = \lfloor l \rfloor.$$

  Their Coq proofs are automatically generated by inspecting the logical structure of classes and patterns in $\mathcal{S}$. For this, we need to derive a very important property: the decoder always decodes a bit string l back to the same sequence of components. We achieve this goal by combining proofs in Coq with SMT solving of verification conditions that are automatically derived from well-formed specifications.
- The proof of soundness of the encoder and decoder. The soundness theorems are stated as follows:

$$\forall\, \mathcal{K}\ k\ l\ l', \mathbb{E}_{\mathcal{K}}(k) = \lfloor l \rfloor \Longrightarrow \mathbb{R}[\![\mathcal{K}]\!]\ k\ l.$$
$$\forall\, \mathcal{K}\ k\ l\ l', \mathbb{D}_{\mathcal{K}}(l{+\!+}l') = \lfloor (k, l') \rfloor \Longrightarrow \mathbb{R}[\![\mathcal{K}]\!]\ k\ l.$$

  As we shall see later, $\mathbb{E}_{\mathcal{K}}$ and $\mathbb{R}[\![\mathcal{K}]\!]$ are both defined recursively on the definition of classes in $\mathcal{S}$. Their main difference is that the former is a function while the latter is a relation. Therefore, it is easy to prove the first soundness theorem by induction on k. By using the second consistency theorem and the first soundness theorem, we can easily prove the second soundness theorem.

As we shall see in the following sections, the actual implementations of encoders and decoders and their consistency and soundness theorems are more complicated than presented here. Nevertheless, the above discussion covers the high-level ideas of our framework.

Note that in Fig. 3, $\mathcal{S}$ and $\mathcal{G}$ are not formalized and hence not in the trusted base. The consistency and soundness of $\mathbb{E}$ and $\mathbb{D}$ are independently *validated* by using Coq and SMT solvers. If the validation of either property fails, the framework reports a failed attempt to generate the encoder and decoder. This often indicates that the instruction specification is not well-formed.

4 The Specification Language

The key idea underlying the design of CSLED is to record explicitly the structures of components in instruction specifications, instead of normalizing them into tokens as did in SLED. In this way, CSLED specifications accurately capture the key characteristics of instruction formats described in Sect. 2.1, hence the bijections embedded in instruction formats. In this section, we present the syntax of CSLED, explain the ideas underlying its design, and use the running example to illustrate how CSLED specifications are written. We also introduce the syntactical and relational interpretations of CSLED specifications and present the well-formedness conditions for the bijections to exist.

4.1 The Syntax

$$
\begin{aligned}
\mathcal{S} &::= \langle\texttt{empty}\rangle \\
&\quad\mid \mathcal{S}\ \mathcal{D} \\[1em]
\mathcal{D} &::= \texttt{token}\ \mathit{tid} = \mathcal{T}; \\
&\quad\mid \texttt{field}\ \mathit{fid} = \mathcal{F}; \\
&\quad\mid \texttt{class}\ \mathit{kid} = \mathcal{K}; \\[1em]
\mathcal{T} &::= \texttt{(}n\texttt{)} \\
\mathcal{F} &::= \mathit{tid}\texttt{(}n_1 : n_2\texttt{)} \\
\mathcal{K} &::= \mathcal{B} \\
&\quad\mid \mathcal{K}\ \texttt{|}\ \mathcal{B} \\
\mathcal{B} &::= \texttt{constr}\ \mathit{cid}\ [\mathit{aid}]\ (\mathcal{P})
\end{aligned}
$$

(a) Definitions

$$
\begin{aligned}
\mathcal{P} &::= \mathcal{J} \\
&\quad\mid \mathcal{P}\,\texttt{;}\,\mathcal{J} \\
\mathcal{J} &::= \mathcal{A} \\
&\quad\mid \mathcal{J}\,\texttt{\&}\,\mathcal{A} \\
\mathcal{A} &::= \mathcal{O} \\
&\quad\mid \texttt{cls}\ \%i \\
\mathcal{O} &::= \epsilon\,\texttt{:}\,\mathit{tid} \\
&\quad\mid \mathit{fid} = n \\
&\quad\mid \mathit{fid} \neq n \\
&\quad\mid \texttt{fld}\ \%i \\
&\quad\mid \mathcal{O}\ \texttt{\&}\ \mathcal{O} \\
&\quad\mid \mathcal{O}\ \texttt{;}\ \mathcal{O}
\end{aligned}
$$

(b) Patterns

Fig. 4. The syntax of CSLED

The syntax of CSLED is shown in Fig. 4. A CSLED specification (denoted by $\mathcal{S}$) consists of a list of *definitions* (denoted by $\mathcal{D}$). The three kinds of definitions are for tokens (denoted by $\mathcal{T}$), fields (denoted by $\mathcal{F}$) and classes (denoted by $\mathcal{K}$). Every definition is bound to a unique identifier where *tid*, *fid* and *kid* represents the identifiers of tokens, fields and classes, respectively.

Tokens represent consecutive segments of bytes and are the basic elements for forming instructions. They are necessary for distinguishing the same sequence of bytes with different interpretations. Their definitions have the form (n) where n must be divisible by 8 which denotes a token of n-bits or $n/8$ bytes. Definitions of fields have the form $\mathit{tid}\texttt{(}n_1 : n_2\texttt{)}$ which denotes a field occupying the n_2-th to n_1-th bits in the token *tid*.

Classes represent specific types of components. They play a central role in the specifications by accurately capturing the component-based abstraction we discussed in Sect. 2.1. A class consists of a collection of *branches* (denoted by $\mathcal{B}$) each of which denotes a possible form of components in the class. Definitions of branches have the form `constr` *cid* $[aid]$ $(\mathcal{P})$ where *cid* is a unique identifier for the branch (denoting a constructor) and $[aid]$ is a list of *fid* or *kid* denoting the sub-components or fields for constructing a component (i.e., the arguments to the constructor). These arguments capture the nested structures of components where a bigger component may be constructed from smaller ones or basic fields.

A branch is associated with a single *pattern* $\mathcal{P}$. A pattern plays two roles: it determines the types of a sequence of tokens that concretely forms components of this branch, and it describes a relation between these tokens (and their fields) with the abstract arguments of the branch. This relation essentially encodes the bijection between the abstract and binary forms of components in this branch.

At the top-most level, $\mathcal{P}$ is a sequence of *judgments* (denoted by $\mathcal{J}$) separated by `;`, such that $\mathcal{J}_1\,\texttt{;}\,\ldots\,\texttt{;}\,\mathcal{J}_n$ matches a sequence of tokens concretely represented by a bit string l if and only if $l = l_1{+}{+}l_2{+}{+}\ldots{+}{+}l_n$ and $\mathcal{J}_i$ matches l_i for $1 \leq i \leq n$. This sequential pattern is enough for relating abstract and binary forms of components when each $\mathcal{J}_i$ (and l_i) corresponds to a single (sub-)component. However, according to the discussion in Sect. 2.1, components may be interleaved with each other and $\mathcal{J}_i$ may correspond to multiple components. Therefore, a judgment is a conjunction of *atomic patterns* (denoted by $\mathcal{A}$) each of which matches an interleaved component. In case there is no interleaving, a judgment reduces to a single atomic pattern.

An atomic pattern has two forms: `cls` $\%i$ for relating a sequence of tokens to the i-th argument in $[aid]$ of the corresponding branch which must be a class, and $\mathcal{O}$ for relating tokens to field arguments in $[aid]$ and for further constraining the fields of these tokens. The $\mathcal{O}$ patterns are called *basic patterns*. Among them $\epsilon\texttt{:}\,tid$ matches any token of type *tid*; $fid = n$ ($fid \neq n$) matches a token with the field *fid* whose value is (is not) the constant n; similar to `cls` $\%i$, `fld` $\%i$ relates the i-th argument in $[aid]$ of the branch which must be a field to the concrete value of the field in the matching token. The last two cases of basic patterns indicate that arbitrary sequencing and interleaving of basic patterns are allowed. Despite such free interleaving, a basic pattern can only match with sequences of tokens of the same length and of a unique type because we require that $\mathcal{O}_1$ `&` $\mathcal{O}_2$ be well-formed only if both $\mathcal{O}_1$ and $\mathcal{O}_2$ match sequences of tokens with the same type. Therefore, basic patterns have the same expressiveness as SLED specifications in their normalized forms [15].

In contrast to basic patterns, judgments and atomic patterns are much more expressive as they may match tokens of different lengths and forms. This is because a class pattern `cls` $\%i$ can match components of a class $\mathcal{K}$ with multiple branches, each of which may have different patterns. By introducing class patterns into atomic patterns, we are able to represent the complete structures of components and establish bijections from these structures. This is the key improvement we made in CSLED compared to SLED.

4.2 The CSLED Specification of the Running Example

```
token Opcode = (8);      token Disp = (32);      token Imms = (32);
token ModRM = (8);       token SIB = (8);

field opcode = Opcode(7 : 0);       field disp = Disp(31 : 0);
field imms = Imms(31 : 0);          field mod = ModRM(7 : 6);
field reg_op = ModRM(5 : 3);        field rm = ModRM(2 : 0);
field scale = SIB(7 : 6);           field index = SIB(5 : 3);
field base = SIB(2 : 0);

class Addrmode  =
| constr addr_r [rm]  (mod = 0b11 & fld %1)
| constr addr_ir [rm]  (mod = 0b00 & rm ≠ 0b100 & rm ≠ 0b101 & fld %1)
| constr addr_disp [disp]  (mod = 0b00 & rm = 0b101; fld %1)
| constr addr_sib [scale, index, base]
   (mod = 0b00 & rm = 0b100;
   fld %1 & fld %2 & fld %3 & index ≠ 0b100 & base ≠ 0b101)
...

class Instruction  =
| constr AddGvEv [reg_op, Addrmode]  (opcode = 0x03; fld %1 & cls %2)
| constr AddEvIz [Addrmode, imms]
   (opcode = 0x81; reg_op = 0b000 & cls %1; fld %2)
...
```

Fig. 5. The CSLED specification of the running example

The CSLED specification of our running example is depicted in Fig. 5. The *Addrmode* class specifies the possible addressing modes. Its branches are translated from the addressing modes described in Table 1 one by one, such that their patterns exactly match the binary structures of components in the corresponding branches. For instance, the branch *addr_sib* is translated from the fourth addressing mode in Table 1. Its pattern is a sequence of two judgment. The first judgment is a conjunction of two basic patterns that are the required constraints on the fields *mod* and *rm* of *ModRM* described in Table 1. Therefore, it must match the single token *ModRM*. The second judgment is a conjunction of basic patterns that constrain the fields *index* and *base* of *SIB* and relate arguments of *addr_sib* with the concrete values in the fields *scale*, *index* and *base*. Because these patterns all constrain the fields of *SIB*, the second judgment must match the single token *SIB*.

Similarly, the *Instruction* class specifies the instructions. Its two branches characterize the two kinds of `add` instructions described in Sect. 2.2. Note how conjunctions between the basic patterns for *reg_op* and class patterns for *Addrmode* are used to describe the interleaving of register operands and addressing modes. Note also that in every branch of *Addrmode* the first pattern matches the token *ModRM*, and in any branch of *Instruction* the token *Opcode* is always followed by *Addrmode*. Therefore, *ModRM* always follows *Opcode* as desired.

By this example, we demonstrate the critical feature of CSLED: because the syntax of CSLED is designed to precisely describe instruction formats in ISA manuals, it implicitly captures the embedded bijections. Note that, because of its faithfulness to the ISA manuals, CSLED's syntax contains full details about instruction encoding by nature. However, it is not hard to imagine this syntax being refined to the client's syntax through another straightforward bijection. In fact, this is how we anticipate clients will use CSLED in practice, e.g., to build verified assemblers for X86.

4.3 Interpretation of CSLED Specifications

From a CSLED specification $\mathcal{S}$, we extract *1)* a collection of data types for representing the abstract syntax of components, and *2)* a collection of binary relations between these data types and bit strings for representing the mappings between the abstract and concrete forms of components.

Data Types of Components. We use the operator $\mathbb{T}[\![-]\!]$ to denote the interpretation of basic fields and classes into data types. The translation for fields are simple: given a field definition `field` $\mathit{fid} = \mathit{tid}\texttt{(}n_1 : n_2\texttt{)}$, $\mathbb{T}[\![\mathit{fid}]\!] = \langle n_1 - n_2 + 1\rangle$ where $\langle n\rangle$ represent an unsigned binary integer of n bits. Note that we do not further translate the values of fields as they have straightforward interpretations (such as the mapping from bits to registers described in Sect. 2.1). The interpretation of classes is only slightly more involved. Given a class definition `class` $\mathit{kid} = \mathcal{K}$, $\mathbb{T}[\![\mathit{kid}]\!]$ is an algebraic data type named *kid*. For each branch `constr` $\mathit{cid}\ [\mathit{aid}_1, \ldots, \mathit{aid}_n]\ \mathcal{P}$ of $\mathcal{K}$, there is a constructor *cid* for *kid* that takes n arguments of types $\mathbb{T}[\![\mathit{aid}_1]\!], \ldots, \mathbb{T}[\![\mathit{aid}_n]\!]$.

Relations Derived from CSLED. The translation of CSLED specifications into relations is defined in Fig. 6. Here, *BS* denotes the type of bit strings. When $\mathit{aids} = [\mathit{aid}_1, \ldots, \mathit{aid}_n]$ we write $\mathbb{T}[\![\mathit{aids}]\!]$ to denote the product type of $\mathbb{T}[\![\mathit{aid}_1]\!], \ldots, \mathbb{T}[\![\mathit{aid}_n]\!]$. We use $\equiv$ to denote the definitional equality.

The function $\mathbb{R}[\![\mathit{aid}]\!]$ translates a type of components associated with *aid* into a binary relation between its abstract representation and bit strings, where *aid* may denote a field or a class. The definition for field components is straightforward. $\mathbb{R}[\![\mathit{kid}]\!]\ k\ l$ holds iff there is a branch of *kid* whose interpretation relates k and l, which further requires (by the third rule in Fig. 6) that k is constructed by using the constructor of that branch and the pattern of the branch relates the arguments of the constructor to l. The latter relation is defined by $\mathbb{R}_p[\![-, -]\!]$ such that $\mathbb{R}_p[\![\mathcal{P}, \mathit{aids}]\!]\ \mathit{args}\ l$ holds iff $\mathcal{P}$ matches l and the arguments *args* satisfy the constraints enforced by $\mathcal{P}$ and *aids*. More specifically, $\mathbb{R}_p[\![\mathcal{P}\,\texttt{;}\,\mathcal{J}, \mathit{aids}]\!]\ \mathit{args}\ l$ holds iff $\mathcal{P}$ matches a prefix of l and $\mathcal{J}$ matches the rest of l. The definition of $\mathbb{R}_p[\![\mathcal{J}\,\texttt{\&}\,\mathcal{A}]\!]$ is slightly different in that $\mathbb{R}_p[\![\mathcal{J}\,\texttt{\&}\,\mathcal{A}, \mathit{aids}]\!]\ \mathit{args}\ l$ holds iff $\mathcal{A}$ matches the whole l and $\mathcal{J}$ matches a prefix of l. This is necessary for describing the

$$
\begin{aligned}
\mathbb{R}[\![\mathit{fid}]\!] ::=& \lambda(f : \mathbb{T}[\![\mathit{fid}]\!])\ (l : \mathit{BS}). \\
& \exists(\mathit{tid}\ n_1\ n_2\ n_3), \mathit{tid} \equiv \texttt{(}n_3\texttt{)} \wedge \mathit{fid} \equiv \mathit{tid}\texttt{(}n_1 : n_2\texttt{)} \\
& \wedge \mathit{length}(l) = n_3 \wedge l[n_1 : n_2] = f \\
\mathbb{R}[\![\mathit{kid}]\!] ::=& \lambda(k : \mathbb{T}[\![\mathit{kid}]\!])\ (l : \mathit{BS}). \\
& \exists \mathcal{B}, \mathit{kid} \equiv \ldots \texttt{|}\mathcal{B}\texttt{|} \ldots \wedge \mathbb{R}_b[\![\mathcal{B}, \mathit{kid}]\!]\ k\ l \\
\mathbb{R}_b[\![\mathcal{B}, \mathit{kid}]\!] ::=& \lambda(k : \mathbb{T}[\![\mathit{kid}]\!])\ (l : \mathit{BS}). \\
& \exists \mathit{args}, k = \mathit{cid}\ \mathit{args} \wedge \mathbb{R}_p[\![\mathcal{P}, \mathit{aids}]\!]\ \mathit{args}\ l \\
& (\text{where } \mathcal{B} = \texttt{constr}\ \mathit{cid}\ \mathit{aids}\ \mathcal{P}) \\
\mathbb{R}_p[\![\mathcal{P}\,\texttt{;}\,\mathcal{J}, \mathit{aids}]\!] ::=& \lambda(\mathit{args} : \mathbb{T}[\![\mathit{aids}]\!])\ (l : \mathit{BS}).\exists l_1\ l_2, l = l_1 +\!\!+ l_2 \\
& \wedge \mathbb{R}_p[\![\mathcal{P}, \mathit{aids}]\!]\ \mathit{args}\ l_1 \wedge \mathbb{R}_p[\![\mathcal{J}, \mathit{aids}]\!]\ \mathit{args}\ l_2 \\
\mathbb{R}_p[\![\mathcal{J}\texttt{\&}\mathcal{A}, \mathit{aids}]\!] ::=& \lambda(\mathit{args} : \mathbb{T}[\![\mathit{aids}]\!])\ (l : \mathit{BS}).\exists l_1\ l_2, l = l_1 +\!\!+ l_2 \\
& \wedge \mathbb{R}_p[\![\mathcal{J}, \mathit{aids}]\!]\ \mathit{args}\ l_1 \wedge \mathbb{R}_p[\![\mathcal{A}, \mathit{aids}]\!]\ \mathit{args}\ l \\
\mathbb{R}_p[\![\epsilon\,\texttt{:}\,\mathit{tid}]\!] ::=& \lambda(\mathit{args} : \mathbb{T}[\![\mathit{aids}]\!])\ (l : \mathit{BS}).\exists n, \mathit{tid} \equiv \texttt{(}n\texttt{)} \wedge \mathit{length}(l) = n \\
\mathbb{R}_p[\![\mathit{fid} = n, \mathit{aids}]\!] ::=& \lambda(\mathit{args} : \mathbb{T}[\![\mathit{aids}]\!])\ (l : \mathit{BS})\ .\exists(\mathit{tid}\ \mathit{fid}\ n_1\ n_2\ n_3), \mathit{tid} \equiv \texttt{(}n_3\texttt{)} \\
& \wedge \mathit{fid} \equiv \mathit{tid}\texttt{(}n_1 : n_2\texttt{)} \wedge \mathit{length}(l) = n_3 \wedge l[n_1 : n_2] = n \\
\mathbb{R}_p[\![\mathit{fid} \neq n, \mathit{aids}]\!] ::=& \lambda(\mathit{args} : \mathbb{T}[\![\mathit{aids}]\!])\ (l : \mathit{BS})\ .\exists(\mathit{tid}\ \mathit{fid}\ n_1\ n_2\ n_3), \mathit{tid} \equiv \texttt{(}n_3\texttt{)} \\
& \wedge \mathit{fid} \equiv \mathit{tid}\texttt{(}n_1 : n_2\texttt{)} \wedge \mathit{length}(l) = n_3 \wedge l[n_1 : n_2] \neq n \\
\mathbb{R}_p[\![\texttt{fld}\ \%i, \mathit{aids}]\!] ::=& \lambda(\mathit{args} : \mathbb{T}[\![\mathit{aids}]\!])\ (l : \mathit{BS}).\mathbb{R}[\![\mathit{aids}[i]]\!]\ \mathit{args}[i]\ l \\
\mathbb{R}_p[\![\texttt{cls}\ \%i, \mathit{aids}]\!] ::=& \lambda(\mathit{args} : \mathbb{T}[\![\mathit{aids}]\!])\ (l : \mathit{BS}).\mathbb{R}[\![\mathit{aids}[i]]\!]\ \mathit{args}[i]\ l
\end{aligned}
$$

Fig. 6. Translation of CSLED specifications into relations

interleaving of components. Furthermore, certain constraints need to be satisfied for deriving a bijection as shall discuss in Sect. 4.4. $\mathbb{R}_p[\![\mathcal{O}_1\,\texttt{;}\,\mathcal{O}_2, \mathit{aids}]\!]$ and $\mathbb{R}_p[\![\mathcal{O}_1\texttt{\&}\mathcal{O}_2, \mathit{aids}]\!]$ are not shown in Fig. 6 because they are defined the same as $\mathbb{R}_p[\![\mathcal{P}\,\texttt{;}\,\mathcal{J}, \mathit{aids}]\!]$ and $\mathbb{R}_p[\![\mathcal{J}\texttt{\&}\mathcal{A}, \mathit{aids}]\!]$, respectively. $\mathbb{R}_p[\![\mathit{fid} = n, \mathit{aids}]\!]\ \mathit{args}\ l$ holds iff l is a token containing *fid* whose value is n; similar for $\mathbb{R}_p[\![\mathit{fid} \neq n, \mathit{aids}]\!]$. $\mathbb{R}_p[\![\texttt{fld}\ \%i, \mathit{aids}]\!]$ holds iff the i-th argument in *args* matches with the concrete value found in l; same for $\mathbb{R}_p[\![\texttt{cls}\ \%i, \mathit{aids}]\!]$. Note how the last two definitions make use of *args* for getting the values of arguments.

4.4 Well-Formedness of Specifications

The binary relation we define in the last section denotes a bijection only when the CSLED specification under investigation satisfies certain well-formedness conditions. These conditions guarantee that, given any bit string l, there is at most one abstract object related to l via the defined binary relation. Well-formedness is the composition of three properties which we call *disjointness*, *compatibility*, and *uniqueness*. We give and explain their definitions below. The logic for checking these conditions is embedded in the generation algorithms we will discuss in the next section and will be exploited for the validation of the generated encoders and decoders.

Disjointness. Given a pattern $\mathcal{P}_1 \& \mathcal{P}_2$, it satisfies disjointness if $\mathcal{P}_1$ and $\mathcal{P}_2$ match disjoint fields.[1] To understand this, suppose $\mathcal{P}_1$ and $\mathcal{P}_2$ relate different abstract arguments a_1 and a_2 to overlapping bits in a bit string l. Then, we cannot determine if the values in the overlapping bits are for a_1 or a_2. Hence, the derived binary relation cannot possibly be a bijection. Disjointness rules out such possibility.

Compatibility. We call the types of sequences of tokens a pattern $\mathcal{P}$ matches the "shapes" of $\mathcal{P}$. Given a pattern $\mathcal{P}_1 \& \mathcal{P}_2$, it satisfies compatibility if every possible shape of $\mathcal{P}_1$ is in a prefix of every possible shape of $\mathcal{P}_2$ when $\mathcal{P}_2$ is a class pattern (and vice versa). Enforcing compatibility simplifies the interpretation of $\mathcal{P}_1 \& \mathcal{P}_2$ when $\mathcal{P}_1$ or $\mathcal{P}_2$ is a class pattern with multiple branches that may match bit strings with different shapes. Compatibility makes sense because for common instruction formats it is always the case that the components matched by $\mathcal{P}_1$ are embedded in the *longest common prefixes* of all the possible shapes of $\mathcal{P}_2$ when $\mathcal{P}_2$ is a class pattern (and vice versa). For example, in the example depicted in Fig. 2, **Reg_op** is always embedded into the common prefix of all the possible shapes of addressing modes, i.e., the **ModRM** token.

Uniqueness. Given a class pattern $\mathcal{K}$, it satisfies uniqueness if for any bit string l, at most one of its branches matches l. Uniqueness is essential for ensuring the determinacy of decoders in presences of class patterns. Fortunately, it implicitly holds for common instruction formats as they are designed with determinacy of decoding in mind. To concretely check the uniqueness implied by instruction formats, we first define the *structural condition* for a branch with pattern $\mathcal{P}$ as the conjunction of the statically known constraints in $\mathcal{P}$, denoted by $[\![\mathcal{P}]\!]_{cond}$. We then require that no structure conditions for any two branches of a class can be satisfied simultaneously. This requirement allows us to uniquely determine the branch used to construct a class component. For example, the structural conditions of the first three branches of *Addrmode* are ($mod = \texttt{0b11}$), ($mod = \texttt{0b00}\ \&\ rm \neq \texttt{0b100}\ \&\ rm \neq \texttt{0b101}$) and ($mod = \texttt{0b00}\ \&\ rm = \texttt{0b101}$). Obviously, any pairwise combination of these conditions cannot possibly be satisfied. This is true even if we consider all the branches of *Addrmode*. Therefore, there is at most one way to decode any addressing mode.

5 Generation of Encoders and Decoders

We discuss the algorithm for generating encoders and decoders from CSLED specifications. The structures of these encoders and decoders closely match the relations derived from specifications. Furthermore, every operation in an encoder has a counterpart in the corresponding decoder, and vice versa.

[1] We abuse the notation by using $\mathcal{P}$ to denote suitable patterns such as $\mathcal{J}$, $\mathcal{A}$ or $\mathcal{O}$.

5.1 Generation of Encoders

$$\begin{aligned}
\mathcal{G}_{\mathbb{E}}[\![\epsilon : tid, bs, args]\!] &::= \lfloor bs \rfloor \\
\mathcal{G}_{\mathbb{E}}[\![\mathit{fid} = n, bs, args]\!] &::= write_{\mathit{fid}}\ bs\ n \\
\mathcal{G}_{\mathbb{E}}[\![\mathit{fid} \neq n, bs, args]\!] &::= assert(read_{\mathit{fid}}\ bs \neq n) \\
\mathcal{G}_{\mathbb{E}}[\![\texttt{fld}\ \%i, bs, args]\!] &::= write_{\mathit{fid}}\ bs\ args[i] \qquad \text{(where } \mathit{fid} \text{ is the field id of } args[i]\text{)} \\
\mathcal{G}_{\mathbb{E}}[\![\texttt{cls}\ \%i, bs, args]\!] &::= \mathbb{E}_{\mathcal{K}}(args[i], bs) \qquad \text{(where } \mathcal{K} \text{ is the class of } args[i]\text{)} \\
\mathcal{G}_{\mathbb{E}}[\![\mathcal{O}_1\ \texttt{;}\ \mathcal{O}_2, bs, args]\!] &::= l_1 \leftarrow \mathit{first\_n}(bs, [\![\mathcal{O}_1]\!]_{tokens}); l_2 \leftarrow \mathit{skip\_n}(bs, [\![\mathcal{O}_1]\!]_{tokens}) \\
&\ bs_1 \leftarrow \mathcal{G}_{\mathbb{E}}[\![\mathcal{O}_1, l_1, args]\!]; bs_2 \leftarrow \mathcal{G}_{\mathbb{E}}[\![\mathcal{O}_2, l_2, args]\!]; \lfloor bs_1 +\!\!+ bs_2 \rfloor \\
\mathcal{G}_{\mathbb{E}}[\![\mathcal{O}_1\ \texttt{\&}\ \mathcal{O}_2, bs, args]\!] &::= bs_1 \leftarrow \mathcal{G}_{\mathbb{E}}[\![\mathcal{O}_1, bs, args]\!]; \mathcal{G}_{\mathbb{E}}[\![\mathcal{O}_2, bs_1, args]\!] \\
\mathcal{G}_{\mathbb{E}}[\![\mathcal{P}\,\texttt{;}\,\mathcal{J}, bs, args]\!] &::= bs_1 \leftarrow \mathcal{G}_{\mathbb{E}}[\![\mathcal{P}, bs, args]\!]; bs' \leftarrow \mathit{skip\_n}(bs, |bs_1|); \\
&\ bs_2 \leftarrow \mathcal{G}_{\mathbb{E}}[\![\mathcal{J}, bs', args]\!]; \lfloor bs_1 +\!\!+ bs_2 \rfloor \\
\mathcal{G}_{\mathbb{E}}[\![\mathcal{J}\texttt{\&}\mathcal{A}, bs, args]\!] &::= bs_1 \leftarrow \mathcal{G}_{\mathbb{E}}[\![\mathcal{J}, bs, args]\!]; \mathcal{G}_{\mathbb{E}}[\![\mathcal{A}, bs_1, args]\!]
\end{aligned}$$

Fig. 7. Generation of encoders from patterns

From every class $\mathcal{K}$, we extract an encoder $\mathbb{E}_{\mathcal{K}}$ for its components. It is a partial function that takes two arguments—a component k and a bit string l representing the result previously generated by encoders—and outputs an updated bit string if the encoding succeeds. We shall write $\mathbb{E}_{\mathcal{K}}(k, l) = \lfloor l' \rfloor$ to denote that l' is the result of encoding k on top of l.

$\mathbb{E}_{\mathcal{K}}(k, l)$ is defined by recursion on the structure of k. For every branch $\mathcal{B}$ of $\mathcal{K}$, we generate a piece of Coq code from the pattern $\mathcal{P}$ of $\mathcal{B}$ for encoding k. We then insert it into the definition of $\mathbb{E}_{\mathcal{K}}(k, l)$. We write $\mathcal{G}_{\mathbb{E}}[\![\mathcal{P}, bs, args]\!]$ to denote the code snippet so generated, where bs is the name of the generated bit string at this point and $args$ contains the names of the arguments to the constructor. $\mathcal{G}_{\mathbb{E}}[\![\mathcal{P}, bs, args]\!]$ is defined in Fig. 7 where we use the option monad for sequencing the encoding operations. The first case is obvious. Code generated by $\mathcal{G}_{\mathbb{E}}[\![\mathit{fid} = n, bs, args]\!]$ writes the constant n into the field associated with fid. $\mathcal{G}_{\mathbb{E}}[\![\mathit{fid} \neq n, bs, args]\!]$ checks whether the corresponding field contains the constant n and returns none if the checking fails. $\mathcal{G}_{\mathbb{E}}[\![\texttt{fld}\ \%i, bs, args]\!]$ writes the value of the i-th argument into the corresponding field. $\mathcal{G}_{\mathbb{E}}[\![\texttt{cls}\ \%i, bs, args]\!]$ calls the encoder for the class corresponding to $\texttt{cls}\ \%i$. $\mathcal{G}_{\mathbb{E}}[\![\mathcal{O}_1\ \texttt{;}\ \mathcal{O}_2, bs, args]\!]$ encodes its two parts recursively and concatenates the results together, where $\mathit{first\_n}(bs, n)$ returns the first n bits in bs and $\mathit{skip\_n}(bs, n)$ skips the first n bits in bs and returns the remaining ones. $\mathcal{G}_{\mathbb{E}}[\![\mathcal{O}_1\texttt{\&}\mathcal{O}_2, bs, args]\!]$ first encodes data matching $\mathcal{O}_1$, and then passes the result to the encoding for $\mathcal{O}_2$. The last two cases are similar. Note that if the generated code occurs at the beginning of a branch, then bs coincides with the input argument l. Otherwise, bs denotes intermediate results. As we can see, all these cases follow the logical structure of CLSED specifications we have described before.

5.2 Generation of Decoders

From every class $\mathcal{K}$, we extract a decoder $\mathbb{D}_{\mathcal{K}}$. It is a partial function such that $\mathbb{D}_{\mathcal{K}}(l) = \lfloor(k, l_1, l_2)\rfloor$ holds iff $l = l' {+\!+} l_2$, l' is the binary representation of k, and l_1 is the result of inverting the encoding operation, i.e., setting every bit the decoder touches in l' to 0. This extra return value is introduced to help with the verification as we shall see in Sect. 6.

$$
\begin{aligned}
\mathcal{G}_{\mathbb{D}}\llbracket \epsilon : tid, bs, args \rrbracket &::= remains \leftarrow skip\_n(bs, tid);\ \lfloor(bs,\ remains)\rfloor \\
\mathcal{G}_{\mathbb{D}}\llbracket fid = n, bs, args \rrbracket &::= ori \leftarrow clear_{fid}\ bs; remains \leftarrow skip\_n(bs, tid); \\
&\ \lfloor(ori,\ remains)\rfloor \qquad (\text{where } fid \equiv tid\texttt{(}n_1 : n_2\texttt{)}) \\
\mathcal{G}_{\mathbb{D}}\llbracket fid \neq n, bs, args \rrbracket &::= ori \leftarrow clear_{fid}\ bs; remains \leftarrow skip\_n(bs, tid); \\
&\ \lfloor(ori,\ remains)\rfloor \qquad (\text{where } fid \equiv tid\texttt{(}n_1 : n_2\texttt{)}) \\
\mathcal{G}_{\mathbb{D}}\llbracket \texttt{fld}\ \%i, bs, args \rrbracket &::= argi \leftarrow read_{fid}\ bs; ori \leftarrow clear_{fid}\ bs; \\
&\ remains \leftarrow skip\_n(bs, tid); \lfloor(ori,\ remains)\rfloor \\
&\ (\text{where } fid \text{ is the field id of } args[i]) \\
\mathcal{G}_{\mathbb{D}}\llbracket \texttt{cls}\ \%i, bs, args \rrbracket &::= argi, origin, remains \leftarrow \mathbb{D}_{\mathcal{K}}(bs); \lfloor(origin,\ remains)\rfloor \\
&\ (\text{where } \mathcal{K} \text{ is the class of } args[i]) \\
\mathcal{G}_{\mathbb{D}}\llbracket \mathcal{O}_1\ \texttt{;}\ \mathcal{O}_2, bs, args \rrbracket &::= ori_1, remains_1 \leftarrow \mathcal{G}_{\mathbb{D}}\llbracket \mathcal{O}_1, bs, args \rrbracket; \\
&\ ori_2, remains_2 \leftarrow \mathcal{G}_{\mathbb{D}}\llbracket \mathcal{O}_2, remains_1, args \rrbracket; \\
&\ \lfloor(ori_1 {+\!+} ori_2,\ remains_2)\rfloor \\
\mathcal{G}_{\mathbb{D}}\llbracket \mathcal{O}_1\ \texttt{\&}\ \mathcal{O}_2, bs, args \rrbracket &::= remains \leftarrow skip\_n(bs, \llbracket \mathcal{O}_2 \rrbracket_{tokens}); \\
&\ ori, \_ \leftarrow \mathcal{G}_{\mathbb{D}}\llbracket \mathcal{O}_2, bs, args \rrbracket; \\
&\ orilst, \_ \leftarrow \mathcal{G}_{\mathbb{D}}\llbracket \mathcal{O}_1, ori, args \rrbracket; \\
&\ \lfloor(orilst,\ remains)\rfloor \\
\mathcal{G}_{\mathbb{D}}\llbracket \mathcal{P}\texttt{;}\mathcal{J}, bs, args \rrbracket &::= ori_1, remains_1 \leftarrow \mathcal{G}_{\mathbb{D}}\llbracket \mathcal{P}, bs, args \rrbracket; \\
&\ ori_2, remains_2 \leftarrow \mathcal{G}_{\mathbb{D}}\llbracket \mathcal{J}, remains_1, args \rrbracket; \\
&\ \lfloor(ori_1 {+\!+} ori_2, remains_2)\rfloor \\
\mathcal{G}_{\mathbb{D}}\llbracket \mathcal{J}\texttt{\&}\mathcal{A}, bs, args \rrbracket &::= ori, remains \leftarrow \mathcal{G}_{\mathbb{D}}\llbracket \mathcal{A}, bs, args \rrbracket; \\
&\ orilst, \_ \leftarrow \mathcal{G}_{\mathbb{D}}\llbracket \mathcal{J}, ori, args \rrbracket; \\
&\ \lfloor(orilst, remains)\rfloor
\end{aligned}
$$

Fig. 8. Generation of decoders from patterns

The first step of $\mathbb{D}_{\mathcal{K}}$ is to decide which branch of $\mathcal{K}$ should be chosen for decoding l. It can be done by checking the structural conditions derived from the patterns of branches (which we have introduced in Sect. 4.4) against l. Specifically, for the pattern $\mathcal{P}$ of each branch of $\mathcal{K}$, we translate its structural condition $\llbracket \mathcal{P} \rrbracket_{cond}$ into a decision procedure in Coq (a function returning boolean values) in a straightforward manner. We then insert an if-statement to check if $\llbracket \mathcal{P} \rrbracket_{cond}$ can be satisfied. If so, we start the decoding process for this branch. Otherwise, we repeatedly check other branches until a matching case is found. Note also that by uniqueness, there is at most one structural condition that can be satisfied. Therefore, $\mathbb{D}_{\mathcal{K}}$ is deterministic in choosing branches.

Once a matching branch is found, we use the algorithm $\mathcal{G}_{\mathbb{D}}[\![\mathcal{P}, bs, args]\!]$ (the counterpart of $\mathcal{G}_{\mathbb{E}}[\![\mathcal{P}, bs, args]\!]$) to generate a piece of Coq code for decoding the arguments of this branch. It is defined in Fig. 8. Similar to encoding, the generated code snippet follows the logical structure of CSLED specifications. The function $clear_{fid}\ bs$ set the bits of the field *fid* in *bs* to 0. Note that the decoding operations are exactly the inversion of those in Fig. 7. Note also that the fourth and fifth cases in Fig. 8 are responsible for decoding the arguments and storing them in *argi*. By applying the corresponding constructor to these arguments, we get the output component k, which together with the two values returned by $\mathcal{G}_{\mathbb{D}}$ form the final output of $\mathbb{D}_{\mathcal{K}}$.

5.3 Generation for the Running Example

We show the representative cases of the generated encoder and decoder for our running example in Fig. 9. They include the encoding and decoding procedures for the fourth branch of *Addrmode* (the most complicated one). We can see that the encoding and decoding operations are exactly the inverses of each other. The encoder first writes the fields in *ModRM* and then those in *SIB*. Conversely, the decoder first reads the fields in *ModRM* and then those in *SIB*. Finally, it forms the component and returns the reverted and remaining bits. The function `BF_addr_sib` is the decision procedure generated from the structural condition for the fourth branch of *Addrmode*. We also show the encoding and decoding procedures for the first `add` instruction in Fig. 9. Their structures are very similar to those of *Addrmode*.

6 Validation of Encoders and Decoders

In this section, we discuss how to exploit the logical structure of and the well-formedness conditions for CSLED specifications to automatically synthesize the proofs of consistency and soundness for encoders and decoders.

6.1 Synthesizing the Proof of Consistency

The consistency between encoders and decoders is composed of two properties and stated as follows:

Theorem 1 (Consistency between Encoders and Decoders). *Given any class $\mathcal{K}$, its encoder $\mathbb{E}_{\mathcal{K}}$ and decoder $\mathbb{D}_{\mathcal{K}}$ are consistent with each other if they invert each other. That is, the following properties hold:*

$$\forall\ k\ l\ r\ l', valid\_input_{\mathcal{K}}(l) \Longrightarrow \mathbb{E}_{\mathcal{K}}(k,l) = \lfloor r \rfloor \Longrightarrow \mathbb{D}_{\mathcal{K}}(r{+}{+}l') = \lfloor (k,l,l') \rfloor.$$
$$\forall\ k\ l\ r\ l', \mathbb{D}_{\mathcal{K}}(r{+}{+}l') = \lfloor (k,l,l') \rfloor \Longrightarrow \mathbb{E}_{\mathcal{K}}(k,l) = \lfloor r \rfloor.$$

```
Definition encode_addrmode instance input :=
  match instance with
  ...
  | addr_sib arg1 arg2 arg3 ⇒
    (* Encode ModRM *)
    let ModRM := input in
    let tmp := write_mod ModRM b["00"] in
    let tmp := write_rm tmp b["100"] in
    let result0 := tmp in
    (* Encode SIB *)
    let SIB := zeros 8 in
    let tmp := write_scale SIB arg1 in
    let tmp := write_index tmp arg2 in
    let tmp := write_base tmp arg3 in
    let index := read_index tmp in
    let base := read_base tmp in
    do _ ← assert(index ≠ b["100"]);
    do _ ← assert(base ≠ b["101"]);
    let result1 := tmp in
    (* Concatenate the results of
       encoding ModRM and SIB *)
    Some (result0++result1)
  | ...
  end.
```

```
Definition decode_addrmode bs :=
  ...
  if BF_addr_sib bs then
    (* Revert the encoding of ModRM *)
    let ori := clear_mod bs in
    let ori := clear_rm ori in
    let ori1 := ori in
    do remains ← skipn bs 8; (* Skip ModRM *)
    (* Decode SIB to get the arguments
       and revert the encoding of SIB *)
    let bs := remains in
    let arg3 := read_base bs in
    let ori := clear_base bs in
    let arg2 := read_index ori in
    let ori := clear_index ori in
    let arg1 := read_scale ori in
    let ori := clear_scale ori in
    let ori2 := ori in
    do remains ← skipn bs 8;  (* Skip SIB *)
    (* Return the result *)
    Some(addr_sib arg1 arg2 arg3,
         ori1++ori2, remains)
  else if BF_addr_r bs then ...
    ...
```

```
Definition encode_instr instance input :=
  match instance with
  |AddGvEv arg1 arg2 ⇒
    ...
    let tmp := write_reg_op ModRM arg1 in
    do tmp ← encode_addrmode arg2 tmp;
    ...
  | ...
  end.

Definition decode_instr bs :=
  if BF_AddGvEv bs then
    ...
    do arg2, ori, remains ←
       decode_addrmode bs;
    let arg1 := read_reg_op ori in
    let ori := clear_reg_op ori in
    ...
```

```
Definition BF_addr_sib bs :=
  let ModRM := firstn bs 8 in
  (* mod = 0b00 ∧ rm = 0b100 *)
  let result0 :=
    (ModRM & b["11000111"]) = b["00000100"] in
  let tmp := skipn bs 8 in
  let SIB := firstn tmp 8 in
  (* index ≠ 0b100 *)
  let result10 :=
    (SIB & b["00111000"]) ≠ b["00100000"] in
  (* base ≠ 0b101 *)
  let result11 :=
    (SIB & b["00000111"]) ≠ b["00000101"] in
  result0 ∧ result10 ∧ result11.

Definition BF_AddGvEv bs :=
  let Opcode := firstn bs 8 in
  (Opcode & b["11111111"]) = b["00000011"].
```

Fig. 9. Encoders and decoders generated from the running example

We first discuss how the proof for the first property in Theorem 1 is generated. Here, the assumption $valid\_input_{\mathcal{K}}(l)$ asserts that all the bits in l that may be modified by $\mathbb{E}_{\mathcal{K}}$ must be 0. This is necessary to ensure that the decoder can revert the resulting bit string back to its initial state by setting them to 0 (i.e., the second result of decoding is the same as l).

The proof proceeds by induction on the structure of k. For each branch $\mathcal{B}$ with the pattern $\mathcal{P}$, we generate a lemma and its proof that the decision procedure generated from $[\![\mathcal{P}]\!]_{cond}$ as described in Sect. 5.2 always returns true given any bit string generated by the encoder for $\mathcal{P}$. With this lemma, the proof for the "symmetric" case where the decoder takes the same branch as the encoder reduces to proving that the encoder and decoder generated from $\mathcal{P}$ are inverses of each other. This proof is straightforward by the definitions of $\mathcal{G}_{\mathbb{E}}$ and $\mathcal{G}_{\mathbb{D}}$

in Sect. 5. An important point to note is that, for any pattern `cls` $\%i$, we need to recursively apply the consistency lemma for its corresponding class, which in turn requires us to establish a *valid_input* assumption. By the disjointness property in Sect. 4.4, we can easily conclude that the encoding of sub-components does not interfere with each other, thereby the desired *valid_input* assumption can be derived.

To finish the proof, we need to show that the "asymmetric" cases are not possible. For each asymmetric branch $\mathcal{B}'$ with the pattern $\mathcal{P}'$, we have that $[\![\mathcal{P}']\!]_{cond}$ holds by the decision procedure guarding this branch. Furthermore, by the above reasoning, $[\![\mathcal{P}]\!]_{cond}$ holds. We hence have that the conjunction of $[\![\mathcal{P}]\!]_{cond}$ and $[\![\mathcal{P}']\!]_{cond}$ holds. However, this contradicts with the uniqueness property given in Sect. 4.4. Therefore, the decoder can never go into a branch different from the encoder. Continue with our running example, suppose we are proving the consistency of the encoder and decoder for *Addrmode*. Further suppose we are working on the branch with the constructor *addr_sib*. Then, the verification condition for the asymmetric case with the constructor *addr_r* is

$$\forall bs, (read_{mod}\ bs = \texttt{0b00} \wedge read_{rm}\ bs = \texttt{0b100}\ldots) \wedge (read_{mod}\ bs = \texttt{0b11})$$

which cannot possibly hold (for simplicity we omit the conditions for *index* and *base*). We note that such condition can be easily checked by any SMT solver with the theory of bit-vectors, and we use Z3 [5] to validate them. This checking can also be directly formalized in Coq, which we plan to do in the future.

Finally, the second property in Theorem 1 can be proved by induction on k in a similar fashion. We elide a discussion of its proof.

6.2 Synthesizing the Proof of Soundness

As we have discussed in Sect. 4.3, the relational specifications extracted from CSLED specifications are tightly related to the actual instruction formats. Thus, it is reasonable to check the soundness of the generated encoders and decoders against these specifications. The relational specifications are easily translated into Coq definitions and we shall use the same notations. The soundness of encoders and decoders is then stated as follows:

Theorem 2 (Soundness of Encoders and Decoders). *Given any class $\mathcal{K}$, its encoder $\mathbb{E}_{\mathcal{K}}$ is sound if the following property holds:*

$$\forall\ k\ l\ r\ l', \mathbb{E}_{\mathcal{K}}(k,l) = \lfloor r \rfloor \Longrightarrow \mathbb{R}[\![\mathcal{K}]\!]\ k\ r.$$

Similarly, its encoder $\mathbb{D}_{\mathcal{K}}$ is sound if the following holds:

$$\forall\ k\ l\ r\ l', \mathbb{D}_{\mathcal{K}}(r{+\!+}l') = \lfloor (k,l,l') \rfloor \Longrightarrow \mathbb{R}[\![\mathcal{K}]\!]\ k\ r.$$

The soundness of encoder is easily proved by induction on the structure of k. We need to exploit the well-formedness conditions of CSLED specifications as for the consistency proofs at relevant points. The soundness of decoder is a corollary of the soundness of encoder and the second consistency property.

7 Evaluation

Besides the CSLED language, our framework has two major parts: *1)* the algorithms for generating encoders, decoders and their proofs and *2)* a Coq library containing the definitions and properties of basic types (including bits, bytes and bit strings) and a collection of automation tactics (Ltac definitions) for proof synthesis. The generation algorithms amount to 5,193 lines of C++ code (excluding comments and empty lines, and likewise for the following statistics). The Coq library amounts to 1,036 lines of Coq code (written in Coq 8.11.0 and counted using `coqwc`). We also make use of the monad definitions and some basic data formats in CompCert's library [13]. The whole framework took six person months to develop.

Table 2. The lines of generated Coq code

Component	Lines of definitions	Lines of proofs
Relational specification	1762	0
AST, encoder and decoder	5677	0
Verification conditions	37011	4402
Consistency proof	295	30841
Soundness proof	60	7193
Total	44805	42436

To evaluate the effectiveness of our framework, we have written a CSLED specification for a total of 186 representative X86-32 instructions which cover the operands with the most complicated formats (e.g., addressing modes) and are sufficient for supporting the assembling process in CompCert's X86-32 backend. The specification is very succinct, containing only 260 lines of CSLED code. From this specification, our framework *automatically* generates around 87k lines of Coq code which form the verified encoder and decoder. The lines of Coq definitions and proofs for individual components are shown in Table 2. Note that the verification conditions account for a major part of the definitions because we need to consider all the possible combinations of structural conditions for the proofs of consistency and soundness. The Coq proofs related to verification conditions are for identifying the concrete forms of structural conditions. As expected, the consistency proof is the most complicated one among all the proofs.

To evaluate the performance of the generated encoder and decoder, we randomly generate four sets of instructions, encode them into bit strings, and decode the bit strings back. The executable encoder and decoder are obtained by extracting Coq definitions into OCaml programs and compiling with OCaml 4.08.0. We repeat this experiment for 30 times on a machine with Intel(R) i7-4980HQ CPU@2.8 GHz and 16 GB memory. For comparison, we conduct the same experiments on the hand-written encoder and decoder in the X86-32 back-end of CompCertELF [20]. The results are shown in Table 3. For each test case, it shows the

Table 3. Performance evaluation

No. of Instr.	CSLED				Hand-Written			
	Enc. Time (s)		Dec. Time (s)		Enc. Time (s)		Dec. Time (s)	
	Med	Var.(%)	Med	Var.(%)	Med	Var.(%)	Med	Var.(%)
6000	0.32	0.00	0.56	0.00	0.01	0.00	0.01	0.00
12000	0.64	0.00	1.12	0.00	0.01	0.00	0.02	0.00
18000	0.98	0.03	1.70	0.15	0.02	0.00	0.03	0.01
60000	3.11	0.16	5.43	0.01	0.08	0.00	0.09	0.01

numbers of randomly generated instructions and the median time (in seconds) and the variance (in percentage) for encoding and decoding. We observe that the automatically generated encoder and decoder perform reasonably well, but significantly slower than the hand-written ones. This is because 1) the hand-written encoder and decoder in CompCertELF currently supports significantly less instructions (about 20) than the CLSED ones due to the complexity in manual implementation, and 2) the hand-written ones are manually optimized while the auto-generated ones are not optimized at all. We plan to solve the above issues by optimizing our generation algorithms in the future.

8 Related Work and Conclusion

We compare our framework with existing work on specification languages of instruction sets, verified parsing and pretty printing, and formalized ISAs.

There exists a lot of work on developing languages for specifying ISAs. Their major deficiency is the lack of formal guarantees. For example, the nML specification language employs attribute grammars to describe instruction sets [7]. For another example, EEL uses machine independent primitives to provide syntactic and semantic information of instructions [12]. The most relevant work in this category is the SLED language which our CSLED is based upon [15]. The patterns in SLED can only describe constraints on tokens and fields. By contrast, CSLED contains class patterns for accurately characterizing the structures of components. This extension enables CSLED to capture the bijection between the abstract and concrete forms of instructions.

Instruction decoding and encoding are special cases of parsing and pretty printing, respectively. Although there was early work on verifying that parsing and pretty-printing are inverses of each other by formulating them as bijections [1,10], this requirement was perceived as too strong [16]. Most of the recent work on verified parsing and pretty printing are dedicated to verify parser generators based on context-free grammars, regular expressions, parser combinators, or general data formats [3,11,17]. Some of them are also specialized work on verifying the encoder-decoder pairs [6,14,19,21]. They mostly deal with general and ambiguous grammars or specifications where bijection is difficult (if not impossible) to establish. By contrast, we intentionally restrict the expressiveness of CSLED specifications to make proving consistency possible. Specifically, the syntax presented in Fig. 4 implies that CSLED specifications can only match sequences of tokens with finite lengths and shapes, making it strictly weaker than regular expressions, yet sufficiently strong for precisely capture the common instruction formats.

There is also abundant work on the development of formal ISA specifications (e.g., [2,4,8,9]). However, almost all of them focus on the problem of rigorously defining the *semantics* of ISAs (such as their sequential behaviors, concurrency models and interrupt behaviors). Although formalized encoders or decoders (or both) are sometimes generated (e.g., in Coq or Isabelle/HOL), there is no formal verification of the soundness or consistency of instruction encoding and decoding which only concerns the *syntax* of instructions.

In this paper, we have presented a framework for specifying instruction formats and for automatically generating and verifying encoders and decoders based on such specifications. The verified encoders and decoders are consistent with each other (being inverses of each other) and sound (conforming to high-level specifications). Consistency is provable in our framework because our specifications capture the bijections inherently embedded in instruction formats. In the future, we would like to apply this framework to a major part of X86-32 and X86-64 instructions and also to other ISAs, thereby to demonstrate the versatility and scalability of our framework.

Acknowledgments. We thank Zhong Shao for his comments and suggestions on this project when it was at an early stage. We are also indebted to anonymous reviewers for their detailed comments. This work was supported by the National Natural Science Foundation of China (NSFC) under Grant No. 62002217.

References

1. Alimarine, A., Smetsers, S., Weelden, A., Eekelen, M., Plasmeijer, R.: There and back again: arrows for invertible programming. In: Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell, pp. 86–97. ACM (2005). https://doi.org/10.1145/1088348.1088357
2. Armstrong, A., et al.: ISA semantics for ARMv8-a, RISC-v, and CHERI-MIPS. Proc. ACM Program. Lang. **3**(POPL), 71:1–71:31 (2019). https://doi.org/10.1145/3290384
3. Barthwal, A., Norrish, M.: Verified, executable parsing. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 160–174. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00590-9_12
4. Dasgupta, S., Park, D., Kasampalis, T., Adve, V.S., Roşu, G.: A complete formal semantics of x86–64 user-level instruction set architecture. In: PLDI 2019, pp. 1133–1148. ACM (2019). https://doi.org/10.1145/3314221.3314601
5. De Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24

6. Delaware, B., Suriyakarn, S., Pit-Claudel, C., Ye, Q., Chlipala, A.: Narcissus: correct-by-construction derivation of decoders and encoders from binary formats. Proc. ACM Program. Lang. **3**(ICFP), 82:1–82:29 (2019). https://doi.org/10.1145/3341686
7. Fauth, A., Van Praet, J., Freericks, M.: Describing instruction set processors using NML. In: Proceedings of the European Design and Test Conference, pp. 503–507. IEEE (1995). https://doi.org/10.1109/EDTC.1995.470354
8. Fox, A.: Directions in ISA specification. In: Beringer, L., Felty, A. (eds.) ITP 2012. LNCS, vol. 7406, pp. 338–344. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32347-8_23
9. Fox, A., Myreen, M.O.: A trustworthy monadic formalization of the armv7 instruction set architecture. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 243–258. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14052-5_18
10. Jansson, P., Jeuring, J.: Polytypic compact printing and parsing. In: Swierstra, S.D. (ed.) ESOP 1999. LNCS, vol. 1576, pp. 273–287. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-49099-X_18
11. Jourdan, J.H., Pottier, F., Leroy, X.: Validating lr(1) parsers. In: Seidl, H. (ed.) ESOP 2012. LNCS, vol. 7211, pp. 397–416. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28869-2_20
12. Larus, J.R., Schnarr, E.: Eel: machine-independent executable editing. In: PLDI 1995, pp. 291–300. ACM (1995). https://doi.org/10.1145/207110.207163
13. Leroy, X.: The CompCert Verified Compiler (2005–2021). https://compcert.org/
14. Ramananandro, T., et al.: Everparse: verified secure zero-copy parsers for authenticated message formats. In: USENIX Security Symposium, pp. 1465–1482. USENIX Association (2019)
15. Ramsey, N., Fernández, M.F.: Specifying representations of machine instructions. ACM Trans. Program. Lang. Syst. **19**(3), 492–524 (1997). https://doi.org/10.1145/256167.256225
16. Rendel, T., Ostermann, K.: Invertible syntax descriptions: Unifying parsing and pretty printing. In: Proceedings of the 3rd ACM Haskell Symposium on Haskell, pp. 1–12. ACM (2010). https://doi.org/10.1145/1863523.1863525
17. Ridge, T.: Simple, functional, sound and complete parsing for all context-free grammars. In: Jouannaud, J.P., Shao, Z. (eds.) CPP 2011. LNCS, vol. 7086, pp. 103–118. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25379-9_10
18. Tan, Gang, Morrisett, Greg: Bidirectional grammars for machine-code decoding and encoding. J. Autom. Reason. **60**(3), 257–277 (2017). https://doi.org/10.1007/s10817-017-9429-1
19. Van Geest, M., Swierstra, W.: Generic packet descriptions: verified parsing and pretty printing of low-level data. In: Proceedings of the 2nd ACM SIGPLAN Workshop on Type-Driven Development, pp. 30–40. ACM (2017). https://doi.org/10.1145/3122975.3122979
20. Wang, Y., Xu, X., Wilke, P., Shao, Z.: Compcertelf: verified separate compilation of c programs into elf object files. Proc. ACM Program. Lang. **4**(OOPSLA), 197:1–197:28 (2020). https://doi.org/10.1145/3428265
21. Ye, Q., Delaware, B.: A verified protocol buffer compiler. In: CPP 2019, pp. 222–233. ACM (2019). https://doi.org/10.1145/3293880.3294105

An SMT Encoding of LLVM's Memory Model for Bounded Translation Validation

Juneyoung Lee[1(✉)], Dongjoo Kim[1], Chung-Kil Hur[1], and Nuno P. Lopes[2]

[1] Seoul National University, Seoul, South Korea
juneyoung.lee@sf.snu.ac.kr
[2] Microsoft Research, Cambridge, UK

Abstract. Several automatic verification tools have been recently developed to verify subsets of LLVM's optimizations. However, none of these tools has robust support to verify memory optimizations.

In this paper, we present the first SMT encoding of LLVM's memory model that 1) is sufficiently precise to validate all of LLVM's intra-procedural memory optimizations, and 2) enables bounded translation validation of programs with up to hundreds of thousands of lines of code. We implemented our new encoding in Alive2, a bounded translation validation tool, and used it to uncover 21 new bugs in LLVM memory optimizations, 10 of which have been already fixed. We also found several inconsistencies in LLVM IR's official specification document (LangRef) and fixed LLVM's code and the document so they are in agreement.

1 Introduction

Ensuring that LLVM is correct is crucial for the safety and reliability of the software ecosystem. There has been significant work towards this goal including, e.g., formally specifying the semantics of the LLVM IR (intermediate representation). This entails describing precisely what each instruction does and how it handles special cases such as integer overflows, division by zero, or dereferencing out-of-bounds pointers [8,24,26,29,47]. There has also been work on automatic verification of classes of optimizations, such as peephole optimizations [25,31], semi-automated proofs [48], translation validation [20,35,42,44], and fuzzing [23,46]. All this work uncovered several hundred bugs in LLVM.

While there has been great success in improving correctness of scalar optimizations, current verification tools only support basic memory optimizations, if any. Since memory operations can take a significant fraction of a program's run time, memory optimizations are very important for performance. The implementation of these optimizations and related pointer analyses tends to be complex, which further justifies the investment in verifying them.

Verifying programs with memory operations is very challenging and it is hard to scale automatic verification tools that handle these. The main issue lies with pointer aliasing: which objects does a given memory operation access? Without any prior information, a verifier must consider that each operation *may* load or

A. Silva and K. R. M. Leino (Eds.): CAV 2021, LNCS 12760, pp. 752–776, 2021.
https://doi.org/10.1007/978-3-030-81688-9_35

store from any live object (global variables and stack/heap allocations). This creates a big case split for the underlying constraint solver to (attempt to) solve.

Since automatic verification of the source code of memory optimizations is out of reach at the moment, we focus on bounded translation validation [30, 40] (BTV) instead. (Bounded) translation validation consists in verifying that an optimization was correct for a particular input program (up to a bounded unrolling of loops) rather than verifying its correctness for all input programs.

In this paper, we present the first SMT encoding of LLVM's memory model [24] that is precise enough to validate all of LLVM's intraprocedural memory optimizations. The design of the encoding was guided by practical insights of the common aliasing cases in BTV to achieve better performance. For example, we observed that in most cases we can cheaply infer whether a pointer aliases with a locally-allocated or a global object (but not both). Therefore, our encoding case-splits itself on this property rather than leaving that to the SMT solver, as we can cheaply resolve the case split for over 95% of the cases.

The second contribution of this paper is a new semantics for heap allocation for the verification of optimizations for real-world C/C++ programs. Although LLVM's memory model has a reasonable semantics for heap allocations [24], we realized it was not suitable for verifying optimizations. In some programming styles, the result of functions such as `malloc` is not checked against `NULL` and the resulting pointer is dereferenced right away. Since `malloc` can return `NULL` in some executions, we could end up proving that some undesirable optimizations were correct since the program triggers undefined behavior in at least one execution. We propose a new semantics for heap allocations in this paper that is better suited for the verification of optimizations.

The third contribution is the identification of approximations to the SMT encoding such that it is still sufficiently precise to verify (and find bugs) in LLVM's memory optimizations. This is possible since for translation validation we only need to be as precise as LLVM's static analyses (e.g., in the encoding of aliasing rules), and therefore we do not need to consider extremely precise analyses nor arbitrary transformations. Compilers have limited reasoning power by construction in order to keep compilation time reasonable.

We implemented our new SMT encoding of LLVM's memory model in Alive2 [30], a bounded translation validation tool for LLVM. We used Alive2 to find and report 21 previously unknown bugs in LLVM memory optimizations, 10 of which have already been fixed.

To summarize, the contributions of this paper are as follows.

1. The first SMT encoding of LLVM's memory model that is precise enough to verify all of LLVM's intraprocedural memory optimizations.
2. A new semantics for heap allocations for the verification of optimizations of real-world C/C++ programs (Sect. 5.1).
3. A set of approximations to the SMT encoding to further improve the performance of verification without introducing false positives or false negatives in practice (Sect. 9).

4. Thorough evaluation of LLVM's memory model against LLVM's implementation, which uncovered deviations from the model (Sect. 10.3).
5. Identification of 21 previously unknown bugs in LLVM. We present a few examples in Sect. 10.1.

2 Overview

Consider the functions below in `C`:[1] a source (original) function on the left and a target (optimized) function on the right. According to the semantics of high-level languages, and also of LLVM IR, a pointer received as argument or a callee cannot guess the address of a memory region allocated within a function. That is, pointer `q` is not aliased with `p`, `r`, nor touched by `g(p+1)`. Although the caller of `f` may guess the address of `q` in practice, that behavior is excluded by the language semantics because `p`'s object (*provenance*) cannot be a fresh one like `q`. If `p` happens to alias `q`, accessing such pointer triggers undefined behavior (UB).

```
1  int f(int *p) {
2    int *q = malloc(4);
3    *q = 42;
4    int *r = g(p+1);
5    *r = 37;
6    return *q;
7  }
```

```
1′ int f(int *p) {
2′   // q removed
3′
4′   int *r = g(p+1);
5′   *r = 37;
6′   return 42;
7′ }
```

The provenance rules allow LLVM to forward the stored value in line 3 to line 6, and therefore line 6′ simply returns 42. As the value stored to `*q` is not used anymore and pointer `q` does not escape, LLVM also removes the heap allocation.

Next we show how to verify this example. Note that we do not require the two programs to be aligned; the example is aligned to make it easier to understand.

2.1 Verifying the Example Transformation

We start by defining two auxiliary functions that encode the effect of memory operations on the program state. Let state $S = (m, ub)$ be a pair, where m is a memory and ub a boolean that tracks whether the program has already executed UB or not. Let p be the accessed pointer, and v the stored value. The definition of functions $\overline{\textbf{load}}$ and $\overline{\textsf{store}}$ is as follows:

$$\overline{\textbf{load}}\ p\ S ::= (\ \boxed{\textbf{load}(p, S.\mathsf{m})}\ , (S.\mathsf{m},\ S.\mathsf{ub} \vee \neg\ \boxed{\textbf{deref}(p, \texttt{sizeof}(*p), S.\mathsf{m})}\ ))$$
$$\overline{\textsf{store}}\ p\ v\ S ::= (\ \boxed{\textbf{store}(p, v, S.\mathsf{m})}\ , S.\mathsf{ub} \vee \neg\ \boxed{\textbf{deref}(p, \texttt{sizeof}(*p), S.\mathsf{m})}\ )$$

$\overline{\textbf{load}}$ returns a pair with the loaded value and the updated state, where ub is further constrained to ensure that pointer p is dereferenceable for at least the size of the loaded type. Similarly, $\overline{\textsf{store}}$ returns the updated state. The gray boxes ($\boxed{\cdots}$) encode SMT expressions; we describe these in the next section.

[1] We use the syntax of `C` for many of the examples in this paper to make them easier to read, even though we consider the semantics of LLVM IR.

Table 1. States and axioms after executing each of the lines of `f`.

#	**Inputs:** p, m_0, ub_0	#	**Inputs:** p', m'_0, ub'_0
2	$S_1 := (m_0, ub_0)$ $\quad A_1 :=$ q is fresh	$2'$	-
3	$S_2 := \overline{\textbf{store}}\ q\ 42\ S_1$	$3'$	-
4	$S_3 := (m_\mathrm{g}, S_2.\mathsf{ub} \lor ub_\mathrm{g})$ $A_2 :=$ r is not aliased with q $\land$ m_g agrees with $S_2.\mathsf{m}$ on q	$4'$	$S'_1 := (m'_\mathrm{g}, ub'_0 \lor ub'_\mathrm{g})$
5	$S_4 := \overline{\textbf{store}}\ r\ 37\ S_3$	$5'$	$S'_2 := \overline{\textbf{store}}\ r'\ 37\ S'_1$
6	$\mathrm{O} := \overline{\textbf{load}}\ q\ S_4$	$6'$	$\mathrm{O}' := (42, S'_2)$

1. Encoding the output states. Table 1 shows the state after executing each of the programs' lines. p, m_0, and ub_0 are SMT variables for the input pointer, and function `f` caller's memory and UB flag, respectively. The target's corresponding variables are primed. Meta variables are upper-cased and SMT variables are lower-cased.

On line 2, q is assigned a pointer to a new object (encoded in axiom A_1). On line 3, '`*q = 42`' updates the state using $\overline{\textbf{store}}$.

On line 4, the return value, output memory, and UB of `g(p+1)` are represented with fresh variables r, m_g, and ub_g, respectively. Axiom A_2 encodes the provenance rules: the return value cannot alias with locally non-escaped pointers (`q`) and only the remaining objects are modified. Line $4'$ does not need these axioms because there are no locally-allocated objects in the target function.

Finally, the outputs O and O' are a pair of return value and state.

2. Relating the source and target's states. To prove correctness of a transformation, we must first establish refinement between the input states of the source/-target functions. Refinement ($\sqsupseteq$) is used rather than equality because it is allowed for the source's caller to give less defined inputs than the target's.

$$A_\mathrm{in} := p \sqsupseteq p' \land m_0 \sqsupseteq m'_0 \land (ub'_0 \implies ub_0)$$

The inputs and outputs of function calls are also related using refinement. For any pair of calls in the source and target functions, if the target's inputs refine those of the source, the target's output also refines the source's output. The example only has one function call pair:

$$A_\mathrm{call} := \Big(S_2.\mathsf{m} \sqsupseteq m'_0 \land p+1 \sqsupseteq p'+1 \implies m_\mathrm{g} \sqsupseteq m'_\mathrm{g} \land r \sqsupseteq r' \land (ub'_\mathrm{g} \implies ub_\mathrm{g})\Big)$$

We can now state the correctness theorem for the example transformation. For any input, if the axioms hold, the output of the target must refine that of the source for some internal nondeterminism in the source (e.g., the address of pointer `q`). Output is refined iff (i) the source triggers UB, or (ii) the target triggers no UB, and the target's return value and memory refine those of the source.

$$\forall p, p', m_0, m_0', ub_0, ub_0', m_g, m_g', ub_g, ub_g' \,.\; \exists q \,.\; (A_1 \wedge A_2 \wedge A_{\text{in}} \wedge A_{\text{call}}) \implies O \sqsupseteq O'$$

2.2 Efficiently Encoding LLVM's Memory Model and Refinement

We now present our key ideas for efficiently encoding LLVM's memory model and refinement (the gray boxes) in SMT, which is one of our main contributions.

1. Pointers. We represent a pointer as a pair (bid, o) of a block id (i.e., its provenance) and an offset within, so that we can easily detect out-of-bound accesses: accessing (bid, o) in memory m triggers UB unless $0 \leq o < m[bid]$.size, from which $\boxed{\mathbf{deref}((bid, o), sz, m)}$ naturally follows.

2. Bounding the number of blocks. Our first observation is that we can safely bound the number of memory blocks for *bounded* translation validation since loops are unrolled for a fixed number of iterations. As a result, we can use a (fixed-length) bit-vector to encode block ids.

For the example source function, four blocks are sufficient: three for pointers `p`, `q`, `r` as they may all point to different blocks, and an extra to represent all the other blocks that are not syntactically present but are accessible by function `g`.

For the sake of simplifying the example, we ignore that `p`, `q`, `r` may be **null**. Our model does not make such assumption; we explain later how null is handled.

3. Aliasing rules. Several of the aliasing rules are encoded for free as we can distinguish most blocks by construction. First, we use the most significant bit of the block ids to distinguish local (1) from non-local (0) blocks. Second, we assign constant ids whenever possible (e.g., global variables and stack allocations).

For the example source function, (without loss of generality) we set the block ids of `q`, `p` and the extra block to $100_{(2)}$, $000_{(2)}$, and $011_{(2)}$ (in binary format), respectively. However, we cannot fix the block id of `r` and instead give the constraint that it should be either $000_{(2)}$ or $001_{(2)}$ since `r` may alias with `p` but not with `q`. This establishes the alias constraints in A_1 and A_2 for free.

4. Memory accesses. In order to leverage the fact that each pointer may range over a small number of blocks as seen above, we use one SMT array per block (from an offset to a byte) instead of using a single global array (from a pointer to a byte). For the latter, it becomes harder to exploit non-aliasing guarantees since all stores to different blocks are grouped together.

For the example source function, m_0 consists of four arrays $m_0^{(100)}$, $m_0^{(000)}$, $m_0^{(001)}$, $m_0^{(011)}$ for the four blocks. Then since q's block id is $100_{(2)}$, $\overline{\mathbf{store}}\ q\ 42\ S_1$ at line 3 only updates the array $m_0^{(100)}$, leaving the others unchanged. Similarly, $\overline{\mathbf{store}}\ r\ 2\ S_3$ at line 5 only updates $m_0^{(000)}$ and $m_0^{(001)}$ using the SMT if-then-else expression on r's block id. Finally, $\overline{\mathbf{load}}\ q\ S_4$ at line 6 reads from the updated array at $100_{(2)}$, thereby easily realizing that the read value is 42.

5. Refinement. The value/memory refinement $\sqsupseteq$ is defined based on a mapping between source and target blocks, which we efficiently encode leveraging the alignment information between source and target as much as possible (Sect. 7).

3 LLVM's Memory Model

In this section, we give a brief introduction to LLVM's memory model [24]. In this paper we only consider logical pointers (i.e., integer-to-pointer casts are not supported) and a single address space.

Memory Block. A memory block is the unit of memory allocation: each stack or global variable has a distinct block, and heap allocation functions like **malloc** create a fresh block each time they are called. Each block is uniquely identified with a non-negative integer (bid), and has associated properties, including size, alignment, whether it can be written to, whether it is alive, allocation type (heap, stack, global), physical address, and value.

Pointer. A pointer is defined as a triple $(\mathsf{bid}, \mathsf{off}, \mathsf{attrs})$, where off is an offset within the block bid, and attrs is a set of attributes that constrain dereferenceability and which operations are allowed.

Pointer arithmetic operations (**gep**) only change the offset, with bid and attrs being carried over. Unlike C, an offset is allowed to go out-of-bounds (OOB). Such pointer, however, cannot be dereferenced like in C (triggers undefined behavior—UB), but can be used for pointer comparisons for example.

LLVM supports several pointer attributes. For example, a **readonly** pointer p cannot be used to store data. However, it is possible to use a non-**readonly** pointer q to store data to the same location as p (provided the block is writable). A **nocapture** pointer cannot escape from a function. For example, when a function returns, no global variable may have a **nocapture** pointer stored (otherwise it is UB).

LLVM has three constant pointers. The **null** pointer is defined as $(0, 0, \emptyset)$. Block 0 is defined as zero sized and not alive. The **undef**[2] pointer is defined as $(\beta, \delta, \emptyset)$, with β, δ being fresh variables for each observation of the pointer. There is also a **poison**[3] pointer.

Instructions. We consider the following LLVM memory-related instructions:

- Memory access: **load**, **store**
- Memory allocation: **malloc**, **calloc**, **realloc**, **alloca** (stack allocation)
- Lifetime: **start_lifetime** (for stack blocks), **free** (stack/heap deallocation)

[2] In LLVM, **undef** values are arbitrary values of a given type with the additional property that they can yield a different value each time they are observed. **undef** values can be replaced with any value of the same type, except **poison** values.

[3] A **poison** value taints whole expression trees (e.g., **poison** + 1 = **poison**), and branching on it is UB. Similarly, dereferencing a **poison** pointer is UB.

- Pointer-related: **gep** (pointer arithmetic), **icmp** (pointer comparison)
- Library functions: **memcpy**, **memset**, **memcmp**, **strlen**
- Others: **ptrtoint** (pointer-to-integer cast), **call** (function call).

Unsupported memory instructions are: integer-to-pointer casts, and atomic and volatile memory accesses.

4 Encoding Memory Blocks and Pointers in SMT

We describe our new encoding of LLVM's memory model in SMT over the next few sections. We use the theories of UFs (uninterpreted functions), BVs (bit-vectors), and arrays with lambdas [7], with first order quantification. Moreover, we consider that the scope of verification is a single function without loops (or where loops have been previously unrolled).

4.1 Memory Blocks

Each memory block is assigned a distinct identifier (a bit-vector number). We further split memory blocks into local and non-local. Local blocks are all those that are allocated within the function under consideration, either on the stack or the heap. Non-local blocks are the remaining ones, including global variables, heap/stack allocations in callers and heap allocations in callees (stack allocations in callees are not observable, since they are deallocated when the called function returns, hence there is no need to consider them).

We use the most significant bit (MSB) to encode whether a block is local (1) or non-local (0). This representation allows the null block to have $\mathsf{bid} = 0$ and be non-local. We refer to the short block id, or $\widetilde{\mathsf{bid}}$, to refer to bid without the MSB. This is used in cases where it has already been established whether the block is local or not. Example with 4-bit block ids:

```
int g;                  // bid(g) = 0001
void f(int *p) {        // bid(p) = 0xyz (with xyz = arbitrary)
  int a[2];             // bid(a) = 1000
  int *q = malloc(4);   // bid(q) = 1001
}
```

The separation of local and non-local block ids is an efficient way to encode the constraint that pointers of these groups cannot alias with each other. In the example above, argument `p` cannot alias with either `a` or `q`.

As we only consider functions without loops, block ids can be statically assigned for each allocation site.

4.2 Pointers

A pointer $\mathsf{ptr} = (\mathsf{bid}, \mathsf{off}, \mathsf{attrs})$ is encoded as a single bit-vector consisting in the concatenation of the three elements. The offset is interpreted as a *signed* number (which is why blocks cannot be larger than half of the address space). Each attribute (such as **readonly**) is encoded with a bit. Example with 2-bit block ids and offsets, and a single attribute (we use `.` to visually separate the elements):

```
void f(char readonly *p, char *q) { // p = 0x.ab.1, q = 0y.cd.0
  char *r = p + 2;                  // r = 0x.(ab+2).1
  char *s = q + 3;                  // s = 0y.(cd+3).0
  char *t = malloc(4);              // t = 10.00.0
}
```

Let $\widetilde{\mathsf{off}}$ be a truncated offset where the least significant bits corresponding to the greatest common divisor of the alignment and sizes of all memory operations are removed. For example, if all operations are 4-byte aligned and they access either 4- or 8-byte values, then $\widetilde{\mathsf{off}}$ has less 2 bits than off (as these are guaranteed to be always zero when accessing the memory).

4.3 Block Properties

Each block has seven associated properties: size, alignment, read-only, liveness, allocation type (heap, stack, global), physical address, and value. Block properties are looked up and updated by memory operations. For example, when doing a store, we need to check if the access is within the bounds of the block.

Except for liveness and value, properties are fixed at allocation time. Liveness is encoded with a bit-vector (one bit per block), and value with arrays (indexed on $\widetilde{\mathsf{off}}$). We use a multi-memory encoding, where we have one array per bid.

The encoding of fixed properties differs for local and non-local blocks. For non-local blocks, we use a UF symbol per property, taking $\widetilde{\mathsf{bid}}$ as argument. For local blocks, we cannot use UFs because for the refinement check some of these would have to be quantified (c.f. Sect. 7) and most, if not all, SMT solvers do not support quantification of UF symbols. Therefore, we encode each of the remaining properties of local blocks as an if-then-else (ITE) expression, which is tailored for each use (e.g., each time an operation needs to lookup a local block's size, we build an ITE expression for the given $\widetilde{\mathsf{bid}}$).

Using ITE expressions to encode properties is less concise than using UFs. However, it is not a disaster for two reasons. Firstly, we only need to consider the local blocks that have been allocated beforehand, since the program cannot access blocks allocated afterward. Secondly, pointers are usually not fully arbitrary. Oftentimes we know statically which type of block they refer to, and even what is the block id, given that pointer arithmetic operations do not change the block id. Therefore, the ITE expressions are usually small in practice. Example with 4-bit block ids and offsets of a source program:

```
int g;                      // g = 0001.0000, size_src(001) = 4
void f() {
  char p[2];                // p = 1000.0000
  char q[3];                // q = 1001.0000
```

```
    char *r = ... p or q or g ...
    r[2] = 0;
    char t[1];                        // t = 1010.0000
  }
```

The store in this program is only well defined if the size of block pointed by `r` is greater than 2. This is encoded in SMT as follows:

$$\mathsf{ite}(\mathbf{islocal}(r), \mathsf{ite}(\widetilde{\mathbf{bid}}(r) = 0, 2, 3), \mathsf{size}_{src}(\widetilde{\mathbf{bid}}(r))) > 2$$

Function $\mathbf{islocal}(p)$ is encoded with the SMT `extract` expression to fetch the MSB of the pointer. Similarly, $\widetilde{\mathbf{bid}}(p)$ extracts the relevant bits from a pointer. The expression for local blocks only needs to consider local blocks 0 and 1, since block 2 (`t`) is only allocated afterward. This allows a simple single pass through the code to generate optimized ITE expressions.

Value. Value is defined as an array from short offset to byte (described later in Sect. 6.1). For non-local blocks, only those that are constant are initialized with the respective value. The remaining blocks are allowed to take almost any value. The exception is for pointers: non-local blocks cannot initially have local pointers stored, since the calling environment cannot fabricate local pointers.

Local blocks are initialized with **poison** values using a constant array (i.e., an array that yields the same value for all indexes).

4.4 Physical Addresses

If a program observes addresses (through, e.g., pointer-to-integer casting), we need additional constraints to ensure that addresses of blocks that overlap in time are disjoint. Since we are doing translation validation, we have two programs with potentially different sets of locally allocated blocks. Therefore, we need to ensure that non-local blocks' addresses are disjoint from those of local blocks of both programs. This makes the disjointness constraints quite complex.

As an optimization, we split the address space in two: local blocks have MSB = 1 and non-locals have MSB = 0. Since the encoding of address disjointness is quadratic in the worst case (cross-product of blocks), halving the number of blocks is significant. This optimization, however, is an under-approximation of the program's behavior (Sect. 9). After investigating LLVM's optimizations, we believe it is highly unlikely this approximation will cause false negatives.

If a program does not observe any pointer's physical address, neither the block's physical address property nor the disjointness axioms are instantiated. However, when dereferencing a pointer, we need to check if the physical address is sufficiently aligned. When physical addresses are not created, we resort to checking alignment of both of the pointer's block and offset. Since in this case physical addresses are not observed (and therefore not constrained by the program using, e.g., pointer comparisons), a block's physical address can take any value, and therefore blocks and offsets must be both sufficiently aligned to ensure that physical pointers are aligned in all program executions. This argument justifies why we can soundly discard physical addresses.

Table 2. Comparison of two semantics for pointer comparison.

	Integer comparison	Non-deterministic
Fold $\mathsf{p} = \mathsf{q}$ to false if $\mathsf{p}.\text{bid} \neq \mathsf{q}.\text{bid}$	No	Yes
Fold $\mathsf{p} + i = \mathsf{q} + i$ to $\mathsf{p} = \mathsf{q}$	Yes	No
Fold $(int)\mathsf{p} = (int)\mathsf{q}$ to $\mathsf{p} = \mathsf{q}$	Yes	No
Fold $\mathsf{p} < \mathsf{q} \wedge \mathsf{p} \neq \mathsf{q}$ to $\mathsf{p} < \mathsf{q}$	Yes	No
Fold $\mathsf{p} < \mathsf{q} \wedge \mathsf{q} \neq null$ to $\mathsf{p} < \mathsf{q}$	Yes	Potentially
Run-time aliasing checks	Yes	Correct, but not useful
Analysis of pointers cast from integers	Harder	Easy

4.5 Pointer Comparison

Given two pointers p and q, if a program learns that q is placed right after p in memory, the program can potentially change the contents of q without the compiler realizing it. Detecting the existence of such code is impossible in general, hence restricting the ways a program can learn the layout of objects in memory is important to make pointer analyses fast yet precise.

A way the memory layout can leak is through pointer comparison. For example, what should $\mathsf{p} < \mathsf{q}$ return if these point to different memory blocks? If it is a well-defined operation (i.e., simply compares their integer values), it leaks memory layout information. An alternative is to return a non-deterministic value to prevent layout leaks, the formal semantics of which is defined at [24].

We found that there are pros and cons of both semantics for the comparison of pointers of different blocks, and that neither of them covers all optimizations that LLVM performs. Table 2 summarizes the effects on each of the optimizations.

We decided to implement the integer comparison semantics, as LLVM performs all the optimizations above and its alias analyses (AA) mostly give up when they encounter an integer-to-pointer cast. In summary, we have to remove the first optimization from LLVM to make it sound. Additionally, we make it harder to improve LLVM's AA algorithms w.r.t. to pointers cast from integers.

4.6 Bounding the Maximum Number of Blocks

Since we assume that programs do not have loops, we can statically bound the maximum number of both local and non-local blocks a program may observe.

The maximum number of local blocks in the source and target programs, respectively, N^{src}_{local} and N^{tgt}_{local}, is computed by counting the number of heap and stack allocation instructions. Note that this is an upper-bound because not all allocation sites may be reachable in practice.

For non-local blocks, we cannot see their definitions as with local blocks, except for global variables. Nevertheless, we can still bound the maximum number of observed blocks. It is sufficient to count the number of instructions that may return non-local pointers, such as function calls and pointer loads. In addition, we consider a null block when needed (if the null pointer may be observed).

To encode the behavior of source and target programs, we need $N^{src}_{nonlocal} + N^{tgt}_{nonlocal}$ non-local blocks in the worst case, as all referenced pointers may be distinct. However, correct transformations will not have the target program observe more blocks than the source. If the target observes a pointer to a non-local block that was not observed in the source, we can set that pointer to **poison** because its value is not restricted by the source. Therefore, $N^{src}_{nonlocal}$ non-local blocks are sufficient to allow the target to exhibit *an* incorrect behavior.

The bit-width of $\widetilde{\mathsf{bid}}$ is: $w_{\widetilde{bid}} = \lceil \log_2(max(N^{src}_{nonlocal}, max(N^{src}_{local}, N^{tgt}_{local}))) \rceil$. When only local or non-local pointers are used, $w_{bid} = w_{\widetilde{bid}}$, as we know statically if the pointer is local or not. Otherwise, $w_{bid} = w_{\widetilde{bid}} + 1$.

5 Memory Allocation

In LLVM, memory blocks can be allocated on the stack (**alloca**), in the heap (e.g., **malloc**, **calloc**, etc.), or as global variables. It is surprisingly non-trivial to find a semantics for memory allocations that allows all of LLVM's optimizations, and rejects undesired transformations. For example, we have to support allocation removal and splitting, introduce new stack allocations and new constant global variables, etc. We explore multiple semantics and show their merits and shortcomings in the context of proving correctness of program transformations.

5.1 Heap Allocation

Heap allocation is done through functions such as **malloc**, **calloc**, C++'s `new` operator, etc. We describe semantics for **malloc**; remaining functions can be described in terms of it.

First of all, it is important to note that there are two common idioms used in practice by C programmers when doing memory allocation:

```
int *p = malloc(4);      |     int *p = malloc(4);
*p = 0;                  |     if (p) { *p = 0; }
```

In some programs, like the example on the left, **malloc** is assumed to never return **null**, say non-null assumption. This is mainly because the program does not consume too much memory and it is expected that the computer has enough memory/swap space. In other programs like the one on the right, **malloc** is expected to sometimes return **null**, say may-null assumption. Therefore, the program performs null-ness checks.

Since both programming styles are prevalent, we would like optimizations to be correct for both. This is non-trivial, as the two assumptions are conflicting: with the non-null assumption, it is sound to eliminate **null** checks, but not with the may-null assumption. We now explore several possible semantics to find one that works for both programming styles.

A. Malloc always succeeds. Based on the non-null assumption, in this semantics we only consider executions where there is enough space for all allocations to succeed. Regardless of whether the target uses more or less memory than the source, all calls to **malloc** yield non-null pointers. Therefore, for example, deleting unused **malloc** calls is allowed.

However, removing **null** checks of **malloc** is also allowed in this semantics. For example, optimizing the right example above into the left one is sound. This transformation, however, is obviously undesirable.

B. Malloc only succeeds if there is enough free space. To solve the problem just described, based on the may-null assumption, we can simulate the behavior of dynamic memory allocation and define **malloc** to return a pointer to a newly created block if there is an empty space in memory, and **null** otherwise. This semantics prevents the removal of **null** checks of **malloc** as it may return **null**.

However, this semantics does not explain removal of unused allocations. It aligns both source and target programs' allocations such that any change in the allocation sequence disrupts the program alignment and thus makes verification fail. For example, the following transformation removing unused **malloc** instructions and replacing comparisons of their output with **null** is not supported:

```
int *x = malloc(4);              ⇒     // remove x (unused)
if (x != nullptr) { ... }              if (true) { ... }
```

In case there were 0 bytes left in memory, `x` would be **null**, but since LLVM assumes that the program cannot observe the state of the allocator it folds the comparison `x != nullptr` to `true` after eliminating the allocation. This optimization would be flagged as incorrect in this semantics.

LLVM assumes very little about the run-time behavior of memory allocators. This is to support, for instance, garbage collectors, where an allocation may fail but if repeated it may succeed because memory was reclaimed in between. This explains why LLVM folds comparisons with **null** of unused memory blocks, and also contradicts the linear view of allocations of this semantics.

C. Malloc non-deterministically returns null. This semantics abstracts the behavior of the memory allocator by (1) allowing **malloc** to non-deterministically return **null** even if there is available space, and (2) only considering executions where there is enough space for all allocations to succeed. This semantics prevents the removal of null checks of **malloc**, which fixes the shortcomings of semantics A, and also allows the removal of unused allocations, which fixes those of semantics B. However, this semantics is too weak and therefore allows other undesirable transformations, like the following:

```
p = malloc(4);          ⇒          exit();
*p = 0;
```

For the sake of proving refinement (Sect. 7), we need just one trace triggering UB (i.e., one particular realization of the non-deterministic choices) for a given

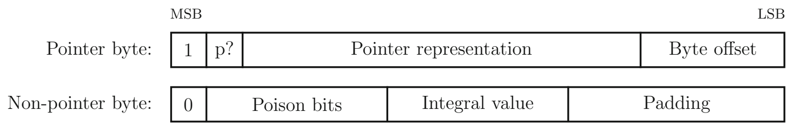


Fig. 1. Bit-wise representation of a byte. A pointer byte is poison if 'p?' is zero. A non-pointer byte tracks poison bit-wise.

input to be able to transform the source program into anything for that input. Informally speaking, refinement always picks the worst-case execution for each input. Since the source program executes UB when p is **null**, it is correct to transform the source into any program although that is obviously undesirable.

This semantics is too weak in practice since many programs are written without **null** checks, either assuming the program will not run out of memory, or assuming the program will terminate if it runs out memory. It is not reasonable in practice to allow compilers to break all such programs.

Our Solution. As we have seen, there is no single semantics that both allows all desired transformations and rejects undesired ones. While semantics B prevents desired optimizations like allocation removal, semantics A and C allow undesired optimizations, but in a complementary way. For example, removing null checks of **malloc** is allowed in A but not in C. On the other hand, transforming an access of a **malloc**-allocated block without a **null** check beforehand into arbitrary code is allowed in C but not in A.

Therefore, we obtain a good semantics by requiring both A and C: an optimization is correct if it passes the refinement criteria with each of the two semantics. Intuitively, this definition requires the compiler to support the two considered coding styles: semantics A supports the non-null assumption, while semantics C the may-null assumption.

5.2 Stack Allocation

The semantics of **alloca**, the stack-allocation instruction, is slightly different from that of **malloc**. LLVM assumes that stack allocations always succeed, since the program will likely crash if there is a stack overflow. That is, **alloca** never returns a **null** pointer.

LLVM performs more optimizations on stack allocations than on heap ones. For example, LLVM can split an allocation into multiple smaller ones or increase the alignment. These transformations can increase memory consumption.

6 Encoding Loads and Stores in SMT

We encode the value of memory blocks with several arrays (one per bid): from short offset to byte. We next give the definition of byte and the encoding of memory accessing instructions in SMT.

6.1 Byte

There are two types of bytes: *pointer* bytes and *non-pointer* bytes, cf. Fig. 1.

A **pointer byte** has the most significant bit (MSB) set to one. The following bit states whether the byte is poison or not. Next is the pointer representation as described in Sect. 4.2 $(\mathsf{bid}, \mathsf{off}, \mathsf{attrs})$.

Pointers are often longer than one byte, so when storing a pointer to memory we write multiple consecutive bytes. Each of these bytes records the same pointer, but with a different byte offset (the last bits of the byte) to distinguish between the partial bytes of the pointer.

For **non-pointer bytes**, we track whether each of the bits is poison or not. This is not required for pointers, since LLVM does not allow pointer values to be manipulated bit-wise. Non-pointer values can be manipulated bit-wise (e.g., using vectors with element types smaller than 8 bits). Each bit of the integral value is only significant if the corresponding poison bit is zero.

6.2 Load and Store Instructions

Load and store instructions are trivially encoded using SMT arrays. These arrays store bytes as described in the previous section. We next describe how LLVM values are encoded to and decoded from our byte representation.

We define two functions, $ty{\Downarrow}(v)$ and $ty{\Uparrow}(b)$, which convert a value v into a byte array and a byte array b back to value, respectively. We show below $ty{\Downarrow}(v)$ when $v \neq$ **poison**. $\mathbf{i}sz$ stands for the integer type with bit-width sz. If sz is not a multiple of 8 bits, v is zero-extended first. When v is poison, all poison bits are set to one. $\mathrm{BitVec}(n, b)$ stands for number n with bit-width b. Pointer's byte offset is 3 bits because we assume 64-bit pointers.

$$\begin{aligned} \mathbf{i}sz{\Downarrow}(v) \text{ or } \mathbf{float}{\Downarrow}(v) &= \lambda i.\, 0 +\!\!+\, 0^8 +\!\!+\, \mathrm{bitrepr}(v)[8\times i \ldots 8\times(i+1)-1] +\!\!+\, \mathrm{padding} \\ ty{*}{\Downarrow}(v) &= \lambda i.\, 1^2 +\!\!+\, \mathrm{bitrepr}(v) +\!\!+\, \mathrm{BitVec}(i, 3) \end{aligned}$$

$\mathbf{i}sz{\Uparrow}(b)$ and $\mathbf{float}{\Uparrow}(b)$ return **poison** if any bit is **poison**, or if any of the bytes is a pointer. Otherwise, these functions return the concatenation of the integral values of the bytes.

$ty{*}{\Uparrow}(b)$ returns **poison** if any of the bytes is **poison** or not a pointer, there is more than one distinct pointer value in b, or one of the bytes has an incorrect byte offset (they have to be consecutive, from zero to byte size minus one). An exception is reading a non-pointer zero byte, which is interpreted as a null pointer byte. This allows initialization of, e.g., arrays with null pointers with **memset** (which is an idiom commonly used in LLVM IR).

6.3 Multi-array Memory

As already described, we use a multi-array encoding for memory, with one array per block id, each indexed on $\widetilde{\mathsf{off}}$. A simpler encoding would have used a single array indexed on ptr. The multi-array encoding is beneficial when we can cheaply compute small aliasing sets for each memory access. In that case, we reduce the

$\text{Num}(sz) ::= \{i \mid 0 \le i < 2^{sz}\}$	$\text{BlockID} ::= \mathbb{N}$	$\text{Addr} ::= \text{Num}(64)$	$\text{Offset} ::= \text{Num}(64)$
$\text{PtrAttr} ::= \{\texttt{nocapture}, \texttt{readonly}, \texttt{readnone}\}$	$\text{Pointer} ::= \text{BlockID} \times \text{Offset} \times 2^{\text{PtrAttr}}$		
$\text{Value} ::= \text{Aggregate} \uplus \text{Int} \uplus \text{Pointer} \uplus \text{Float} \uplus \{\mathbf{poison}\}$	$\text{Aggregate} ::= \text{list Value}$		
$\text{PtrByte} ::= (\text{Pointer} \times \{i \mid 0 \le i < 8\}) \uplus \{\mathbf{poison}\}$	$\text{NonPtrByte} ::= \text{Num}(8) \times \text{Num}(8)$		
$\text{Byte} ::= \text{PtrByte} \uplus \text{NonPtrByte}$	$\text{Bytes} ::= \text{Offset} \rightarrow \text{Byte}$	$\text{Size} ::= \text{Num}(64)$	
$\text{Align} ::= \{i \mid 0 \le i < 64\}$	$\text{Kind} ::= \{\texttt{stack}, \texttt{malloc}, \texttt{new}, \texttt{global}\}$	$\text{Live} ::= \text{bool}$	
$\text{Writable} ::= \text{bool}$	$\text{MemBlock} ::= \text{Addr} \times \text{Align} \times \text{Kind} \times \text{Live} \times \text{Writable} \times \text{Size} \times \text{Bytes}$		
$\text{Memory} ::= \text{BlockID} \rightarrow \text{MemBlock}$	$\text{UB} ::= \text{bool}$	$\text{FinalState} ::= \text{Value} \times \text{Memory} \times \text{UB}$	

$p \in \text{Pointer}$	$ag \in \text{Aggregate}$	$v \in \text{Value}$	$pb \in \text{PtrByte}$	$nb \in \text{NonPtrByte}$
$b \in \text{Byte}$	$mb \in \text{MemBlock}$	$M \in \text{Memory}$	$ub \in \text{UB}$	$\mu \in \text{BlockID} \nrightarrow \text{BlockID}$

Fig. 2. Type definitions and variable naming conventions.

$$\text{(VALUE-POISON)}\ \frac{v \in \text{Value}}{\mathbf{poison} \sqsupseteq^{\mu} v} \qquad \text{(VALUE-NONPTR)}\ \frac{v \in \text{Int} \uplus \text{Float}}{v \sqsupseteq^{\mu} v} \qquad \text{(VALUE-PTR)}\ \frac{p \sqsupseteq^{\mu}_{\text{ptr}} p'}{p \sqsupseteq^{\mu} p'} \qquad \text{(VALUE-AGGREGATE)}\ \frac{|ag| = |ag'| \qquad \forall i,\ ag[i] \sqsupseteq^{\mu} ag'[i]}{ag \sqsupseteq^{\mu} ag'}$$

$$\text{(FINAL-STATE-UB)}\ \frac{}{(v, M, \mathbf{true}) \sqsupseteq_{\text{st}} (v', M', ub')} \qquad \text{(FINAL-STATE)}\ \frac{ub = ub' \qquad \exists \mu,\ v \sqsupseteq^{\mu} v' \wedge M \sqsupseteq^{\mu}_{\text{mem}} M'}{(v, M, ub) \sqsupseteq_{\text{st}} (v', M', ub')}$$

Fig. 3. Refinement of value and final state.

case-splitting work on bid that the SMT solver needs to do, and it enables further formula simplifications like store forwarding.

The multi-array encoding may, however, end up in a larger encoding overall if several of the accesses may alias with too many blocks. For load operations that alias multiple blocks the resulting expression is a linear combination of the loads of each block, e.g., $\mathsf{ite}(\mathsf{bid} = 0, \mathbf{load}(m_0, \widetilde{\mathsf{off}}), \mathsf{ite}(\mathsf{bid} = 1, \mathbf{load}(m_1, \widetilde{\mathsf{off}}), \ldots))$. In this case, it would be more compact to use the single-array encoding. Note that even if we do not know the specific block id, we often know whether a pointer refers to a local or non-local block (e.g., pointers received as argument have unknown block id, but are known to be non-local), and hence splitting the memory in two is usually a good idea (c.f. Sect. 10).

We perform several optimizations that are enabled with this multi-array encoding. We do partial-order reduction (POR) to shrink the potential aliasing of pointers with unknown block id. For example, consider a function with two pointer arguments (`x` and `y`) and one global variable. We assign $\mathsf{bid} = 1$ to the global variable. Then, we estipulate that `x` can only alias blocks with $\mathsf{bid} \le 2$, which is sufficient to access the global variable or another unknown block. Argument `y` is also constrained to only alias blocks with $\mathsf{bid} \le 3$, allowing it to alias with the global variable, the same block as `x`, or a different block. The same is

$$\textsc{(pointer)}\quad \frac{\begin{array}{c} p.\text{block.live} \Rightarrow p'.\text{block.live} \\ p.\text{offset} = p'.\text{offset} \\ \left[\begin{array}{l} (\text{isNonLocal}(\{p, p'\}) \wedge p.\text{block.id} = p'.\text{block.id}) \\ \vee\ (\text{isLocal}(\{p, p'\}) \wedge p.\text{block.id} = \mu[p'.\text{block.id}]) \end{array}\right] \end{array}}{p \sqsupseteq^{\mu}_{\text{ptr}} p'}$$

$$\textsc{(memory-map)}\quad \frac{\begin{array}{c} \left[\begin{array}{l} \forall bid,\ \text{isNonLocal}(bid) \\ \quad \Longrightarrow\ M[bid] \sqsupseteq^{\mu}_{\text{blk}} M'[bid] \end{array}\right] \\ \left[\begin{array}{l} \forall bid,\ \text{isLocal}(bid) \wedge \mu[bid] \text{ defined} \\ \Longrightarrow\ M[\mu[bid]] \sqsupseteq^{\mu}_{\text{blk}} M'[bid] \end{array}\right] \end{array}}{M \sqsupseteq^{\mu}_{\text{mem}} M'}$$

$$\textsc{(byte-ptr)}\quad \frac{\begin{array}{c} pb.\text{byteoff} = pb'.\text{byteoff} \\ pb.\text{ptr} \sqsupseteq^{\mu}_{\text{ptr}} pb'.\text{ptr} \end{array}}{pb \sqsupseteq^{\mu}_{\text{byte}} pb'} \qquad \textsc{(byte-nonptr)}\quad \frac{\begin{array}{c} nb'.\text{p} \mid nb.\text{p} = nb.\text{p} \\ nb.\text{v} \mid nb.\text{p} = nb'.\text{v} \mid nb.\text{p} \end{array}}{nb \sqsupseteq^{\mu}_{\text{byte}} nb'}$$

$$\textsc{(byte-zero)}\quad \frac{\begin{array}{c} \text{isZeroByte}(b) \\ \text{isZeroByte}(b') \end{array}}{b \sqsupseteq^{\mu}_{\text{byte}} b'} \qquad \textsc{(byte-poison)}\quad \frac{\text{isPoisonByte}(b)}{b \sqsupseteq^{\mu}_{\text{byte}} b'}$$

$$\textsc{(bytes)}\quad \frac{\left[\begin{array}{l} \forall\, 0 \leq i < mb.\text{size}, \\ mb.\text{bytes}[i] \sqsupseteq^{\mu}_{\text{byte}} mb'.\text{bytes}[i] \end{array}\right]}{mb \sqsupseteq^{\mu}_{\text{bytes}} mb'}$$

$$\textsc{(block)}\quad \frac{\begin{array}{cc} mb.\text{live} \Rightarrow mb'.\text{live} & mb.\text{size} = mb'.\text{size} \\ mb.\text{kind} = mb'.\text{kind} & mb.\text{writable} = mb'.\text{writable} \\ mb.\text{align} \leq mb'.\text{align} & mb.\text{live} \Rightarrow mb \sqsupseteq^{\mu}_{\text{bytes}} mb' \end{array}}{mb \sqsupseteq^{\mu}_{\text{blk}} mb'}$$

Fig. 4. Refinement of memory and pointers.

done for function calls that return pointers. This POR technique greatly reduces the potential aliasing of unknown pointers without losing precision.

7 Verifying Correctness of Optimizations

To verify correctness of LLVM optimizations, we establish a refinement relation between source (or original) and target (or optimized) functions. Equivalence is not used due to undefined behavior and nondeterminism. Compilers are allowed to reduce the set of possible behaviors from the source.

Given functions f_{src} and f_{tgt}, set of input and output variables I_{src}/I_{tgt} and O (which include, e.g., memory and the return value), and set of non-determinism variables N_{src}/N_{tgt}, f_{src} is refined by f_{tgt} iff:

$$\begin{aligned} \forall I_{src}, I_{tgt}, O_{tgt}\,.\ \ & \text{valid}(I_{src}, I_{tgt})\ \wedge\ I_{src} \sqsupseteq I_{tgt}\ \wedge\ \exists N_{src}\,.\,\text{pre}_{\text{src}}(I_{src}, N_{src})\ \wedge \\ & \big(\exists N_{tgt}\,.\,\text{pre}_{\text{tgt}}(I_{tgt}, N_{tgt})\ \wedge\ [\![f_{tgt}]\!](I_{tgt}, N_{tgt}) = O_{tgt}\big) \\ \Longrightarrow\ \ & (\exists N_{src}\,.\,\text{pre}_{\text{src}}(I_{src}, N_{src})\ \wedge\ [\![f_{src}]\!](I_{src}, N_{src}) \sqsupseteq_{\text{st}} O_{tgt}) \end{aligned}$$

Predicate valid(I_{src}, I_{tgt}) encodes the global precondition of the input memory and arguments such as disjointness of non-local blocks. Function's preconditions, pre_{src} and pre_{tgt}, include the constraint for disjointness of local blocks. The existential pre_{src} constrains the input such that the source function has at least one possible execution. $\sqsupseteq_{\text{st}}$ is the refinement between final states.

Figure 2 shows the definition of final program state which is a tuple of return value, return memory, and UB. A memory is a function from block id to a memory block. A memory block has seven attributes that are described in Sect. 4.3.

Figure 3 shows the definition of refinement of value and final state. For pointers, we cannot simply use equality because local pointers in source and target are internal to each of the functions. Even if they have the same block identifier, they may refer to different allocation sites in the functions (VALUE-PTR). Similarly, the refinement of the final state should consider this difference between local pointers. To address this, we track a mapping μ between escaped local blocks of the two functions (described next).

7.1 Refinement of Memory

Checking refinement of non-local memory blocks is simple as blocks are the same in the source and target functions (e.g., global variables have the same ids in the two functions). Therefore, one just needs to compare blocks of source and target functions with the same id pairwise.

Checking refinement of local blocks is harder but needed when, e.g., the function returns a locally-allocated heap block. This is legal, but block ids in the two functions may not be equal as allocations may have happened in a different order. Therefore, we cannot simply compare local blocks with the same ids.

To check refinement of local blocks, we need to align the two functions' allocations, i.e., we need to find a correspondence between local blocks of the two functions. We introduce a mapping $\mu \in \text{BlockID} \nrightarrow \text{BlockID}$ between target and source local block ids.

Local blocks become related on function calls and return statements, which is when local pointers may be observed. For example, if a function is called with a pointer to a local block as the first argument, μ should relate that pointer with the first argument of an equivalent function call in the target function.

Figure 4 gives the definition of memory refinement, $M \sqsupseteq^{\mu}_{\text{mem}} M'$, as well as other related relations between memory blocks and pointers. The first rule POINTER describes refinement between source pointer p and target pointer p' with respect to μ. The following four rules define refinement between bytes b and b'. In rule BYTE-NONPTR, '$a \,|\, b$' is the bitwise OR operation, and it is used to check the equality of only those bits that are not **poison**. Predicate isZeroByte(b) holds if b is a **null** pointer or if it is a zero-valued non-pointer byte. This is needed because stores of **null** pointers can be optimized to **memset** instructions.

Rules BYTES and BLOCK define refinement between memory blocks' values and memory blocks, respectively. Rule MEMORY-MAP describes memory refinement with respect to local block mapping μ. $M[bid]$ stands for the memory block with block id *bid*.

The well-formedness of μ is established in the refinement rules for function calls and return statements. We show these for function calls in the next section. We note that there might be multiple well-formed μ due to non-determinism.

$$\textsc{(nonptr-arg)}\ \frac{\begin{bmatrix} v, v' \notin \\ \text{Pointer} \end{bmatrix} \quad v \sqsupseteq^{\mu} v'}{v \sqsupseteq^{\mu,sz}_{\text{arg}} v'} \qquad \textsc{(ptr-arg-mapped)}\ \frac{p \sqsupseteq^{\mu}_{\text{ptr}} p'}{p \sqsupseteq^{\mu,sz}_{\text{arg}} p'} \qquad \textsc{(ptr-arg-unmapped)}\ \frac{\begin{array}{c} \text{isLocal}(\{p, p'\}) \\ p.\text{offset} = p'.\text{offset} \\ M[p.\text{bid}] \sqsupseteq^{\mu}_{\text{blk}} M'[p'.\text{bid}] \end{array}}{p \sqsupseteq^{\mu,sz}_{\text{arg}} p'}$$

$$\textsc{(ptr-arg-byval)}\ \frac{\begin{array}{c} sz > 0 \quad o = p.\text{offset} \quad o' = p'.\text{offset} \\ mb = M[p.\text{bid}] \qquad mb' = M'[p'.\text{bid}] \\ \begin{bmatrix} \forall 0 \leq i < sz, \\ mb.\text{bytes}[o+i] \sqsupseteq^{\mu}_{\text{byte}} mb'.\text{bytes}[o'+i] \end{bmatrix} \end{array}}{p \sqsupseteq^{\mu,sz}_{\text{arg}} p'}$$

Fig. 5. Refinement between function arguments.

8 Function Calls

A call to an unknown function may change the memory arbitrarily (except for, e.g., constant variables and non-escaped local blocks). The outputs in the source and target are, however, related: if the target's inputs refine those of the source, refinement holds between their outputs as well. Alive2 already supported function calls; this section shows how it was extended to support memory.

Let (M_{in}, v_{in}) and (M_{out}, v_{out}) be the input and output of a function call in the source, and their primed versions, (M'_{in}, v'_{in}) and (M'_{out}, v'_{out}), those of a function call in the target. Let μ_{in} be a local block mapping before executing the calls. To state that the outputs are refined if the inputs are refined, we add the following formula to the target's precondition:

$$\left(M_{in} \sqsupseteq^{\mu_{in}}_{\text{mem}} M'_{in} \wedge \forall i \,.\, v_{in}[i] \sqsupseteq^{\mu_{in}, sz[i]}_{\text{arg}} v'_{in}[i]\right) \implies \left(M_{out} \sqsupseteq^{\mu_{out}}_{\text{mem}} M'_{out} \wedge v_{out} \sqsupseteq^{\mu_{out}} v'_{out}\right)$$

A call to a function with a pointer to a local block as argument escapes this block, as the callee may, e.g., store that pointer to a global variable. Moreover, any pointer stored in this block also escapes as the callee may traverse the block and grab any pointer stored there, and do so transitively. The updated mapping $\mu_{out} = \text{extend}(\mu_{in}, M_{in}, M'_{in}, v_{in}, v'_{in})$ returns μ_{in} updated with the relationship between the newly escaped blocks in source and target functions.

Figure 5 shows the definition of refinement between function call arguments in source and target programs. The first rule relates non-pointer arguments. The second one handles pointers that have escaped before these calls. The third rule handles local pointers of blocks that did not escape before these calls, and therefore we need to check if the contents of these block are refined.

The fourth refinement rule handles **byval** pointer arguments. These arguments get a freshly allocated block and the contents of the pointer are copied from the pointer's offset onwards.

9 Approximating Program Behavior

In order to speedup verification, we approximate programs' behaviors, which can result in false positives and false negatives. We believe none of these approximations has a significant impact for two reasons: (1) we only need to be as precise as

LLVM's static analyses, i.e., we do not need to support arbitrary optimizations, and (2) we do not consider the compiler to be malicious (which may not be true in certain contexts). Moreover, we conducted an extensive evaluation to support these claims, on which we report in the next section.

Under-Approximations

1. Physical addresses of local memory blocks have the MSB set to 1, and non-locals set to 0. This is reasonable if we assume the compiler is not malicious and therefore will not exploit our approximation.
2. We do not consider the case where a (portion of a) global variable is initially **undef**, only **poison** or a regular value.
3. Library functions **strlen**, **memcmp**, and **bcmp** are unrolled for a constant number of times. A precondition is added to constrain the input to be smaller than the unroll factor. In the case of **strlen**, the input pointer is often a constant array. We compute the result straight away in this case.

Over-Approximations. The set of local blocks that escape (e.g., whose address is stored into a global variable) is computed per function. This may over-approximate the set of escaped pointers at times because, e.g., a pointer may only escape in a particular branch. LLVM also computes the set of escaped pointers per function.

10 Evaluation

We implemented our new memory model in Alive2 [30]. The implementation of the memory model consists in about 3.0 KLoC plus an additional 0.4 KLoC for static analyses for optimization.

We run two set of experiments to both validate our implementation and the formal semantics, and to identify bugs in LLVM. First, we did translation validation of LLVM's unit tests (`test/Transforms`) to increase confidence that we match LLVM's behavior in practice. Second, we run five benchmarks: bzip2, gzip, oggenc, ph7, and SQLite3.

Benchmarks were compiled with `-O3`. Moreover, we disabled type-based aliasing because there is no formal model for this feature yet. During compilation, we emitted pairs of IR files before and after each intra-procedural optimization. We discarded syntactically equal pairs as well as pairs without memory operations.

We used a machine with two Intel Xeon E5-2630 v2 CPUs (total of 12 cores). We set Z3's timeout to 1 min and memory limit to 1 GB. Loops were unrolled once. We used LLVM from 11/Dec (`5e31e22`) and Z3 [33] from 16/Dec (`11477f`).

10.1 LLVM Unit Tests

LLVM's `Transforms` unit test suite consists in 6,600 tests totaling 36,600 functions. Alive2 takes about 2.5 h (in parallel) to validate these. By running LLVM's unit tests, we found 21 new bugs in memory optimizations.

Table 3. Statistics and results for the single-file benchmarks.

Program	LoC	Pairs	Time (hours)	Correct	Incorrect	TO	OOM	Unsupported pairs
bzip2	5.1k	2.3k	1.9	316	9	574	175	1.2k
gzip	5.3k	2.6k	2.0	908	4	922	45	737
oggenc	48k	1.8k	2.0	433	5	617	49	701
ph7	43k	5.6k	3.4	1.2K	23	1.5K	15	2.8k
sqlite3	141k	12k	7.5	2.2k	38	2.2K	48	7.8k

We show below an example of a bug we found. This optimization was shrinking the store from 64 to 32 bits, which is incorrect since the last 32 bits were not copied. This happened because of the mismatch in the load/store's sizes.

```
// i32 *x, *y, *z;                         // i32 *x, *y, *z;
i32 *p = (*x < *y ? x : y);   ⇏   i32 r = (*x < *y ? *x : *y);
*(i64*)z = *(i64*)p;                       *z = r;
```

10.2 Benchmarks

Table 3 shows the statistics and results for translation validation. The Pairs column indicates the number of source/optimized function pairs considered for validation. We discarded pairs where the two functions were syntactically equal, as the transformation is then trivially correct. The last column indicates the number of skipped pairs because they use features Alive2 does not yet support.

All the 79 incorrect pairs are due to mismatches between LLVM and the formal semantics. Of these, 74 are related with incorrect handling of **undef** and **poison** values, and the remaining 5 are caused by incorrect load type punning optimizations. This shows that our tool has no false positives.

10.3 Specification Bugs

While testing our tool, we found a mismatch in the semantics of the **nonnull** attribute between LLVM's documentation and LLVM's code. The documentation specified that passing a null pointer to a **nonnull** argument triggered UB. However, as illustrated below, LLVM adds **nonnull** to a pointer that may be **poison**. This is incorrect because **poison** can be optimized into any value including null.

```
p = gep inbounds q, 1           p = gep inbounds q, 1
f(p)                    ⇒       f(nonnull p) ; UB if p poison
```

We proposed a new semantics to the LLVM developers, where non-conforming pointers would be considered **poison** rather than UB. This was accepted and we have contributed patches to fix the docs and the incorrect optimizations.

10.4 Alias Sets

To show that splitting the memory into multiple arrays is beneficial, we gathered statistics of the alias sets in our benchmarks. More than 96% of the dereferenced pointers turned out to be only local or non-local, but not both. This shows that splitting the memory into local and non-local simplifies the memory encoding.

We also counted the number of memory blocks pointers may alias with. Half of the pointers were aliased with just one block. About 80% of the pointers aliased with at most 3 blocks. This is much less than the median number of blocks functions have. The median of the number of memory blocks was $7 \sim 13$ (varying over programs), and only 10% of the functions had fewer than 3 blocks.

11 Related Work

Semantics of LLVM IR. The official LLVM IR's specification is written in prose [1]. Vellvm [47] and K-LLVM [29] formalized large subsets of the IR in Coq and K, respectively. [26] clarifies the semantics of **undef** and **poison** and proposes a new **freeze** instruction. [24] formalizes various memory instructions of LLVM. [32] presents a C memory model that supports compilation to that LLVM model.

Translation validation. [38] presents a translation validation infrastructure for GCC's intermediate language, using a set of arithmetic/aliasing rules for showing equivalence. LLVM-MD [44] and Peggy [42] verify LLVM optimizations by showing equivalence of source and targets with rewrite rules/equality axioms. They suffer, however, from incomplete axioms for aliasing.

In order to simplify the work of translation validation tools, it is possible to extend the compiler to produce hints (witnesses) [18,36,38,41]. One of these tools, Crellvm [20], is formally verified in Coq.

Verifying programs with memory using SMT solvers. SMT solvers have been used before to check equivalence of programs with memory [11,14,21,25,31]. [12] give an encoding of some (but not all) aliasing constraints needed to do translation validation of assembly generated by C compilers.

Other memory models encoded in SMT include one for Solidity (Etherium smart contracts) [16], and for separation logic [37,39]. Several verification tools include SAT/SMT-based (partial) memory models for C [2,9,10] and Java [43].

Several automatic software verification tools, often based on CHCs (constrained Horn clauses), support memory programs [6,13]. For example, both SeaHorn and Cascade use a field-sensitive alias analysis to split the memory [15,45].

SLAYER [4] is an automatic tool for analyzing memory safety of a C program using Z3. Smallfoot [3] verifies assertions written in separation logic.

There have been recent advances in speeding up verification of (SMT) array programs [17,22], from which we could likely benefit.

CompCert [27] splits the memory into local (private) and non-local (public) blocks, similarly to what we do, but assumes that allocations never fail [28]. Work on verifying peephole optimizations for CompCert does not support memory [34].

To support integer-to-pointer casts in CompCert, [5] proposes extending integer values to carry block ids as well. In this model, arithmetic on pointer values yields a symbolic expression. [19] makes the pointer-to-integer cast an instruction that assigns a physical address to the block. Neither of these models supports several optimizations performed by LLVM.

12 Conclusion

We presented the first SMT encoding of LLVM's memory model that is sufficiently precise to validate all of LLVM's intra-procedural memory optimizations.

Using our new encoding, we found and reported 21 previously unknown bugs in LLVM memory optimizations, 10 of which have already been fixed.

Acknowledgement. This work was supported in part by the Basic Science Research Program through the National Research Foundation of Korea (NRF-2020R1A2C2011947).

References

1. LLVM language reference manual. https://llvm.org/docs/LangRef.html
2. Ball, T., Bounimova, E., Levin, V., de Moura, L.: Efficient evaluation of pointer predicates with Z3 SMT solver in SLAM2. Technical Report MSR-TR-2010-24, Microsoft Research (2010), https://www.microsoft.com/en-us/research/publication/efficient-evaluation-of-pointer-predicates-with-z3-smt-solver-in-slam2/
3. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: modular automatic assertion checking with separation logic. In: FMCO (2006). https://doi.org/10.1007/11804192_6
4. Berdine, J., Cook, B., Ishtiaq, S.: Slayer: memory safety for systems-level code. In: CAV (2011). https://doi.org/10.1007/978-3-642-22110-1_15
5. Besson, F., Blazy, S., Wilke, P.: A concrete memory model for CompCert. In: ITP (2015). https://doi.org/10.1007/978-3-319-22102-1_5
6. Bjørner, N., McMillan, K., Rybalchenko, A.: On solving universally quantified horn clauses. In: SAS (2013). https://doi.org/10.1007/978-3-642-38856-9_8
7. Bryant, R.E., Lahiri, S.K., Seshia, S.A.: Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In: CAV (2002). https://doi.org/10.1007/3-540-45657-0_7
8. Chakraborty, S., Vafeiadis, V.: Formalizing the concurrency semantics of an LLVM fragment. In: CGO (2017). https://doi.org/10.1109/CGO.2017.7863732

9. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS (2004). https://doi.org/10.1007/978-3-540-24730-2_15
10. Cordeiro, L., Fischer, B., Marques-Silva, J.: SMT-based bounded model checking for embedded ANSI-C software. In: ASE (2009). https://doi.org/10.1109/ASE.2009.63
11. Dahiya, M., Bansal, S.: Black-box equivalence checking across compiler optimizations. In: APLAS (2017). https://doi.org/10.1007/978-3-319-71237-6_7
12. Dahiya, M., Bansal, S.: Modeling undefined behaviour semantics for checking equivalence across compiler optimizations. In: HVC (2017). https://doi.org/10.1007/978-3-319-70389-3_2
13. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: PLDI (2012). https://doi.org/10.1145/2254064.2254112
14. Gupta, S., Saxena, A., Mahajan, A., Bansal, S.: Effective use of SMT solvers for program equivalence checking through invariant-sketching and query-decomposition. In: SAT (2018). https://doi.org/10.1007/978-3-319-94144-8_22
15. Gurfinkel, A., Navas, J.A.: A context-sensitive memory model for verification of C/C++ programs. In: SAS (2017). https://doi.org/10.1007/978-3-319-66706-5_8
16. Hajdu, Á., Jovanović, D.: SMT-friendly formalization of the solidity memory model. In: ESOP (2020)
17. Ish-Shalom, O., Itzhaky, S., Rinetzky, N., Shoham, S.: Putting the squeeze on array programs: Loop verification via inductive rank reduction. In: VMCAI (2020). https://doi.org/10.1007/978-3-030-39322-9_6
18. Kanade, A., Sanyal, A., Khedker, U.P.: Validation of GCC optimizers through trace generation. SP&E **39**(6), 611–639 (2009). https://doi.org/10.1002/spe.913
19. Kang, J., Hur, C.K., Mansky, W., Garbuzov, D., Zdancewic, S., Vafeiadis, V.: A formal C memory model supporting integer-pointer casts. In: PLDI (2015). https://doi.org/10.1145/2737924.2738005
20. Kang, J., et al.: Crellvm: Verified credible compilation for LLVM. In: PLDI (2018). https://doi.org/10.1145/3192366.3192377
21. Klebanov, V., Rümmer, P., Ulbrich, M.: Automating regression verification of pointer programs by predicate abstraction. Formal Methods Syst. Des. **52**(3), 229–259 (2018). https://doi.org/10.1007/s10703-017-0293-8
22. Komuravelli, A., Bjørner, N., Gurfinkel, A., McMillan, K.L.: Compositional verification of procedural programs using horn clauses over integers and arrays. In: FMCAD (2015). https://doi.org/10.1109/FMCAD.2015.7542257
23. Le, V., Afshari, M., Su, Z.: Compiler validation via equivalence modulo inputs. In: PLDI (2014). https://doi.org/10.1145/2594291.2594334
24. Lee, J., Hur, C.K., Jung, R., Liu, Z., Regehr, J., Lopes, N.P.: Reconciling high-level optimizations and low-level code in LLVM. In: Proceedings of the ACM on Programming Languages 2(OOPSLA), November 2018. https://doi.org/10.1145/3276495
25. Lee, J., Hur, C.K., Lopes, N.P.: AliveInLean: a verified LLVM peephole optimization verifier. In: CAV (2019). https://doi.org/10.1007/978-3-030-25543-5_25
26. Lee, J., et al.: Taming undefined behavior in LLVM. In: PLDI (2017). https://doi.org/10.1145/3062341.3062343
27. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (2009). https://doi.org/10.1145/1538788.1538814
28. d Leroy, X., Appel, A.W., Blazy, S., Stewart, G.: The CompCert memory model, version 2. Technical Report RR-7987, INRIA, June 2012. http://hal.inria.fr/hal-00703441

29. Li, L., Gunter, E.L.: -LLVM: a relatively complete semantics of LLVM IR. ECOOP (2020). https://doi.org/10.4230/LIPIcs.ECOOP.2020.7
30. Lopes, N.P., Lee, J., Hur, C.K., Liu, Z., Regehr, J.: Alive2: bounded translation validation for LLVM. In: PLDI (2021). https://doi.org/10.1145/3453483.3454030
31. Lopes, N.P., Menendez, D., Nagarakatte, S., Regehr, J.: Provably correct peephole optimizations with Alive. In: PLDI (2015). https://doi.org/10.1145/2737924.2737965
32. Memarian, K., et al.: Exploring C semantics and pointer provenance. Proc. ACM Program. Lang. **3**(POPL) (2019). https://doi.org/10.1145/3290380
33. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: TACAS (2008). https://doi.org/10.1007/978-3-540-78800-3_24
34. Mullen, E., Zuniga, D., Tatlock, Z., Grossman, D.: Verified peephole optimizations for CompCert. In: PLDI (2016). https://doi.org/10.1145/2908080.2908109
35. Namjoshi, K.S., Tagliabue, G., Zuck, L.D.: A witnessing compiler: a proof of concept. In: RV (2013). https://doi.org/10.1007/978-3-642-40787-1_22
36. Namjoshi, K.S., Zuck, L.D.: Witnessing program transformations. In: SAS (2013). https://doi.org/10.1007/978-3-642-38856-9_17
37. Navarro Pérez, J.A., Rybalchenko, A.: Separation logic modulo theories. In: APLAS (2013). https://doi.org/10.1007/978-3-319-03542-0_7
38. Necula, G.C.: Translation validation for an optimizing compiler. In: PLDI (2000). https://doi.org/10.1145/349299.349314
39. Piskac, R., Wies, T., Zufferey, D.: Automating separation logic using SMT. In: CAV (2013). https://doi.org/10.1007/978-3-642-39799-8_54
40. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: TACAS (1998). https://doi.org/10.1007/BFb0054170
41. Rinard, M.C., Marinov, D.: Credible compilation with pointers. In: RTRV (1999)
42. Stepp, M., Tate, R., Lerner, S.: Equality-based translation validator for LLVM. In: CAV (2011). https://doi.org/10.1007/978-3-642-22110-159
43. Torlak, E., Vaziri, M., Dolby, J.: MemSAT: checking axiomatic specifications of memory models. In: PLDI (2010). https://doi.org/10.1145/1806596.1806635
44. Tristan, J.B., Govereau, P., Morrisett, J.G.: Evaluating value-graph translation validation for LLVM. In: PLDI (2011). https://doi.org/10.1145/1993316.1993533
45. Wang, W., Barrett, C., Wies, T.: Partitioned memory models for program analysis. In: VMCAI (2017). https://doi.org/10.1007/978-3-319-52234-0_29
46. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in C compilers. In: PLDI (2011). https://doi.org/10.1145/1993498.1993532
47. Zhao, J., Nagarakatte, S., Martin, M.M., Zdancewic, S.: Formalizing the LLVM intermediate representation for verified program transformations. In: POPL (2012). https://doi.org/10.1145/2103656.2103709
48. Zhao, J., Nagarakatte, S., Martin, M.M., Zdancewic, S.: Formal verification of SSA-based optimizations for LLVM. In: PLDI (2013). https://doi.org/10.1145/2491956.2462164

Automatically Tailoring Abstract Interpretation to Custom Usage Scenarios

Muhammad Numair Mansur[1](✉), Benjamin Mariano[2], Maria Christakis[1], Jorge A. Navas[3], and Valentin Wüstholz[4]

[1] MPI-SWS, Kaiserslautern and Saarbrücken, Germany
{numair,maria}@mpi-sws.org
[2] The University of Texas at Austin, Austin, USA
bmariano@cs.utexas.edu
[3] SRI International, Menlo Park, USA
jorge.navas@sri.com
[4] ConsenSys, Kaiserslautern, Germany
valentin.wustholz@consensys.net

Abstract. In recent years, there has been significant progress in the development and industrial adoption of static analyzers, specifically of abstract interpreters. Such analyzers typically provide a large, if not huge, number of configurable options controlling the analysis precision and performance. A major hurdle in integrating them in the software-development life cycle is tuning their options to custom usage scenarios, such as a particular code base or certain resource constraints.

In this paper, we propose a technique that automatically tailors an abstract interpreter to the code under analysis and any given resource constraints. We implement this technique in a framework, tAIlor, which we use to perform an extensive evaluation on real-world benchmarks. Our experiments show that the configurations generated by tAIlor are vastly better than the default analysis options, vary significantly depending on the code under analysis, and most remain tailored to several subsequent code versions.

1 Introduction

Static analysis inspects code, without running it, in order to prove properties or detect bugs. Typically, static analysis approximates code behavior, for instance, because checking the correctness of most properties is undecidable. *Performance* is another important reason for this approximation. Typically, the closer the approximation is to the actual code behavior, the less efficient and the more *precise* the analysis is, that is, the fewer false positives it reports. For less tight approximations, the analysis tends to become more efficient but less precise.

Recent years have seen tremendous progress in the development and industrial adoption of static analyzers. Notable successes include Facebook's Infer [7,8] and AbsInt's Astrée [5]. Many popular analyzers, such as these, are based on *abstract interpretation* [12], a technique that abstracts the concrete program

A. Silva and K. R. M. Leino (Eds.): CAV 2021, LNCS 12760, pp. 777–800, 2021.
https://doi.org/10.1007/978-3-030-81688-9_36

semantics and reasons about its abstraction. In particular, program states are abstracted as elements of *abstract domains*. Most abstract interpreters offer a wide range of abstract domains that impact the precision and performance of the analysis. For instance, the Intervals domain [11] is typically faster but less precise than Polyhedra [16], which captures linear inequalities among variables.

In addition to the domains, abstract interpreters usually provide a large number of other options, for instance, whether backward analysis should be enabled or how quickly a fixpoint should be reached. In fact, the sheer number of option combinations (over 6M in our experiments) is bound to overwhelm users, especially non-expert ones. To make matters worse, the best option combinations may vary significantly depending on the code under analysis and the resources, such as time or memory, that users are willing to spend.

In light of this, we suspect that most users resort to using the default options that the analysis designer pre-selected for them. However, these are definitely not suitable for all code. Moreover, they do not adjust to different stages of software development, e.g., running the analysis in the editor should be much faster than running it in a continuous integration (CI) pipeline, which in turn should be much faster than running it prior to a major release. The alternative of enabling the (in theory) most precise analysis can be even worse, since in practice it often runs out of time or memory as we show in our experiments. As a result, the widespread adoption of abstract interpreters is severely hindered, which is unfortunate since they constitute an important class of practical analyzers.

Our Approach. To address this issue, we present the first technique that automatically tailors a generic abstract interpreter to a custom usage scenario. With the term *custom usage scenario*, we refer to a particular piece of code and specific resource constraints. The key idea behind our technique is to phrase the problem of customizing the abstract-interpretation configuration to a given usage scenario as an optimization problem. Specifically, different configurations are compared using a cost function that penalizes those that prove fewer properties or require more resources. The cost function can guide the configuration search of a wide range of existing optimization algorithms. This problem of tuning abstract interpreters can be seen as an instance of the more general problem of *algorithm configuration* [31]. In the past, algorithm configuration has been used to tune algorithms for solving various hard problems, such as SAT solving [32,33], and more recently, training of machine-learning models [3,18,52].

We implement our technique in an open-source framework called tAIlor[1], which configures a given abstract interpreter for a given usage scenario using a given optimization algorithm. As a result, tAIlor enables the abstract interpreter to prove as many properties as possible within the resource limit without requiring any domain expertise on behalf of the user.

Using tAIlor, we find that tailored configurations vastly outperform the default options pre-selected by the analysis designers. In fact, we show that this is possible even with very simple optimization algorithms. Our experiments

[1] The tool implementation is found at https://github.com/Practical-Formal-Methods/tailor and an installation at https://doi.org/10.5281/zenodo.4719604.

also demonstrate that tailored configurations vary significantly depending on the usage scenario—in other words, there cannot be a single configuration that fits all scenarios. Finally, most of the generated configurations remain tailored to several subsequent code versions, suggesting that re-tuning is only necessary after major code changes.

Contributions. We make the following contributions:

1. We present the first technique for automatically tailoring abstract interpreters to custom usage scenarios.
2. We implement our technique in an open-source framework called tAIlor.
3. Using a state-of-the-art abstract interpreter, Crab [25], with millions of configurations, we show the effectiveness of tAIlor on real-world benchmarks.

2 Overview

We now illustrate the workflow and tool architecture of tAIlor and provide examples of its effectiveness.

Terminology. In the following, we refer to an abstract domain with all its options (e.g., enabling backward analysis or more precise treatment of arrays etc.) as an *ingredient*.

As discussed earlier, abstract interpreters typically provide a large number of such ingredients. To make matters worse, it is also possible to combine different ingredients into a sequence (which we call a *recipe*) such that more properties are verified than with individual ingredients. For example, a user could configure the abstract interpreter to first use Intervals to verify as many properties as possible and then use Polyhedra to attempt verification of any remaining properties. Of course, the number of possible configurations grows exponentially in the length of the recipe (over 6M in our experiments for recipes up to length 3).

Workflow. The high-level architecture of tAIlor is shown in Fig. 1. It takes as input the code to be analyzed (i.e., any program, file, function, or fragment), a user-provided resource limit, and optionally an optimization algorithm. We focus on time as the constrained resource in this paper, but our technique could be easily extended to other resources, such as memory.

The optimization engine relies on a recipe generator to generate a fresh recipe. To assess its quality in terms of precision and performance, the recipe evaluator computes a cost for the recipe. The cost is computed by evaluating how precise and efficient the abstract interpreter is for the given recipe. This cost is used by the optimization engine to keep track of the best recipe so far, i.e., the one that proves the most properties in the least amount of time. tAIlor repeats this process for a given number of iterations to sample multiple recipes and returns the recipe with the lowest cost.

Zooming in on the evaluator, a recipe is processed by invoking the abstract interpreter for each ingredient. After each analysis (i.e., one ingredient), the evaluator collects the new verification results, that is, the verified assertions. All

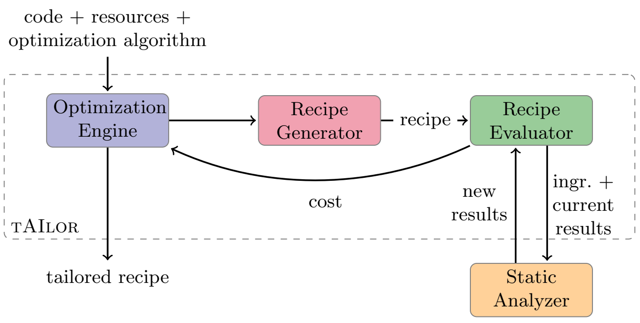


Fig. 1. Overview of our framework.

verification results that have been achieved so far are subsequently shared with the analyzer when it is invoked for the next ingredient. Verification results are shared by converting all verified assertions into assumptions. After processing the entire recipe, the evaluator computes a cost for the recipe, which depends on the number of unverified assertions and the total analysis time.

In general, there might be more than one recipe tailored to a particular usage scenario. Naïvely, finding one requires searching the space of all recipes. Section 4.3 discusses several optimization algorithms for performing this search, which tAIlor already incorporates in its optimization engine.

Examples. As an example, let us consider the usage scenario where a user runs the Crab abstract interpreter [25] in their editor for instant feedback during code development. This means that the allowed time limit for the analysis is very short, say, 1 s. Now assume that the code under analysis is a program file[2] of the multimedia processing tool ffmpeg, which is used to evaluate the effectiveness of tAIlor in our experiments. In this file, Crab checks 45 assertions for common bugs, i.e., division by zero, integer overflow, buffer overflow, and use after free.

Analysis of this file with the default Crab configuration takes 0.35 s to complete. In this time, Crab proves 17 assertions and emits 28 warnings about the properties that remain unverified. For this usage scenario, tAIlor is able to tune the abstract-interpreter configuration such that the analysis time is 0.57 s and the number of verified properties increases by 29% (i.e., 22 assertions are proved). Note that the tailored configuration uses a completely different abstract domain than the one in the default configuration. As a result, the verification results are significantly better, but the analysis takes slightly longer to complete (although remaining within the specified time limit). In contrast, enabling the most precise analysis in Crab verifies 26 assertions but takes over 6 min to complete, which by far exceeds the time limit imposed by the usage scenario.

While it takes tAIlor 4.5 s to find the above configuration, this is time well invested; the configuration can be re-used for several subsequent code versions. In fact, in our experiments, we show that generated configurations can remain

[2] https://github.com/FFmpeg/FFmpeg/blob/master/libavformat/idcin.c

tailored for at least up to 50 subsequent commits to a file under version control. Given that changes in the editor are typically much more incremental, we expect that no re-tuning would be necessary at all during an editor session. Re-tuning may be beneficial after major changes to the code under analysis and can happen offline, e.g., between editor sessions, or in the worst case overnight.

As another example, consider the usage scenario where Crab is integrated in a CI pipeline. In this scenario, users should be able to spare more time for analysis, say, 5 min. Here, let us assume that the analyzed code is a program file[3] of the curl tool for transferring data by URL, which is also used in our evaluation. The default Crab configuration takes 0.23 s to run and only verifies 2 out of 33 checked assertions. tAIlor is able to find a configuration that takes 7.6 s and proves 8 assertions. In contrast, the most precise configuration does not terminate even after 15 min.

Both scenarios demonstrate that, even when users have more time to spare, the default configuration cannot take advantage of it to improve the verification results. At the same time, the most precise configuration is completely impractical since it does not respect the resource constraints imposed by these scenarios.

3 Background: A Generic Abstract Interpreter

Many successful abstract interpreters (e.g., Astrée [5], C Global Surveyor [53], Clousot [17], Crab [25], IKOS [6], Sparrow [46], and Infer [8]) follow the generic architecture in Fig. 2. In this section, we describe its main components to show that our approach should generalize to such analyzers.

Memory Domain. Analysis of low-level languages such as C and LLVM-bitcode requires reasoning about pointers. It is, therefore, common to design a *memory domain* [42] that can simultaneously reason about pointer aliasing, memory contents, and numerical relations between them.

Pointer domains resolve aliasing between pointers, and *array domains* reason about memory contents. More specifically, array domains can reason about individual memory locations (cells), infer universal properties over multiple cells, or both. Typically, reasoning about individual cells trades performance for precision unless there are very few array elements (e.g., [22,42]). In contrast, reasoning about multiple memory locations (*summarized cells*) trades precision for performance. In our evaluation, we use *Array smashing* domains [5] that abstract different array elements into a single summarized cell. *Logico-numerical domains* infer relationships between program and *synthetic* variables, introduced by the pointer and array domains, e.g., summarized cells.

Next, we introduce domains typically used for proving the absence of runtime errors in low-level languages. *Boolean domains* (e.g., flat Boolean, BDDApron [1]) reason about Boolean variables and expressions. *Non-relational domains* (e.g., Intervals [11], Congruence [23]) do not track relations among different variables, in contrast to *relational domains* (e.g., Equality [35], Zones [41],

[3] https://github.com/curl/curl/blob/master/lib/cookie.c

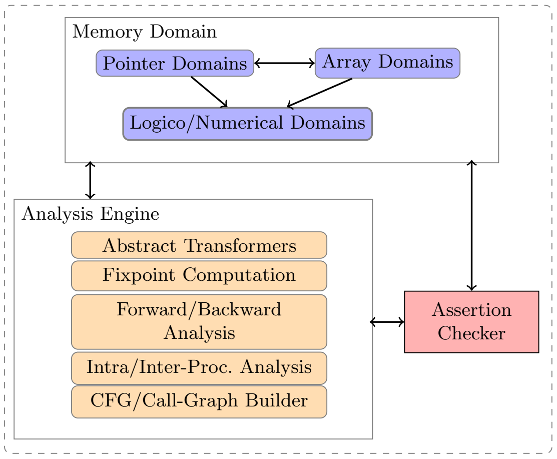


Fig. 2. Generic architecture of an abstract interpreter.

Octagons [43], Polyhedra [16]). Due to their increased precision, relational domains are typically less efficient than non-relational ones. *Symbolic domains* (e.g., Congruence closure [9], Symbolic constant [44], Term [21]) abstract complex expressions (e.g., non-linear) and external library calls by uninterpreted functions. *Non-convex domains* express disjunctive invariants. For instance, the DisInt domain [17] extends Intervals to a finite disjunction; it retains the scalability of the Intervals domain by keeping only non-overlapping intervals. On the other hand, the Boxes domain [24] captures arbitrary Boolean combinations of intervals, which can often be expensive.

Fixpoint Computation. To ensure termination of the fixpoint computation, Cousot and Cousot introduce *widening* [12,14], which usually incurs a loss of precision. There are three common strategies to reduce this precision loss, which however sacrifice efficiency. First, *delayed widening* [5] performs a number of initial fixpoint-computation iterations in the hope of reaching a fixpoint before resorting to widening. Second, *widening with thresholds* [37,40] limits the number of program expressions (thresholds) that are used when widening. The third strategy consists in applying *narrowing* [12,14] a certain number of times.

Forward and Backward Analysis. Classically, abstract interpreters analyze code by propagating abstract states in a *forward* manner. However, abstract interpreters can also perform *backward* analysis to compute the execution states that lead to an assertion violation. Cousot and Cousot [13,15] define a *forward-backward refinement* algorithm in which a forward analysis is followed by a backward analysis until no more refinement is possible. The backward analysis uses invariants computed by the forward analysis, while the forward analysis does not explore states that cannot reach an assertion violation based on the backward analysis. This refinement is more precise than forward analysis alone, but it may also become very expensive.

Algorithm 1: Optimization engine.

```
1  Function OPTIMIZE($P$, $r_{max}$, $l_{max}$, $i_{dom}$, $i_{set}$, $rec_{init}$, GENERATERECIPE, ACCEPT) is
2      // Phase 1 (optimize domains)
3      $rec_{best} := rec_{curr} := rec_{init}$
4      $cost_{best} := cost_{curr} :=$ EVALUATE($P$, $r_{max}$, $rec_{best}$)
5      for $l := 1$ to $l_{max}$ do
6          for $i := 1$ to $i_{dom} \cdot l$ do
7              $rec_{next} :=$ GENERATERECIPE($rec_{curr}$, $l$)
8              $cost_{next} :=$ EVALUATE($P$, $r_{max}$, $rec_{next}$)
9              if $cost_{next} < cost_{best}$ then
10                 $rec_{best}, cost_{best} := rec_{next}, cost_{next}$
11             if ACCEPT($cost_{curr}$, $cost_{next}$) then
12                 $rec_{curr}, cost_{curr} := rec_{next}, cost_{next}$
13     // Phase 2 (optimize settings)
14     for $i := 1$ to $i_{set}$ do
15         $rec_{mut} :=$ MUTATESETTINGS($rec_{best}$)
16         $cost_{mut} :=$ EVALUATE($P$, $r_{max}$, $rec_{mut}$)
17         if $cost_{mut} < cost_{best}$ then
18             $rec_{best}, cost_{best} := rec_{mut}, cost_{mut}$
19     return $rec_{best}$
```

Intra- and Inter-procedural Analysis. An *intra-procedural* analysis analyzes a function ignoring the information (i.e., call stack) that flows into it, while an *inter-procedural* analysis considers all flows among functions. The former is much more efficient and easy to parallelize, but the latter is usually more precise.

4 Our Technique

This section describes the components of tAIlor in detail; Sects. 4.1, 4.2, 4.3 explain the optimization engine, recipe evaluator, and recipe generator (Fig. 1).

4.1 Recipe Optimization

Algorithm 1 implements the optimization engine. In addition to the code P and the resource limit r_{max}, it also takes as input the maximum length of the generated recipes l_{max} (i.e., the maximum number of ingredients), a function to generate new recipes GenerateRecipe (i.e., the recipe generator from Fig. 1), and four other parameters, which we explain later.

A tailored recipe is found in two phases. The first phase aims to find the best abstract domain for each ingredient, while the second tunes the remaining analysis settings for each ingredient (e.g., whether backward analysis should be enabled). Parameters i_{dom} and i_{set} control the number of iterations of each phase. Note that we start with a search for the best domains since they have the largest impact on the precision and performance of the analysis.

During the first phase, the algorithm initializes the best recipe rec_{best} with an initial recipe rec_{init} (line 3). The cost of this recipe is evaluated with function Evaluate, which implements the recipe evaluator from Fig. 1. The subsequent nested loop (line 5) samples a number of recipes, starting with the shortest recipes ($l := 1$) and ending with the longest recipes ($l := l_{max}$). The inner loop generates i_{dom} ingredients for each ingredient in the recipe (i.e., $i_{dom} \cdot l$ total iterations) by invoking function GenerateRecipe, and in case a recipe with lower cost is found, it updates the best recipe (lines 9–10). Several optimization algorithms, such as hill climbing and simulated annealing, search for an optimal result by mutating some of the intermediate results. Variable rec_{curr} stores intermediate recipes to be mutated, and function Accept decides when to update it (lines 11–12).

As explained earlier, the purpose of the first phase is to identify the best sequence of abstract domains. The second phase (lines 13–18) focuses on tuning the other settings of the best recipe so far. This is done by randomly mutating the best recipe via MutateSettings (line 15), and updating the best recipe if better settings are found (lines 17–18). After exploring i_{set} random settings, the best recipe is returned to the user (line 19).

4.2 Recipe Evaluation

The recipe evaluator from Fig. 1 uses a cost function to determine the quality of a fresh recipe with respect to the precision and performance of the abstract interpreter. This design is motivated by the fact that analysis imprecision and inefficiency are among the top pain points for users [10].

Therefore, the cost function depends on the number of generated warnings w (that is, the number of unverified assertions), the total number of assertions in the code w_{total}, the resource consumption r of the analyzer, and the resource limit r_{max} imposed on the analyzer:

$$cost(w, w_{total}, r, r_{max}) = \begin{cases} \dfrac{w + \dfrac{r}{r_{max}}}{w_{total}}, & \text{if } r \leq r_{max} \\ \infty, & \text{otherwise} \end{cases}$$

Note that w and r are measured by invoking the abstract interpreter with the recipe under evaluation. The cost function evaluates to a lower cost for recipes that improve the precision of the abstract interpreter (due to the term w/w_{total}). In case of ties, the term r/r_{max} causes the function to evaluate to a lower cost for recipes that result in a more efficient analysis. In other words, for two recipes resulting in equal precision, the one with the smaller resource consumption is assigned a lower cost. When a recipe causes the analyzer to exceed the resource limit, it is assigned infinite cost.

4.3 Recipe Generation

In the literature, there is a broad range of optimization algorithms for different application domains. To demonstrate the generality and effectiveness of tAIlor, we instantiate it with four adaptations of three well-known optimization algorithms, namely random sampling [38], hill climbing (with regular restarts) [48], and simulated annealing [36,39]. Here, we describe these algorithms in detail, and in Sect. 5, we evaluate their effectiveness.

Before diving into the details, let us discuss the suitability of different kinds of optimization algorithms for our domain. There are algorithms that leverage mathematical properties of the function to be optimized, e.g., by computing derivatives as in Newton's iterative method. Our cost function, however, is evaluated by running an abstract interpreter, and thus, it is not differentiable or continuous. This constraint makes such analytical algorithms unsuitable. Moreover, evaluating our cost function is expensive, especially for precise abstract domains such as Polyhedra. This makes algorithms that require a large number of samples, such as genetic algorithms, less practical.

Now recall that Algorithm 1 is parametric in how new recipes are generated (with GenerateRecipe) and accepted for further mutations (with Accept). Instantiations of these functions essentially constitute our search strategy for a tailored recipe. In the following, we discuss four such instantiations. Note that, in theory, the order of recipe ingredients matters. This is because any properties verified by one ingredient are converted into assumptions for the next, and different assumptions may lead to different verification results. Therefore, all our instantiations are able to explore different ingredient orderings.

Random Sampling. Random sampling (rs) just generates random recipes of a certain length. Function Accept always returns *false* as each recipe is generated from scratch, and not as a result of any mutations.

Domain-Aware Random Sampling. rs might generate recipes containing abstract domains of comparable precision. For instance, the Octagons domain is typically strictly more precise than Intervals. Thus, a recipe consisting of these domains is essentially equivalent to one containing only Octagons.

Now, assume that we have a partially ordered set (poset) of domains that defines their ordering in terms of precision. An example of such a poset for a particular abstract interpreter is shown in Fig. 3. An optimization algorithm can then leverage this information to reduce the search space of possible recipes. Given such a poset, we therefore define domain-aware random sampling (dars), which randomly samples recipes that do not contain abstract domains of comparable precision. Again, Accept always returns *false*.

Simulated Annealing. Simulated annealing (sa) searches for the best recipe by mutating the current recipe rec_{curr} in Algorithm 1. The resulting recipe (rec_{next}), if accepted on line 12, becomes the new recipe to be mutated. Algoirthm 2 shows an instantiation of GenerateRecipe, which mutates a given recipe such that the poset precision constraints are satisfied (i.e., there are no domains of comparable precision). A recipe is mutated either by adding new ingredients with

Algorithm 2: A recipe-generator instantiation.

```
1  Function GenerateRecipe($rec$, $l_{max}$) is
2      $act$ := RandomAction({ADD: 0.2, MOD: 0.8})
3      if $act$ = ADD $\wedge$ Len($rec$) $< l_{max}$ then
4          $ingr_{new}$ := RandomPosetLeastIncomparable($rec$)
5          $rec_{mut}$ := AddIngredient($rec$, $ingr_{new}$)
6      else
7          $ingr$ := RandomIngredient($rec$)
8          $act_m$ := RandomAction({GT: 0.5, LT: 0.3, INC: 0.2})
9          if $act_m$ = GT then
10             $ingr_{new}$ := PosetGreaterThan($ingr$)
11         else if $act_m$ = LT then
12             $ingr_{new}$ := PosetLessThan($ingr$)
13         else
14             $rec_{rem}$ := RemoveIngredient($rec$, $ingr$)
15             $ingr_{new}$ := RandomPosetLeastIncomparable($rec_{rem}$)
16         $rec_{mut}$ := ReplaceIngredient($rec$, $ingr$, $ingr_{new}$)
17     if $\neg$PosetCompatible($rec_{mut}$) then
18         $rec_{mut}$ := GenerateRecipe($rec$, $l_{max}$)
19     return $rec_{mut}$
```

20% probability or by modifying existing ones with 80% probability (line 2). The probability of adding ingredients is lower to keep recipes short.

When adding a new ingredient (lines 4–5), Algorithm 2 calls RANDOMPOSETLEASTINCOMPARABLE, which considers all domains that are incomparable with the domains in the recipe. Given this set, it randomly selects from the domains with the least precision to avoid adding overly expensive domains. When modifying a random ingredient in the recipe (lines 7–16), the algorithm can replace its domain with one of three possibilities: a domain that is immediately more precise (i.e., not transitively) in the poset (via POSETGREATERTHAN), a domain that is immediately less precise (via POSETLESSTHAN), or an incomparable domain with the least precision (via RANDOMPOSETLEASTINCOMPARABLE). If the resulting recipe does not satisfy the poset precision constraints, our algorithm retries to mutate the original recipe (lines 17–18).

For simulated annealing, ACCEPT returns *true* if the new cost (for the mutated recipe) is less than the current cost. It also accepts recipes whose cost is higher with a certain probability, which is inversely proportional to the cost increase and the number of explored recipes. That is, recipes with a small cost increase are likely to be accepted, especially at the beginning of the exploration.

Hill Climbing. Our instantiation of hill climbing (HC) performs regular restarts. In particular, it starts with a randomly generated recipe that satisfies the poset precision constraints, generates 10 new valid recipes, and restarts with a random recipe. ACCEPT returns *true* only if the new cost is lower than the best cost, which is equivalent to the current cost.

5 Experimental Evaluation

To evaluate our technique, we aim to answer the following research questions:

RQ1: Is our technique effective in tailoring recipes to different usage scenarios?
RQ2: Are the tailored recipes optimal?
RQ3: How diverse are the tailored recipes?
RQ4: How resilient are the tailored recipes to code changes?

5.1 Implementation

We implemented tAIlor by extending Crab [25], a parametric framework for modular construction of abstract interpreters[4]. We extended Crab with the ability to pass verification results between recipe ingredients as well as with the four optimization algorithms discussed in Sect. 4.3.

Table 1 shows all settings and values used in our evaluation. The first three settings refer to the strategies discussed in Sect. 3 for mitigating the precision loss incurred by widening. For the initial recipe, tAIlor uses Intervals and the Crab default values for all other settings (in bold in the table). To make the search more efficient, we selected a representative subset of all possible setting values.

Crab uses a DSA-based [26] pointer analysis and can, optionally, reason about array contents using array smashing. It offers a wide range of logico-numerical domains, shown in Fig. 3. The `bool` domain is the flat Boolean domain, `ric` is a reduced product of Intervals and Congruence, and `term(int)` and `term(disInt)` are instantiations of the Term domain with `intervals` and `disInt`, respectively. Although Crab provides a bottom-up inter-procedural analysis, we use the default intra-procedural analysis; in fact, most analyses deployed in real usage scenarios are intra-procedural due to time constraints [10].

5.2 Benchmark Selection

For our evaluation, we systematically selected popular and (at some point) active C projects on GitHub. In particular, we chose the six most starred C repositories

Table 1. Crab settings and their possible values as used in our experiments. Default settings are shown in bold.

Setting	Possible values
NUM_DELAY_WIDEN	$\{\mathbf{1}, 2, 4, 8, 16\}$
NUM_NARROW_ITERATIONS	$\{1, \mathbf{2}, 3, 4\}$
NUM_WIDEN_THRESHOLDS	$\{\mathbf{0}, 10, 20, 30, 40\}$
Backward Analysis	$\{\mathbf{OFF}, \mathit{ON}\}$
Array Smashing	$\{\mathit{OFF}, \mathbf{ON}\}$
Abstract Domains	All domains in Fig. 3

[4] Crab is available at https://github.com/seahorn/crab.

Table 2. Overview of projects.

Project	Description
CURL	Tool for transferring data by URL
DARKNET	Convolutional neural-network framework
FFMPEG	Multimedia processing tool
GIT	Distributed version-control tool
PHP-SRC	PHP interpreter
REDIS	Persistent in-memory database

with over 300 commits that we could successfully build with the Clang-5.0 compiler. We give a short description of each project in Table 2.

For analyzing these projects, we needed to introduce properties to be verified. We, thus, automatically instrumented these projects with four types of assertions that check for common bugs; namely, division by zero, integer overflow, buffer overflow, and use after free. Introducing assertions to check for runtime errors such as these is common practice in program analysis and verification.

As projects consist of different numbers of files, to avoid skewing the results in favor of a particular project, we randomly and uniformly sampled 20 LLVM-bitcode files from each project, for a total of 120. To ensure that each file was neither too trivial nor too difficult for the abstract interpreter, we used the number of assertions as a complexity indicator and only sampled files with at least 20 assertions and at most 100. Additionally, to guarantee all four assertion types were included and avoid skewing the results in favor of a particular assertion type, we required that the sum of assertions for each type was at least 70 across all files—this exact number was largely determined by the benchmarks.

Overall, our benchmark suite of 120 files totals 1346 functions, 5557 assertions (on average 4 assertions per function), and 667927 LLVM instructions (Table 3).

5.3 Results

We now present our experimental results for each research question. We performed all experiments on a 32-core Intel ® Xeon ® E5-2667 v2 CPU @ 3.30 GHz machine with 264 GB of memory, running Ubuntu 16.04.1 LTS.

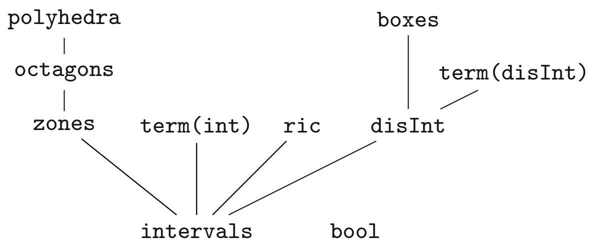


Fig. 3. Comparing logico-numerical domains in Crab. A domain d_1 is less precise than d_2 if there is a path from d_1 to d_2 going upward, otherwise d_1 and d_2 are incomparable.

Table 3. Benchmark characteristics (20 files per project). The last three columns show the number of functions, assertions, and LLVM instructions in the analyzed files.

Project	**Functions**	**Assertions**	**LLVM instructions**
CURL	306	787	50 541
DARKNET	130	958	55 847
FFMPEG	103	888	27 653
GIT	218	768	102 304
PHP-SRC	268	1031	305 943
REDIS	321	1125	125 639
Total	1346	5557	667 927

RQ1: Is Our Technique Effective in Tailoring Recipes to Different Usage Scenarios? We instantiated tAIlor with the four optimization algorithms described in Sect. 4.3: RS, DARS, SA, and HC. We constrained the analysis time to simulate two usage scenarios: 1 s for instant feedback in the editor, and 5 min for feedback in a CI pipeline. We compare tAIlor with the default recipe (DEF), i.e., the default settings in Crab as defined by its designer after careful tuning on a large set of benchmarks over the years. DEF uses a combination of two domains, namely, the reduced product of Boolean and Zones. The other default settings are in Table 1.

For this experiment, we ran tAIlor with each optimization algorithm on the 120 benchmark files, enabling optimization at the granularity of files. Each algorithm was seeded with the same random seed. In Algorithm 1, we restrict recipes to contain at most 3 domains ($l_{max} = 3$) and set the number of iterations for each phase to be 5 and 10 ($i_{dom} = 5$ and $i_{set} = 10$).

The results are presented in Fig. 4, which shows the number of assertions that are verified with the best recipe found by each algorithm as well as by the default recipe. All algorithms outperform the default recipe for both usage scenarios, verifying almost twice as many assertions on average. The random-

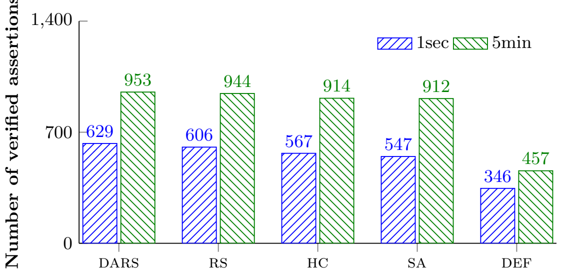


Fig. 4. Comparison of the number of assertions verified with the best recipe generated by each optimization algorithm and with the default recipe, for varying timeouts.

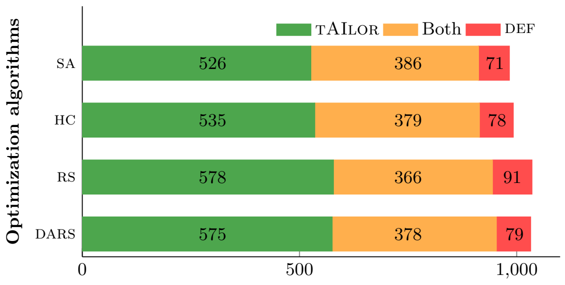


Fig. 5. Comparison of the number of assertions verified by a tailored vs. the default recipe.

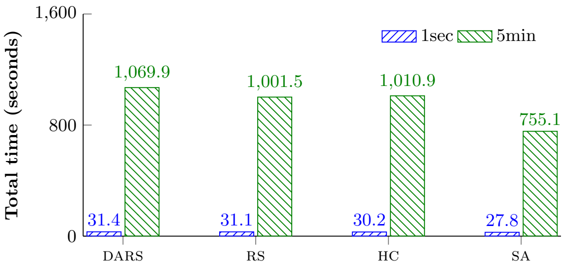


Fig. 6. Comparison of the total time (in sec) that each algorithm requires for all iterations, for varying timeouts.

sampling algorithms are shown to find better recipes than the others, with DARS being the most effective. Hill climbing is less effective since it gets stuck in local cost minima despite restarts. Simulated annealing is the least effective because it slowly climbs up the poset toward more precise domains (see Algorithm 2). However, as we explain later, we expect the algorithms to converge on the number of verified assertions for more iterations.

Figure 5 gives a more detailed comparison with the default recipe for the time limit of 5 min. In particular, each horizontal bar shows the total number of assertions verified by each algorithm. The orange portion represents the assertions verified by both the default recipe and the optimization algorithm, while the green and red portions represent the assertions only verified by the algorithm and default recipe, respectively. These results show that, in addition to verifying hundreds of new assertions, tAIlor is able to verify the vast majority of assertions proved by the default recipe, regardless of optimization algorithm.

In Fig. 6, we show the total time each algorithm takes for all iterations. DARS takes the longest. This is due to generating more precise recipes thanks to its domain knowledge. Such recipes typically take longer to run but verify more assertions (as in Fig. 4). On average, for all algorithms, tAIlor requires only 30 s to complete all iterations for the 1-s timeout and 16 min for the 5-min timeout. As discussed in Sect. 2, this tuning time can be spent offline.

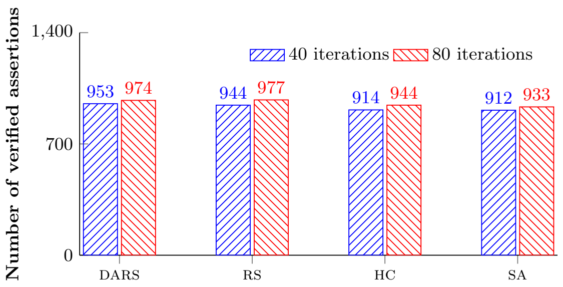


Fig. 7. Comparison of the number of assertions verified with the best recipe generated by the different optimization algorithms, for different numbers of iterations.

Figure 7 compares the total number of assertions verified by each algorithm when tAIlor runs for 40 ($i_{dom} = 5$ and $i_{set} = 10$) and 80 ($i_{dom} = 10$ and $i_{set} = 20$) iterations. The results show that only a relatively small number of additional assertions are verified with 80 iterations. In fact, we expect the algorithms to eventually converge on the number of verified assertions, given the time limit and precision of the available domains.

As DARS performs best in this comparison, we only evaluate DARS in the remaining research questions. We use a 5-min timeout.

RQ1 takeaway: tAIlor verifies between 1.6–2.1× the assertions of the default recipe, regardless of optimization algorithm, timeout, or number of iterations. In fact, even very simple algorithms (such as RS) significantly outperform the default recipe.

RQ2: Are the Tailored Recipes Optimal? To check the optimality of the tailored recipes, we compared them with the most precise (and least efficient) Crab configuration. It uses the most precise domains from Fig. 3 (i.e., `bool`, `polyhedra`, `term(int)`, `ric`, `boxes`, and `term(disInt)`) in a recipe of 6 ingredients and assigns the most precise values to all other settings from Table 1. We generously gave a 30-min timeout to this recipe.

For 21 out of 120 files, the most precise recipe ran out of memory (264 GB). For 86 files, it terminated within 5 min, and for 13, it took longer (within 30 min)—in many cases, this was even longer than tAIlor's tuning time in Fig. 6. We compared the number of assertions verified by our tailored recipes (which do not exceed 5 min) and by the most precise recipe. For the 86 files that terminated within 5 min, our recipes prove 618 assertions, whereas the most precise recipe proves 534. For the other 13 files, our recipes prove 119 assertions, whereas the most precise recipe proves 98.

Consequently, our (in theory) less precise and more efficient recipes prove more assertions in files where the most precise recipe terminates. Possible explanations for this non-intuitive result are: (1) Polyhedra coefficients may overflow, in which case the constraints are typically ignored by abstract interpreters, and

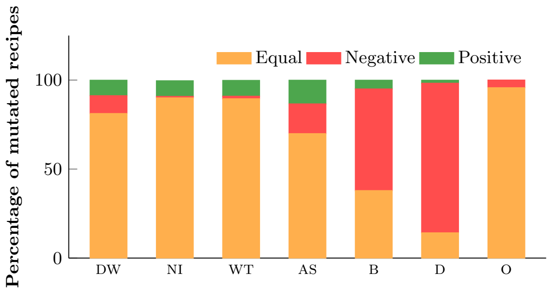


Fig. 8. Effect of different settings on the precision and performance of the abstract interpreter. (DW: `NUM_DELAY_WIDEN`, NI: `NUM_NARROW_ITERATIONS`, WT: `NUM_WIDEN_-THRESHOLDS`, AS: array smashing, B: backward analysis, D: abstract domain, O: ingredient ordering).

(2) more precise domains with different widening operations may result in less precise results [2,45].

We also evaluated the optimality of tailored recipes by mutating individual parts of the recipe and comparing to the original. In particular, for each setting in Table 1, we tried all possible values and replaced each domain with all other comparable domains in the poset of Fig. 3. For example, for a recipe including `zones`, we tried `octagons`, `polyhedra`, and `intervals`. In addition, we tried all possible orderings of the recipe ingredients, which in theory could produce different results. We observed whether these changes resulted in a difference in the precision and performance of the analyzer.

Figure 8 shows the results of this experiment, broken down by setting. Equal (in orange) indicates that the mutated recipe proves the same number of assertions within ± 5 s of the original. Positive (in green) indicates that it either proves more assertions or the same number of assertions at least 5 s faster. Negative (in red) indicates that the mutated recipe either proves fewer assertions or the same number of assertions at least 5 seconds slower.

The results show that, for our benchmarks, mutating the recipe found by TAILOR rarely led to an improvement. In particular, at least 93% of all mutated recipes were either equal to or worse than the original recipe. In the majority of these cases, mutated recipes are equally good. This indicates that there are many optimal or close-to-optimal solutions and that TAILOR is able to find one.

RQ2 takeaway: As compared to the most precise recipe, TAILOR verified more assertions across benchmarks where the most precise recipe terminated. Furthermore, mutating recipes found by TAILOR resulted in improvement only for less than 7% of recipes.

RQ3: How Diverse are the Tailored Recipes? To motivate the need for optimization, we must show that tailored recipes are sufficiently diverse such that they could not be replaced by a well-crafted default recipe. To better understand the characteristics of tailored recipes, we manually inspected all of them.

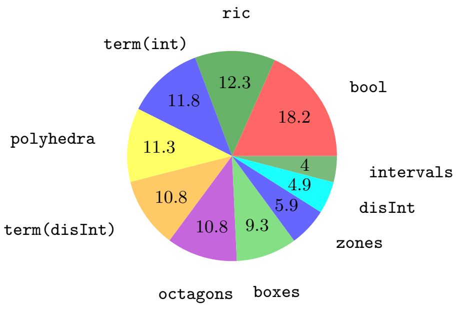


Fig. 9. Occurrence of domains (in %) in the best recipes for all assertion types.

tAIlor generated recipes of length greater than 1 for 61 files. Out of these, 37 are of length 2 and 24 of length 3. For 77% of generated recipes, `NUM_DELAY_WIDEN` is not set to the default value of 1. Additionally, 55% of the ingredients enable array smashing, and 32% enable backward analysis.

Figure 9 shows how often (in percentage) each abstract domain occurs in a best recipe found by tAIlor. We observe that all domains occur almost equally often, with 6 of the 10 domains occurring in between 9% and 13% of recipes. The most common domain was `bool` at 18%, and the least common was `intervals` at 4%. We observed a similar distribution of domains even when instrumenting the benchmarks with only one assertion type, e.g., checking for integer overflow.

We also inspected which domain combinations are frequently used in the tailored recipes. One common pattern is combinations between `bool` and numerical domains (18 occurrences). Similarly, we observed 2 occurrences of `term(disInt)` together with `zones`. Interestingly, the less powerful variants of combining `disInt` with `zones` (3 occurrences) and `term(int)` with `zones` (6 occurrences) seem to be sufficient in many cases. Finally, we observed 8 occurrences of `polyhedra` or `octagons` with `boxes`, which are the most precise convex and non-convex domains. Our approach is, thus, not only useful for users, but also for designers of abstract interpreters by potentially inspiring new domain combinations.

RQ3 takeaway: The diversity of tailored recipes prevents replacing them with a single default recipe. Over half of the tailored recipes contain more than one ingredient, and ingredients use a variety of domains and their settings.

RQ4: How Resilient are the Tailored Recipes to Code Changes? We expect tailored recipes to be resilient to code changes, i.e., to retain their optimality across several changes without requiring re-tuning. We now evaluate if a recipe tailored for one code version is also tailored for another, even when the two versions are 50 commits apart.

For this experiment, we took a random sample of 60 files from our benchmarks and retrieved the 50 most recent commits per file. We only sampled 60 out of

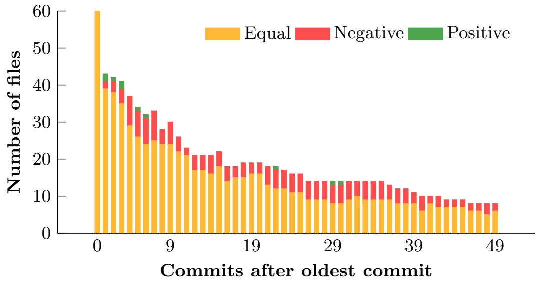


Fig. 10. Difference in the safe assertions across commits.

120 files as building these files for each commit is quite time consuming—it can take up to a couple of days. We instrumented each file version with the four assertion types described in Sect. 5.2. It should be noted that, for some files, we retrieved fewer than 50 versions either because there were fewer than 50 total commits or our build procedure for the project failed on older commits. This is also why we did not run this experiment for over 50 commits.

We analyzed each file version with the best recipe, R_o, found by TAILOR for the oldest file version. We compared this recipe with new best recipes, R_n, that were generated by TAILOR when run on each subsequent file version. For this experiment, we used a 5-min timeout and 40 iterations.

Note that, when running TAILOR with the same optimization algorithm and random seed, it explores the same recipes. It is, therefore, very likely that recipe R_o for the oldest commit is also the best for other file versions since we only explore 40 different recipes. To avoid any such bias, we performed this experiment by seeding TAILOR with a different random seed for each commit. The results are shown in Fig. 10.

In Fig. 10, we give a bar chart comparing the number of files per commit that have a positive, equal, and negative difference in the number of verified assertions, where commit 0 is the oldest commit and 49 the newest. An equal difference (in orange) means that recipe R_o for the oldest commit proves the same number of assertions in the current file version, f_n, as recipe R_n found by running TAILOR on f_n. To be more precise, we consider the two recipes to be equal if they differ by at most 1 verified assertion or 1% of verified assertions since such a small change in the number of safe assertions seems acceptable in practice (especially given that the total number of assertions may change across commits). A positive difference (in green) means that R_o achieves better verification results than R_n, that is, R_o proves more assertions safe (over 1 assertion or 1% of the assertions that R_n proves). Analogously, a negative difference (in red) means that R_o proves fewer assertions. We do not consider time here because none of the recipes timed out when applied on any file version.

Note that the number of files decreases for newer commits. This is because not all files go forward by 50 commits, and even if they do, not all file versions build. However, in a few instances, the number of files increases going forward in time. This happens for files that change names, and later, change back, which we do not account for.

For the vast majority of files, using recipe R_o (found for the oldest commit) is as effective as using R_n (found for the current commit). The difference in safe assertions is negative for less than a quarter of the files tested, with the average negative difference among these files being around 22% (i.e., R_o proved 22% fewer assertions than R_n in these files). On the remaining three quarters of the files tested however, R_o proves at least as many assertions as R_n, and thus, R_o tends to be tailored across code versions.

Commits can result in both small and large changes to the code. We therefore also measured the average difference in the number of verified assertions per changed line of code with respect to the oldest commit. For most files, regardless of the number of changed lines, we found that R_o and R_n are equally effective, with changes to 1000 LOC or more resulting in little to no loss in precision. In particular, the median difference in safe assertions across all changes between R_o and R_n was 0 (i.e., R_o proved the same number of assertions safe as R_n), with a standard deviation of 15 assertions. We manually inspected a handful of outliers where R_o proved significantly fewer assertions than R_n (difference of over 50 assertions). These were due to one file from GIT where R_o is not as effective because the widening and narrowing settings have very low values.

RQ4 takeaway: For over 75% of files, TAILOR's recipe for a previous commit (from up to 50 commits previous) remains tailored for future versions of the file, indicating the resilience of tailored recipes across code changes.

5.4 Threats to Validity

We have identified the following threats to the validity of our experiments.

Benchmark Selection. Our results may not generalize to other benchmarks. However, we selected popular GitHub projects from different application domains (see Table 2). Hence, we believe that our benchmark selection mitigates this threat and increases generalizability of our findings.

Abstract Interpreter and Recipe Settings. For our experiments, we only used a single abstract interpreter, CRAB, which however is a mature and actively supported tool. The selection of recipe settings was, of course, influenced by the available settings in CRAB. Nevertheless, CRAB implements the generic architecture of Fig. 2, used by most abstract interpreters, such as those mentioned at the beginning of Sect. 3. We, therefore, expect our approach to generalize to such analyzers.

Optimization Algorithms. We considered four optimization algorithms, but in Sect. 4.3, we explain why these are suitable for our application domain. Moreover, TAILOR is configurable with respect to the optimization algorithm.

Assertion Types. Our results are based on four types of assertions. However, these cover a wide range of runtime errors that are commonly checked by static analyzers.

6 Related Work

The impact of different abstract-interpretation configurations has been previously evaluated [54] for Java programs and partially inspired this work. To the best of our knowledge, we are the first to propose tailoring abstract interpreters to custom usage scenarios using optimization.

However, optimization is a widely used technique in many engineering disciplines. In fact, it is also used to solve the general problem of algorithm configuration [31], of which there exist numerous instantiations, for instance, to tune hyper-parameters of learning algorithms [3,18,52] and options of constraint solvers [32,33]. Existing frameworks for algorithm configuration differ from ours in that they are not geared toward problems that are solved by sequences of algorithms, such as analyses with different abstract domains. Even if they were, our experience with tAIlor shows that there seem to be many optimal or close-to-optimal configurations, and even very simple optimization algorithms such as rs are surprisingly effective (see RQ2); similar observations were made about the effectiveness of random search in hyper-parameter tuning [4].

In the rest of this section, we focus on the use of optimization in program analysis. It has been successfully applied to a number of program-analysis problems, such as automated testing [19,20], invariant inference [50], and compiler optimizations [49].

Recently, researchers have started to explore the direction of enriching program analyses with machine-learning techniques, for example, to automatically learn analysis heuristics [27,34,47,51]. A particularly relevant body of work is on adaptive program analysis [28–30], where existing code is analyzed to learn heuristics that trade soundness for precision or that coarsen the analysis abstractions to improve memory consumption. More specifically, adaptive program analysis poses different static-analysis problems as machine-learning problems and relies on Bayesian optimization to solve them, e.g., the problem of selectively applying unsoundness to different program components (e.g., different loops in the program) [30]. The main insight is that program components (e.g., loops) that produce false positives are alike, predictable, and share common properties. After learning to identify such components for existing code, this technique suggests components in unseen code that should be analyzed unsoundly.

In contrast, tAIlor currently does not adjust soundness of the analysis. However, this would also be possible if the analyzer provided the corresponding configurations. More importantly, adaptive analysis focuses on learning analysis heuristics based on existing code in order to generalize to arbitrary, unseen code. tAIlor, on the other hand, aims to tune the analyzer configuration to a custom usage scenario, including a particular program under analysis. In addition, the custom usage scenario imposes user-specific resource constraints, for instance by limiting the time according to a phase of the software-engineering life cycle. As we show in our experiments, the tuned configuration remains tailored to several versions of the analyzed program. In fact, it outperforms configurations that are meant to generalize to arbitrary programs, such as the default recipe.

7 Conclusion

In this paper, we have proposed a technique and framework that tailors a generic abstract interpreter to custom usage scenarios. We instantiated our framework with a mature abstract interpreter to perform an extensive evaluation on real-world benchmarks. Our experiments show that the configurations generated by tAIlor are vastly better than the default options, vary significantly depending on the code under analysis, and typically remain tailored to several subsequent code versions. In the future, we plan to explore the challenges that an inter-procedural analysis would pose, for instance, by using a different recipe for computing a summary of each function or each calling context.

Acknowledgements. We are grateful to the reviewers for their constructive feedback. This work was supported by DFG grant 389792660 as part of TRR 248 (see https://perspicuous-computing.science). Jorge Navas was supported by NSF grant 1816936.

References

1. The BDDAPRON logico-numerical abstract domains library. http://www.inrialpes.fr/pop-art/people/bjeannet/bjeannet-forge/bddapron
2. Amato, G., Rubino, M.: Experimental evaluation of numerical domains for inferring ranges. ENTCS **334**, 3–16 (2018)
3. Bergstra, J., Bardenet, R., Bengio, Y., Kégl, B.: Algorithms for hyper-parameter optimization. In: NIPS, pp. 2546–2554 (2011)
4. Bergstra, J., Bengio, Y.: Random search for hyper-parameter optimization. JMLR **13**, 281–305 (2012)
5. Blanchet, B., et al.: A static analyzer for large safety-critical software. In: PLDI, pp. 196–207. ACM (2003)
6. Brat, G., Navas, J.A., Shi, N., Venet, A.: IKOS: a framework for static analysis based on abstract interpretation. In: Giannakopoulou, D., Salaün, G. (eds.) SEFM 2014. LNCS, vol. 8702, pp. 271–277. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10431-7_20
7. Calcagno, C., Distefano, D.: Infer: an automatic program verifier for memory safety of C programs. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 459–465. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_33
8. Calcagno, C., et al.: Moving fast with software verification. In: Havelund, K., Holzmann, G., Joshi, R. (eds.) NFM 2015. LNCS, vol. 9058, pp. 3–11. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-17524-9_1
9. Chang, B.-Y.E., Leino, K.R.M.: Abstract interpretation with alien expressions and heap structures. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 147–163. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30579-8_11

10. Christakis, M., Bird, C.: What developers want and need from program analysis: an empirical study. In: ASE, pp. 332–343. ACM (2016)
11. Cousot, P., Cousot, R.: Static determination of dynamic properties of programs. In: ISOP, pp. 106–130. Dunod (1976)
12. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, pp. 238–252. ACM (1977)
13. Cousot, P., Cousot, R.: Abstract interpretation and application to logic programs. JLP **13**, 103–179 (1992)
14. Cousot, P., Cousot, R.: Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In: Bruynooghe, M., Wirsing, M. (eds.) PLILP 1992. LNCS, vol. 631, pp. 269–295. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-55844-6_142
15. Cousot, P., Cousot, R.: Refining model checking by abstract interpretation. Autom. Softw. Eng. **6**, 69–95 (1999)
16. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL, pp. 84–96. ACM (1978)
17. Fähndrich, M., Logozzo, F.: Static contract checking with abstract interpretation. In: Beckert, B., Marché, C. (eds.) FoVeOOS 2010. LNCS, vol. 6528, pp. 10–30. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18070-5_2
18. Falkner, S., Klein, A., Hutter, F.: BOHB: robust and efficient hyperparameter optimization at scale. In: ICML. PMLR, vol. 80, pp. 1436–1445. PMLR (2018)
19. Fu, Z., Su, Z.: Mathematical execution: a unified approach for testing numerical code. CoRR abs/1610.01133 (2016)
20. Fu, Z., Su, Z.: Achieving high coverage for floating-point code via unconstrained programming. In: PLDI, pp. 306–319. ACM (2017)
21. Gange, G., Navas, J.A., Schachte, P., Søndergaard, H., Stuckey, P.J.: An abstract domain of uninterpreted functions. In: Jobstmann, B., Leino, K.R.M. (eds.) VMCAI 2016. LNCS, vol. 9583, pp. 85–103. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49122-5_4
22. Gershuni, E., et al.: Simple and precise static analysis of untrusted Linux kernel extensions. In: PLDI, pp. 1069–1084. ACM (2019)
23. Granger, P.: Static analysis of arithmetical congruences. Int. J. Comput. Math. **30**, 165–190 (1989)
24. Gurfinkel, A., Chaki, S.: Boxes: a symbolic abstract domain of boxes. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 287–303. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15769-1_18
25. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 343–361. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_20
26. Gurfinkel, A., Navas, J.A.: A context-sensitive memory model for verification of C/C++ programs. In: Ranzato, F. (ed.) SAS 2017. LNCS, vol. 10422, pp. 148–168. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66706-5_8
27. Heo, K., Oh, H., Yang, H.: Learning a variable-clustering strategy for octagon from labeled data generated by a static analysis. In: Rival, X. (ed.) SAS 2016. LNCS, vol. 9837, pp. 237–256. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53413-7_12
28. Heo, K., Oh, H., Yang, H.: Resource-aware program analysis via online abstraction coarsening. In: ICSE, pp. 94–104. IEEE Computer Society/ACM (2019)

29. Heo, K., Oh, H., Yang, H., Yi, K.: Adaptive static analysis via learning with Bayesian optimization. TOPLAS **40**, 14:1–14:37 (2018)
30. Heo, K., Oh, H., Yi, K.: Machine-learning-guided selectively unsound static analysis. In: ICSE, pp. 519–529. IEEE Computer Society/ACM (2017)
31. Hutter, F.: Automated Configuration of Algorithms for Solving Hard Computational Problems. Ph.D. thesis, The University of British Columbia, Canada (2009)
32. Hutter, F., Babic, D., Hoos, H.H., Hu, A.J.: Boosting verification by automatic tuning of decision procedures. In: FMCAD, pp. 27–34. IEEE Computer Society (2007)
33. Hutter, F., Hoos, H.H., Stützle, T.: Automatic algorithm configuration based on local search. In: AAAI, pp. 1152–1157. AAAI (2007)
34. Jeong, S., Jeon, M., Cha, S.D., Oh, H.: Data-driven context-sensitivity for points-to analysis. PACMPL **1**, 100:1–100:28 (2017)
35. Karr, M.: Affine relationships among variables of a program. Acta Inf. **6**, 133–151 (1976)
36. Kirkpatrick, S., Gelatt, C.D., Jr., Vecchi, M.P.: Optimization by simulated annealing. Science **220**, 671–680 (1983)
37. Lakhdar-Chaouch, L., Jeannet, B., Girault, A.: Widening with thresholds for programs with complex control graphs. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 492–502. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24372-1_38
38. Mátyáš, I.: Random optimization. Avtomat. i Telemekh. **26**, 246–253 (1965)
39. Metropolis, N., Rosenbluth, A.W., Rosenbluth, M.N., Teller, A.H., Teller, E.: Equation of state calculations by fast computing machines. J. Chem. Phys. **21**, 1087–1092 (1953)
40. Mihaila, B., Sepp, A., Simon, A.: Widening as abstract domain. In: Brat, G., Rungta, N., Venet, A. (eds.) NFM 2013. LNCS, vol. 7871, pp. 170–184. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38088-4_12
41. Miné, A.: A few graph-based relational numerical abstract domains. In: Hermenegildo, M.V., Puebla, G. (eds.) SAS 2002. LNCS, vol. 2477, pp. 117–132. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45789-5_11
42. Miné, A.: Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In: LCTES, pp. 54–63. ACM (2006)
43. Miné, A.: The Octagon abstract domain. HOSC **19**, 31–100 (2006)
44. Miné, A.: Symbolic methods to enhance the precision of numerical abstract domains. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 348–363. Springer, Heidelberg (2005). https://doi.org/10.1007/11609773_23
45. Monniaux, D., Le Guen, J.: Stratified static analysis based on variable dependencies. ENTCS **288**, 61–74 (2012)
46. Oh, H., Heo, K., Lee, W., Lee, W., Yi, K.: Design and implementation of sparse global analyses for C-like languages. In: PLDI, pp. 229–238. ACM (2012)
47. Raychev, V., Vechev, M.T., Krause, A.: Predicting program properties from 'big code'. CACM **62**, 99–107 (2019)
48. Russell, S.J., Norvig, P.: Artificial Intelligence: A Modern Approach. Pearson Education (2010)
49. Schkufza, E., Sharma, R., Aiken, A.: Stochastic superoptimization. In: ASPLOS, pp. 305–316. ACM (2013)
50. Sharma, R., Aiken, A.: From invariant checking to invariant inference using randomized search. In: CAV. LNCS, vol. 8559, pp. 88–105. Springer (2014)

51. Singh, G., Püschel, M., Vechev, M.: Fast numerical program analysis with reinforcement learning. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 211–229. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_12
52. Thornton, C., Hutter, F., Hoos, H.H., Leyton-Brown, K.: Auto-WEKA: combined selection and hyperparameter optimization of classification algorithms. In: KDD, pp. 847–855. ACM (2013)
53. Venet, A., Brat, G.P.: Precise and efficient static array bound checking for large embedded C programs. In: PLDI, pp. 231–242. ACM (2004)
54. Wei, S., Mardziel, P., Ruef, A., Foster, J.S., Hicks, M.: Evaluating design tradeoffs in numeric static analysis for Java. In: ESOP. LNCS, vol. 10801, pp. 653–682. Springer (2018)

Functional Correctness of C Implementations of Dijkstra's, Kruskal's, and Prim's Algorithms

Anshuman Mohan(✉), Wei Xiang Leow, and Aquinas Hobor

School of Computing, National University of Singapore, Singapore, Republic of Singapore
amohan@cs.cornell.edu

Abstract. We develop machine-checked verifications of the full functional correctness of C implementations of the eponymous graph algorithms of Dijkstra, Kruskal, and Prim. We extend Wang *et al.*'s CertiGraph platform to reason about labels on edges, undirected graphs, and common spatial representations of edge-labeled graphs such as adjacency matrices and edge lists. We certify binary heaps, including Floyd's bottom-up heap construction, heapsort, and increase/decrease priority.

Our verifications uncover subtle overflows implicit in standard textbook code, including a nontrivial bound on edge weights necessary to execute Dijkstra's algorithm; we show that the intuitive guess fails and provide a workable refinement. We observe that the common notion that Prim's algorithm requires a connected graph is wrong: we verify that a standard textbook implementation of Prim's algorithm can compute minimum spanning forests without finding components first. Our verification of Kruskal's algorithm reasons about two graphs simultaneously: the undirected graph undergoing MSF construction, and the directed graph representing the forest inside union-find. Our binary heap verification exposes precise bounds for the heap to operate correctly, avoids a subtle overflow error, and shows how to recycle keys to avoid overflow.

Keywords: Separation logic · Graph algorithms · Coq · VST

1 Introduction

Dijkstra's eponymous shortest-path algorithm [22] finds the cost-minimal paths from a distinguished *source* vertex to all reachable vertices in a directed graph. Prim's [61] and Kruskal's [42] algorithms return minimal spanning trees for undirected graphs. Binary heaps are the first priority queue one typically encounters. These algorithms/structures are classic and ubiquitous, appearing widely in textbooks [20,33,36,65,66,68] and in real routing protocol libraries.

In addition to decades of use and textbook analysis, recent efforts have verified one or more of these algorithms in proof assistants and formally proved claims about their behavior [12,15,30,45,53]. A reasonable person might think that all that can be said, has been. However, we have found that textbook code glosses over a cornucopia of issues that routinely crop up in real-world settings: under/overflows, integration with performant data structures, manual memory (de-)allocation, error handling, casts, memory alignment, *etc.* Further, previous verification efforts with formal checkers often operate within idealized formal environments, which likewise leads them to ignore the same kinds of issues.

In our work, we provide C implementations of each of these algorithms/data structures, and prove in Coq [71] the functional correctness of the same with respect to the formal semantics of CompCert C [50]. By "functional correctness" we mean natural algorithmic specifications; we do not prove resource bounds. Although our C code is developed from standard textbooks, we uncover several subtleties that are absent from the algorithmic and formal methods literature:

- §3.2 an overflow in Dijkstra's algorithm, avoiding which requires a nontrivial refinement to the algorithm's precondition to bound edge weights;
- §4.2 that the specification of Prim's algorithm can be improved to apply to disconnected graphs without any change to textbook (pseudo-)code;
- §4.2 the presence of a wholly unneeded line of (pseduo-)code in Prim's algorithm, and an associated unneeded function argument;
- §5 several potential overflows in binary heaps equipped with Floyd's linear-time build-heap function and an edit-priority operation.

We wish to develop general and reusable techniques for verifying graph-manipulating programs written in real programming languages. This is a significant challenge, and so we choose to leverage and/or extend three large existing proof developments to state and prove the full functional correctness of our code in Coq: CompCert; the Verified Software Toolchain [4] (VST) separation logic [59] deductive verifier; and our own previous efforts [73], hereafter dubbed the CertiGraph project. Our primary extensions are to the third, and include:

- §2.1 pure/abstract reasoning for graphs with edge labels, (*e.g.*, a distinguished edge-label value for "infinity" that indicates invalid/absent edges);
- §2.2 spatial representations and associated reasoning for edge-labeled graphs (several flavors of adjacency matrices as well as edge lists);
- §2.3 pure reasoning for undirected graphs (*e.g.*, notions of connectedness).

We prove that our pure machinery and our spatial machinery are well-isolated from each other by verifying several implementations (of each of Dijkstra and Prim) that represent graphs differently in memory but reuse the entire pure portion of the proof. Likewise, we show that our spatial reasoning is generic by reusing graph representations across Dijkstra and Prim. Our verification of Kruskal proves that we can reason about two graphs simultaneously: a directed graph with vertex labels for union-find and an undirected graph with edge labels for which we are building a spanning forest. In addition to our verification of

Dijkstra, Prim, and Kruskal, we develop increased lemma support for the preexisting CertiGraph union-find example [73]. Our extension to "base VST" (*e.g.*, verifications without graphs) primarily consists of our verified binary heap.

The remainder of this paper is organized as follows:

- §2 We explain our extensions to CertiGraph: edge-labeled graphs, spatial representations of such graphs, and undirected graphs.
- §3 We explain our verification of Dijkstra's algorithm in some detail, discuss a potential overflow, and refine the precondition to avoid it.
- §4 We overview our verifications of the Minimum Spanning Tree/Forest algorithms of Prim and Kruskal, focusing on high-level points such as our improved novel specification of Prim's.
- §5 We overview our verification of binary heaps, including a discussion of Floyd's bottom-up heap construction and the `edit_priority` operation.
- §6 We briefly discuss engineering, *e.g.* statistics for our formal development.
- §7 We discuss related work, outline future research directions, and conclude.

Our results are completely machine-checked in Coq and publicly available [1].

2 Extensions to CertiGraph

We begin with the briefest of introductions to CertiGraph's core structure and then detail the extensions we make to various levels of CertiGraph in service of our Dijkstra, Prim, and Kruskal verifications. Ignoring modularity and eliding elements not used in this work, a mathematical graph in CertiGraph is a tuple: ($\mathcal{V}$, $\mathcal{E}$, `vvalid`, `evalid`, `src`, `dst`, `vlabel`, `elabel`, `sound`). Here $\mathcal{V}/\mathcal{E}$ are the carrier types of vertices/edges, `vvalid`/`evalid` place restrictions specifying whether a vertex/edge is valid[1], and `src`/`dst` : $\mathcal{E} \rightarrow \mathcal{V}$ map edges to their source/destination. Labels are allowed on vertices and edges, and a `sound`ness condition allows custom application-specific restrictions [73]. Mathematical graphs connect to graphs in computer memory via spatial predicates in separation logic.

2.1 Pure Reasoning for Adjacency Matrix-Represented Graphs

Two of our algorithms operate over graphs represented as adjacency matrices. Not every legal graph can be represented as an adjacency matrix, so we develop a unified, reusable, and extendable `sound`ness condition `SoundAdjMat` that a graph must satisfy in order for it to be represented as an adjacency matrix.

`SoundAdjMat` is parameterized by the graph's `size` and a distinguished number `inf`. We restrict most fields in the tuple: ($\mathcal{V} = \mathbb{Z}$, $\mathcal{E} = \mathbb{Z} \times \mathbb{Z}$, `vvalid` $= \lambda v.\ 0 \leq v <$ `size`, `evalid` $= \ldots$, `src` $=$ *fst*, `dst` $=$ *snd*, `vlabel`, `elabel`, `sound` $= \ldots$). We also restrict the carrier type of vertex labels to `unit`

[1] Validity denotes presence in the graph: *e.g.*, if we are using $\mathbb{Z}$ as the carrier type $\mathcal{V}$, and have only 7 vertices, then `vvalid`(x) is probably the proposition $0 \leq x < 7$).

and edge labels to $\mathbb{Z}$. We require the parameters `size` and `inf` be strictly positive and representable on the machine. Most critical, however, is the semantics of `evalid`: a valid edge must have a machine-representable label and that label cannot have value `inf`; an invalid edge *must* have label `inf`. Last, the graph must be finite.

The restriction on edge labels is necessary because we are working with labeled adjacency matrices on a real system: we need to set aside a distinguished number `inf` such that edgeweight `inf` indicates the *absence* of an edge. We cannot prescribe some `inf` because client needs can vary widely. For instance, our verifications of Dijkstra's and Prim's algorithms require subtly different `inf`s.

`SoundAdjMat` guarantees spatial representability as an adjacency matrix, but it can be extended with further algorithm-specific restrictions before being plugged in for `sound`. Dijkstra's algorithm requires nonnegative edge weights, and—as we will discuss in §3.2—nontrivial restrictions on `size` and `inf`.

2.2 New Spatial Representations for Edge-Labeled Graphs

We give predicates for adjacency matrices and edge lists for edge-labeled graphs.

Adjacency Matrices. Adjacency matrices enable efficient label access for edge-labeled graphs. We support three common adjacency matrix representations: a stack-allocated 2D array `int graph[size][size]`, a stack-allocated 1D array `int graph[size×size]`, and a heap-allocated 2D array `int **graph`. To the casual observer, these are essentially interchangeable, but that is a mistake when thinking spatially. Apart from the arithmetic that the second flavor uses to access cells, there is a more subtle point: the first and second enjoy a contiguous block of memory, but the third does not: it is an allocated "spine" with pointers to separately-allocated rows. For a taste, the spatial representation of the first is:

$$\begin{aligned}
\mathit{arr\_addr}(\mathit{ptr}, i, \texttt{size}) &\triangleq \mathit{ptr} + (i \times \texttt{size})\\
\mathsf{array}(\mathit{ptr}, \mathit{list}) &\triangleq \mathop{\Large\ast}_{i \in [0, |\mathit{list}|)} (\mathit{ptr} + i) \mapsto \mathit{list}[i])\\
\mathsf{arr\_rep}(\gamma, i, \mathit{ptr}) &\triangleq \texttt{let } \mathit{row} \texttt{ := } \mathsf{graph2mat}(\gamma)[i] \texttt{ in}\\
&\qquad \mathsf{array}(\mathit{arr\_addr}(\mathit{ptr}, i, |\mathit{row}|), \mathit{row})\\
\mathsf{graph\_rep}(\gamma, \mathit{g\_addr}, \_) &\triangleq \mathop{\Large\ast}_{v \in \gamma} \mathsf{arr\_rep}(\gamma, v, \mathit{g\_addr})
\end{aligned}$$

We use the separation logic $*$ in its iterated form to say that the arrays are separate in memory. We elide details relating to object sizes, pointer alignment, and so forth, although our formal proofs handle such matters. Of particular note are graph2mat, which performs two projections to drag out the graph's nested edge labels into a 2D matrix, and *arr_addr*, which in this instance simply computes the address of any legal row i from the base address of the graph. Notice that this graph_rep predicate ignores its third argument. To represent a heap-allocated 2D array we can still use graph2mat but can no longer use address arithmetic; the third parameter is then a list of pointers to the row sub-arrays.

While ironing out these spatial wrinkles, we develop utilities that easily unfold and refold our adjacency matrices, thus smoothing user experience when reading and writing arrays and cells. Of course these utilities themselves vary by flavor of representation, but the net effect is that users of our adjacency matrices really can be agnostic to the style of representation they are using (see §3.1).

Edge Lists. Edge lists are the representation of choice for sparse graphs. Our C implementation defines an `edge` as a `struct` containing `src`, `dst`, and `weight`, and defines a `graph` as a `struct` containing the graph's size, edge count, and an array of `edges`. Our spatial representation follows this pattern:

$$\mathsf{graph\_rep}(\gamma, g\_addr, e\_addr) \overset{\Delta}{=} \big(g\_addr \mapsto (|\gamma.V|, |\gamma.E|, e\_addr)\big) * \mathsf{array}(e\_addr, \gamma.E)$$

2.3 Undirectedness in a Directed World

The CertiGraph library presented in [73] supports only directed graphs, and, as we have seen, bakes direction-reliant idioms such as `src` and `dst` deep into its development. Our challenge is to add support for undirected graphs atop of this.

Our approach is to observe that every directed graph can be treated as an undirected graph by ignoring edge direction. We develop a lightweight layer of "undirected flavored" definitions atop of the existing "directed flavored" definitions, state and prove connections between these, and then build the undirected infrastructure we need. The result is that we retain full access to CertiGraph's graph theory formalizations modulo some mathematical bridging.

Our basic "undirected flavored" definitions are standard [20]. Vertices u and v are `adjacent` if there is an edge between them in either direction; vertices are self-adjacent. A valid `upath` (undirected path) is list of valid vertices that form a pairwise-adjacent chain. Two vertices are `connected` when a valid `upath` features them as head and foot (essentially the transitive closure of `adjacenct`).

The definitions above sync up with preexisting "directed flavored" definitions. Intuitively, undirectedness is more lax than directedness, and so it is unsurprising that these connections are straightforward weakenings of directed properties. We next give standard definitions [20] that culminate in `minimum_spanning_forest`, which is exactly our postcondition of both Prim's and Kruskal's algorithms.[2]

An undirected cycle (`ucycle`) is a valid non-empty `upath` whose first and last vertices are equal. A `connected_graph` means that any two valid vertices are `connected`. `is_partial_graph f g` means everything in `f` is in `g`. We proceed:

```
Definition uforest g :=
 (∀ e, evalid g e → strong_evalid g e) ∧
 (∀ p l, ¬ucycle g p l).
Definition spanning g g' :=
 ∀ u v, connected g u v ↔ connected g' u v.
```

[2] That Prim's postcondition has a *forest* may raise an eyebrow. See §4.2.

```
Definition spanning_uforest f g :=
  is_partial_graph f g ∧ uforest f ∧ spanning f g.
```

The `strong_evalid` predicate means that the `src` and `dst` of the edge are also valid, so *e.g.*, a valid edge cannot point to a deleted/absent vertex. The second conjunct of `uforest` is critical: a forest has no undirected cycles. The other definitions are straightforward from there, and `minimum_spanning_forest f g` means that no other spanning forest has lower total edge cost than `f`.

Our undirected work is also compatible with our new developments in §2.1 and §2.2. An adjacency matrix-representable undirected graph has all the pure properties discussed in `SoundAdjMat`, and also has symmetry across the left diagonal. We extend `SoundAdjMat` into `SoundUAdjMat` by requiring that all valid edges have `src` ≤ `dst`. This effectively "turns off" the matrix on one half of the diagonal and avoids double-counting. Prim's algorithm uses `SoundUAdjMat` and places no further restrictions. Further, spatial representations and fold/unfold utilities are shared across directed and undirected adjacency matrices.

3 Shortest Path

We verify a standard C implementation of Dijkstra's algorithm. We first sketch our proof in some detail with an emphasis on our loop invariants, then uncover and remedy a subtle overflow bug, and finish with a discussion of related work.

3.1 Verified Dijkstra's Algorithm in C

Figure 1 shows the code and proof sketch of Dijkstra's algorithm. Red text is used in the figure to highlight changes compared to the annotation immediately prior. Our code is implemented exactly as suggested by CLRS [20], so we refer readers there for a general discussion of the algorithm. The adjacency-matrix-represented graph γ of `size` vertices is passed as the parameter `g` along with the source vertex `src` and two allocated arrays `dist` and `prev`. The spatial predicate $\mathsf{array}(\mathtt{x}, \boldsymbol{v})$, which connects an array pointer `x` with its contents $\boldsymbol{v}$, is standard and unexciting. $\mathsf{PQ}(\mathsf{pq}, \mathit{heap})$ is the spatial representation of our priority queue (PQ) and $\mathsf{Item}(\mathtt{i}, (\mathit{key}, \mathit{pri}, \mathit{data}))$ lays out a struct that we use to interact with the PQ; we leave the management of the PQ to the operations described in§ 5. Of greater interest is $\mathsf{AdjMat}(\mathtt{g}, \gamma)$, which as explained in §2.2, links the concrete memory values of `g` to an abstract mathematical graph γ, which in turn exposes an interface in the language of graph theory (*e.g.*, vertices, edges, labels). Graph γ contains the general adjacency matrix restrictions given in §2.1 along with some further Dijkstra-specific restrictions to be explained in §3.2. We verify Dijkstra three times using different adjacency-matrix representations as explained in §2.2. Thanks to some careful engineering, the C code and the Coq verification are both almost completely agnostic to the form of representation. The only variation between implementations is when reading a cell (line 15), so we refactor this out into a straightforward helper method and verify it separately; accordingly, the proof bases for the three variants differ by less than 1%.

```
 1 void dijkstra (int **g, int src, int *dist,
 2               int *prev, int size, int inf {
 3 // $\{\mathsf{AdjMat}(\mathtt{g},\gamma) * \mathsf{array}(\mathtt{dist},\_) * \mathsf{array}(\mathtt{prev},\_) \wedge \mathtt{src} \in \gamma \wedge \mathit{connected}(\gamma,\mathtt{src})\}$
 4  Item* temp = (Item*) mallocN(sizeof(Item));
 5  int* keys = mallocN (size * sizeof (int));
 6  PQ* pq = pq_make(size); int i, u, cost;
 7  for (i = 0; i < size; i++)
 8  { dist[i] = inf; prev[i] = inf; keys[i] = pq_push(pq,inf,i); }
 9  dist[src]= 0; prev[src]= src; pq_edit_priority(pq,keys[src],0);
10  while (pq_size(pq) > 0) {
11 // $\left\{\begin{array}{l}\exists \mathit{dist}, \mathit{prev}, \mathit{popped}, \mathit{heap}.\ \mathsf{AdjMat}(\mathtt{g},\gamma) * \mathsf{PQ}(\mathtt{pq},\mathit{heap}) * \mathsf{Item}(\mathtt{temp},\_) * \\ \mathsf{array}(\mathtt{dist},\mathit{dist}) * \mathsf{array}(\mathtt{prev},\mathit{prev}) * \mathsf{array}(\mathtt{keys},\mathit{keys}) \wedge \\ \mathit{linked\_correctly}(\gamma,\mathit{heap},\mathit{keys},\mathit{dist},\mathit{popped}) \wedge \\ \mathit{dijk\_correct}(\gamma,\mathtt{src},\mathit{popped},\mathit{prev},\mathit{dist})\end{array}\right\}$
12   pq_pop(pq, temp); u = temp->data;
13   for (i = 0; i < size; i++) {
14 // $\left\{\begin{array}{l}\exists \mathit{dist}', \mathit{prev}', \mathit{heap}'.\ \mathsf{AdjMat}(\mathtt{g},\gamma) * \mathsf{PQ}(\mathtt{pq},\mathit{heap}') * \\ \mathsf{array}(\mathtt{dist},\mathit{dist}') * \mathsf{array}(\mathtt{prev},\mathit{prev}') * \mathsf{array}(\mathtt{keys},\mathit{keys}) * \\ \mathsf{Item}(\mathtt{temp},(\mathtt{keys[u]},\mathtt{dist[u]},\mathtt{u})) \wedge \mathit{min}(\mathtt{dist[u]},\mathit{heap}') \wedge \\ \mathit{linked\_correctly}(\gamma,\mathit{heap}',\mathit{keys},\mathit{dist}',\mathit{popped} \uplus \{\mathtt{u}\}) \wedge \\ \mathit{dijk\_correct\_weak}(\gamma,\mathtt{src},\mathit{popped} \uplus \{\mathtt{u}\},\mathit{prev}',\mathit{dist}',\mathtt{i},\mathtt{u})\end{array}\right\}$
15    cost = getCell(g, u, i);
16    if (cost < inf) {
17     if (dist[i] > dist[u] + cost) {
18      dist[i] = dist[u] + cost; prev[i] = u;
19      pq_edit_priority(pq, keys[i], dist[i]);
20   }}}} // $\left\{\begin{array}{l}\exists \mathit{dist}'', \mathit{prev}''.\ \mathsf{AdjMat}(\mathtt{g},\gamma) * \mathsf{PQ}(\mathtt{pq},\emptyset) * \mathsf{Item}(\mathtt{temp},\_) * \\ \mathsf{array}(\mathtt{dist},\mathit{dist}'') * \mathsf{array}(\mathtt{prev},\mathit{prev}'') * \mathsf{array}(\mathtt{keys},\mathit{keys}) \wedge \\ \forall \mathit{dst}.\ \mathit{dst} \in \gamma \rightarrow \mathit{inv\_popped}(\gamma,\mathit{src},\gamma.V,\mathit{prev}'',\mathit{dist}'',\mathit{dst})\end{array}\right\}$
21  freeN (temp); pq_free (pq); freeN (keys); return; }
```

Fig. 1. C code and proof sketch for Dijkstra's algorithm.

Dijkstra's algorithm uses a PQ to greedily choose the cheapest unoptimized vertex on line 12. The best-known distances to vertices are expected to improve as various edges are relaxed, and such improvements need to be logged in the PQ: Dijkstra's algorithm implicitly assumes that its PQ supports the additional operation `decrease_priority`. Our "advanced" PQ (§5.3) supports this operation in logarithmic time with the `pq_edit_priority` function[3].

The first nine lines are standard setup. The *keys* array, assigned on line 8, is thereafter a mathematical constant. The pure predicate *linked_correctly* contains the plumbing connecting the various mathematical arrays. The verification turns on the loop invariants on lines 11 and 14. The pure `while` invariant

[3] Because `decrease_priority` is relatively complex to implement, several popular workarounds (*e.g.* [12]) use simpler PQs at the cost of decreased performance.

$\textit{dijk\_correct}(\gamma, \textit{src}, \textit{popped}, \textit{prev}, \textit{dist})$ essentially unfolds into:

$$\begin{aligned}\forall \textit{dst}.\ \textit{dst} \in \gamma \rightarrow\ & \textit{inv\_popped}(\gamma, \textit{src}, \textit{popped}, \textit{prev}, \textit{dist}, \textit{dst}) \wedge \\ & \textit{inv\_unpopped}(\gamma, \textit{src}, \textit{popped}, \textit{prev}, \textit{dist}, \textit{dst}) \wedge \\ & \textit{inv\_unseen}(\gamma, \textit{src}, \textit{popped}, \textit{prev}, \textit{dist}, \textit{dst})\end{aligned}$$

That is, a destination vertex *dst* falls into one of three categories:

1. *inv_popped*: if $\textit{dst} \in \textit{popped}$, then *dst* has been fully processed, *i.e.*, *dst* is reachable from *src* via a globally-optimal path *p* whose vertices are all in *popped*. Path *p* has been logged in *prev* and *p*'s cost is given in *dist*.
2. *inv_unpopped*: if $\textit{dst} \notin \textit{popped}$, but its known *dist*ance is less than `inf`, then *dst* is reachable in one step from a popped vertex *mom*. This route is locally optimal: we cannot improve the cost via an alternate popped vertex. Moreover, *prev* logs *mom* as the best-known way to reach *dst*, and *dist* logs the path cost via *mom* as the best-known cost.
3. *inv_unseen*: if $\textit{dst} \notin \textit{popped}$ and its known *dist*ance is `inf`, then there is no edge from any *popped* vertex to *dst*; in other words, *dst* is located deeper in the graph than has yet been explored.

After line 12, the above invariant is no longer true: a minimum-cost item *u* has been popped from the PQ, and so the *dist* and *prev* arrays need to be updated to account for this pop. The `for` loop does exactly this repair work. Its pure invariant $\textit{dijk\_correct\_weak}(\gamma, \textit{src}, \textit{popped}, \textit{prev}, \textit{dist}, u, i)$ essentially unfolds into:

$$\begin{aligned}&\big(\forall \textit{dst}.\ \textit{dst} \in \gamma && \rightarrow \textit{inv\_popped}(\gamma, \textit{src}, \textit{popped}, \textit{prev}, \textit{dist}, \textit{dst})\big) \wedge \\ &\big(\forall \textit{dst}.\ 0 \le \textit{dst} < i && \rightarrow \textit{inv\_unpopped}(\gamma, \textit{src}, \textit{popped}, \textit{prev}, \textit{dist}, \textit{dst}) \wedge \\ & && \quad\ \textit{inv\_unseen}(\gamma, \textit{src}, \textit{popped}, \textit{prev}, \textit{dist}, \textit{dst})\big) \wedge \\ &\big(\forall \textit{dst}.\ i \le \textit{dst} < \texttt{size} && \rightarrow \textit{inv\_unpopped\_weak}(\gamma, \textit{src}, \textit{popped}, \textit{prev}, \textit{dist}, \textit{dst}, u) \wedge \\ & && \quad\ \textit{inv\_unseen\_weak}(\gamma, \textit{src}, \textit{popped}, \textit{prev}, \textit{dist}, \textit{dst}, u)\big)\end{aligned}$$

We now have five cases, many of which are familiar from *dijk_correct*:

1. *inv_popped*: as before; if $\textit{dst} \in \textit{popped}$, then it has been fully processed. For all "previously-popped vertices" (*i.e.,* except for *u*), this is trivial from *dijk_correct*. For *u* itself, we reach the heart of Dijkstra's correctness: the locally-optimal path to the cheapest unpopped vertex is *globally* optimal.
2. *inv_unpopped* (less than *i*): as before; if *dst* is reachable in one hop from a popped vertex *mom*, where now *mom* could be *u*. Initially this is trivial since $i = 0$, and we restore it as *i* increments by updating costs when they can be improved, as on lines 18 and 19.
3. *inv_unseen* (less than *i*): as before; some previously unseen neighbors of *u* may be transferred to unpopped status. This is also restored as *i* increments.
4. *inv_unpopped_weak* (between *i* and `size`): if *dst* is reachable in one hop from a previously-popped vertex *mom*, with potentially further improvements possible via *u*. As *i* increments, we strengthen it into *inv_unpopped* after considering whether routing via *u* improves the best-known cost to *dst*.

5. *inv_unseen_weak* (between i and `size`): no edge exists from any previously-popped vertex to *dst*, but there may be one from u. As i increments, we consider whether routing via u reveals a path to *dst*. This is strengthened into *inv_unpopped* if so, and into *inv_unseen* if not.

At the end of the `for` loop the fourth and fifth cases fall away (i = size), and the PQ and the *dist* and *prev* arrays finish "catching up" to the pop on line 12. This allows us to infer the `while` invariant *dijk_correct*, and thus continue the `while` loop. The `while` loop itself breaks when all vertices have been popped and processed. The second and third clauses of the `while` loop invariant *dijk_correct* then fall away, as seen on line 20: all vertices satisfy *inv_popped*, and are either optimally reachable or altogether unreachable. We are done.

3.2 Overflow in Dijkstra's Algorithm

Dijkstra's algorithm clearly cannot work when a path cost is more than `INT_MAX`. A reasonable-looking restriction is to bound edge costs by $\left\lfloor \frac{\texttt{INT\_MAX}}{\texttt{size}-1} \right\rfloor$, since the longest optimal path has $\texttt{size} - 1$ links and so the most expensive possible path costs no more than `INT_MAX`. However, this has two flaws.

First, since we are writing real code in C, rather than pseudocode in an idealized setting, we must reserve some concrete `int` value `inf` for "infinity". Suppose we set $\texttt{inf} = \texttt{INT\_MAX}$, and that $\texttt{size} - 1$ divides `INT_MAX`. Now the longest path can have cost $(\texttt{size} - 1) \cdot \left\lfloor \frac{\texttt{INT\_MAX}}{\texttt{size}-1} \right\rfloor = \texttt{INT\_MAX} = \texttt{inf}$. This creates an unpleasant ambiguity: we cannot tell if the farthest vertex is unreachable, or if it is reachable with legitimate cost `INT_MAX`. We need to adjust our maximum edge weights to leave room for `inf`; using $\left\lfloor \frac{\texttt{INT\_MAX}-1}{\texttt{size}-1} \right\rfloor$ solves this first issue.

Second, even though the best-known distances start at `inf` (see line 8) and only ever decrease from there, the code can overflow on lines 17 and 18. Consider applying Dijkstra's algorithm on a 32-bit unsigned machine to the graph in Fig. 2. The `size` of the graph is 3 nodes, and the proposed edge-weight upper bound is $\left\lfloor \frac{\texttt{INT\_MAX}-1}{\texttt{size}-1} \right\rfloor = \left\lfloor \frac{(2^{32}-1)-1}{3-1} \right\rfloor = 2^{31} - 1$, for example as in the graph pictured in Fig. 2. A glance at the figure shows that the true distance from the source A to vertices B and C are $2^{31} - 1$ and $2^{32} - 2$ respectively. Both values are representable with 32 bits, and neither distance is $\texttt{inf} = 2^{32} - 1$, so naïvely all seems well. Unfortunately, Dijkstra's algorithm does not exactly work like that.

After processing vertices A and B, $2^{31} - 1$ and $2^{32} - 2$ *are* the costs reflected in the `dist` array for B and C respectively—*but unfortunately vertex C is still in the priority queue.* After vertex C is popped on line 12, we fetch its neighbors in the `for` loop; the cost from C to B ($2^{31} - 1$) is fetched on line 15. On line 17 the currently optimal cost to B ($2^{31} - 1$) is compared with the sum of the optimal cost to C ($2^{32} - 2$) plus the just-retrieved cost of the edge from C to B ($2^{31} - 1$). Naïvely, $(2^{32} - 2) + (2^{31} - 1)$ is *greater than* the currently optimal cost $2^{31} - 1$, so the algorithm should stick with the latter. However, $(2^{32} - 2) + (2^{31} - 1)$ overflows, with $\big((2^{32} - 2) + (2^{31} - 1)\big) \texttt{ mod } 2^{32} = 2^{31} - 3$, which is *less than*

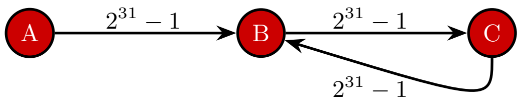


Fig. 2. A graph that will result in overflow on a 32-bit machine.

$2^{31}-1$! Thus the code decides that a new cheaper path from A to B exists (in particular, A⇝B⇝C⇝B) and then trashes the `dist` and `prev` arrays on line 18.

Our code uses signed `int` rather than `unsigned int` so we have undefined behavior rather than defined-but-wrong behavior, but the essence of the overflow is identical. We ensure that the "probing edge" does not overflow by restricting the maximum edge cost further, from $\left\lfloor \frac{\texttt{INT\_MAX}-1}{\texttt{size}-1} \right\rfloor$ to $\left\lfloor \frac{\texttt{INT\_MAX}}{\texttt{size}} \right\rfloor$. In Fig. 2, edge weights should be bounded by $\left\lfloor \frac{2^{32}-1}{3} \right\rfloor = 1{,}431{,}655{,}765$; call this value w. Suppose we change the edge weights in Fig. 2 from $2^{31}-1$ to w. Now vertex B has distance w and C has distance $2 \cdot w$. When we remove C from the priority queue, the comparison on line 17 is between the known best cost to B (*i.e.*, w) and the candidate best cost to B via C (*i.e.*, $3 \cdot w = 2^{32}-1 = \texttt{INT\_MAX}$). There is no overflow, so the candidate is rejected and the code behaves as advertised.

We fold these new restrictions into the mathematical graph γ. In addition to the bounds discussed above, we require a few other more straightforward bounds: edge costs be non-negative, as is typical for Dijkstra; $4 \cdot \texttt{size} \leq \texttt{INT\_MAX}$, to ensure that the multiplication in the `malloc` on line 5 does not overflow; and that $\left\lfloor \frac{\texttt{INT\_MAX}}{\texttt{size}} \right\rfloor \cdot (\texttt{size}-1) < \texttt{inf}$, so no valid path has cost `inf`. These bounds are optimal: if the input is any less restricted, the postcondition will fail. The last restriction on `inf` is not sufficient when $\texttt{size} = 1$, so in that special case we further require that any (self-loop) edges cost less than `inf`. Whenever $0 < 4 \cdot \texttt{size} \leq \texttt{INT\_MAX}$, the restrictions on `inf` are satisfiable with $\texttt{inf} \overset{\Delta}{=} \texttt{INT\_MAX}$.

3.3 Related Work on Dijkstra in Algorithms and Formal Methods

We were not able to find a reference that gives a robust, precise, and full description of the overflow issue we describe above. Dijkstra's original paper [22] ignores the issue, as do the standard textbooks *Introduction to Algorithms* (*a.k.a.* CLRS) by Cormen *et al.* [20] and *Algorithm Design* by Kleinberg and Tardos [38]. Sedgewick's book on graph algorithms in C [66] sidesteps the overflow in line 17 by requiring weights be in `double`, which *does* have a well-defined positive infinity value and cannot overflow in the traditional sense; Sedgewick and Wayne's *Algorithms* textbook in Java does the same [67]. However, Sedgewick's sidestep entails enduring the inevitable round-off intrinsic to floating-point arithmetic, and so his algorithm computes approximate optimal costs rather than exact ones. Sedgewick does not specify any bounds on input edge weights, and accordingly does not (and cannot) provide any bound on this accumulated error. Sedgewick is silent on how to handle an `int`-weighted input graph. Skiena's *Algorithm*

Design Manual [68] contains code with exactly the bug we identify: he uses integer weights and does not specify any bounds. To its credit, Heineman *et al.*'s *Algorithms in a Nutshell* [33] takes `int` edge weights as inputs and mentions overflow as a possibility. Heineman *et al.* hustle their way around this overflow by performing the arithmetic in line 17 in `long`. However, this cast does not really handle the problem in a fundamental way: if edge weights are given in `long` rather than `int`, then it would be necessary to cast to `long long`; if edge weights are given in `long long`, then Heineman's hustle breaks as there is no bigger type to which to cast. Moreover, Heineman *et al.* do not bound edge weights, so when the cumulative edge weights are too high their code fails silently.

Chen verified Dijkstra in Mizar [15], Gordon *et al.* formalized the reachability property in HOL [29], Moore and Zhang verified it in ACL2 [53], Mange and Kuhn verified it in Jahob [52], Filliâtre in Why3 [25], and Klasen verified it in KeY [37]. Liu *et al.* took an alternative SMT-based approach to verify a Java implementation of Dijkstra [51]. The most recent effort (2019) is by Lammich *et al.*, working within Isabelle/HOL, although they only return the weight of the shortest path rather than the path itself [45]. In general the previous mechanized proofs on Dijkstra verify code defined within idealized formal environments, *e.g.* with unbounded integers rather than machine `int`s and a distinguished non-integer value for infinity. No previous work mentions the overflow we uncover.

4 Minimum Spanning Trees

Here we discuss our verifications of the classic MST algorithms Prim and Kruskal. Although our machine-checked proofs are about real C code, in this section we take a higher-level approach than we did in §3, focusing on our key algorithmic findings and overall experience. Accordingly, we only provide pseudocode for Prim's algorithm rather than a decorated program and do not show any code for Kruskal's. Our development contains our C code and formal proofs [1].

4.1 Prim's Algorithm

We put the pseudocode for Prim's algorithm in Fig. 3; the code on the left-hand side is directly from CLRS, whereas the code on the right omits line 5 and will be discussed in §4.2. Note that line 12 contains an implicit call to the PQ's `edit_priority`. Since the pseudocode only compares `keys` (*i.e.*, edge weights) rather than doing arithmetic on them *à la* Dijkstra, there are no potential overflows and it is reasonable to set `INF` to `INT_MAX` in C.

Indeed, our initial verifications of C code were largely "turning the crank" once we had the definitions and associated lemma support for pure/abstract undirected graphs, forests, *etc.* discussed in §2.3. Accordingly, our initial contribution was a demonstration that this new graph machinery was sufficient to verify real code. We also proved that our extensions to CertiGraph from §2 were generic rather than verification-specific by reusing much pure and spatial reasoning that had been originally developed for our verification of Dijkstra.

```
1  MST-PRIM(G,w,r):
2   for each u in G.V
3    u.key = INF
4    u.parent = NIL
5   r.key = 0
6   Q = G.V
7   while Q ≠ ∅
8    u = EXTRACT-MIN(Q)
9    for each v in G.Adj[u]
10    if v ∈ Q and w(u,v) < v.key
11     v.parent = u
12     v.key = w(u,v)
```

```
MST-NOROOT-PRIM(G,w):
 for each u in G.V
  u.key = INF
  u.parent = NIL

 Q = G.V
 while Q ≠ ∅
  u = EXTRACT-MIN(Q)
  for each v in G.Adj[u]
   if v ∈ Q and w(u,v) < v.key
    v.parent = u
    v.key = w(u,v)
```

Fig. 3. Left: Prim's algorithm from CLRS [20]. Right: the same omitting line 5.

4.2 Prim's Algorithm Handles Multiple Components Out of the Box

Textbook discussions of Prim's algorithm are usually limited to single-component input graphs (*a.k.a.* connected graphs), producing a minimum spanning tree. It is widely believed that Prim's is not directly applicable to graphs with multiple components, which should produce a minimum spanning forest. For example, both Rozen [65] and Sedgewick *et al.* [66,67] leave the extension to multiple components as a formal exercise for the reader, whereas Kepner and Gilbert suggest that multiple-component graphs should be handled by first finding the components and then running Prim on each component [36].

After we completed our initial verification, a close examination of our formal invariants showed us that the algorithm *exactly as given by standard textbooks* will properly handle multi-component graphs *in a single run*. The confusion starts because, in a fully connected graph, any vertex `u` removed from the PQ on line 8 must have `u.key` < `INF`; *i.e.*, `u` must be immediately reachable from the spanning tree that is in the process of being built. However, nothing in the code relies upon this connectedness fact! All we need is that `u` is the "closest vertex" to the "current component." If `u.key` = `INF` *and* `u` is a minimum of the PQ, then it simply means that the "previous component" is done, and we have started spanning tree construction on a new unconnected component "rooted" at `u`, yielding a forest. The node `u`'s parent will remain `NIL`, at it was after the setup loop on line 4, indicating that it is the root of a spanning tree. Its `key` will be `INF` rather than 0, but the `key`s are *internal to Prim's algorithm*: clients only get back the spanning forest as encoded in the `parent` pointers[4].

Having made this discovery, we updated our proofs to support the new weaker precondition, which is what we currently formally verify in Coq [71]. A little further thought led to the realization that since Prim can handle arbitrary numbers

[4] The `key`s simply record the edge-weight connecting a vertex to its candidate parent; recall that line 12 is really a call to the PQ's `edit_priority`. If a client wishes to know this edge weight, it can simply look up the edge in the graph.

of components, the initialization of the root's `key` in line 5 is in fact unnecessary. Accordingly, if we remove this line and the associated function argument `r` from `MST-PRIM` (*i.e.*, the code on the right half of Fig. 3), the algorithm still works correctly. Moreover, *the program invariants become simpler* because we no longer need to treat a specified vertex (`r`) in a distinguished manner. Our formal development verifies this version of the algorithm as well [1].

4.3 Related Work on Prim in Algorithms and Formal Methods

Prim's algorithm was in fact first developed by the Czech mathematician Vojtěch Jarník in 1930 [35] before being rediscovered by Robert Prim in 1957 [61] and a third time by Edsger W. Dijkstra in 1959 [22]. Both Prim's and Dijkstra's treatment explicitly assumes a connected graph; although we cannot read Czech, some time with Google translate suggests that Jarník's treatment probably does the same. The textbooks we surveyed [20,36,38,65–68] seem to derive from Prim's and/or Dijkstra's treatment. More casual references such as Wikipedia [3] and innumerable lecture slides are presumably derived from the textbooks cited. We have not found any references that state that Prim's algorithm *without modification* applies to multi-component graphs, even when executable code is provided: *e.g.*, Heineman *et al.* provide C++ code that aligns closely with our C code [33], but do not mention that their code works equally well on multi-component graphs. Sadly, many sources promulgate the false proposition that modifications to the algorithm are needed to handle multi-component graphs (*e.g.*, [3,36,65–67]). Likewise, we have found no reference that removes the initialization step (line 5 in Fig. 3) from the standard algorithm.

Prim's algorithm has been the focus of a few previous formalization efforts. Guttman formalised and proved the correctness of Prim's algorithm using Stone-Kleene relation algebras in Isabelle/HOL [30]. He works in an idealized formal environment that does not require the development of explicit data structures; his code does not appear to be executable. Lammich *et al.* provided a verification of Prim's algorithm [45]. Lammich *et al.* also work within the idealized formal environment of Isabelle/HOL, but, in contrast to Guttman, develop efficient purely functional data structures and extract them to executable code. Both Guttman and Lammich explicitly require that the input graph be connected.

4.4 Kruskal's Algorithm

Although Kruskal's algorithm is sometimes presented as taking connected graphs and producing spanning trees, the literature also discusses the more general case of multi-component input graphs and spanning forests. However, Kruskal has only recently been the focus of formal verification efforts, partly because it relies on the notoriously difficult-to-verify union-find algorithm; fortunately, the CertiGraph project has an existing fully-verified union-find implementation that we can leverage [73]. Kruskal also requires a sorting function; we implemented `heapsort` as explained in §5.2. Kruskal is optimized for compact representations of sparse graphs, so the $O(1)$ space cost of `heapsort` is a reasonable fit.

The primary interest of our verification of Kruskal is in our proof engineering. Kruskal inputs graphs as edge lists rather than adjacency matrices. In addition to requiring an addition to our spatial graph predicate menu, this means that Kruskal's input graphs can have multiple edges between a given pair of vertices (*i.e.*, a "multigraph"). Pleasingly, we can reuse most of the undirected graph definitions (§2.3), demonstrating that they are generic and reusable.

Another challenge is integrating the pre-existing CertiGraph verification of union-find. We are pleased to say that no change was required for CertiGraph's existing union-find definitions, lemmas, specifications and verification. Kruskal actually manipulates two graphs simultaneously: a directed graph with vertex labels (to store parent pointers and ranks) within union-find, and an undirected multigraph with edge labels (for which the algorithm is constructing a spanning forest). Beyond showing that CertiGraph was capable of this kind of systems-integration challenge, we had to develop additional lemma support to bridge the directed notion of "reachability," used within the directed union-find graph to the undirected notion of "connectedness," used in the MSF graph (§2.3).

4.5 Related Work on Kruskal in Algorithms and Formal Methods

Joseph Kruskal published his algorithm in 1956 [42] and it has appeared in numerous textbooks since (*e.g.*, [20,38,66,68]). Kruskal's algorithm is usually preferred over Prim's for sparse graphs, and is sometimes presented as "the right choice" when confronted with multi-component graphs under the mistaken assumption that Prim's first requires a component-finding initial step.

Guttman generalized minimum spanning tree algorithms using Stone relation algebras [31], and provided a proof of Kruskal's algorithm formatted in said algebras. Like in his work on Prim's [30], Guttmann works within Isabelle/HOL and does not include concrete data structures such as priority-queues and union-find, instead capturing their action as equivalence relations in the underlying algebras. In Guttmann's Kruskal paper, he mentions that his Prim paper axiomatizes the fact that "every finite graph has a minimum spanning forest," which he is then able to prove *using his Kruskal algorithm*. Interestingly, our Prim verification needs the same fact, but we prove it directly.

In a similar vein, Haslbeck *et al.* verified Kruskal's algorithm [32] by building on Lammich *et al.*'s earlier work on Prim [45]. Like Lammich *et al.*, Haslbeck *et al.* work within Isabelle/HOL with a focus on purely functional data structures.

One of the stumbling blocks in verifying Kruskal's algorithm is the need to verify union-find. In addition to CertiGraph [73], two recent efforts to certify union-find are by Charguéraud and Pottier, who also prove time complexity [14]; and by Filliâtre [26], whose proof benefits from a high degree of automation.

5 Verified Binary Heaps in C

A binary heap embeds a heap-ordered tree in an array and uses arithmetic on indices to navigate between a parent and its left and right children [20]. In addition to providing the standard `insert` and `remove-min`/`remove-max` operations (depending on whether it is a min- or max-ordered heap) in logarithmic time, binary heaps can by upgraded to support two nontrivial operations. First, Floyd's `heapify` function builds a binary heap from an unordered array in linear time, and as a related upgrade, `heapsort` performs a worst-case linearithmic-time sort using only constant additional space. Second, binary heaps can be upgraded to support logarithmic-time `decrease-` and `increase-priority` operations, which we generalize straightforwardly into `edit_priority`.

Binary heaps are a good fit for our graph algorithms because Dijkstra's and Prim's algorithms need to edit priorities, and a constant-space `heapsort` is appropriate for the sparse edge-list-represented graphs typically targeted by Kruskal's. The C language has poor support for polymorphic higher-order functions, and a binary heap that supports `edit_priority` is half as fast as a binary heap that does not. Accordingly, we implement binary heaps in C three times:

Name	Heap order	`edit_priority`	`heapify`	Payload
basic	min	no	yes	`void*`
advanced	min	yes	no	`int`
Kruskal	max	no	yes	`int, int` (*i.e.*, unboxed)

Priorities are of type `int`. The Kruskal-specific implementation is stripped down to the bare minimum required to implement `heapsort` (*e.g.*, it does not support `insert`). We next overview these verifications in three parts: basic heap operations, `heapify` and `heapsort` operations, and the `edit_priority` operation.

5.1 The Basic Heap Operations of Insertion and Min/Max-Removal

Because we are juggling three implementations, we take some care to factor our verification to maximize reuse. First, each C implementation has its own exchange and comparison functions that handle the nitty-gritty of the payload and choose between a min or max heap. The following lines are from the "basic" implementation, in which the "payload" (`data` field) is of type `void*`:

```c
void exch(unsigned int j, unsigned int k, Item arr[]) {
 int priority = arr[j].priority; void* data = arr[j].data;
 arr[j].priority = arr[k].priority; arr[j].data = arr[k].data;
 arr[k].priority = priority; arr[k].data = data; }
int less(unsigned int j, unsigned int k, Item arr[]) {
 return (arr[j].priority <= arr[k].priority); }
```

These C functions are specified as refinements of Gallina functions that exchange polymorphic data in lists and compare objects in an abstract preordered set; we verify them in VST after a little irksome engineering. The payoff is that the key heap operations, which, following Sedgewick [66], we call `swim` and `sink`, can use identical C code (up to alpha renaming) in all three implementations:

```c
void swim(unsigned int k, Item arr[]) {
 while (k > ROOT_IDX && less (k, PARENT(k), arr)) {
  exch(k, PARENT(k), arr); k = PARENT(k);         } }
void sink (unsigned int k, Item arr[], unsigned int available) {
```

```
15 while (LEFT_CHILD(k) < available) {
16  unsigned j = LEFT_CHILD(k);
17  if (j+1 < available && less(j+1, j, arr)) j++;
18  if (less(k, j, arr)) break; exch(k, j, arr); k = j;     } }
```

These functions involve a number of complexities, both at the algorithms level and at the semantics-of-C level. At the C level, there is the potential for a rather subtle bug in the macros ROOT_IDX, PARENT, etc. Abstractly, these are simple: the root is in index 0; the children of x at roughly $2x$ and the parent at roughly $\frac{x}{2}$, with ± 1 as necessary. The danger is thinking that because the variables are `unsigned int`, all arithmetic will occur in this domain; in fact we must force the associated constants into `unsigned int` as well:

```
#define ROOT_IDX   0u
#define PARENT(x)  (x-1u)/2u
#define LEFT_CHILD(x)  (2u*x)+1u
#define RIGHT_CHILD(x)  2u*(x+1u)
```

A second C-semantics issue is the potential for overflow within LEFT_CHILD and RIGHT_CHILD (as well as the increments on line 17), and underflow within the PARENT macro (if x should ever be 0). To avoid this overflow, the precondition of sink requires that when k is in bounds (*i.e.*, $\mathtt{k} < \mathtt{available}$), then $2 \cdot (\mathtt{available} - 1) \leq \mathtt{max\_unsigned}$. An edge case occurs when deleting the last element from a heap ($\mathtt{k} = \mathtt{available}$); we then require $2 \cdot \mathtt{k} \leq \mathtt{max\_unsigned}$.

At the algorithmic level, both the swim and sink functions involve nontrivial loop invariants; sink is complicated by the further need to support Floyd's heapify, during which a large portion of the array is unordered. Accordingly, we build Gallina models of both functions and show that they restore heap order given a mostly-ordered input heap. There are two different versions of "mostly-ordered". Specifically, swim uses a "bottom-up" version:

```
Definition weak_heapOrdered2 (L : list A) (j : nat) : Prop :=
 (∀ i b, i ≠ j → nth_error L i = Some b →
   ∀ a, nth_error L (parent i) = Some a → a ≼ b) ∧
 (grandsOk L j root_idx).
```

whereas sink uses a "top-down" version:

```
Definition weak_heapOrdered_bounded (L:list A) (k:nat) (j:nat) :=
 (∀ i a, i ≥ k → i ≠ j → nth_error L i = Some a →
  (∀ b, nth_error L (left_child i) = Some b → a ≼ b) ∧
  (∀ c, nth_error L (right_child i) = Some c → a ≼ c)) ∧
 (grandsOk L j k).
```

The parameter j indicates a "hole", at which the heap may not be heap-ordered; grandsOk bridges this hole by ordering the parent and the children of j:

```
Definition grandsOk (L : list A) (j : nat) (k : nat) : Prop :=
 j ≠ root_idx → parent j ≥ k →
   ∀ gs bb, parent gs = j → nth_error L gs = Some bb →
     ∀ a, nth_error L (parent j) = Some a → a ≼ bb.
```

The parameter k is used to support Floyd's heapify: it bounds the portion of the list in which elements are heap-ordered (with the exception of j). The proofs that the Gallina `swim` and `sink` can restore (bounded) heap-orderedness involve a number of edge cases, but given the above definitions go through. The invariants of the C versions of `swim` and `sink` are stated via the associated Gallina versions, thereby delegating all heap-ordering proofs to the Gallina versions.

The insertion and remove functions we verify are in fact "non-checking" versions (`insert_nc` and `remove_nc`): their preconditions assume there is room in the heap to add or an item in the heap to remove. In the context of Dijkstra and Prim, these preconditions can be proven to hold. The associated verifications involve a little separation logic hackery (specifically, to FRAME away the "junk" part of the heap-array from the "live" part), but are straightforward using VST. We avoid the overflow issue in `sink` by bounding the maximum capacity of the heap: $4 \leq 12 \cdot \texttt{capacity} \leq \texttt{max\_unsigned}$; the magic number 12 comes from the size of the underlying data structure in C. We require users to prove this bound on heap creation, and thereafter handle it under the hood.

5.2 Bottom-Up Heapify and Heapsort

Floyd's bottom-up procedure for constructing a binary heap in linear time, and using a binary heap to sort, are classics of the literature [20,66]. Happily, while the asymptotic bound on heap construction is nontrivial, the implementations of both are basically repeated calls to `sink` (and exchanges to remove the root):

```c
19 void build_heap(Item arr[], unsigned int size) {
20  unsigned int start = PARENT(size);
21  while(1) { sink(start, arr, size);
22             if (start == 0) break; start--;  } }
23 void heapsort_rev(Item* arr, unsigned int size) {
24  build_heap(arr,size);
25  while (size > 1) { size--;
26   exch(ROOT_IDX, size, arr); sink(ROOT_IDX, arr, size); } }
```

Given that in §5.1 we already generalized the specification for `sink` to handle a portion of the array being unordered, the verification of these functions is straightforward. There is, however, the possibility of a subtle underflow on line 20, in the case when building an empty heap (*i.e.*, `size` = 0). In turn, this means that `heapsort_rev` as given above cannot sort empty lists; in our "basic" implementation we strengthen the precondition accordingly, whereas in our "Kruskal" implementation we add a line before 24 that `return`s when `size` = 0. We use a max-heap for Kruskal because heapsort yields a *reverse* sorted list.

5.3 Modifying an Element's Priority

To support edit-priority, each live item is associated not only with its usual `int` priority but also given a unique `unsigned int` "key", generated during `insert` and returned to the client. The binary heap internally maintains a secondary array `key_table` that maps each key to the current location of the associated item within the primary heap array. The client calls `edit_priority` by supplying the key for the item that it wishes to modify, which the binary heap looks up in the `key_table` to locate the item in the heap array before calling `sink` or `swim`. To keep everything linked together, the `key_table` is modified during `exchange`.

To generate the keys on insert, we store a key field within each heap-item in the main array. These keys are initialized to 0..(`capacity` − 1), and thereafter are never modified other than when two cells are swapped during `exchange`. An invariant can then be maintained that the keys from the "live" and "junk" parts have no duplicates. On insertion, we "recycle" the key of the first "junk" item, which is by the invariant known to be appropriately fresh.

5.4 Related Work on Binary Heaps in Algorithms and Formal Methods

J. W. J. Williams published the binary heap data structure, along with heapsort, in June 1964 [28]. Floyd proposed his linear-time bottom-up method to construct such heaps that December [27]. Since then, binary heaps, including Floyd's construction and heapsort, have become a staple of the introductory data structure diet [20]. On the other hand, standard textbooks are surprisingly vague on the implementation of `edit_priority` [20,38,66], and completely silent on the generation of fresh keys during insertion. Our method above of "recycling keys" avoids a subtle overflow in a naïve approach, and does not appear in the literature we examined. The naïve idea is to have a global counter starting at 0, which is then increased on each insert. Unfortunately, this is unsound: during (very) long runs involving both `insert` and `remove-min`, this key counter will overflow. Although overflow is defined in C for `unsigned int`, this overflow is fatal algorithmically: multiple live items could be assigned the same key.

Binary heaps have been verified several times in the literature. They were problem 2 of the VACID-0 benchmark [49], and solved in this regard as well by the Why3 team [69]. These solutions did not implement bottom-up heap construction or edit priority. Summers verified heapsort in Viper, again without bottom-up heap construction [56]. Lammich verified Introsort, which includes a heapsort subroutine [44]. Previous formal work ignores nitty-gritty C issues such as the difference between signed and unsigned arithmetic. We believe we are the first formally verified binary heap to support edit-priority.

6 Engineering Considerations

Verifying real code is meaningfully harder than verifying toy implementations. On top of such challenges, verifying graph algorithms requires a significant amount of mathematical machinery: there are many plausible ways to define basic notions such as reachability, but not all of them can handle the challenges of verifying real code [72]. Moreover, we would like our mathematical, spatial, and verification machinery to be generic and reusable.

All of the above suggests that it is important to work within existing formal proof developments due a strong desire to not reinvent very large wheels (the existing proof bases we work with contain hundreds of thousands of lines of formal proof). We chose to work with the CompCert certified compiler [50]; the Verified Software Toolchain [4], which provides significant tactic support for separation logic-based deductive verification of CompCert C programs; and the CertiGraph framework [73], which provides much pure and spatial reasoning support for verifying graph-manipulating programs within VST. We did so because these frameworks can handle the challenges of real code and because the CertiGraph included several fully verified implementations of union-find that we wished to reuse in our verification of Kruskal's algorithm.

Modular formal proof development involves major software engineering challenges [64]. Accordingly, we took care factoring our extensions to CertiGraph into generic and reusable pieces. This factoring allows us to reuse machinery between verifications, including in the mathematical, spatial, and verification levels. So, *e.g.*, we share significant pure and spatial machinery between Dijkstra, Prim, and Kruskal. Moreover, we maintain good separation between pure and spatial reasoning. So, *e.g.*, both our Dijkstra and Prim verifications can handle multiple spatial variants of adjacency matrices without significant change.

On the other hand, working within existing developments involves some challenges, primarily in that some design decisions have been already made and are hard to change. Moreover, our verifications tickled numerous bugs within VST, including: overly-aggressive automatic entailment simplifying, poor error messages, improper handling of C `structs`, and performance issues. We have been fortunate that the VST team has been willing to work with us to fix such bugs, although some work still remains. Performance remains one area of focus: for example, checking our verification of Kruskal with a 3.7 GHz processor and 32 gb of memory takes more than 22 min even after all of the generic pure and spatial reasoning has been checked, *i.e.* approximately 7 s per line of C code (including whitespace and comments). This performance is unviable for verifying an industrial-sized application of equivalent difficulty: *e.g.*, it would take 13 years for Coq to check the proof for 1,000,000 lines of C. Before some optimizations to our proof structure, the time was significantly longer still.

Our contributions to CertiGraph include pieces that are reused repeatedly and pieces that are more bespoke. Below, we give a sense of both the size of our development (lines of formal Coq proof) and the mileage we get out of our own work via reuse. Items "added with +" are very similar (within 1%) of each other; Prim #4 is the version that does not set the root, *i.e.* on the right in Fig. 3.

Name	Used	LoC	Name	LoC
MathAdjMat	7x	165	DijkSpec1+2+3	301
Undirected	5x	2,139	VerifDijk1+2+3	3,554
MathUAdjMat	4x	1,024	PrimSpec1+2+3+4	508
SpaceAdjMat1+2+3	7x	499	VerifPrim1+2+3+4	7,455
EdgeListGraph	1x	911	KruskalSpec	302
MathDijkGraph	3x	165	VerifKruskal	1,606
DijkPureProof	3x	2,124	VerifHeapSort	568
UndirectedUF	1x	183	VerifBasicBinaryHeap	777
BinaryHeapModel	1x	1,870	VerifAdvBinaryHeap	2,253
Total (pure/spatial)		9,080	Total (verifications)	17,234

In total we have 26,314 novel lines of Coq proof to verify 1,155 lines of C code divided among 12 files, including 3 variants of Dijkstra, 4 variants of Prim, 1 of Kruskal (which includes its `heapsort`), and 2 binary heaps.

7 Concluding Thoughts: Related and Future Work

We have already discussed work directly related to Dijkstra's (§3.3), Prim's (§4.3), and Kruskal's (§4.5) algorithms, as well as binary heaps (§5.4). Summarizing briefly to the point of unreasonableness, our observations about Dijkstra's overflow and Prim's specification are novel, and existing formal proofs focus on code working within idealized environments rather than handling the real-world considerations that we do. We have also discussed the three formal developments we build upon and extend: CompCert, VST, and CertiGraph (Sect. 6). Our goal now is to discuss mechanized graph reasoning and verification more broadly.

Reasoning About Mathematical Graphs. There is a 30+ year history of mechanizing graph theory, beginning at least with Wong [74] and Chou [19] and continuing to the present day; Wang discusses many such efforts [72, §3.3]. The two abstract frameworks that seem closest to ours are those by Noschinski [58]; and by Lammich and Nipkow [45]. The latter is particularly related to our work, because they too start with a directed graph library and must extend it to handle undirected graphs so that they can verify Prim's algorithm.

More-Automated Verification. Broadly speaking, mechanized verification of software falls in a spectrum between more-automated-but-less-precise verifications and less-automated-but-more-precise verifications. Although VST contains some automation, we fall within the latter camp. In the former camp, landmark initial separation logic [63] tools such as Smallfoot [7] have grown into Facebook's industrial-strength Infer [11]. Other notable relatively-automated separation logic-based tools include HIP/SLEEK [17], Bedrock [18], KIV [24], VerCors [9], and Viper [57]. More-automated solutions that use techniques other than separation logic include Boogie [6], Blast [8], Dafny [48], and KeY [2]. In Sect. 3.3 we discuss how some of these more-automated approaches have been applied to verify Dijkstra's algorithm. Petrank and Hawblitzel's Boogie-based verification of a garbage collector [60], Bubel's KeY-based verification of the Schorr-Waite algorithm, and Chen *et al.*'s Tarjan's strongly connected components algorithm in (among others) Why3 [16] are three examples of more-automated verification of graph algorithms. Müller verified *binomial* (not binary) heaps in Viper, although his implementation did not support an edit-priority function [55]. The VOCAL project has verified a number of data structures, including binary and other heaps (all without edit-priority) and union-find [13].

We are not confident that more-automated tools would be able to replicate our work easily. We prove full functional correctness, whereas many more-automated tools prove only more limited properties. Moreover, our full functional correctness results rely upon a meaningful amount of domain-specific knowledge about graphs, which automated tools usually lack. Even if we restrict ourselves to more limited domains such as overflows, several more automated efforts did not uncover the overflow that we described in Sect. 3.3. The proof that certain bounds on edge weights and `inf` suffice depends on an intimate understanding of Dijkstra's algorithm (in particular, that it explores one edge beyond the optimum paths); overall the problem seems challenging in highly-automated settings. The more powerful specification we discover for Prim's algorithm in Sect. 4.2 is likewise not something a tool is likely to discover: human insight appears necessary, at least given the current state of machine learning techniques.

In contrast, several of the potential overflows in our binary heap might be uncovered by more-automated approaches, especially those related to the `PARENT` and `LEFT_CHILD` macros from Sect. 5.1. Although the arithmetic involves both addition/subtraction and multiplication/division, we suspect a tool such as Z3 [54] could handle it. Moreover, a sufficiently-precise tool would probably spot the necessity of forcing the internal constants into `unsigned int`. The issue of sound key generation described in Sect. 5.3 might be a bit trickier. On the one hand, `unsigned int` overflow is defined in C, so real code sometimes relies upon it. Accordingly, merely observing that the counter could overflow does not guarantee that the code is necessarily buggy. On the other hand, some tools might flag it anyway out of caution (*i.e.* right answer, wrong reason).

Less-Automated Verification. Although as discussed above some more-automated tools have been applied to verify graph algorithms, the problem domain is sufficiently complex that many of the verifications discussed in Sect. 3.3, Sect. 4.3, and Sect. 4.5 use less-automated techniques. Two basic approaches are popular. The "shallow embedding" approach is to write the algorithm in the native language of a proof assistant. The "deep embedding" approach is to write the algorithm in another language whose semantics has been precisely defined in the proof assistant. VST uses a deep embedding, and so we do too; one of VST's more popular competitors in the deep embedding style is "Iris Proof Mode" [39]. In contrast, Lammich *et al.* have produced a series of results verifying a variety of graph algorithms using a shallow embedding (*e.g.*, [32,43,45–47]). From a bird's-eye view Lammich *et al.*'s work is the most related to our results in this paper: they verify all three algorithms we do and are able to extract fully-executable code, even if sometimes their focus is a bit different, *e.g.* on novel purely-functional data structures such as a priority queue with `edit_priority`.

Pen-and-Paper Verification of Graph Algorithms. We use separation logic [63] as our base framework. Initial work on graph algorithms in separation logic was minimal; Bornat *et al.* is an early example [10]. Hobor and Villard developed the technique of ramification to verify graph algorithms [34], using a particular "star/wand" pattern to express heap update. Wang *et al.* later integrated ramification into VST as the CertiGraph project we use [73]. Krishna *et al.* [40] have developed a flow algebraic framework to reason about local and global properties of *flow graphs* in the program heap; their flow algebra is mainly used to tackle local reasoning of global graphs in program heaps. Flow algebras should be compatible with existing separation logics; implementation and integration with the Iris project appears to be work in progress [41].

Krishna *et al.* are interested in concurrency [40]; Raad *et al.* provide another example of pen-and-paper reasoning about concurrent graph algorithms [62].

Future Work. We see several opportunities for decreasing the effort and/or increasing the automation in our approach. At the level of Hoare tuples, we see opportunities for improved VST tactics to handle common cases we encounter in graph algorithms. At the level of spatial predicates, we can continue to expand our library of graph constructions, for example for adjacency lists. We also believe there are opportunities to increase modularity and automation at the interface between the spatial and the mathematical levels, *e.g.* we sometimes compare C pointers to heap-represented graph nodes for equality, and due to the nature of our representations this equality check will be well-defined in C when the associated nodes are present in the mathematical graph, so this check should pass automatically.

We believe that more automation is possible at the level of mathematical graphs: for example reachability techniques based on regular expressions over matrices and related semirings [5,23,70]. We are also intrigued by the recent development of various specialized graph logics such as by Costa *et al.* [21] and hope that these kinds of techniques will allow us to simplify our reasoning. The key advantage of having end-to-end machine-checked examples such as the ones we presented above is that they guide the automation efforts by providing precise goals that are known to be strong enough to verify real code.

Conclusion. We extend the CertiGraph library to handle undirected graphs and several flavours of graphs with edge labels, both at the pure and at the spatial levels. We verify the full functional correctness of the three classic graph algorithms of Dijkstra, Prim, and Kruskal. We find nontrivial bounds on edge costs and infinity for Dijkstra and provide a novel specification for Prim. We verify a binary heap with Floyd's `heapify` and `edit_priority`. All of our code is in CompCert C and all of our proofs are machine-checked in Coq.

Acknowledgements. We thank Shengyi Wang for his help and support.

References

1. Functional Correctness of C implementations of Dijkstra's, Kruskal's, and Prim's Algorithms (2021). https://doi.org/10.5281/zenodo.4744664
2. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification-The KeY Book-From Theory to Practice (2016)
3. Anonymous: Prim's algorithm. https://en.wikipedia.org/wiki/Prim%27s_algorithm
4. Appel, A.W., et al.: Program Logics for Certified Compilers. Cambridge University Press, Cambridge (2014)
5. Backhouse, R., Carré, B.: Regular algebra applied to path-finding problems. J. Inst. Math. Appl. **15**, 161–186 (1975)
6. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: a modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006). https://doi.org/10.1007/11804192_17
7. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: modular automatic assertion checking with separation logic. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 115–137. Springer, Heidelberg (2006). https://doi.org/10.1007/11804192_6
8. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker Blast. Int. J. Softw. Tools Technol. Transf. **9**, 505–525 (2007)
9. Blom, S., Huisman, M.: The VerCors tool for verification of concurrent programs. In: Jones, C., Pihlajasaari, P., Sun, J. (eds.) FM 2014. LNCS, vol. 8442, pp. 127–131. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06410-9_9
10. Bornat, R., Calcagno, C., O'Hearn, P.: Local reasoning, separation and aliasing. In: SPACE (2004)
11. Calcagno, C., et al.: Moving fast with software verification. In: NASA Formal Methods Symposium (2015)
12. Charguéraud, A.: Characteristic formulae for the verification of imperative programs. In: ICFP (2011)
13. Charguéraud, A., Filliâtre, J.C., Pereira, M., Pottier, F.: VOCAL - a verified OCaml library. ML Family Workshop (2017)
14. Charguéraud, A., Pottier, F.: Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits. J. Autom. Reason. **62**, 331–365 (2019)
15. Chen, J.C.: Dijkstra's shortest path algorithm. JFM **15**, 237–247 (2003)
16. Chen, R., Cohen, C., Lévy, J., Merz, S., Théry, L.: Formal proofs of Tarjan's strongly connected components algorithm in Why3, Coq and Isabelle. In: ITP (2019)
17. Chin, W.N., David, C., Nguyen, H.H., Qin, S.: Automated verification of shape, size and bag properties via user-defined predicates in separation logic. Sci. Comput. Program. **77**, 1006–1036 (2010)

18. Chlipala, A.: Mostly-automated verification of low-level programs in computational separation logic. In: PLDI (2011)
19. Chou, C.T.: A formal theory of undirected graphs in HOL. In: HOL (1994)
20. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.S.: Introduction to Algorithms, 3rd edn. (2009)
21. Costa, D., Brotherston, J., Pym, D.: Graph decomposition and local reasoning (2020). Under submission
22. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numer. Math. **1**, 269–271 (1959)
23. Dolan, S.: Fun with semirings: a functional pearl on the abuse of linear algebra. In: ICFP (2013)
24. Ernst, G., Pfähler, J., Schellhorn, G., Haneberg, D., Reif, W.: KIV - overview and VerifyThis competition. STTT **17**, 677–694 (2015)
25. Filliâtre, J.C.: Dijkstra's shortest path algorithm in Why3 (2011). http://toccata.lri.fr/gallery/dijkstra.en.html
26. Fillitre, J.C.: Simpler proofs with decentralized invariants. J. Log. Algebraic Methods Program. **121**, 100645 (2021)
27. Floyd, R.W.: Algorithm 245: treesort. Commun. ACM **7**(12), 701 (1964)
28. Forsythe, G.E.: Algorithms. Commun. ACM **7**(6), 347–349 (1964)
29. Gordon, M., Hurd, J., Slind, K.: Executing the formal semantics of the accellera property specification language by mechanised theorem proving. In: Geist, D., Tronci, E. (eds.) CHARME 2003. LNCS, vol. 2860, pp. 200–215. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-39724-3_19
30. Guttmann, W.: Relation-algebraic verification of prim's minimum spanning tree algorithm. In: Sampaio, A., Wang, F. (eds.) ICTAC 2016. LNCS, vol. 9965, pp. 51–68. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46750-4_4
31. Guttmann, W.: Verifying minimum spanning tree algorithms with stone relation algebras. J. Log. Algebraic Methods Program. **101**, 132–150 (2018)
32. Haslbeck, M.P.L., Lammich, P.: Refinement with time - refining the run-time of algorithms in Isabelle/HOL. In: ITP (2019)
33. Heineman, G., Pollice, G., Selkow, S.: Algorithms in a Nutshell. O'Reilly (2008)
34. Hobor, A., Villard, J.: Ramifications of sharing in data structures. In: POPL (2013)
35. Jarník, V.: O jistém problému minimálním. (z dopisu panu o. Borůvkovi) (1930)
36. Kepner, Jeremy; Gilbert, J.: Graph algorithms in the language of linear algebra. Soc. Ind. Appl. Math. (2011)
37. Klasen, V.: Verifying Dijkstra's algorithm with KeY. Diploma thesis (2010)
38. Kleinberg, J.M., Tardos, É.: Algorithm Design. Addison-Wesley (2006)
39. Krebbers, R., Timany, A., Birkedal, L.: Interactive proofs in higher-order concurrent separation logic. In: POPL (2017)
40. Krishna, S., Shasha, D., Wies, T.: Go with the flow: compositional abstractions for concurrent data structures. In: POPL (2017)
41. Krishna, S., Summers, A.J., Wies, T.: Local reasoning for global graph properties. In: ESOP (2020)
42. Kruskal, J.B.: On the shortest spanning subtree of a graph and the traveling salesman problem. Proc. Am. Math. Soc. **7**, 48–50 (1956)
43. Lammich, P.: Verified efficient implementation of Gabow's strongly connected component algorithm. In: Klein, G., Gamboa, R. (eds.) ITP 2014. LNCS, vol. 8558, pp. 325–340. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08970-6_21
44. Lammich, P.: Efficient verified implementation of Introsort and Pdqsort. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) IJCAR 2020. LNCS (LNAI), vol. 12167, pp. 307–323. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51054-1_18

45. Lammich, P., Nipkow, T.: Proof pearl: Purely functional, simple and efficient priority search trees and applications to Prim and Dijkstra. In: ITP (2019)
46. Lammich, P., Sefidgar, S.R.: Formalizing the Edmonds-Karp algorithm. In: Blanchette, J.C., Merz, S. (eds.) ITP 2016. LNCS, vol. 9807, pp. 219–234. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-43144-4_14
47. Lammich, P., Sefidgar, S.R.: Formalizing network flow algorithms: a refinement approach in Isabelle/HOL. J. Autom. Reason. **62**(2), 261–280 (2019)
48. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20
49. Leino, K.R.M., Moskal, M.: VACID-0: verification of ample correctness of invariants of data-structures. Edition 0 (2010)
50. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: POPL (2006)
51. Liu, T., Nagel, M., Taghdiri, M.: Bounded program verification using an SMT solver: a case study. In: ICST (2012)
52. Mange, R., Kuhn, J.: Verifying Dijkstra's algorithm in Jahob (2007)
53. Moore, J.S., Zhang, Q.: Proof Pearl: Dijkstra's shortest path algorithm verified with ACL2. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 373–384. Springer, Heidelberg (2005). https://doi.org/10.1007/11541868_24
54. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
55. Müller, P.: The binomial heap verification challenge in Viper. In: Müller, P., Schaefer, I. (eds.) Principled Software Development. Springer, Heidelberg (2018). https://doi.org/10.1007/978-3-319-98047-8_13
56. Müller, P.: Private correspondence (2021)
57. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: a verification infrastructure for permission-based reasoning. In: Jobstmann, B., Leino, K.R.M. (eds.) VMCAI 2016. LNCS, vol. 9583, pp. 41–62. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49122-5_2
58. Noschinski, L.: A graph library for Isabelle. Math. Comput. Sci. **9**, 23–39 (2015)
59. O'Hearn, P., Reynolds, J., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) CSL 2001. LNCS, vol. 2142, pp. 1–19. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44802-0_1
60. Petrank, E., Hawblitzel, C.: Automated verification of practical garbage collectors. Log. Methods Comput. Sci. **6** (2010)
61. Prim, R.C.: Shortest connection networks and some generalizations. Bell Syst. Tech. J. **36**(6), 1389–1401 (1957)
62. Raad, A., Hobor, A., Villard, J., Gardner, P.: Verifying concurrent graph algorithms. In: Igarashi, A. (ed.) APLAS 2016. LNCS, vol. 10017, pp. 314–334. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47958-3_17
63. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: LICS (2002)
64. Ringer, T., Palmskog, K., Sergey, I., Gligoric, M., Tatlock, Z.: QED at large: a survey of engineering of formally verified software. CoRR (2020)
65. Rosen, K.H.: Discrete Mathematics and Its Applications. 7th edn. (2012)
66. Sedgewick, R.: Algorithms in C, Part 5: Graph Algorithms (2002)
67. Sedgewick, R., Wayne, K.: Algorithms. 4th edn. Addison-Wesley (2011)

68. Skiena, S.: The Algorithm Design Manual, 2nd edn. Springer, Heidelberg (2008)
69. Tafat, A., Marché, C.: Binary heaps formally verified in Why3 (2011)
70. Tarjan, R.E.: A unified approach to path problems. J. ACM **28**(3), 577–593 (1981)
71. Coq development team: The Coq Proof Assistant. https://coq.inria.fr/
72. Wang, S.: Mechanized verification of graph-manipulating programs. Ph.D. thesis, National University of Singapore (2019)
73. Wang, S., Cao, Q., Mohan, A., Hobor, A.: Certifying graph-manipulating C programs via localizations within data structures. In: OOPSLA (2019)
74. Wong, W.: A simple graph theory and its application in railway signaling. In: HOL Theorem Proving System and Its Applications (1991)

Gillian, Part II: Real-World Verification for JavaScript and C

Petar Maksimović[1](✉), Sacha-Élie Ayoun[1], José Fragoso Santos[2], and Philippa Gardner[1]

[1] Imperial College London, London, UK
{p.maksimovic,s.ayoun17,p.gardner}@imperial.ac.uk
[2] INESC-ID/Instituto Superior Técnico, Universidade de Lisboa, Lisbon, Portugal
jose.fragoso@tecnico.ulisboa.pt

Abstract. We introduce verification based on separation logic to Gillian, a multi-language platform for the development of symbolic analysis tools which is parametric on the memory model of the target language. Our work develops a methodology for constructing compositional memory models for Gillian, leading to a unified presentation of the JavaScript and C memory models. We verify the JavaScript and C implementations of the AWS Encryption SDK message header deserialisation module, specifically designing common abstractions used for both verification tasks, and find two bugs in the JavaScript and three bugs in the C implementation.

1 Introduction

Separation logic (SL) [25,40] introduced compositional program verification using Hoare reasoning. Current analysis tools based on ideas from SL include: the automatic tool Infer [8,9] used inside Facebook to find lightweight bugs in Java/C/C++/Obj-C programs; the semi-automatic tool Verifast [26], which provides full verification for fragments of C and Java; the semi-automatic tool JaVerT [21], which provides bug-finding and verification for JavaScript (JS) programs; and the Viper architecture [36,35], which provides a verification backend for multiple programming languages, including Java, Rust, and Python. Our goal is to introduce verification based on SL to Gillian [19], a multi-language platform for symbolic analysis, integrating bug-finding and verification in the spirit of JaVerT and targeting many languages in the spirit of Viper.

Gillian currently supports three types of program analysis: symbolic testing, verification and bi-abduction. In [19], the focus was on symbolic testing, parametrised on complete concrete and symbolic memory models of the target language (TL), and underpinned by a core symbolic execution engine with strong mathematical foundations. Gillian analysis is done on GIL, an intermediate goto language parametric on a set of memory actions, which describe the fundamental ways in which TL programs interact with their memories. To instantiate Gillian to a new TL, a tool developer must: (1) identify the set of the TL memory actions and implement the TL memory models using these actions; and (2) provide a

A. Silva and K. R. M. Leino (Eds.): CAV 2021, LNCS 12760, pp. 827-850, 2021.
https://doi.org/10.1007/978-3-030-81688-9_38

trusted compiler from the TL to GIL, which preserves the TL memory models and the semantics. In [19], Gillian was instantiated to JS and C, and used to find bugs in two real-world data-structure libraries, Buckets.js [43] and Collections-C [41]. Here, we introduce compositional memory models for Gillian, extend Gillian analysis with verification based on separation logic, adapt Gillian-JS and Gillian-C to this compositional setting, and provide verified specifications of the JS and C implementations of the deserialisation module of the AWS Encryption SDK.

The compositional Gillian memory models (§2) are given by the tool developer for each TL instantiation. They are based on partial memories, and formulated using core predicates and the associated consumer and producer actions. Core predicates describe fundamental units of TL memories: e.g., a property of a JS object and a C block cell. Consumers and producers, respectively, frame off and frame on the TL memory resource described by the core predicates. Partiality and frame are familiar concepts from SL [25,40,11]. What is perhaps less familiar is our emphasis on negative resource: i.e., the resource known to be absent from the partial memory. For example, in JS, a new extensible object is known not to contain any property; and, in C, a freed block is known not to be in memory and a cell is known not to exist beyond the block bound. We introduce a methodology for designing Gillian compositional memory models, and apply it to JS and C (§3), resulting in a unexpected similarity between the two models. Our compositional JS memory models follow those given in work on a JS program logic [24] and the JaVerT tool [21], where negative resource was essential for frame preservation, inspired by the use of negative resource to capture stability properties in the CAP concurrent separation logic [14], now used in Iris [27]. Our compositional C memory models are based on the complete CompCert memory model [31]. Despite a large body of work on separation logic for C, we were unable to find a partial C memory model that captures the negative resource in its entirety. The nearest is probably the CH20 formalism [29], which handles freed locations but not block bounds. Negative resource for freed locations has also been used in incorrectness logic [39], and for block bounds in a program logic for WebAssembly [48].

We build Gillian verification on top of our compositional memory models. In particular, using the core predicates, we design an assertion language for writing function specifications in separation logic and, using the consumers and producers, we build a fully parametric spatial entailment engine which enables the use of function specifications in symbolic execution. Gillian also supports user-defined predicates, which allow tool developers to identify the TL language interface familiar to code developers, and code developers to describe and prove properties about the particular data structures in their programs.

We extend Gillian-JS and Gillian-C to enable verification, introducing the JS and C compositional memory models, and using the same trusted compilers as in [19]. With these instantiations, we provide functionally-correct, verified specifications of the message header deserialisation module of the AWS Encryption SDK JS and C implementations (§4, §5). This is stable, critical, industry-grade code (~200loc for JS, ~950loc for C), which uses advanced language features to manipulate complex data structures. To verify this code, we create language-independent predicates to capture the message header, which we then connect without modification to both JS and C memories, giving specifications for the module functions. We also build a library of associated lemmas, used for the verification of both implementations. The verification itself required a substantial improvement of the reasoning capabilities of Gillian, especially when it came to handling arrays of symbolic size. We discovered two bugs in the JS implementation: one a form of prototype poisoning, predicted theoretically in our paper on JaVerT [21]; and another that allowed third parties to potentially alter authenticated, non-secret data. We have also discovered three bugs in the C implementation: one which allowed some malformed headers to be parsed as correct; one over-allocation; and one undefined behaviour. All of these bugs have been fixed.

2 Gillian Verification

We introduce Gillian verification based on separation logic (§2.2), extending the GIL execution engine presented in [19] with compositional memory models (§2.1).

2.1 Compositional Memory Models

GIL is a simple goto intermediate language whose syntax is given below. It is parametric on a set of TL memory actions, $A \ni \alpha$, given per instantiation by the tool developer. GIL values, $v \in \mathcal{V}al$, contain numbers, strings, booleans, uninterpreted symbols (used, e.g., to represent memory locations), simple types (e.g., numbers, strings), function identifiers and lists of values. GIL expressions, $e \in \mathcal{E}xpr$, contain values, program variables, and unary and binary operators (e.g. addition, list concatenation); GIL symbolic expressions, $\hat{e} \in \hat{\mathcal{E}xpr}$, are analogous except that symbolic variables, $\hat{x} \in \hat{\mathcal{X}}$, are used instead of program variables.

GIL Syntax

$$v \in \mathcal{V}al \triangleq i, j, n \in \mathcal{N} \mid s \in \mathcal{S} \mid b \in \mathcal{B} \mid \varsigma \in \mathcal{U} \mid \tau \in \mathcal{T} \mid f \in \mathcal{F} \mid \overline{v} \in List(\mathcal{V}al)$$

$$e \in \mathcal{E}xpr \triangleq v \mid x \in \mathcal{X} \mid \ominus e \mid e_1 \oplus e_2 \qquad \hat{e} \in \hat{\mathcal{E}xpr} \triangleq v \mid \hat{x} \in \hat{\mathcal{X}} \mid \ominus \hat{e} \mid \hat{e}_1 \oplus \hat{e}_2$$

$$\begin{array}{ll} c \in \mathcal{C}md \triangleq x := e \mid \mathsf{ifgoto}\ e\ \ i \mid x := e(e') \mid x := \alpha(e) \mid & func \in \mathcal{F}unc \triangleq f(x)\{\overline{c}\} \\ \quad x := \mathsf{uSym}/\mathsf{iSym}(e) \mid \mathsf{return}\ e \mid \mathsf{fail}\ e \mid \mathsf{vanish} & \mathtt{p} \in \mathcal{P}rog = \mathcal{P}_!(\mathcal{F}unc) \end{array}$$

GIL commands, $c \in \mathcal{C}md$, contain variable assignment, conditional goto, function call, memory actions, allocation of uninterpreted/interpreted symbols, function return, error termination and path cutting. A GIL function, $f(x)\{\overline{c}\}$, comprises an identifier $f \in \mathcal{F}$, a formal parameter x[3], and a body given by a list of commands $\overline{c}$. A GIL program is a set of GIL functions with unique identifiers.

GIL execution is defined in terms of state models, which are parametric on a value set, $\mathsf{V} \supseteq \mathcal{V}al$, and a set of memory actions, A. We distinguish the Boolean value set, $\Pi \subset \mathsf{V}$, and refer to $\pi \in \Pi$ as a context. State models expose an interface consisting of state actions, $A \uplus A_S$, where the actions

[3] The implementation supports multiple parameters.

$$\frac{\begin{array}{c}\textsc{Memory Action - Success}\\ \mathsf{cmd}(\mathsf{p}, cs, i) = x := \alpha(e)\\ \sigma.\mathrm{eval}_e\,(-) \rightsquigarrow \mathsf{v} \qquad \sigma.\alpha(\mathsf{v}) \rightsquigarrow (\sigma', \mathsf{v}')^{\mathcal{S}}\\ \sigma'.\mathrm{setVar}_x\,(\mathsf{v}') \rightsquigarrow \sigma''\end{array}}{\mathsf{p} \vdash \langle \sigma, cs, i\rangle \rightsquigarrow \langle \sigma'', cs, i{+}1\rangle^{\mathtt{S}}}
\qquad
\frac{\begin{array}{c}\textsc{Memory Action - Error}\\ \mathsf{cmd}(\mathsf{p}, cs, i) = x := \alpha(e)\\ \sigma.\mathrm{eval}_e\,(-) \rightsquigarrow \mathsf{v} \qquad \sigma.\alpha(\mathsf{v}) \rightsquigarrow (\sigma', \mathsf{v}')^{r}\\ r \neq \mathcal{S} \qquad o = (\text{if } r = \mathcal{E} \text{ then } \mathtt{E} \text{ else } \mathtt{M})\end{array}}{\mathsf{p} \vdash \langle \sigma, cs, i\rangle \rightsquigarrow \langle \sigma', cs, i\rangle^{o(\mathsf{v}')}}$$

Fig. 1: GIL Execution Semantics: Memory Actions

$A_S = \{\mathrm{setVar}_x\}_{x\in\mathcal{X}} \cup \{\mathrm{setStore}, \mathrm{getStore}\} \cup \{\mathrm{eval}_e\}_{e\in\mathcal{E}xpr} \cup \{\mathrm{assume}, \mathrm{uSym}, \mathrm{iSym}\}$ address store management, expression evaluation, branching, and allocation.

Definition 1 (State Model). *A state model, $S(\mathsf{V}, A) \triangleq \langle |S|, ea\rangle$, comprises: a set of states $\sigma = \langle \mu, \rho, \pi\rangle \in |S|$, containing a memory μ, a variable store ρ, and a (satisfiable) context π*[4]*; and an action execution function, $ea : (A \uplus A_S) \to |S| \to \mathsf{V} \rightharpoonup \mathcal{P}(|S| \times \mathsf{V} \times \mathcal{R})$, with the result $r \in \mathcal{R} = \{\mathcal{S}, \mathcal{E}, \mathcal{M}\}$ denoting success, non-correctible error, or missing information error, pretty-printed $\sigma.\alpha(\mathsf{v}) \to \{(\sigma_i, \mathsf{v}_i)^{r_i}|_{i\in I}\}$ for all outcomes and $\sigma.\alpha(\mathsf{v}) \rightsquigarrow (\mu_i, \sigma_i)^{r_i}$ for a specific outcome, with countable I. The value set of concrete state models is the set of GIL values, $\mathcal{V}al$*[5]*; the value set of symbolic state models is the set of symbolic expressions, $\hat{\mathcal{E}}xpr$.*

Definition 2 (GIL Execution Semantics). *Given a state model S, the GIL execution semantics has judgements of the form:*

$$\mathsf{p} \vdash \langle \sigma, cs, i\rangle^{o} \rightsquigarrow_S \langle \sigma', cs', j\rangle^{o'}$$

with: call stacks, $cs \in \mathcal{C}all_S$; command indexes, $i, j \in \mathbb{N}$; and outcomes, $o \in \mathcal{O}$.

The GIL execution semantics is standard for a goto language, except that it is parametrised by the memory actions. Call stacks capture function-related control flow, with $\mathsf{cmd}(\mathsf{p}, cs, i)$ denoting the i-th command of the currently executing function (cf. [33] for details). Outcomes, $o \in \mathcal{O} \triangleq \mathtt{S} \mid \mathtt{N}(\mathsf{v}) \mid \mathtt{E}(\mathsf{v}) \mid \mathtt{M}(\mathsf{v})$, indicate how the execution is to proceed: $\mathtt{S}$ states that it can continue; $\mathtt{N}(\mathsf{v})$ states that it terminated normally with return value v; and $\mathtt{E}(\mathsf{v})$ and $\mathtt{M}(\mathsf{v})$ state that it failed with either a non-correctible or missing information error described by v. We give the rules for memory action execution in Figure 1; all can be found in [33].

Compositional Memory Models. We move from whole-program memory models [19] to compositional memory models by introducing memory core predicates, $\gamma \in \Gamma$, which represent the fundamental units of the TL memory model (e.g., a memory cell). Core predicates take two lists of parameters, in-parameters (or ins), denoted v_i, and out-parameters (or outs), denoted v_o, such that from the ins we can learn the outs. This concept is similar to predicate parameter modes

[4] States also include allocators (cf. [33] for details), elided to limit clutter.

[5] Note that the only satisfiable concrete context is true, meaning that concrete contexts can be elided and concrete states can be viewed as memory-store pairs, $\langle \mu, \rho\rangle$.

of [37] and we use it to implement a parametric spatial entailment engine. An example of a core predicate is the cell assertion, $x \mapsto \mathsf{v}$, which captures a cell in memory at address x having value v. Its in-parameter is x, and its out-parameter is v, because, if we know x, we can find v by looking it up in the memory.

With each core predicate $\gamma \in \Gamma$, we associate a consumer and a producer memory action, denoted by cons_γ and prod_γ respectively, to obtain the set of predicate actions $A_\Gamma = \bigcup_{\gamma\in\Gamma}\{\mathrm{cons}_\gamma, \mathrm{prod}_\gamma\}$, whose meaning is discussed shortly.

Definition 3 (Compositional Memory Model). *Given value set* V *and core predicate set* Γ*, a compositional memory model,* $M(\mathsf{V},\Gamma) \triangleq \langle |M|, \mathcal{W}f, \underline{\mathit{ea}}_\Gamma\rangle$*, comprises: **(1)** a partial commutative monoid (PCM)*[6]*,* $|M| = (|M|, \bullet, \mathbf{0})$*, where* $\mathbf{0}$ *denotes the (indivisible) empty memory; **(2)** a well-formedness relation,* $\mathcal{W}f \subseteq |M| \times \Pi$*, with* $\mathcal{W}f_\pi(\mu)$ *denoting that memory* μ *is well-formed in (satisfiable) context* π*; and **(3)** a predicate action execution function,* $\underline{\mathit{ea}}_\Gamma : A_\Gamma \times |M| \times \mathsf{V} \times \Pi \rightharpoonup \mathcal{P}(|M| \times \mathsf{V} \times \Pi \times \mathcal{R})$*, pretty-printed* $\mu.\alpha(\mathsf{v})_\pi \rightarrow \{(\mu_i, \mathsf{v}_i)^{r_i}_{\pi_i}|_{i\in I}\}$ *for all outcomes and* $\mu.\alpha(\mathsf{v})_\pi \rightsquigarrow (\mu_i, \mathsf{v}_i)^{r_i}_{\pi_i}$ *for a specific outcome, with countable* I*. The value set of concrete memory models is the set of GIL values,* $\mathcal{V}al$*; the value set of symbolic memory models is the set of symbolic expressions,* $\hat{\mathcal{E}}xpr$*.*

We discuss the most important properties that the components of compositional memory models must satisfy; a full list is available in [33]. The PCM requirement is well-known from separation logic [40,11]. Well-formedness holds only for satisfiable contexts, and describes the separation of symbolic resource and any further TL-specific well-formedness criteria (cf. §3). It must be monotonic with respect to context strengthening, compatible with the PCM composition, and the empty memory must be well-formed in any satisfiable context. The action execution function, $\mu.\alpha(\mathsf{v})_\pi \rightarrow \{(\mu_i, \mathsf{v}_i)^{r_i}_{\pi_i}|_{i\in I}\}$, denotes that, in a memory μ that is well-formed in the context π, executing action α with parameter v yields a countable number of branches characterised by the non-overlapping[7], satisfiable contexts π_i, each of which implies π and makes the corresponding memory μ_i well-formed, and all of which together cover π (i.e., $\pi \Rightarrow \bigvee_{i\in I}\pi_i$). This last property means that memory actions do not drop paths, which is essential for verification.

The intuition behind consumers and producers is that consumers frame off the core predicate resource (CPR), uniquely determined by the core predicate ins, and the producers frame it on. The following properties capture this intuition. First, we define the CPR of a core predicate $\gamma\langle \mathsf{v}_i \cdot \mathsf{v}_o\rangle$ as the memory resulting from its production in $\mathbf{0}$, which must succeed in any satisfiable context:

$$\pi\ \mathtt{SAT} \implies \mathbf{0}.\mathrm{prod}_\gamma(\mathsf{v}_i \cdot \mathsf{v}_o)_\pi \rightsquigarrow (\gamma\langle \mathsf{v}_i \cdot \mathsf{v}_o\rangle, \mathsf{true})^{\mathcal{S}}_\pi \wedge \gamma\langle \mathsf{v}_i \cdot \mathsf{v}_o\rangle \neq \mathbf{0}.$$

overloading notation for the core predicate and its resource. Moreover, we require that any successful production frames on the CPR:

$$\mu.\mathrm{prod}_\gamma(\mathsf{v}_i \cdot \mathsf{v}_o)_\pi \rightsquigarrow (\mu', \mathsf{true})^{\mathcal{S}}_{\pi'} \implies \mu' = \mu \bullet \gamma\langle \mathsf{v}_i \cdot \mathsf{v}_o\rangle$$

[6] A PCM, $X = \langle X, \bullet, \mathbf{0}\rangle$, comprises a carrier set X (overloaded for simplicity), a partial, associative, and commutative composition operator $\bullet$, and unit element $\mathbf{0}$.

[7] Note that this requirement makes concrete memory actions deterministic.

and also that producers cannot return missing information errors, as they are meant to succeed precisely when the CPR is missing. The consumers, on the other hand, must succeed if and only if the CPR is present in memory:

$$\begin{aligned}\mu.\mathrm{cons}_\gamma(\mathsf{v}_i)_\pi \rightsquigarrow (\mu', \mathsf{v}_o)^{\mathcal{S}}_{\pi'} &\implies \pi' \vdash \mu = \mu' \bullet \gamma\langle \mathsf{v}_i \cdot \mathsf{v}_o \rangle \\ \pi \vdash \mu = \mu' \bullet \gamma\langle \mathsf{v}_i \cdot \mathsf{v}_o \rangle \wedge \mathcal{W}f_\pi(\mu) &\implies \mu.\mathrm{cons}_\gamma(\mathsf{v}_i)_\pi \rightsquigarrow (\mu', \mathsf{v}_o)^{\mathcal{S}}_{\pi}\end{aligned}$$

with the resulting context π' having enough information to isolate the CPR[8]. Interestingly, erroneous executions cannot be fully characterised in terms of CPR presence or absence, because of TL-specific error cases: for example, in C, attempting to either get or set the value of a block cell that is beyond the block bound raises an out-of-bounds error (cf. §3). What we require instead is that consumed CPR can always be re-produced, that producers fail in a memory in which consumers succeed, and that producers succeed in a memory in which consumers return a missing information error (and vice versa for the latter):

$$\begin{aligned}\mu.\mathrm{cons}_\gamma(\mathsf{v}_i)_\pi \rightsquigarrow (\mu', \mathsf{v}_o)^{\mathcal{S}}_{\pi'} &\implies \mu'.\mathrm{prod}_\gamma(\mathsf{v}_i \cdot \mathsf{v}'_o)_{\pi'} \rightsquigarrow (\mu'', \mathsf{true})^{\mathcal{S}}_{\pi'} \\ \mu.\mathrm{cons}_\gamma(\mathsf{v}_i)_\pi \rightsquigarrow (\mu', \mathsf{v}_o)^{\mathcal{S}}_{\pi'} &\implies \mu.\mathrm{prod}_\gamma(\mathsf{v}_i \cdot -)_\pi \rightsquigarrow (\mu, \mathsf{false})^{\mathcal{E}}_{\pi'} \\ \mu.\mathrm{cons}_\gamma(\mathsf{v}_i)_\pi \rightsquigarrow (\mu, \mathsf{false})^{\mathcal{M}}_{\pi'} &\iff \mu.\mathrm{prod}_\gamma(\mathsf{v}_i \cdot \mathsf{v}_o)_\pi \rightsquigarrow (\mu \bullet \gamma\langle \mathsf{v}_i \cdot \mathsf{v}_o\rangle, \mathsf{true})^{\mathcal{S}}_{\pi'}\end{aligned}$$

The properties given so far allow us, for example, to prove that well-formed memories cannot contain duplicated CPR. The final property below requires that non-missing executions of consumers and erroneous executions of producers must be frame-preserving, with the former formulated as follows:

$$\begin{aligned}&\mu.\mathrm{cons}_\gamma(\mathsf{v}_i)_\pi \rightsquigarrow (\mu', \mathsf{v}_o)^{r}_{\pi'} \wedge r \neq \mathcal{M} \wedge (\pi'' \Rightarrow \pi') \wedge \mathcal{W}f_{\pi''}(\mu \bullet \mu_f) \\ &\quad \implies (\mu \bullet \mu_f).\mathrm{cons}_\gamma(\mathsf{v}_i)_{\pi''} \rightsquigarrow (\mu' \bullet \mu_f, \mathsf{v}_o)^{r}_{\pi''}\end{aligned}$$

where π'' effectively maintains well-formedness constraints for μ, adds on further ones required for $\mu \bullet \mu_f$ to be defined and also isolates the consumed CPR. Note that neither missing executions of consumers nor successful executions of producers can be frame preserving, as framing on the appropriate CPR could result in success for the former, and a duplicated resource error for the latter.

Using the consumers and producers, we are able to derive getter and setter actions, $A \triangleq \{\mathrm{get}_\gamma, \mathrm{set}_\gamma : \gamma \in \Gamma\}$, which perform frame-preserving CPR lookup and mutation, as given below. We discuss getters and setters further in §3, in the context of our JS and C instantiations.

Getter: Success

$$\frac{\mu.\mathrm{cons}_\gamma(\mathsf{v}_i)_\pi \rightsquigarrow (\mu', \mathsf{v}_o)^{\mathcal{S}}_{\pi'} \qquad \mu'.\mathrm{prod}_\gamma(\mathsf{v}_i \cdot \mathsf{v}_o)_{\pi'} \rightsquigarrow (\mu'', \mathsf{true})^{\mathcal{S}}_{\pi'}}{\mu.\mathrm{get}_\gamma(\mathsf{v}_i)_\pi \rightsquigarrow (\mu'', \mathsf{v}_o)^{\mathcal{S}}_{\pi'}}$$

Setter: Success

$$\frac{\mu.\mathrm{cons}_\gamma(\mathsf{v}_i)_\pi \rightsquigarrow (\mu', -)^{\mathcal{S}}_{\pi'} \qquad \mu'.\mathrm{prod}_\gamma(\mathsf{v}_i \cdot \mathsf{v}_o)_{\pi'} \rightsquigarrow (\mu'', \mathsf{true})^{\mathcal{S}}_{\pi'}}{\mu.\mathrm{set}_\gamma(\mathsf{v}_i \cdot \mathsf{v}_o)_\pi \rightsquigarrow (\mu'', \mathsf{true})^{\mathcal{S}}_{\pi'}}$$

Getter: Non-Success

$$\frac{\mu.\mathrm{cons}_\gamma(\mathsf{v}_i)_\pi \rightsquigarrow (\mu, \mathsf{false})^{r}_{\pi'} \qquad r \neq \mathcal{S}}{\mu.\mathrm{get}_\gamma(\mathsf{v}_i)_\pi \rightsquigarrow (\mu, \mathsf{false})^{r}_{\pi'}}$$

Setter: Non-Success

$$\frac{\mu.\mathrm{cons}_\gamma(\mathsf{v}_i)_\pi \rightsquigarrow (\mu, \mathsf{false})^{r}_{\pi'} \qquad r \neq \mathcal{S}}{\mu.\mathrm{set}_\gamma(\mathsf{v}_i \cdot \mathsf{v}_o)_\pi \rightsquigarrow (\mu, \mathsf{false})^{r}_{\pi'}}$$

[8] The $\pi \vdash \ldots$ denotes reasoning under context π. In the concrete case, it can be ignored.

Compositional State Models. Compositional memory models lift to compositional state models, in a similar way to the lifting of the complete memory models illustrated in [19]; see [33] for details. Here, we focus on memory action execution, which is lifted as follows to state action execution, given a memory model $M(\mathsf{V}, \Gamma)$ and $\alpha \in A_\Gamma \uplus A$:

$$ea(\alpha, \langle \mu, \rho, \pi \rangle, \mathsf{v}) \triangleq \{(\langle \mu', \rho, \pi' \rangle, \mathsf{v}')^r \mid \mu.\alpha(\mathsf{v})_\pi \rightsquigarrow (\mu', \mathsf{v}')^r_{\pi'}\}.$$

Observe how the context of the state is passed to the memory execution function, which may then strengthen it before passing it back to the resulting state. We can show that the PCM and well-formedness relation on memories lift to a PCM and well-formedness relation on states, and that state action execution maintains properties analogous to those given for memory models.

2.2 GIL Verification

We give an overview of Gillian verification based on separation logic (SL); see [33] for details. We describe GIL assertions, parameterised by the core predicates of the TL, define assertion satisfiability in a novel, parametric way using the core predicate producers, and provide a mechanism for using verified function specifications in GIL execution.

GIL Assertion Syntax

$$p, q \in \mathcal{A} \triangleq \mathsf{emp} \mid p * q \mid \gamma\langle \hat{e}_i \cdot \hat{e}_o \rangle \mid \delta\langle \hat{e}_i \cdot \hat{e}_o \rangle$$
$$P, Q \in \mathcal{A}srt \triangleq \{p \wedge \pi \mid p \in \mathcal{A}, \pi \in \Pi\}$$
$$pred \in \mathcal{P}red \triangleq \text{pred } \delta\langle \hat{x}_i \cdot \hat{x}_o \rangle := P_1; \ldots; P_n;$$

A compositional memory model with core predicates Γ induces an SL-assertion language given on the right. GIL memory assertions, $p, q \in \mathcal{A}$, are formed using the empty assertion, the separating conjunction, the core predicates, and user-defined predicates, whose names come from a dedicated set, $\Delta \ni \delta$. The empty assertion and the separating conjunction are standard. Core predicate assertions are lifted from memory core predicates. User-defined predicates, introduced by example in §3 and §4, are used by tool developers to characterise the interface of the TL, and by code developers to describe the data structures in their programs. They have in- and out-parameters like core predicates, and can have multiple definitions, separated by a semi-colon. Assertions, $P, Q \in \mathcal{A}srt$, extend memory assertions with pure first-order assertions, π, conflated with Boolean symbolic expressions.

Satisfiability. To define assertion satisfiability, we lift memory consumers and producers from core predicates to memory assertions, denoted by $\mu.\mathrm{cons}_\theta(p)$ and $\mu.\mathrm{prod}_\theta(p)$, and then to states and arbitrary assertions, denoted by $\sigma.\mathrm{cons}_\theta(P)$ and $\sigma.\mathrm{prod}_\theta(P)$, using substitutions $\theta : \hat{\mathcal{X}} \mapsto \mathsf{V}$ (extended to symbolic expressions inductively, in the standard way) to map core predicate assertions, with parameters given by symbolic expressions, to the core predicates of the memory model, with parameters given by values. We highlight the successful base case of the memory assertion consumers, where the returned context requires the out-parameters of the assertion to match the ones found in memory:

$$\frac{\mu.\mathrm{cons}_\gamma(\theta(\hat{e}_i))_\pi \rightsquigarrow (\mu', \mathsf{v}'_o)^{\mathcal{S}}_{\pi'} \qquad \pi'' = (\pi' \wedge \mathsf{v}'_o = \theta(\hat{e}_o)) \qquad \pi'' \ \mathtt{SAT}}{\mu.\mathrm{cons}_\theta(\gamma\langle \hat{e}_i \cdot \hat{e}_o \rangle)_\pi \rightsquigarrow (\mu', \mathsf{true})^{\mathcal{S}}_{\pi''}}$$

and the successful consumption of an arbitrary assertion $P = p \wedge \pi$:

$$\frac{\mu'.\mathrm{cons}_\theta(p)_{\pi'} \rightsquigarrow (\mu'', \mathsf{true})^{\mathcal{S}}_{\pi''} \qquad \pi'' \vdash \theta(\pi)}{\langle \mu', \rho, \pi' \rangle.\mathrm{cons}_\theta(p \wedge \pi) \rightsquigarrow (\langle \mu'', \rho, \pi'' \rangle, \mathsf{true})^{\mathcal{S}}}$$

Definition 4 (Satisfiability). *The satisfiability relation, stating that memory μ' and context π' satisfy assertion $p \wedge \pi$ under substitution θ, is defined by:*

$$\mu', \pi', \theta \models p \wedge \pi \iff \mathbf{0}.\mathit{prod}_\theta(p)_{\mathsf{true}} \rightsquigarrow (\mu_p, \mathsf{true})^{\mathcal{S}}_{\pi_p} \wedge \pi' \vdash (\mu' = \mu_p \wedge \pi_p \wedge \theta(\pi))$$

and is lifted to states as: $\langle \mu', \rho, \pi' \rangle, \theta \models p \wedge \pi$ if and only if $\mu', \pi', \theta \models p \wedge \pi$.

In Definition 4, the production, when successful, creates the (unique) memory μ_p that corresponds to the resource of the assertion p, with its (unique) well-formedness constraints, π_p. In the concrete case, as the only allowed context is true, the formulation simplifies to the more intuitive $\mathbf{0}.\mathrm{prod}_\theta(p) \to (\mu', \mathsf{true})^{\mathcal{S}} \wedge \theta(\pi)$.

Specifications. Gillian function specifications have the form $\{\hat{x}, P\} f(x) \{Q\}^{\hat{e}}$, where f is the function identifier, x is the function parameter, $\hat{x}$ is the symbolic variable holding the value of x, P is the pre-condition, Q is the post-condition, and $\hat{e}$ is the return value of the function, with the following, well-known, constraints:

1. program variables do not appear in the pre- or the post-condition, and the function parameter x is accessed using the symbolic variable $\hat{x}$;
2. symbolic variables that appear in a pre-condition are implicitly universally quantified, and can be re-used in the corresponding post-condition; and
3. symbolic variables that appear only in a post-condition are implicitly existentially quantified.

We extend GIL programs with function specifications, accessible via $\mathsf{p}.\mathrm{specs}$, and the GIL execution semantics with rules for folding and unfolding user-defined predicates, as well as with a rule for calling function specifications, the success case of which is given below. Gillian verifies a specification $\{\hat{x}, P\} f(x) \{Q\}^{\hat{e}}$ if, given the identity substitution $\hat{\theta}$ and a symbolic state $\hat{\sigma}$ with store $\{x \mapsto \hat{\theta}(\hat{x})\}$ such that $\hat{\sigma}, \hat{\theta} \models P$, the symbolic execution of f starting from $\hat{\sigma}$ always terminates, for all final symbolic states $\hat{\sigma}_i$ there exists some $\hat{\theta}_i \geq \hat{\theta}$ such that $\hat{\sigma}_i, \hat{\theta}_i \models Q$, and the corresponding return value equals $\hat{\theta}_i(\hat{e})$ under the context of $\hat{\sigma}_i$. We can prove that if Gillian verifies a specification, then its standard SL interpretation holds.

Spec Call - Success

$$\frac{\begin{array}{ll}
\mathsf{cmd}(\mathsf{p}, cs, i) = y := e(e') \text{ with } \theta & \text{function call with substitution } \theta \\
\sigma.\mathrm{eval}_e(-) \rightsquigarrow f \quad \sigma.\mathrm{eval}_{e'}(-) \rightsquigarrow \mathsf{v}' & \text{get function id and parameter value} \\
\{\hat{x}, P\} f(x) \{Q\}^{\hat{e}} \in \mathsf{p}.\mathrm{specs} & \text{get one of the function specifications} \\
\theta' = \theta[\hat{x} \mapsto \mathsf{v}'] & \text{extend substitution with parameter value} \\
\sigma.\mathrm{cons}_{\theta'}(P) \to \{(\sigma_j, \mathsf{true})^{\mathcal{S}}|_{j \in J}\} & \text{consume pre-condition} \\
j \in J & \text{select a branch} \\
\sigma_j.\mathrm{prod}_{\theta'}(Q) \rightsquigarrow (\sigma'_j, \mathsf{true})^{\mathcal{S}} & \text{produce post-condition} \\
\sigma'_j.\mathrm{setVar}_y(\theta'(\hat{e})) \rightsquigarrow \sigma' & \text{assign return value}
\end{array}}{\mathsf{p} \vdash \langle \sigma, cs, i \rangle \rightsquigarrow \langle \sigma', cs, i{+}1 \rangle}$$

Note that for this rule to succeed, the consumption of P must succeed. The rule is slightly simplified for presentation. First, it assumes to have the substitution upfront; in the implementation, we have a unification algorithm that, starting from the function parameter and using the consumers, learns the substitution. Second, it assumes that the post-condition does not introduce fresh symbolic variables; these are handled using allocators and added to the substitution.

Remark. Due to space constraints, we have not been able to give the full technical details of Gillian verification. These are available in the Gillian technical report [33], where we demonstrate that the overall GIL execution using compositional memory models is frame-preserving (up to the usual renaming of allocated memory locations) and prove a standard verification soundness result.

3 Compositional Memory Models: JavaScript and C

We present the compositional memory models of JS and C, giving the basic actions and core predicates, and some of the user-defined predicates that capture the intuitive interfaces of these languages. The key ideas behind compositional JS memory models were introduced in the JaVerT project [21,20,22]; we transfer them to Gillian. We introduce the compositional C memory models, building on the concrete block-offset memory model of CompCert [31], simplifying the presentation.[9] In doing so, we highlight a striking similarity between the JS and C models that is the result of our emphasis on negative resource.

The JS and C concrete compositional memory models are made up of building blocks that are assigned a unique location (or identifier) from a set of uninterpreted symbols, $\mathcal{L} \subset \mathcal{U}$: for JS, the building blocks are the extensible objects; for C, they are the blocks of linear memory of a given size. Each building block is divided into at least one component. For JS, each object has three components: a property table, $h : \mathcal{S} \rightharpoonup \mathcal{V}al$, partially mapping property names (strings) to values; a domain, $d : \mathcal{P}(\mathcal{S})$, discussed shortly; and metadata, $m : \mathcal{V}al$, which keeps track of internal JS properties for that object [22]. For C, each block has two components: the block contents $k : \mathbb{N} \rightharpoonup \mathcal{V}al$, partially mapping offsets (natural numbers) to values; and a bound, $n : \mathbb{N}$, discussed shortly. Finally, the memory units are, intuitively, the parts of the memory components that cannot be separated further: for JS, these are single object properties, domains, and metadata; for C, these are single block cells and bounds. These memory units directly correspond to the core predicates given in Definitions 6 and 7.

Compositional memory models must keep track of negative resource, which can come from two sources: allocation and deallocation. For JS and C, the negative information originating from allocation has infinite representation: in JS, a freshly created object is known to not have any properties; in C, a freshly allocated block of a given size in C is known not to have offsets beyond that size. This infinite information is captured, for JS, by the object domain whose meaning

[9] We assume that values have the same size in memory and omit permissions. Gillian-C implements the full models, eliding the concurrency-related aspects of permissions.

is that any property not in the domain is absent, and, for C, by the block bound whose meaning is that any accesses beyond that bound result in a buffer overrun error. The negative information originating from deallocation is easier to handle, tracked by a dedicated uninterpreted symbol, $\varnothing \in \mathcal{U}$. In JS, deallocation is at the unit level: only object properties are deleted. This is captured by extending the co-domain of property tables with $\varnothing$: that is, $h : \mathcal{S} \rightharpoonup \mathcal{V}al_{\varnothing}$. In C, deallocation is at the building-block level: only entire blocks can be deleted. This is captured by extending the co-domain of blocks with $\varnothing$, indicating that a block has been freed.

Due to compositionality, any building block, component or unit can be missing. In the theory, we capture this either implicitly, via absence from the domain of a mapping (e.g., a missing object property for JS or a missing block cell for C), or explicitly, using the symbol $\perp$ (e.g. a missing domain, metadata, or bound).

Definition 5 (Compositional JS and C Memories). *The PCMs of compositional concrete JS and C memories, $|M_{JS}|$ and $|M_C|$, are given by the sets*

$$\begin{aligned} \mu \in |M_{JS}| &: \mathcal{L} \rightharpoonup ((\mathcal{S} \rightharpoonup \mathcal{V}al_{\varnothing}) \times \mathcal{P}(\mathcal{S})_{\perp} \times \mathcal{V}al_{\perp}), \\ \mu \in |M_C| &: \mathcal{L} \rightharpoonup ((\mathbb{N} \rightharpoonup \mathcal{V}al) \times \mathbb{N}_{\perp})_{\varnothing}, \end{aligned}$$

composition defined as disjoint union, and empty memory $\emptyset$. The PCMs of compositional symbolic JS and C memories, $|\hat{M}_{JS}|$ and $|\hat{M}_C|$, are given by the sets

$$\begin{aligned} \hat{\mu} \in |\hat{M}_{JS}| &: \hat{\mathcal{E}xpr} \rightharpoonup ((\hat{\mathcal{E}xpr} \rightharpoonup \hat{\mathcal{E}xpr}) \times \hat{\mathcal{E}xpr}_{\perp} \times \hat{\mathcal{E}xpr}_{\perp}), \\ \hat{\mu} \in |\hat{M}_C| &: \hat{\mathcal{E}xpr} \rightharpoonup ((\hat{\mathcal{E}xpr} \rightharpoonup \hat{\mathcal{E}xpr}) \times \hat{\mathcal{E}xpr}_{\perp})_{\varnothing}, \end{aligned}$$

with composition defined as (syntactic) disjoint union, and empty memory $\emptyset$.

In the above definition, symbolic memory models are simple liftings of the concrete ones. In the implementation, we employ heavy optimisation: for example, in Gillian-C, we have developed a complex tree representation of symbolic blocks inspired by [29], enabling tractable reasoning about arrays of symbolic size.

Well-formedness of concrete memories addresses the relationship between positive and negative information, given for JS and C below:

$$\begin{aligned} \mathcal{W}f^{\text{JS}}(\mu) &\triangleq \forall (h, d, -) \in \mathsf{codom}(\mu).\ d \neq \perp \implies \mathsf{dom}(h) \subseteq d \\ \mathcal{W}f^{\text{C}}(\mu) &\triangleq \forall (k, n) \in \mathsf{codom}(\mu).\ n \neq \perp \implies \mathsf{dom}(k) \subseteq [0, n) \end{aligned}$$

Well-formedness of symbolic memories additionally has to address separation of locations and separation in any other mappings with symbolic expressions in its domain (e.g. object properties for JS and offsets for C). We give the well-formedness criterion for the symbolic C memory:

$$\hat{\mathcal{W}f}^{\text{C}}_{\pi}(\hat{\mu}) \triangleq \pi \vdash \bigwedge_{\substack{\hat{l}, \hat{l}' \in \mathsf{dom}(\hat{\mu}) \\ \hat{l} \not\equiv \hat{l}'}} \hat{l} \neq \hat{l}' \wedge \bigwedge_{\substack{(\hat{k}, -) \in \mathsf{codom}(\hat{\mu}) \\ \hat{o}, \hat{o}' \in \mathsf{dom}(\hat{k}), \hat{o} \not\equiv \hat{o}'}} \hat{o} \neq \hat{o}' \wedge \bigwedge_{\substack{(\hat{k}, \hat{n}) \in \mathsf{codom}(\hat{\mu}) \\ \hat{o} \in \mathsf{dom}(\hat{k}), \hat{n} \not\equiv \perp}} \hat{o} < \hat{n}$$

For our JS and C instantiations, the core predicates follow straightforwardly from the units of their memory models.

$$\textsc{CConsCell - Found}\quad \frac{\mu(l) = (k, n) \qquad k(o) = v \qquad k' = k \setminus \{o\} \qquad \mu' = \mu[l \mapsto (k', n)]}{\mu.\mathsf{consCell}([l, o]) \rightsquigarrow (\mu', v)^{\mathcal{S}}}$$

$$\textsc{SConsCell - Use After Free}\quad \frac{\hat{\mu}(\hat{l}') = \varnothing \qquad \pi' = (\hat{l} = \hat{l}') \qquad (\pi \wedge \pi')\ \mathtt{SAT}}{\hat{\mu}.\mathsf{consCell}\,([\hat{l}, \hat{o}])_{\pi} \rightsquigarrow (\hat{\mu}, \mathsf{false})^{\mathcal{E}}_{\pi \wedge \pi'}}$$

$$\textsc{SConsCell - Found}\quad \frac{\hat{\mu}(\hat{l}') = (\hat{k}, \hat{n}) \quad \hat{k}(\hat{o}') = \hat{v} \qquad \pi' = ([\hat{l}, \hat{o}] = [\hat{l}', \hat{o}']) \quad (\pi \wedge \pi')\ \mathtt{SAT} \qquad \hat{k}' = \hat{k} \setminus \{\hat{o}'\} \quad \hat{\mu}' = \hat{\mu}[\hat{l}' \mapsto (\hat{k}', \hat{n})]}{\hat{\mu}.\mathsf{consCell}\,([\hat{l}, \hat{o}])_{\pi} \rightsquigarrow (\hat{\mu}', \hat{v})^{\mathcal{S}}_{\pi \wedge \pi'}}$$

$$\textsc{SConsCell - Missing Cell}\quad \frac{\hat{\mu}(\hat{l}') = (\hat{k}, \hat{n}) \qquad \pi_k = (\hat{l} = \hat{l}') \wedge \hat{o} \notin \mathsf{dom}(\hat{k}) \qquad \pi_n = (\hat{n} = \bot) \vee (\hat{n} \geq \hat{o}) \quad (\pi \wedge \pi_k \wedge \pi_n)\ \mathtt{SAT}}{\hat{\mu}.\mathsf{consCell}\,([\hat{l}, \hat{o}])_{\pi} \rightsquigarrow (\hat{\mu}, \mathsf{false})^{\mathcal{M}}_{\pi \wedge \pi_k \wedge \pi_n}}$$

Fig. 2: Selected rules for the consCell consumer.

Definition 6 (JS Core Predicates). *JS has three core predicates, $\gamma_{JS} \in \Gamma_{JS}$:*

- *the object-property predicate, $(\hat{l}, \hat{p}) \mapsto \hat{v}$, which states that property $\hat{p}$ of object at location $\hat{l}$ contains value $\hat{v}$ (including $\varnothing$ denoting property absence);*
- *the domain predicate, $\mathsf{domain}(\hat{l}, \hat{d})$, which states that object at location $\hat{l}$ has no properties outside the finite set $\hat{d}$;*
- *the metadata predicate, $\mathsf{metadata}(\hat{l}, \hat{m})$, which states that object at location $\hat{l}$ has metadata $\hat{m}$.*

Definition 7 (C Core Predicates). *C has three core predicates, $\gamma_C \in \Gamma_C$ [10]:*

- *the cell predicate, $(\hat{l}, \hat{o}) \mapsto \hat{v}$, which states that the cell at offset $\hat{o}$ in the block at location $\hat{l}$ contains value $\hat{v}$ (which, this time, does not include $\varnothing$);*
- *the bounds predicate, $\mathsf{bound}(\hat{l}, \hat{n})$, which states that any cell beyond offset $\hat{n}$ in block at location $\hat{l}$ is not there;*
- *the freed predicate, $\hat{l} \mapsto \varnothing$, which states that block at location $\hat{l}$ has been freed.*

We illustrate the C predicate action execution functions, $\underline{\mathsf{ea}}_{C}$ and $\underline{\hat{\mathsf{ea}}}_{C}$, respectively, with a selection of rules for the C cell-predicate consumer, consCell, given in Figure 2. The remaining rules, as well as the rules for their JS counterparts, $\underline{\mathsf{ea}}_{JS}$ and $\underline{\hat{\mathsf{ea}}}_{JS}$, can be found in the Gillian technical report [33]. With this information, we can define the compositional concrete and symbolic JS and C memory models.

Definition 8 (JS Memory Models). *The compositional concrete and symbolic JS memory models are defined, respectively, as $M_{JS}(\mathcal{V}al, \Gamma_{JS}) = \langle |M_{JS}|, \mathcal{W}f^{JS}, \underline{\mathsf{ea}}_{JS} \rangle$ and $\hat{M}_{JS}(\hat{\mathcal{E}}xpr, \Gamma_{JS}) = \langle |\hat{M}_{JS}|, \hat{\mathcal{W}f}^{JS}, \underline{\hat{\mathsf{ea}}}_{JS} \rangle$.*

Definition 9 (C Memory Models). *The compositional concrete and symbolic C memory models are defined, respectively, as $M_C(\mathcal{V}al, \Gamma_{JS}) = \langle |M_C|, \mathcal{W}f^{C}, \underline{\mathsf{ea}}_{C} \rangle$ and $\hat{M}_C(\hat{\mathcal{E}}xpr, \Gamma_{JS}) = \langle |\hat{M}_C|, \hat{\mathcal{W}f}^{C}, \underline{\hat{\mathsf{ea}}}_{C} \rangle$.*

[10] In full C and the Gillian-C implementation, memory values may be of different sizes, and holes may exist between these values due to alignment restrictions. To address this, the implemented cell assertion carries additional information related to, e.g., size and type, similarly to that of [4], and there also exists a `hole` core predicate.

The getters and setters for JS and C are defined using the methodology described in §2. In particular, the JS getters and setters are given by $A_{\text{JS}} = \{\mathsf{getProp}, \mathsf{setProp}, \mathsf{getDomain}, \mathsf{setDomain}, \mathsf{getMetadata}, \mathsf{setMetadata}\}$, and the summary of the execution of the symbolic $\mathsf{getProp}(\hat{l}, \hat{p})$ getter is illustrated below:

$$\begin{tikzcd}
\hat{l} \in \mathrm{dom}(\hat{\mu}) \arrow[r, "\hat{\mu}(\hat{l}) = (\hat{h}, \hat{d}, \hat{m})", "\text{yes}"'] \arrow[d, "\text{no}"] & \hat{p} \in \mathrm{dom}(\hat{h}) \arrow[r, "\text{no}"'] \arrow[d, "\text{yes}"] & \hat{d} = \bot \arrow[r, "\text{yes}"] \arrow[d, "\text{no}"] & \text{✗}\ (\hat{l}, \hat{p})\ \text{Missing property} \\
\text{✗}\ (\hat{l}, \hat{p})\ \text{Missing object} & \text{✓}\ \hat{h}(\hat{p}) & \hat{p} \in \hat{d} \arrow[ur, "\text{yes}"] \arrow[r, "\text{no}"'] & \emptyset
\end{tikzcd}$$

Similarly, the C getters and setters are given by $A_{\text{C}} = \{\mathsf{getCell}, \mathsf{setCell}, \mathsf{getBound}, \mathsf{setBound}, \mathsf{getFreed}, \mathsf{setFreed}\}$ and the summary of the execution of the symbolic $\mathsf{getCell}(\hat{l}, \hat{o})$ getter is illustrated below:

$$\begin{tikzcd}
\hat{l} \in \mathrm{dom}(\hat{\mu}) \arrow[r, "\text{yes}"] \arrow[d, "\text{no}"] & \hat{\mu}(\hat{l}) = \emptyset \arrow[r, "\hat{\mu}(\hat{l}) = (\hat{k}, \hat{n})", "\text{no}"'] \arrow[d, "\text{yes}"] & \hat{o} \in \mathrm{dom}(\hat{k}) \arrow[r, "\text{no}"'] \arrow[d, "\text{yes}"] & \hat{n} = \bot \arrow[r, "\text{yes}"] \arrow[d, "\text{no}"] & \text{✗}\ (\hat{l}, \hat{o})\ \text{Missing cell} \\
\text{✗}\ (\hat{l}, \hat{o})\ \text{Missing block} & \text{⚡ Use after free} & \text{✓}\ \hat{k}(\hat{o}) & \hat{o} \geq \hat{n} \arrow[ur, "\text{no}"] \arrow[r, "\text{yes}"'] & \text{⚡ Buffer overrun}
\end{tikzcd}$$

The similarities in the two diagrams are evident, with the main difference being that JS getters do not throw errors, whereas C getters do.

User-defined JS and C Predicates. Core predicates describe fundamental units of the TL memory model. On top, user-defined predicates build layers of abstraction to describe memory components and building blocks, standard library interfaces, all the way to complex data structures for particular code such as the AWS message header. Using Gillian notation, we present some of the JS and C user-defined predicates; in this notation: $*$ and $\wedge$ are conflated to $*$, with automatic differentiation between spatial and pure assertions[11]; predicate definitions are separated with a semi-colon; and logical variables are prefixed with the # symbol and are implicitly existentially quantified in predicate definitions.

Gillian-JS inherits many user-defined predicates from JaVerT [21], including simple ones for describing JS objects and their properties, as well as advanced ones for specifying scoping, function closures and prototype chains. We focus here on the new `FrozenObject(o, proto, pvs)` predicate, which describes a frozen object[12] `o` with prototype `proto` and property-value pairs `pvs`. We first define the predicate `FrozenObjectProps(o, pvs)` to grab the resource of the object properties:

```
pred FrozenObjectProps(o, pvs) : pvs = [ ];
    pvs = [#p, #v] :: #rpvs * DataPropConst(o, #p, #v) *
    FrozenObjectProps(o, #rpvs);
```

where `DataPropConst(o, #p, #v)` states that the object `o` has a non-writable property `#p` with value `#v`. We then add information about the object prototype and its non-extensibility using the `JSObject(o, proto, ext)` predicate, and also state that the object has no properties other than `pvs` using the domain core predicate:

[11] From the separation logic literature, the pure assertions can be regarded as dotted.
[12] A JS object is frozen if it cannot be extended and all its properties are non-writable.

```
pred FrozenObject(o, proto, pvs) :
    JSObject(o, proto, false) * FrozenObjectProps(o, pvs) *
    FirstProj(pvs, #ps) * ListToSet(#ps, #pss) * domain(o, #pss)
```

where FirstProj(pvs, #ps) means that the list #ps is the first projection of the list of pairs pvs, and ListToSet(#ps, #pss) means that the elements of the list #ps form the set #pss.

Gillian-C, on the other hand, comes with user-defined predicates capturing, for example, arrays and blocks in memory, as well as automatically-generated predicates describing C structs, with support for nested structs. In particular, the array(b, off, c) predicate describes a contiguous fragment of a block b, starting from offset off, with contents described by the mathematical list c:

```
array(b, off, c) : c = [];
                   (b, off) -> #c * array(b, off+1, #d) * c = #c :: #d
```

and the block(b, c) predicate captures an entire C block with contents c:

```
block(b, c) : array(b, 0, c) * bound(b, |c|)
```

In the implementation, arrays also exist as core predicates. This allows us to reason about arrays automatically in the symbolic memory (e.g., to split an array into sub-arrays), supported by our tree representation of symbolic blocks, instead of requiring manual application of lemmas.

Finally, we illustrate automatically generated struct-related predicates using the aws_byte_cursor structure given below, which contains two fields: an unsigned integer len; and a nullable pointer to an array of 8-bit unsigned integers buf. This struct is used for traversing the AWS message header (cf. §4), and is intended to capture an array in memory that starts at buf and has length len.

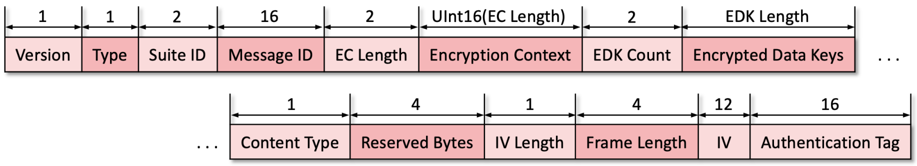
```
struct aws_byte_cursor {  pred struct_aws_byte_cursor(cur, len, buf) :
  size_t len;               (cur == [#b, #o]) * ((#b, #o) -int64-> len) *
  uint8_t *buf;             ((#b, #o +p 8) -int64-> buf) *
}                           is_ptr_or_null(buf)
```

The generated predicate describes the struct's layout in memory and gives basic typing information: it states that an aws_byte_cursor, starting from the position given by the pointer cur, occupies 16 bytes in memory (8 + 8, given by the type annotation int64), with the first 8 bytes taken by len, and the second 8 bytes (note the pointer addition +p) taken by buf, which is either a pointer or null.

4 AWS Encryption SDK Message Header Specification

The encrypted data handled by the AWS Encryption SDK is stored within a structure called a message [3]. The message format has two versions of similar complexity: we verify version 1; version 2 was introduced very recently. Messages consist of a header, a body, and a footer. Here, we describe only the structure of the header, as we are verifying header deserialisation.

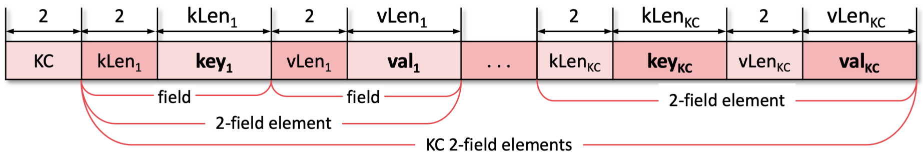
The AWS Encryption SDK message header is a sequence of bytes (buffer) divided into sections, as illustrated below; above each section is its length in bytes.



Our approach is to abstract the header contents into a list and formulate pure predicates that describe its structure in a language-independent way. This allows us to then use the same abstractions as part of further, language-dependent, abstractions for both JS and C. Our design of the abstractions was informed by existing code annotations found in the implementations, which describe simple first-order properties of the code and, in the case of C, can also link to the CBMC [30] bounded model checker. However, these annotations are limited by the expressivity of JS and C, particularly when it comes to reflecting on the memory contents. Our predicates have no such limitations.

We narrow down our exposition to the encryption context, as it illustrates well the language-independent and language-dependent aspects of our specification, and is also the section in which we discovered bugs in both implementations.

Pure Specification of the Encryption Context. The encryption context (EC) is a sequence of bytes that describes a set of key-value pairs. Its structure is given in the diagram below.

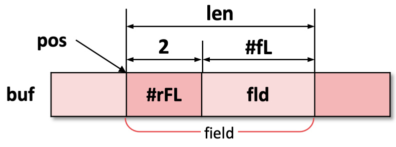


The first two bytes represent the number of key-value pairs, denoted by `KC`, and the rest describe the `KC` key-value pairs themselves. Keys and values are represented by sequences of bytes and, as they are of variable length, are serialised by first having *two bytes* that represent the length, followed by *that many bytes* of the actual key or value; we refer to this pattern as a field, and to a sequence of n fields as an n-element. Then, a key-value pair is serialised as a 2-field element, and all of the key-value pairs form a sequence of `KC` 2-field elements.

We specify the EC by building layers of abstraction, from fields to elements to element sequences to the EC, each of which can either be complete, incomplete (partial, but with correct structure), or malformed (with incorrect structure). In the implementation, these are specified separately and are joined together in appropriate over-arching abstractions. Here, we focus on complete variants only.

The `Field`(`buf`, `pos`, `fld`, `len`) predicate states that the buffer (list of bytes) `buf`, at index `pos`, holds a field with contents `fld` (list of bytes) and total length `len`:

```
pred Field(buf, pos, fld, len) :
 (0 <= pos) * (#rFL = sub(buf, pos, 2)) *
 UInt16(#rFL, #fL) *
 (fld = sub(buf, pos+2, #fL)) *
 (len = 2+#fL) * (pos+len <= |buf|)
```



This predicate uses the GIL operator `sub(l, s, n)`, which returns the sublist of list `l` starting from index `s` and of length `n`, and also the `UInt16(rn, n)` predicate, which states that `n` is a 16-bit big-endian interpretation of the raw 2-byte list `rn`. The `Element(buf, pos, fC, elem, len)` predicate states that buffer `buf` at index `pos` holds a sequence of `fC` fields, with contents `elem` (a list of the appropriate field contents) and total length `len`. It is defined similarly to a standard linked-list predicate, with the 'link' being the fact that the list members are contiguous in memory:

```
pred Element(buf, pos, fC, elem, len) :
  (fC = 0) * (0 <= pos) * (pos <= |buf|) * (elem = [ ]) * (len = 0);
  (0 < fC) * Field(buf, pos, #fld, #fL) * Element(buf, pos+#fL, fC-1, #rFs,
  #rL) * (elem = #fld :: #rFs) * (len = #fL+#rL)
```

Next, analogously to `Element`, we define the `Elements(buf, pos, eC, fC, elems, len)` predicate, which states that the buffer `buf`, at index `pos`, holds a sequence of `eC` elements, each with `fC` fields, with contents `elems` (a list of the appropriate element contents) and of total length `len`. Finally, the `EncryptionContext(buf, KVs)` predicate states that the entire buffer `buf` is an EC with key-value pairs `KVs`, with all keys being unique:

```
pred EncryptionContext(buf, KVs) : (buf = [ ]) * (KVs = [ ]);
     (#rKC = sub(buf, 0, 2)) * UInt16(#rKC, #KC) * (0 < #KC) *
     Elements(buf, 2, #KC, 2, KVs, #len) *
     FirstProj(KVs, #Ks) * Unique(#Ks) * (2+#len = |buf|)
```

Next, we show how this pure specification of the EC contents can be connected <u>without modification</u> to both the JS and C memories.

Encryption Context in JS. In JS, the EC is serialised as an ArrayBuffer, which is a raw binary data buffer in memory, and accessed using a Uint8Array, which is a view on top of that ArrayBuffer starting from a given offset and of a given length, treating the raw data underneath as 8-bit unsigned integers. This Uint8Array view is similar in function to the `aws_byte_cursor` C structure (cf. §3). Abstracting ArrayBuffer contents to lists, we connect these data structures in JS memory to our pure EC specification (cf. Figure 3, top and centre):

```
pred JSSerEC(o, EC, KVs) :
     Uint8Array(o, #aBuf, #off, #len) * ArrayBuffer(#aBuf, #data) *
     (EC = sub(#data, #off, #len)) * EncryptionContext(EC, KVs)
```

In JS, the EC is deserialised into a frozen JS object with prototype `null`, whose properties represent the keys and hold the values. This is done by converting the keys and the values to UTF-8 strings, and is specified as follows:

```
pred JSDeserEC(o, KVs) : toUtf8(KVs, #sKVs) * FrozenObject(o, null, #sKVs)
```

where `toUtf8` converts the list `KVs` point-wise to strings, obtaining `#sKVs`.

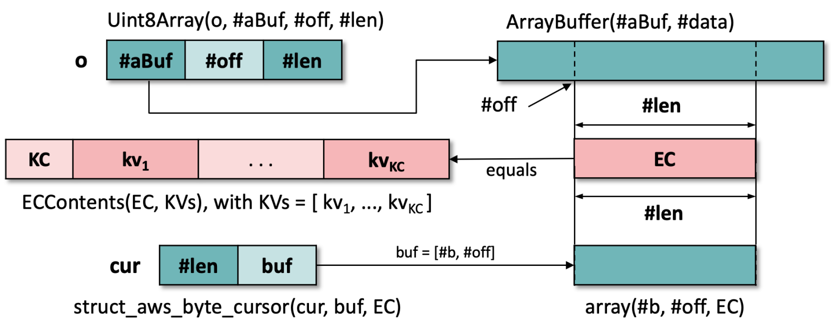

Uint8Array(o, #aBuf, #off, #len)
o
#aBuf
#off
#len
ArrayBuffer(#aBuf, #data)
#off
#len
KC
kv_1
. . .
kv_{KC}
equals
EC
ECContents(EC, KVs), with KVs = [kv_1, ..., kv_{KC}]
#len
cur
#len
buf
buf = [#b, #off]
struct_aws_byte_cursor(cur, buf, EC)
array(#b, #off, EC)

Fig. 3: Serialised Encryption Context: language-independent pure part (red; middle) and language-specific resource (green; JS above, C below)

Finally, the specification of the `decodeEncryptionContext` function states that the EC deserialisation is performed correctly.

```
{ JSSerEC(eEC, #EC, #KVs)                          }
  function decodeEncryptionContext(eEC)
{ PRE-CONDITION * JSDeserEC(ret, #KVs)             }
```

Encryption Context in C. In C, the EC is serialised as a block in memory, and is traversed using an AWS byte cursor. Using the auto-generated predicate given in §3, we define the `aws_byte_cursor`(`cur`, `buf`, `c`) predicate, stating that `cur` points to a byte cursor which has access to an array starting from `buf`, and holding contents `c`, making the length implicit:

```
  pred aws_byte_cursor(cur, buf, c) :
    struct_aws_byte_cursor(cur, #len, buf) * (buf = [#b, #off]) *
    array(#b, #off, c) * (#len = |c|)
```

A serialised EC can then be described as a valid byte cursor whose contents represent the EC key-value pairs (cf. Figure 3, centre and bottom):

```
pred CSerEC(cur, buf, EC, KVs) :
  aws_byte_cursor(cur, buf, EC) * EncryptionContext(EC, KVs)
```

In C, the EC is deserialised into an AWS hash table, whose keys and values directly correspond to the key/value pairs of the EC, specified as follows, eliding the internal structure of the hash tables due to space constraints:

```
pred CDeserEC(ht, KVs) : valid_hash_table(ht, KVs)
```

The specification of the EC deserialisation function is more complex than for JS. In particular, the byte cursor that originally pointed to the EC ends up shifted to the end of the byte buffer, exposing the array underneath the `CSerEC` predicate.

```
{   empty_hash_table(ec) * CSerEC(cur, #buf, #EC, #KVs)            }
    int aws_cryptosdk_enc_ctx_deserialize(
        struct aws_hash_table *ec, struct aws_byte_cursor *cur)
{   (ret = 0) * CDeserEC(ec, #KVs) * (#buf = [#b, #off]) *
    array(#b, #off, #EC) * aws_byte_cursor(cur, #buf +p |#EC|, [ ])  }
```

5 AWS Encryption SDK Message Header Verification

Using Gillian-JS and Gillian-C, together with the specifications given in §4, we verify full functional correctness of the header deserialisation module of the AWS Encryption SDK JS [2] (~200loc) and C [1] (~950loc) implementations. In particular, we verify that the deserialisation of a complete header is correct, and the deserialisation of an incomplete or a malformed header raises an appropriate error.

Verification Effort and Performance. The JS verification took 3 person-months and the C verification took 2 person-months, with the latter taking less time because a large part of the infrastructure developed for JS could be re-used. We substantially improved the first-order solver of Gillian to reason automatically about complex operations on lists of symbolic length, first used in the modelling of JS ArrayBuffers and then for C dynamic arrays. We created a collection of language-independent predicates and lemmas about their inductive properties (~1.2kloc) that cover the project-specific AWS header, but also re-usable first-order concepts such as list element uniqueness, projections of lists of pairs, conversion from bytes to numbers, and conversion from raw bytes to strings. Similarly, we also had to create language-dependent abstractions and associated lemmas for the JS and C manipulation of the AWS message header (~1.2kloc). Finally, we had to: annotate the code with specifications and loop invariants, with the latter often having more than twenty components; manually apply lemmas to prove numerous complex entailments; and manually unfold user-defined predicates at times (the folding is automated) (~1.1kloc).

On a machine with an Intel Core i7-4980HQ CPU 2.80 GHz, DDR3 RAM 16GB, and a 256GB solid-state hard-drive running macOS, the JS verification takes approximately 45 seconds and the C verification takes approximately six minutes. The C time is longer, in part due to the larger codebase, but mainly due to the complexity of the implementation of the full C memory model, which is able to reason about arrays of symbolic size. This requires frequent satisfiability checks and (for the moment) branching on non-zero array size. These times could both be improved with the implementation of basic merging techniques.

JS Verification: Bugs/Improvements. We discovered two bugs and improved one function implementation to link better with the underlying data structure.

– In the `decodeEncryptionContext` function, the object representing the deserialised EC originally had prototype `Object.prototype` which, in this case, due to the prototype inheritance of JavaScript, meant that if an EC key coincided with a property of `Object.prototype`, an error would be thrown incorrectly. This bug was predicted theoretically in [21], and has since been found in several real-world libraries [42], including `cash` and `jQuery`.
– In the same function, in one of the branches the deserialised EC was returned non-frozen, which constituted a potential vulnerability in that third parties could alter non-secret, but authenticated data.
– The `readElements(eC, fC, buf, pos)` function, which reads `eC` elements with `fC` fields from buffer `buf` at index `pos` into a JS array of arrays, was misaligned

with the underlying data structures. Its parameters were non-intuitive (it received `eC·fC`, `buf`, and `pos`), and used complex array operations to re-form the final return value. We re-implemented this function to construct the returned array of arrays efficiently, simplifying specification and verification, and our implementation was integrated into the codebase.

JS Verification: Caveats. Our JS verification is correct up to the following caveats. First, as the AWS SDK JS implementation is written in TypeScript, we elide types to obtain JS; this could be automated, potentially generating predicates from the types. Next, some ES6 features, such as patterns in function parameters, are not yet supported by Gillian-JS; these we rewrite to ES5 Strict, preserving their meaning. Next, we use axiomatic specifications of the ArrayBuffer, DataView, and UInt8Array ES6 built-in libraries, as well as of the `Object.freeze` and `Array.prototype.map` built-in functions. These would ideally be accompanied with implementations, tested against the official Test262 test suite [16] and verified against their specifications. Finally, as Gillian does not support higher-order reasoning, we axiomatise the `toUtf8` function, passed into the deserialisation module as a parameter, as an injective function from raw bytes to JS strings.

C Verification: Bugs. We discovered three bugs: one logical error; one undefined behaviour; and one over-allocation.

– The deserialisation of the EC mishandled the case when there is not enough data to read it entirely, continuing to read the EDK instead of reporting an error. This allows some malformed headers to be parsed as well-formed.
– The function `aws_byte_cursor_advance`, when called with a `NULL` cursor and a length of 0, resulted in `NULL` + `0` being computed, which is undefined behaviour, although not problematic for most compilers.
– The deserialised EC was stored using `aws_string`, which extends C strings with certain metadata. It is implemented using a structure that includes a flexible array member. We discovered that string creation over-allocated this array by 8 bytes, because our (correct) predicate describing `aws_string`s was not allowing the verification to go through.

C Verification: Caveats. Our C verification is correct up to the following caveats. First, we do not use the `aws_byte_cursor_advance_nospec` function, which advances the byte cursor, but also uses complex computation to protect against the Spectre bug. We instead use `aws_byte_cursor_advance`, which has equivalent behaviour, as our specifications are not expressive enough to capture this distinction. Next, we axiomatise the functions of the AWS hash tables and array list libraries, as their verification is of comparable complexity to the entire deserialisation module. Finally, the AWS allocators of the C implementation, which are passed into some of the functions, contain pointers to memory management functions; this is higher-order in nature. In the verification, we assume those functions are `malloc`, `calloc`, and `realloc`.

6 Related Work

The literature explores many techniques and tools for verifying JS [44,18,22,21] and C [23,26,28,13,7]. We describe: multi-language verification architectures; JS and C verification tools based on separation logic; C memory models related to our models; and other analyses applied to the AWS Encryption SDK.

Multi-Language Verification Architectures. The multi-language verification architectures closest to Gillian are coreStar [6] and Viper [36,35]. Both of these architectures were designed to serve as verification back-ends for TLs and both have at their core a simple intermediate representation with a dedicated symbolic execution engine[13]. However, they work with the TL in different ways.

In coreStar, TL core assertions are modelled as abstract predicates and memory actions as function calls. The function specifications play the role of our consumer and producer actions. The user also has to provide logical axioms, describing properties of the abstract predicates. The Gillian equivalent of these axioms are the implementations of the memory actions using consumers and producers, which can be optimised, but require understanding of the inner workings of Gillian. Like Gillian, coreStar's symbolic execution engine is parametric on the underlying logical theory and can thus be used to reason about any memory model representable using abstract predicates. It is, however, unclear how efficiently this can be done. coreStar has been used inside the tool jStar [15], which has verified implementations of several Java design patterns but was not pushed to more complex Java code. In [21], the authors observed that coreStar was not able to handle tractably even simple JS programs.

Unlike Gillian and coreStar, Viper [35,36] comes with a fixed intermediate language, also called Viper. The user encodes their memory model and corresponding core assertions into the memory model and assertion language of Viper. A key advantage of Viper lies in its expressive permission model, which includes fractional, recursive, and abstract read permissions, as well as in its support for custom mathematical domains, which enable users to extend Viper with their own first-order theories, tailored to the data structures at hand. Viper has mechanisms similar to our consumer and producer actions, called *inhale* and *exhale*. Viper can reason about both sequential and concurrent programs, and has been used to verify programs written in Java, Go, Rust, and Python, but not JS and C. In fact, it is not clear to us how difficult it would be to use Viper to reason about JS objects and the linear memory of C, as neither can be simply expressed using the static objects natively provided by Viper.

Semi-automatic JS and C Verification Tools. There are very few verification tools for JS based on separation logic. For example, JaVerT [21] has been used to verify simple sequential data-structure algorithms. Its successor, JaVerT 2.0 [22], provides whole-program symbolic testing, verification and bi-abductive reasoning [10], unified by a core symbolic execution engine.

[13] Viper includes both a symbolic execution engine and a verification condition generator based on Boogie [5] for its intermediate language.

JaVerT 2.0 verification is more efficient than JaVerT verification, but has still only been applied to simple data-structure algorithms. Gillian [19] builds on JaVerT 2.0, taking the highly non-trivial step of designing the intermediate language, correctness results, and implementation to be parametric on the TL memory models. Despite this generalisation, Gillian substantially outperforms JaVerT 2.0, both for symbolic testing [19] and for verification.

Verifast [26] and the tool in [7] are prominent examples of semi-automatic tools that provide functionally-correct verification of C programs using separation-logic specifications. These tools work with C fragments and simplified memory models. While the tool in [7] has not been applied to real-world code, Verifast has been used to verify, e.g., an implementation of a Policy Enforcement Point (PEP) for Network Admission Control scenarios [38]. One difference between these tools and Gillian is that Gillian specifications can express negative resource, allowing us to differentiate missing resource errors from use-after-free errors. However, Verifast, unlike Gillian, supports reasoning about concurrent programs. There is also much work on using theorem provers to verify both sequential and concurrent C code using separation logic: see, for example, the DeepSpec project [45] and the Iris project [47], which we do not describe here.

Related Formal C Memory Models. Our compositional C memory models were inspired by CompCert [32] and the CH20 formalisation of Krebbers [29]. In particular, our concrete C model is adapted from the complete model of CompCert, which supports reasoning about programs that access in-memory data representations. This feature is used by the AWS deserialisation algorithm, which reads the buffer contents at the byte-granularity.

We present our compositional symbolic C memory model in this paper as a simple lifting of the concrete one. Our implementation is more complex, however, representing blocks as trees holding symbolic values and combining the concepts of memory trees and abstract values from the concrete memory model of the CH2O formalisation. Although not mentioned in [29], CH2O does keep track of some negative resource in that it maintains freed locations, but not block bounds.

Analysis of the AWS Encryption SDK. Amazon has recently directed considerable effort towards the formal analyses of their codebase, with a number of tools incorporated into their CI pipeline. For example, the main cryptographic algorithms of the AWS Encryption SDK have certified implementations in the specification language Cryptol [17], underpinned by SAW [12]. These implementations, however, have not yet been proven equivalent to the corresponding C implementation. In addition, the C implementation of the AWS Encryption SDK includes a symbolic test suite run using CBMC [30]. This implementation makes heavy use of the aws-c-common data-structure library, which is annotated with first-order assertions checked by CBMC. CBMC is a mature, industrial-strength tool, likely to outperform and have broader coverage than the symbolic testing of Gillian-C, with substantially fewer annotations than Gillian verification. However, as CBMC is a bounded model checker, it provides weaker correctness guarantees and is not compositional. Its expressivity is also somewhat constrained by the expressivity of the C runtime. For example, it does not allow reasoning about the size of allocated memory. Gillian specifications have this expressivity, as highlighted by the discovered over-allocation bug. The subtle logical bug found by Gillian also demonstrates the importance of being able to express full, functionally-correct specifications. We believe there has been no previous analysis of the JS implementation of AWS Encryption SDK.

7 Conclusions

We have introduced compositional verification to the Gillian platform. Our work includes a methodology for designing compositional TL memory models, distinguishing negative resource from missing resource and using the JS and C memory models as demonstrator examples. It also includes a novel, parametric approach to assertion interpretation, independent of the TL, enabling compositional use of function specifications in verification. We have been able to push the Gillian verification to self-contained, critical, real-world AWS JS and C code. The bugs and suggestions for code improvements that arose during this verification process have all been accepted by the developers and incorporated into the codebase. To our knowledge, this is the first time that industry-grade JS code has been fully verified and the first time that, in one verification platform, the same abstractions were used to verify industry code from languages as different as JS and C. The artifact accompanying this paper can be found at [34], and the entire Gillian development at [46]. In future, we will publish correctness results for Gillian verification [33], as part of an in-depth theoretical study of program correctness and incorrectness for symbolic testing, verification and bi-abductive reasoning being developed in Gillian.

Acknowledgments. We thank AWS engineers and Mike Dodds from Galois for several inspiring meetings which led to our focus on verifying the JS implementation of the AWS Encryption SDK message header deserialisation module. We would especially like to thank Ryan Emery for many detailed discussions about his JS code. We thank the reviewers, whose comments have improved the overall quality of the paper. Gardner and Maksimović were supported by the EPSRC Fellowship 'VetSpec: Verified Trustworthy Software Specification' (EP/R034567/1). Fragoso Santos was supported by national funds through Fundação para a Ciência e a Tecnologia (UIDB/50021/2020, INESC-ID multi-annual funding) and project INFOCOS (PTDC/CCI-COM/32378/2017). Ayoun was supported by a Department of Computing PhD Scholarship from Imperial.

References

1. Amazon Web Services: AWS Encryption SDK: C Implementation. https://github.com/aws/aws-encryption-sdk-c (2020)
2. Amazon Web Services: AWS Encryption SDK: JS Implementation. https://github.com/aws/aws-encryption-sdk-javascript (2020)

3. Amazon Web Services: AWS Encryption SDK: Message Format. https://docs.aws.amazon.com/encryption-sdk/latest/developer-guide/message-format.html (2020)
4. Appel, A.W., Blazy, S.: Separation Logic for Small-Step Cminor. In: TPHOL (2007)
5. Barnett, M., Chang, B.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) FMCO (2005)
6. Botinčan, M., Distefano, D., Dodds, M., Grigore, R., Naudžiūnienė, D., Parkinson, M.J.: coreStar: The core of jstar. In: Boogie (2011)
7. Botinčan, M., Parkinson, M.J., Schulte, W.: Separation Logic Verification of C Programs with an SMT Solver. Electr. Notes Theor. Comput. Sci. **254**, 5–23 (2009)
8. Calcagno, C., Distefano, D.: Infer: An Automatic Program Verifier for Memory Safety of C Programs. In: NFM (2011)
9. Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P.W., Papakonstantinou, I., Purbrick, J., Rodriguez, D.: Moving Fast with Software Verification. In: NFM (2015)
10. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional Shape Analysis by Means of Bi-Abduction. JACM **58**, 26:1–26:66 (2011)
11. Calcagno, C., O'Hearn, P.W., Yang, H.: Local action and abstract separation logic. In: LICS (2007)
12. Chudnov, A., Collins, N., Cook, B., Dodds, J., Huffman, B., MacCárthaigh, C., Magill, S., Mertens, E., Mullen, E., Tasiran, S., Tomb, A., Westbrook, E.: Continuous Formal Verification of Amazon s2n. In: CAV (2018)
13. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A Practical System for Verifying Concurrent C. In: TPHOL (2009)
14. Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M.J., Vafeiadis, V.: Concurrent abstract predicates. In: ECOOP (2010)
15. Distefano, D., Parkinson, M.: jStar: Towards practical verification for Java. In: OOPSLA (2008)
16. ECMA TC39: Test262 Test Suite. https://github.com/tc39/test262 (2017)
17. Erkök, L., Matthews, J.: High Assurance Programming in Cryptol. In: CSIIRW (2009)
18. Fournet, C., Swamy, N., Chen, J., Dagand, P.E., Strub, P.Y., Livshits, B.: Fully Abstract Compilation to JavaScript. In: POPL (2013)
19. Fragoso Santos, J., Maksimović, P., Ayoun, S.E., Gardner, P.: Gillian, Part I: A Multi-language Platform for Symbolic Execution. In: PLDI (2020)
20. Fragoso Santos, J., Maksimović, P., Grohens, T., Dolby, J., Gardner, P.: Symbolic Execution for JavaScript. In: PPDP (2018)
21. Fragoso Santos, J., Maksimović, P., Naudžiūnienė, D., Wood, T., Gardner, P.: JaVerT: JavaScript Verification Toolchain. PACMPL **2**(POPL), 50:1–50:33 (2018)
22. Fragoso Santos, J., Maksimović, P., Sampaio, G., Gardner, P.: JaVerT 2.0: Compositional Symbolic Execution for JavaScript. PACMPL **3**(POPL), 66:1–66:31 (2019)
23. Frumin, D., Gondelman, L., Krebbers, R.: Semi-automated Reasoning About Non-determinism in C Expressions. In: PLS (2019)
24. Gardner, P., Maffeis, S., Smith, G.D.: Towards a Program Logic for JavaScript. In: POPL (2012)
25. Ishtiaq, S.S., O'Hearn, P.W.: BI as an Assertion Language for Mutable Data Structures. In: Hankin, C., Schmidt, D. (eds.) POPL (2001)

26. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In: NFM (2011)
27. Jung, R., Krebbers, R., Jourdan, J., Bizjak, A., Birkedal, L., Dreyer, D.: Iris from the ground up: A modular foundation for higher-order concurrent separation logic. J. Funct. Program. **28** (2018)
28. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-c: A software analysis perspective. Formal Aspects of Computing **27**(3), 573–609 (2015)
29. Krebbers, R.: A Formal C Memory Model for Separation Logic. Journal of Automated Reasoning **57**(4), 319–387 (2016)
30. Kroening, D., Tautschnig, M.: CBMC: C bounded model checker. In: TACAS (2014)
31. Leroy, X., Appel, A.W., Blazy, S., Stewart, G.: The CompCert Memory Model, Version 2. Research Report RR-7987, INRIA (2012)
32. Leroy, X.: Formal Verification of a Realistic Compiler. Commun. ACM **52**(7), 107–115 (2009)
33. Maksimović, P., Santos, J.F., Ayoun, S.E., Gardner, P.: Gillian: A Multi-Language Platform for Unified Symbolic Analysis (2021), `http://arxiv.org/abs/2105.14769`
34. Maksimović, P., Ayoun, S.E., Fragoso Santos, J., Gardner, P.: Artifact: Gillian, Part II: Real-World Verification for JavaScript and C (2021), `https://doi.org/10.5281/zenodo.4838116`
35. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A Verification Infrastructure for Permission-Based Reasoning. In: VMCAI (2016)
36. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A Verification Infrastructure for Permission-Based Reasoning. In: Dependable Software Systems Engineering (2017)
37. Nguyen, H.H., Kuncak, V., Chin, W.N.: Runtime checking for separation logic. In: VMCAI. pp. 203–217 (2008)
38. Philippaerts, P., Mülberg, J.T., Penninckx, W., Smans, J., Jacobs, B., Piessens, F.: Software Verification with VeriFast: Industrial Case Studies. Science of Computer Programming **82**, 77–97 (2014)
39. Raad, A., Berdine, J., Dang, H., Dreyer, D., O'Hearn, P.W., Villard, J.: Local Reasoning About the Presence of Bugs: Incorrectness Separation Logic. In: CAV (2020)
40. Reynolds, J.C.: Separation Logic: A Logic for Shared Mutable Data Structures. In: LICS (2002)
41. S. Panić: Collections-C: A Library of Generic Data Structures. `https://github.com/srdja/Collections-C` (2014)
42. Sampaio, G., Santos, J.F., Maksimović, P., Gardner, P.: A Trusted Infrastructure for Symbolic Analysis of Event-Driven Web Applications. In: ECOOP (2020)
43. Santos, M.: Buckets-js: A javascript data structure library. `https://github.com/mauriciosantos/Buckets-JS` (2016)
44. Swamy, N., Weinberger, J., Schlesinger, C., Chen, J., Livshits, B.: Verifying Higher-order Programs with the Dijkstra Monad. In: PLDI (2013)
45. The DeepSpec Team: The DeepSpec Project. `https://deepspec.org/main` (2021)
46. The Gillian Team: Gillian. `https://gillianplatform.github.io` (2020)
47. The Iris Team: The Iris Project. `https://iris-project.org` (2021)
48. Watt, C., Maksimović, P., Krishnaswami, N.R., Gardner, P.: A Program Logic for First-Order Encapsulated WebAssembly. In: ECOOP (2019)

Debugging Network Reachability with Blocked Paths

S. Bayless(✉), J. Backes, D. DaCosta, B. F. Jones, N. Launchbury, P. Trentin, K. Jewell, S. Joshi, M. Q. Zeng, and N. Mathews

Amazon Web Services, Seattle, USA
sabayles@amazon.com

Abstract. In this industrial case study we describe a new network troubleshooting analysis used by VPC Reachability Analyzer, an SMT-based network reachability analysis and debugging tool. Our troubleshooting analysis uses a formal model of AWS Virtual Private Cloud (VPC) semantics to identify whether a destination is reachable from a source in a given VPC configuration. In the case where there is no feasible path, our analysis derives a *blocked path*: an infeasible but otherwise complete path that would be feasible if a corresponding set of VPC configuration settings were adjusted.

Our blocked path analysis differs from other academic and commercial offerings that either rely on packet probing (e.g., TCPTRACE) or provide only partial paths terminating at the first component that rejects the packet. By providing a complete (but infeasible) path from the source to destination, we identify for a user all the configuration settings they will need to alter to admit that path (instead of requiring them to repeatedly re-run the analysis after making partial changes). This allows users to refine their query so that the blocked path is aligned with their intended network behavior before making any changes to their VPC configuration.

1 Introduction

This paper describes a new network connectivity troubleshooting analysis used by VPC Reachability Analyzer, a service that analyzes Amazon Web Services' (AWS) Virtual Private Cloud (VPC) configurations.

VPCs are user-configured networks of virtual compute devices and resources. AWS VPC offers dozens of networking components and controls to give users flexibility in configuring their networks. Access to these resources is logically isolated within virtual networks configured by the users. As VPCs grow in size and complexity, users can increasingly benefit from automation to identify and resolve misconfigurations, as well as to validate that applications maintain security and availability invariants through infrastructure changes.

VPC Reachability Analyzer uses the Tiros [2] formal model of AWS VPC networking semantics to identify whether a destination is reachable from a source in a given VPC configuration. If the destination is reachable, then Tiros identifies a *feasible path* from the source to the destination, where a path is a sequence of network components associated with incoming and/or outgoing packet header assignments (protocol, addresses, ports). The outgoing packet header of one component is the incoming packet header of the next component. Paths may also identify relevant VPC configuration details such as the specific routes, firewall rules, or other settings admitting the packet at each step. Each component in a VPC may accept or reject incoming and outgoing packet headers; a *feasible path* is a path in which every component on the path accepts both its incoming and outgoing packet header.

Tiros's analysis is static, *i.e.*, Tiros does not inject traffic into VPC configurations, and is complete for the subset of AWS VPC semantics it supports: if there exists a path connecting the source and destination, Tiros will find it. Since 2018, Tiros has powered the commercially available *Network Reachability* assessment in Amazon Inspector [1], statically identifying ports on EC2 Instances (virtual machines) accessible outside of their VPCs.

In this work, we extend Tiros by introducing a new diagnostic *blocked path* analysis when there is not a feasible path, to help users understand why their query is infeasible. A *blocked path* is a path as defined above, in which at least one component rejects its incoming or outgoing packet, along with one or more *blocking reasons*: elements of the VPC configuration preventing one or more components on the path from accepting packets. The blocked path identifies a sufficient set of blocking reasons, such that if each were addressed the query would be satisfiable.

Previous tools for connectivity diagnosis typically provide a partial path, up to the first component/rule that rejects the packet; in some cases those tools also identify a single blocking reason. Remediations based on a partial path may address that initial blocking reason only to discover that remediations are still necessary, or that the remediation may be working towards a path that the user ultimately will reject. Providing a complete blocked path connecting the source and destination allows users to ensure that their intent is aligned with our diagnosis before taking any corrective actions.

Our contributions in this work are:

1. Identifying the notion of a blocked path as a useful medium for conveying a network diagnosis and aligning it with a user's intent,
2. Demonstrating how blocked paths can be efficiently derived at scale,
3. Describing VPC Reachability Analyzer, a commercial tool based on these insights.

2 Background

2.1 Related Works

Many previous works have proposed network reachability diagnosis tools, including both widely-used industry tools and academic literature. These tools can be broadly divided into model-based and non-model-based approaches.

Non-model-based network diagnostic tools include system applications such as IPTRACE and TCPTRACE, commercial tools such as *Cisco Packet Tracer* [7], and academic works such as *Tulip* [12]. These tools trace live packets through a network or routing device, identifying the sequence of addresses of devices that accept the packet. Packet tracing tools lack visibility into the configuration settings that block and route packets.

Model-based tools [2,5,6,13,16] statically analyze reachability between a specified source and destination in a network or routing device. Rather than transmitting live packets, these tools use formal methods such as constraint solvers to rigorously identify feasible paths. Existing model-based tools provide control-plane level information when there is a feasible path, but produce either no information for unreachable paths, or identify only the first (out of potentially many) reasons why a path is blocked.

Our blocked path analysis is based on deriving minimal correction subsets (described below), which several previous works have proposed for general-purpose SAT-based error diagnosis or repair [4,8,9,17].

2.2 Minimal Correction Subsets

The blocked path analysis we describe in Sect. 3 relies on two related concepts: Maximal Satisfiable Subsets (MSS) and Minimal Correction Subsets (MCS), which we define below. Following the definitions from [14]:

Definition 1 (MSS). *$\mathcal{S} \subseteq \mathcal{F}$ is a Maximal Satisfiable Subset of constraints $\mathcal{F}$ iff $\mathcal{S}$ is satisfiable and $\forall c \in \mathcal{F} \setminus \mathcal{S}, \mathcal{S} \cup \{c\}$ is unsatisfiable.*

Definition 2 (MCS). *$\mathcal{C} \subseteq \mathcal{F}$ is a Minimal Correction Subset of constraints $\mathcal{F}$ iff $\mathcal{F} \setminus \mathcal{C}$ is satisfiable and $\forall c \in \mathcal{C}, (\mathcal{F} \setminus \mathcal{C}) \cup \{c\}$ is unsatisfiable.*

The complement of an MCS, $\mathcal{F} \setminus MCS(\mathcal{F})$, is guaranteed to be a maximal satisfiable subset of $\mathcal{F}$; for this reason the MCS is sometimes called the coMSS.[1]

In general, the MCS and MSS are not guaranteed to be unique. There is a close connection between the definition of a Maximal Satisfiable Subset and MaxSAT [10]: The largest MSS (and therefore smallest MCS) corresponds to a solution to MaxSAT. Indeed, one approach for computing the MCS is to compute MaxSAT and take the complement. Efficient algorithms for directly computing the (not necessarily smallest) MCS without computing MaxSAT are available and are typically much faster than computing MaxSAT; a good survey of MCS algorithms including an empirical evaluation can be found in [14].

In constraint optimization problems, it is common to consider hard and soft constraints, in which only the soft constraints may be relaxed. Definition 2 assumes that all constraints are soft, but can be easily extended to support

[1] Note that a minimal correction subset is a distinct concept from an unsatisfiable core [11]. An unsatisfiable core is always unsatisfiable, but its complement $\mathcal{F} \setminus CORE(\mathcal{F})$ is not guaranteed to be satisfiable; in contrast, an MCS may or may not be satisfiable, but its complement is guaranteed to be satisfiable.

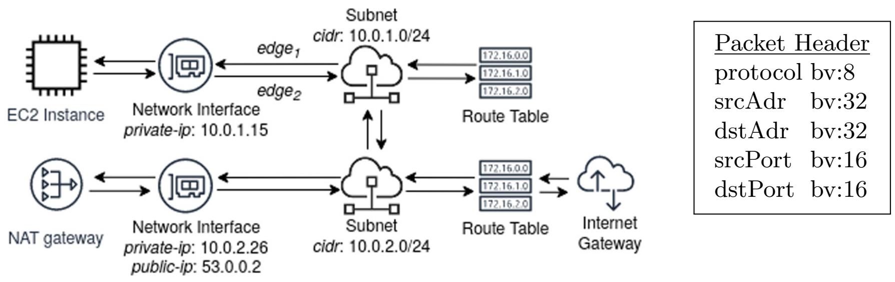
a mix of soft and hard constraints (where the MCS must contain only soft constraints). In this case, the MCS is only well defined if the hard constraints are satisfiable.

In Sect. 4, we will use a function COMPUTEMCS($Soft$, $Hard$) that supports both hard and soft constraints. COMPUTEMCS returns a minimal correction set $\mathcal{C} = MCS(Soft \cup Hard)$, with $\mathcal{C} \subseteq Soft$. Our implementation of COMPUTEMCS uses a simple binary search, similar to FastDiag [4], or Algorithm BFD from [14]. We add activation literals to the soft constraints to allow the underlying solver instance to be re-used incrementally while testing different subsets of soft constraints for satisfiability.

2.3 Network Reachability

We use the SMT-encoding of AWS VPC network semantics previously described in TIROS [2]. In this section, we briefly review this graph-based encoding; we refer readers to [2] for more details.

We take as input a configuration describing one or more user VPCs, and a user-specified reachability query, consisting of a source and destination component in the VPC. For example, the source of the query may be an internet gateway, and the destination may be an EC2 Instance. A query may also optionally specify additional constraints, such as the protocol, a range of source or destination addresses or ports for the packet, or an intermediate component that must (or must not) be on the path.



Packet Header	
protocol	bv:8
srcAdr	bv:32
dstAdr	bv:32
srcPort	bv:16
dstPort	bv:16

$$((dstAdr \neq 10.0.1.15) \implies \neg edge_1)$$
$$((srcAdr \neq 10.0.1.15) \implies \neg edge_2)$$

Fig. 1. Simplified example symbolic graph representation of a VPC (*left*), with symbolic packet header consisting of bitvectors (*right*). Edges in the graph are associated with theory atoms, and are traversable only if those atoms are assigned true. Two example constraints, enforcing that a network interface is only accessible if the packet is addressed to/from that interface are shown. These constraints relate edge atoms in the symbolic graph to the bitvectors in the symbolic packet header to enforce AWS VPC semantics.

We encode VPC configurations as constrained symbolic graphs using the SMT solver MonoSAT [3], with fixed-width bitvectors representing the protocol, port, and addressing information in a symbolic packet header. Figure 1 shows a symbolic graph along with a packet header and example constraints.

VPC components are represented as a nodes in the symbolic graph. Each component has semantics governing which packets it will accept; these semantics are encoded as constraints that restrict which edges incident to that component's node are traversible, depending on the assignment of the packet header variables. A satisfying assignment to the full set of constraints corresponds to a feasible path. In such an assignment, the bitvector variable assignments provide the packet header(s) and the graph theory model provides a path of network component nodes connecting the source and destination of the user's query.

Some components (such as NAT gateways) transform and retransmit packets. Tiros supports this by unrolling the VPC configuration graph into multiple copies with separate packet header variables. Edges from packet-transforming components connect to their components in the next unrolled section of the graph. Tiros unrolls the graph to a sufficient depth to model the behavior of the components for each query.

Query source and destination reachability is enforced with a single graph theory reachability predicate requiring a feasible path in the VPC configuration graph from the source to the destination of the query. Query restrictions requiring intermediate components are enforced using additional reachability predicates. Query restrictions requiring that a given resource not occur on a path are enforced by excluding that resource from the VPC configuration graph representation. Packet header restrictions are enforced using bitvector constraints.

If the constraints are satisfiable, Tiros extracts a reachable path satisfying the query from the satisfying assignment to the constraints. In the next section, we will discuss how we extend Tiros to also provide diagnostic feedback in the case where the constraints are unsatisfiable.

3 Blocked Paths for Network Configuration Diagnosis

We introduce the notion of *blocked path* for analyzing infeasible network connections. As shown in Fig. 2, a blocked path is an infeasible but otherwise complete path from a source to a destination, in which one or more edges or nodes are annotated with *blocking reasons*: configuration settings or network semantics that explain why that transition in the path is infeasible.

Unlike a live packet trace, a blocked path continues past components that reject or redirect the packet so as to reach the user's intended destination, potentially transiting through multiple infeasible steps along the way.

Definition 3 (Blocked path).

1. *A blocked path is a complete (but infeasible) path from a source to a destination in a network, satisfying the user's query.*

2. *A blocked path is* actionable*: it is a path that could, with the right control plane configuration adjustments, be a feasible path.*
3. *A blocked path identifies a sufficient set of* blocking reasons *(network semantics or control-plane settings) that would need to be addressed to admit the packet along that blocked path. This may include multiple blocking reasons along the path, as opposed to just the first blocking reason.*

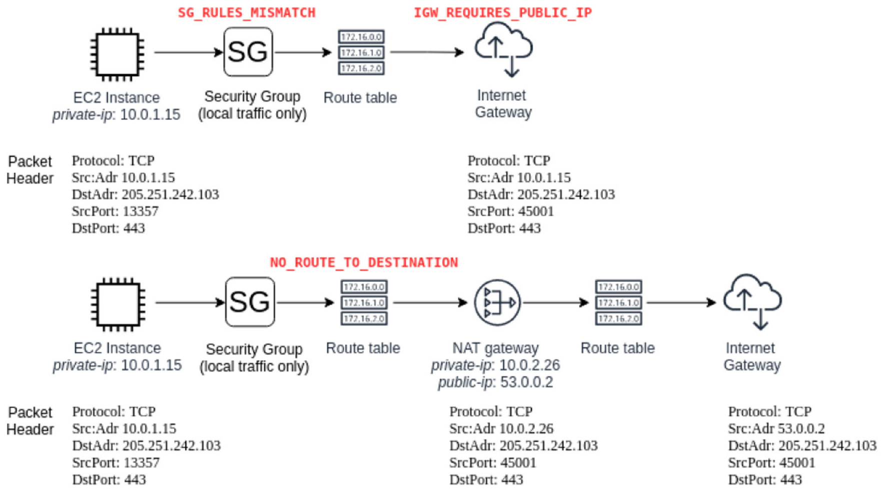


Fig. 2. Two alternative blocked paths from an EC2 instance to an internet gateway. These blocked paths take different routes, and have different *blocking reasons* (shown in red) that explain why those paths are infeasible. In the first blocked path, there are two blocking reasons: the security group egress rule rejects packets destined for the Internet, and the internet gateway requires that the source instance must have a public IP address. Note that although the packet would be rejected by the security group, the blocked path continues past the security group to identify a complete (but infeasible) path to the internet gateway. The second blocked path transitions through an intermediate NAT gateway, which satisfies the security group rule and also has a public IP address. However, this path is still blocked, because the route table does not have an applicable route to the NAT gateway.

Validating User Intent

Showing a complete path from the source to destination, along with all the relevant configuration settings blocking that path, allows users to confirm that this course of action matches their intended network behavior before making any changes. However, in many cases there are multiple ways to adjust a configuration to admit a path, resulting in different blocked paths.

For example, Fig. 2 shows two example blocked paths to an internet gateway from an EC2 instance lacking a public IP address. Our analysis might initially produce for the user the shorter blocked path. Two remediation steps are required to admit this shorter path: The user must adjust the security group rule of the instance to admit egress packets to the public internet, and the user must also associate a public IP address with the source instance. Upon seeing the complete blocked path, the user may immediately determine that this would be the wrong solution for their network.

If the proposed blocked path doesn't match the user's intent, we allow users to submit a refined query so as to generate an alternative blocked path. For instance, the user may specify allowed address or port ranges for the packet, or specify components that must or must not appear on the path. Similarly, the user may submit a refined query specifying that a NAT gateway must be an intermediate component on the path. In this case, we might produce the longer blocked path from Fig. 2.

Actionable Blocked Paths

In some cases, there may not exist any combination of VPC configuration adjustments that would allow a query to be satisfied. For example, under typical conditions in VPCs, route tables cannot be adjusted to redirect packets that are destined for a local address within the VPC. It is possible for users to specify queries that cannot be satisfied without violating this local route restriction.

In principle, it is possible to derive a blocked path with non-user-configurable blocking reasons, however the resulting paths may behave in misleading or confusing ways, and in general will not be possible for users to actually achieve in any real configuration of their VPC. If possible, we want to ensure that the path contains only user-configurable blocking reasons, so that we produce an *actionable* finding for users. However, we still want to be able to provide useful diagnostics in cases where no actionable blocked path is possible (*e.g.*, to explain to the user that the local route restriction will prevent their path).

In Sect. 4, we describe how we determine when it is not possible to produce a blocked path without including non-configurable blocking reasons. In this case, we produce a partial path up to that first non-configurable blocking reason.

Additionally, in some cases a user may specify a query that remains unsatisfiable even if all of the network semantics in our model are relaxed. This can occur if the user specifies components that do not exist, or that are in isolated, disconnected networks (for which no relaxation of the edge constraints will admit a path). In this case, our blocked path analysis fails, and TIROS falls back on other techniques to produce diagnostic information.

In Sect. 5 we show that in most cases, our analysis succeeds and produces an actionable blocked path.

4 Deriving Blocked Paths from Unsatisfiable Queries

We group VPC configuration semantics into three disjoint sets of constraints: $(U \cup N \cup H)$. Set U contains constraints that enforce user-configurable control-plane settings (such as a user-defined route or firewall rule), while set N contains non-configurable for user-visible network semantics (such as the local route restriction).

Set H contains elements of the constraints that are either not user-visible (such as internal implementation details) or that should never be relaxed (such as the reachability predicate or any other constraints defined by the user's query). For example, many of our constraints involve containment comparisons between CIDRs and bitvectors representing IP addresses. An individual CIDR comparison is encoded as a fresh literal represnting the truth value of the comparison, along with multiple clauses that enforce the comparison semantics. The intermediate clauses that enforce the comparison semantics are implementation details that we include in set H, ensuring they are not included in the blocking reasons.

When a query is unsatisfiable, we derive a blocked path and corresponding blocking reasons from a Maximal Satisfiable Subset and Minimal Correction Subset of $(U \cup N \cup H)$, with set H being treated as hard constraints that must not be included in the MCS.

If possible, we want to produce an MCS containing only configurable blocking reasons from U. This ensures that the resulting blocked path is actionable. If we directly compute the MCS of the full constraint set $U \cup N \cup H$, with both U and N as soft constraints, non-configurable constraints from N may be included in the MCS even in cases where there exists an MCS containing only constraints from U. On the other hand, we still want to be able to produce an MCS in the case where the non-configurable and hard constraints $(N \cup H)$ are, by themselves, unsatisfiable.

In Algorithm 1, we resolve this by breaking the computation of the MCS into two steps, initially computing an MCS of $N \cup H$, and only allowing constraints from N into the blocking reasons if MCS$(N \cup H)$ is non-empty.

When $N \cup H$ is satisfiable, Algorithm 1 produces a blocked path that only contains the configurable blocking reasons from U.

Algorithm 1 constructs two correction sets, $MCS_N \subseteq N$ and $MCS_U \subseteq U$, with $MCS_N \cup MCS_U$ a valid MCS of $(U \cup N \cup H)$. We then extract a path p from a satisfying assignment to the corresponding MSS $(U \cup N \cup H) \setminus (MCS_N \cup MCS_U)$. Finally, as shown below, we return either a complete or a partial blocked path, by associating blocking reasons from the MCS with nodes on that path.

Algorithm 1 relies on two helper methods, ExtractPath and BuildPath. ExtractPath retrieves the satisfying theory model (a sequence of edges) for the query reachability predicate from a satifiable formula, using the graph theory in the SMT-solver MonoSAT, and associates packet header assignments with each step of that path from the corresponding bitvector assignments. BuildPath maps the literals of the MCS to descriptive strings representing blocking reasons, and associates those strings with steps on the blocked path.

Algorithm 1. Blocked Path Analysis

```
 1: function DeriveBlockedPath($U, N, H$)   ▷ Precondition: $U \cup N \cup H$ is UNSAT.
 2:     if UNSAT($H$) then
 3:         throw Error: No blocked path can be produced.
 4:     else
 5:         // Note: If $N \cup H$ is SAT, then $MCS_N = \emptyset$.
 6:         $MCS_N \leftarrow$ computeMCS($N, H$)
 7:         // Note: $(N \cup H) \setminus MCS_N$ is SAT; $MCS_U$ is well-defined.
 8:         $MCS_U \leftarrow$ computeMCS($U, (N \cup H) \setminus MCS_N$)
 9:         $p \leftarrow$ ExtractPath($(U \cup N \cup H) \setminus (MCS_N \cup MCS_U)$)
10:         return BuildPath($p, MCS_N, MCS_U$)
11:     end if
12: end function
```

We can see that $MCS_N \cup MCS_U$ meets the definition of a minimal correction set of $U \cup N \cup H$ by observing that:

$$\begin{aligned} \mathrm{SAT}((U \cup N \cup H \setminus (MCS_N)) \setminus (MCS_N \cup MCS_U)) &\quad \text{line 8} \\ \implies \mathrm{SAT}((U \cup N \cup H) \setminus (MCS_N \cup MCS_U))) & \\ \forall c \in MCS_N, \mathrm{UNSAT}((N \cup H) \setminus (MCS_N \setminus \{c\}) &\quad \text{line 6} \\ \forall c \in MCS_U, \mathrm{UNSAT}((U \cup N \cup H) \setminus (MCS_U \setminus \{c\})) &\quad \text{line 8} \\ \implies \forall c \in (MCS_N \cup MCS_U), \mathrm{UNSAT}((U \cup N \cup H) \setminus ((MCS_N \cup MCS_U) \setminus \{c\})) & \end{aligned}$$

If $N \cup H$ is satisfiable, then MCS_N is empty and MCS_U, containing only configurable constraints, is an MCS of $(U \cup N \cup H)$. In this case, BuildPath constructs a complete blocked path consisting entirely of configurable blocking reasons.

If $N \cup H$ is unsatisfiable, then MCS_N is non-empty and $MCS_N \cup MCS_U$ contains at least one non-actionable constraints. In this case, the path p may behave unexpectedly and may not be realizable in a VPC configuration after adjustment. If MCS_N is non-empty, BuildPath forms the blocked path as above, but returns only the prefix of that blocked path up to and including the first edge or node associated with a non-actionable setting.

Above, we discussed the cases where $N \cup H$ is satisfiable or unsatisfiable. There is also a third possibility: The hard constraints H, representing the constraints enforcing the user's query or implementation details of our model, may by themselves be unsatisfiable. For example, H may be unsatisfiable if the user specifies a source and destination that are in separate, disconnected networks.

If H is unsatisfiable, Algorithm 1 fails, and is unable to produce even a partial blocked path. In this case, we fall back on other techniques to provide useful diagnostic information for users. In practice, the typical reason that H is unsatisfiable is that the source and destination are in disconnected VPCs (so the reachability constraint is unsatisfiable). We use a static analysis pass to identify this case and handle it separately in our service.

In the case that Algorithm 1 produces a complete (*resp.* partial) blocked path, the underlying MCS algorithm guarantees that the blocked path will have the fewest possible number of blocking reasons from among all complete (*resp.* partial) blocked paths. In general this blocked path is not unique.

In our implementation of Algorithm 1, the graph-based decision heuristic in MonoSAT will prioritize finding shortest-length paths in most cases, but does not guarantee that a shortest-length path is always found.

5 Evaluation

VPC Reachability Analyzer, a commercial offering available from AWS since December 2020, uses the blocked path analysis we have described to derive findings for queries between unreachable endpoints.

To demonstrate the practical impact of this blocked path analysis, we randomly selected 1000 unreachable queries processed by VPC Reachability Analyzer. We executed the blocked path analysis for those queries on an 'm5.24xlarge' EC2 instance using GNU Parallel [15], running Amazon Linux 2, using MonoSAT version 1.6.0.

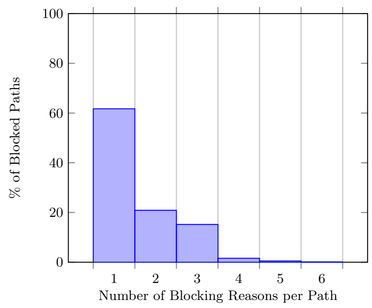


Fig. 3. Number of blocking reasons per blocked path (among the 63% of unreachable queries for which the blocked path analysis produced a complete blocked path). 97% percent of blocked paths have three or fewer blocking reasons; 60% have just a single blocking reason.

Excluding the time to complete the blocked path analysis, the average time required to initially determine satisfiability of the constraints was 2.1 s (P50: 1.7 s, P99: 7.4 s). The blocked path analysis was as fast or faster than the initial solving time, requiring 0.3 s on average (P50: 0.05 s, P99: 6.6 s).

As described in Sect. 4, in some cases, the blocked path analysis can produce only a partial path, or no results at all. Of those 1000 unreachable queries, 63.2% resulted in complete blocked paths, 7.4% resulted in partial blocked paths, and the remainder (29.4%) produced no analysis (in which case VPC Reachability Analyzer applies other techniques so that it can still provide useful diagnostics).[2]

As can be seen in Fig. 3, most blocked paths have just one blocking reason, and 97% have at most three. This demonstrates that our analysis produces actionable, concise findings on real production data, a key requirement of a useful diagnosis service.

6 Conclusion

The blocked path analysis we have introduced provides key advantages over previous network diagnostic techniques. By showing users a blocked path from a source to a destination, we allows users the opportunity to refine their query such that their intended path is aligned with our analysis. Furthermore, showing all blocking reasons on a blocked path allows users to understand the VPC configuration adjustments necessary to realize a path for their query.

Our blocked path analysis is a fully static analysis (requiring no packets to be injected into the network), can be computed efficiently using standard techniques from the formal methods literature, and is now used successfully in production by VPC Reachability Analyzer.

References

1. Amazon Inspector. https://docs.aws.amazon.com/inspector/. Accessed December 2018
2. Backes, J., et al.: Reachability analysis for AWS-based networks. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11562, pp. 231–241. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25543-5_14
3. Bayless, S., Bayless, N., Hoos, H.H., Hu, A.J.: SAT modulo monotonic theories. In: Proceedings of AAAI, pp. 3702–3709 (2015)
4. Felfernig, A., Schubert, M., Zehentner, C.: An efficient diagnosis algorithm for inconsistent constraint sets. Artif. Intell. Eng. Design Anal. Manuf. AI EDAM **26**(1), 53 (2012)
5. Fogel, A., et al.: A general approach to network configuration analysis. In: Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation. pp. 469–483. NSDI 2015, USENIX Association, Berkeley, CA, USA (2015). http://dl.acm.org/citation.cfm?id=2789770.2789803
6. Jayaraman, K., Bjørner, N., Outhred, G., Kaufman, C.: Automated analysis and debugging of network connectivity policies. Microsoft Research, pp. 1–11 (2014)
7. Jazib Frahim, Omar Santos, A.O.: Cisco ASA All-in-One Firewall, IPS, and VPN Adaptive Security Appliance, 3rd edition. Cisco Press (2014)

[2] Of the queries for which no blocked path analysis was performed, 80% were due to users specifying endpoints in disconnected VPCs. We perform a disconnected component analysis to identify this case. Others were due to users specifying resources they lack access to, or that we do not support.

8. Junker, U.: Preferred explanations and relaxations for over-constrained problems. In: AAAI-2004 (2004)
9. Koitz, R., Wotawa, F.: Sat-based abductive diagnosis. In: DX@ Safeprocess, pp. 167–176 (2015)
10. Li, C.M., Manya, F.: Maxsat, hard and soft constraints. Handb. Satisf. **185**, 613–631 (2009)
11. Lynce, I., Marques-Silva, J.P.: On computing minimum unsatisfiable cores (2004)
12. Mahajan, R., Spring, N., Wetherall, D., Anderson, T.: User-level internet path diagnosis. ACM SIGOPS Oper. Syst. Rev. **37**(5), 106–119 (2003)
13. Mai, H., Khurshid, A., Agarwal, R., Caesar, M., Godfrey, B., King, S.T.: Debugging the data plane with anteater. In: Proceedings of the ACM SIGCOMM 2011 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Toronto, ON, Canada, 15–19 August 2011, pp. 290–301 (2011). https://doi.org/10.1145/2018436.2018470, http://doi.acm.org/10.1145/2018436.2018470
14. Marques-Silva, J., Heras, F., Janota, M., Previti, A., Belov, A.: On computing minimal correction subsets. In: Twenty-Third International Joint Conference on Artificial Intelligence. Citeseer (2013)
15. Tange, O.: GNU Parallel 2018. Ole Tange, March 2018. https://doi.org/10.5281/zenodo.1146014
16. Tian, B., et al.: Safely and automatically updating in-network acl configurations with intent language. In: Proceedings of the ACM Special Interest Group on Data Communication, pp. 214–226 (2019)
17. Walter, R., Felfernig, A., Küchlin, W.: Constraint-based and sat-based diagnosis of automotive configuration problems. J. Intell. Inf. Syst. **49**(1), 87–118 (2017)

Lower-Bound Synthesis Using Loop Specialization and Max-SMT

Elvira Albert[1,2], Samir Genaim[1,2], Enrique Martin-Martin[1], Alicia Merayo[1](✉), and Albert Rubio[1,2]

[1] Fac. Informática, Complutense University of Madrid, Madrid, Spain
amerayo@ucm.es

[2] Instituto de Tecnología del Conocimiento, Madrid, Spain

Abstract. This paper presents a new framework to synthesize lower-bounds on the worst-case cost for non-deterministic integer loops. As in previous approaches, the analysis searches for a *metering function* that under-approximates the number of loop iterations. The key novelty of our framework is the *specialization* of loops, which is achieved by restricting their enabled transitions to a subset of the inputs combined with the narrowing of their transition scopes. Specialization allows us to find metering functions for complex loops that could not be handled before or be more precise than previous approaches. Technically, it is performed (1) by using quasi-invariants while searching for the metering function, (2) by strengthening the loop guards, and (3) by narrowing the space of non-deterministic choices. We also propose a Max-SMT encoding that takes advantage of the use of soft constraints to force the solver look for more accurate solutions. We show our accuracy gains on benchmarks extracted from the 2020 Termination and Complexity Competition by comparing our results to those obtained by the LoAT system.

1 Introduction

One of the most important problems in program analysis is to automatically –and accurately– bound the cost of program's executions. The first automated analysis was developed in the 70s [24] for a strict functional language and, since then, a plethora of techniques has been introduced to handle the peculiarities of the different programming languages (see, e.g., for Integer programs [5], for Java-like languages [2,19], for concurrent and distributed languages [16], for probabilistic programs [15,18], etc.) and to increase their accuracy (see, e.g., [10,14,21,22]). The vast majority of these techniques have focused on inferring *upper bounds* on the worst-case cost, since having the assurance that none execution of the program will exceed the inferred amount of resources (e.g., time, memory, etc.) has crucial applications in safety-critical contexts. On the other hand, *lower bounds*

This work was funded partially by the Spanish MCIU, AEI and FEDER (EU) project RTI2018-094403-B-C31, by the CM project S2018/TCS-4314 co-funded by EIE Funds of the European Union, and by the UCM CT42/18-CT43/18 grant.

on the best-case cost characterize the minimal cost of any program execution and are useful in task parallelization (see, e.g., [3,9,10]). There are a third type of important bounds which are the focus of this work: *lower bounds on the worst-case cost*, they bound the worst-case cost from below. Their main application is that, together with the upper bounds on worst-case, allow us to infer tighter worst-case cost bounds (when they coincide ensuring that the inferred cost is exact) what can be crucial in safety-critical contexts. Besides, lower bounds on the worst-case cost will give us families of inputs that lead to an expensive cost, what could be used to detect performance bugs. In what follows, we use the acronyms LB^w and LB^b to refer to *w*orst-case and *b*est-case lower-bounds, resp.

State-of-the-Art in LB^w. An important difference between LB^w and LB^b is that, while the best-case must consider *all* program runs, LB^w holds for (usually infinite) families of the most expensive program executions. This is why the techniques applicable to LB^b inference (e.g., [3,9,10]) are not useful for LB^w in general, since they would provide too inaccurate (low) results. The state-of-the-art in LB^w inference is [12,13] (implemented in the LoAT system) which introduces a variation of ranking functions, called *metering functions*, to under-estimate the number of iterations of *simple* loops, i.e., loops without branching nor nested loops. The core of this method is a simplification technique that allows treating general loops (with branchings and nested loops) by using the so-called *acceleration*: that replaces a transition representing one loop iteration by another rule that collects the effect of applying several consecutive loop iterations using the original rule. Asymptotic lower bounds are then deduced from the resulting simplified programs using a special-purpose calculus and an SMT encoding.

Motivation. Our work is motivated by the limitation of state-of-the-art methods when, by treating each simple loop separately, a LB^w bound cannot be found or it is too imprecise. For example, consider the interleaved loop in Fig. 1, that is a simplification of the benchmark SimpleMultiple.koat from the Termination and Complexity competition. Its *transition system* appears to the right (the transition system is like a control-flow graph (CFG) in which the transitions τ are labeled with the applicability conditions and with the updates for the variables, primed variables denote the updated values). In every iteration x or y can decrease by one, and these behaviors can interleave. The worst case is obtained for instance when x is decreased to 0 (x iterations) and then y is decreased to 0 (y iterations), resulting in $x + y$ iterations, or when y is first decreased to 1 and then x to -1, etc. The approach in [12,13] accelerates independently both τ_1 and τ_4, resulting in accelerated versions $\tau_1^a = x \geq -1 \wedge y > 0 \wedge x' = -1 \wedge y' = y$ with cost $x + 1$ and $\tau_4^a = x \geq 0 \wedge y \geq 0 \wedge x' = x \wedge y' = 0$ with cost y. Applying one accelerated version results in that the other accelerated version cannot be applied because of the final values of the variables. Thus, the overall knowledge extracted from the loop is that it can iterate $x+1$ or y times, whereas the precise LB^w is $x+y$ iterations. Our challenge for inferring more precise LB^w is to devise a method that can handle all loop transitions simultaneously, as disconnecting them leads to a semantics loss that cannot be recovered by acceleration.

```
while (x >= 0 && y > 0) {
  if (*) {
    x = x - 1;
  } else {
    y = y - 1;
  }
}
```

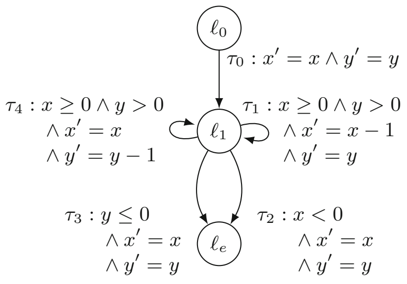

ℓ_0
$\tau_0 : x' = x \wedge y' = y$
$\tau_4 : x \geq 0 \wedge y > 0 \wedge x' = x \wedge y' = y - 1$
ℓ_1
$\tau_1 : x \geq 0 \wedge y > 0 \wedge x' = x - 1 \wedge y' = y$
$\tau_3 : y \leq 0 \wedge x' = x \wedge y' = y$
ℓ_e
$\tau_2 : x < 0 \wedge x' = x \wedge y' = y$

Fig. 1. Interleaved loop (left) and its representation as a transition system (right)

Non-Termination and LB^w. Our work is inspired by [17], which introduces the powerful concept of *quasi-invariant* to find witnesses for non-termination. A quasi-invariant is an invariant which does not necessarily hold on initialization, and can be found as in template-based verification [23]. Intuitively, when there is a loop in the program that can be mapped to a quasi-invariant that forbids executing any of the outgoing transitions of the loop, then the program is non-terminating. This paper leverages such powerful use of quasi-invariants and Max-SMT in non-termination analysis to the more difficult problem of LB^w inference. Non-termination and LB^w are indeed related properties: in both cases we need to find witnesses, resp., for non-terminating the loop and for executing at least a certain number of iterations. For LB^w, we additionally need to provide such under-estimation for the number of iterations and search for LB^w behaviors that occur for a class of inputs rather than for a single input instantiation (since the LB^w for a single input is a concrete (i.e., constant) cost, rather than a parametric LB^w function as we are searching for). Instead, for non-termination, it is enough to find a non-terminating input instantiation.

Our Approach. A fundamental idea of our approach is to *specialize* loops in order to guide the search of the metering functions of complex loops, avoiding the inaccuracy introduced by disconnecting them into simple loops. To this purpose, we propose specializing loops by combining the addition of constraints to their transitions with the restriction of the valid states by means of quasi-invariants. For instance, for the loop in Fig. 1, our approach automatically narrows τ_1 by adding $x > 0$ (so that x is decreased until $x = 0$) and τ_4 by adding $x \leq 0$ (so that τ_4 can only be applied when $x = 0$). This specialized loop has lost many of the possible interleavings of the original loop but keeps the worst case execution of $x+y$ iterations. These specialized guards do not guarantee that the loop executes $x + y$ iterations in every possible state, as the loop will finish immediately for $x < 0$ or $y \leq 0$, thus our approach also infers the quasi-invariant $x \geq 0 \wedge x \leq y$. Combining the specialized guards and the quasi-invariant, we can assure that when reaching the loop in a valid state according to the quasi-invariant, $x + y$ is a lower bound on the number of iterations of the loop, i.e., its cost. Using quasi-invariants that include all (invariant) inequalities syntactically appearing in loop transitions might work for the case of loops with single path. However, for the general case, the specialized guards usually lead to essential quasi-invariants that do not appear in the original loop. The specialization achieved by adding constraints could be also applied in the context of non-termination to increase the accuracy of [17], as only quasi-invariants were used. Therefore, we argue that our work avoids the precision loss caused by the simplification in [12,13] and, besides, introduces a loop specialization technique that can also be applied to gain precision in non-termination analysis [17].

Contributions. Briefly, our main theoretical and practical contributions are:

1. In Sect. 3 we introduce several semantic specializations of loops that enable the inference of *local* metering functions for complex loops by: (1) restricting the input space by means of automatically generated quasi-invariants, (2) narrowing transition guards and (3) narrowing non-deterministic choices.
2. We propose a template-based method in Sect. 4 to automate our technique which is effectively implemented by means of a Max-SMT encoding. Whereas the use of templates is not new [6], our encoding has several novel aspects that are devised to produce better lower-bounds, e.g., the addition of (soft) constraints that force the solver look for larger lower-bound functions.
3. We implement our approach in the LOBER system and evaluate it on benchmarks from the Integer Transition Systems category of the 2020 Termination and Complexity Competition (see Sect. 5). Our experimental results when compared to the existing system LoAT [12] are promising: they show further accuracy of LOBER in challenging examples that contain complex loops.

2 Background

This section introduces some notation on the program representation and recalls the notion of LB^w we aim at inferring.

2.1 Program Representation

Our technique is applicable to sequential non-deterministic programs with integer variables and commands whose updates can be expressed in linear (integer) arithmetic. We assume that the non-determinism originates from non-deterministic assignments of the form "x:=nondet();", where x is a program variable and nondet() can be represented by a fresh non-deterministic variable u. This assumption allows us to also cover non-deterministic branching, e.g., "if (*){..} else {..}" as it can be expressed by introducing a non-deterministic variable u and rewriting the code as "u:=nondet(); if (u≥0){..} else {..}".

Our programs are represented using *transition systems*, in particular using the formalization of [17] that simplifies the presentation of some formal aspects of our work. A transition system (abbrev. TS) is a tuple $\mathcal{S} = \langle \bar{x}, \bar{u}, \mathcal{L}, \mathcal{T}, \Theta \rangle$, where $\bar{x}$ is a tuple of n integer program variables, $\bar{u}$ is a tuple of integer (non-deterministic) variables, $\mathcal{L}$ is a set of locations, $\mathcal{T}$ is a set of transitions, and Θ is a formula that defines the valid input and is specified by a conjunction of linear constraints of the form $\bar{a}\cdot\bar{x}+b\diamond 0$ where $\diamond \in \{>,<,=,\geq,\leq\}$. A transition is of the form $(\ell,\ell',\mathcal{R}) \in \mathcal{T}$ such that $\ell,\ell' \in \mathcal{L}$, and $\mathcal{R}$ is a formula over $\bar{x}$, $\bar{u}$ and $\bar{x}'$ that is specified by a conjunction of linear constraints of the form $\bar{a}\cdot\bar{x}+\bar{b}\cdot\bar{u}+\bar{c}\cdot\bar{x}'+d\diamond 0$ where $\diamond \in \{>,<,=,\geq,\leq\}$, and primed variables $\bar{x}'$ represent the values of the unprimed corresponding variables after the transition. We sometimes write $\mathcal{R}$ as $\mathcal{R}(\bar{x},\bar{u},\bar{x}')$, use $\mathcal{R}(\bar{x})$ to refer to the constraints that involve only variables $\bar{x}$ (i.e., the *guard*), and use $\mathcal{R}(\bar{x},\bar{u})$ to refer to the constraints that involve only variables $\bar{u}$ and (possibly) $\bar{x}$. W.l.o.g., we may assume that constraints involving primed variables are of the form $x_i' = \bar{a}\cdot\bar{x}+\bar{b}\cdot\bar{u}+c$. This is because non-determinism can be moved to $\mathcal{R}(\bar{x},\bar{u})$ – if a primed variable x_i' appears in any expression that is not of this form, we replace x_i' by a fresh non-deterministic variable u_i in such expressions and add the equality $x_i' = u_i$. We require that for any $\bar{x}$ satisfying $\mathcal{R}(\bar{x})$, there are $\bar{u}$ satisfying $\mathcal{R}(\bar{x},\bar{u})$, formally

$$\forall\bar{x}.\exists\bar{u}.\ \mathcal{R}(\bar{x}) \to \mathcal{R}(\bar{x},\bar{u}) \tag{1}$$

This guarantees that for any state $\bar{x}$ satisfying the condition, there are values for the non-deterministic variables $\bar{u}$ such that we can make progress. A transition that does not satisfy this condition is called *invalid*. Note that (1) does not refer to $\bar{x}'$ since they are set in a deterministic way, once the values of $\bar{x}$ and $\bar{u}$ are fixed. W.l.o.g., we assume that all coefficients and free constants, in all linear constraints, are integer; and that there is a single *initial location* $\ell_0 \in \mathcal{L}$ with no incoming transitions, and a single *final location* ℓ_e with no outgoing transitions.

Example 1. The TS graphically presented in Fig. 1 is expressed as follows, considering that all inputs are valid ($\Theta = \mathit{true}$):

$$\begin{aligned}
\mathcal{S} \equiv \langle\ & \{x,y\},\emptyset,\{\ell_0,\ell_1,\ell_e\},\\
& \{(\ell_0,\ell_1,x'=x\wedge y'=y),\\
& \ (\ell_1,\ell_1,x\geq 0\wedge y>0\wedge x'=x-1\wedge y'=y),\\
& \ (\ell_1,\ell_e,x<0\wedge x'=x\wedge y'=y),\\
& \ (\ell_1,\ell_e,y\leq 0\wedge x'=x\wedge y'=y),\\
& \ (\ell_1,\ell_1,x\geq 0\wedge y>0\wedge x'=x\wedge y'=y-1)\},\mathit{true}\rangle
\end{aligned}$$

A configuration C is a pair (ℓ,σ) where $\ell \in \mathcal{L}$ and $\sigma : \bar{x} \mapsto \mathbb{Z}$ is a mapping representing a state. We abuse notation and use σ to refer to $\wedge_{i=1}^{n} x_i = \sigma(x_i)$, and also write σ' for the assignment obtained from σ by renaming the variables to primed variables. There is a transition from (ℓ,σ_1) to (ℓ',σ_2) iff there is $(\ell,\ell',\mathcal{R}) \in \mathcal{T}$ such that $\exists\bar{u}.\sigma_1 \wedge \sigma_2' \models \mathcal{R}$. A (valid) trace t is a (possibly infinite) sequence of configurations $(\ell_0,\sigma_0),(\ell_1,\sigma_1),\ldots$ such that $\sigma_0 \models \Theta$, and for each i there is a transition from (ℓ_i,σ_i) to $(\ell_{i+1},\sigma_{i+1})$. Traces that are infinite or end in a configuration with location ℓ_e are called complete. A configuration (ℓ,σ), where $\ell \neq \ell_e$, is *blocking* iff

$$\sigma \not\models \bigvee_{(\ell,\ell',\mathcal{R})\in\mathcal{T}} \mathcal{R}(\bar{x}) \tag{2}$$

A TS is non-blocking if no trace includes a blocking configuration. We assume that the TS under consideration is non-blocking, and thus any trace is a prefix of a complete one. Throughout the paper, we represent a TS as a CFG, and analyze its strongly connected components (SCC) one by one. An SCC is said to be *trivial* if it has no edge.

2.2 Lower-Bounds

For simplicity, we assume that an execution step (a transition) costs 1. Under this assumption, the cost of a trace t is simply its length $len(t)$ where the length of an infinite trace is ∞. In what follows, the set of all configurations is denoted by $\mathcal{C}$, the set of all valid complete traces (using a transition system $\mathcal{S}$) when starting from configuration $C \in \mathcal{C}$ is denoted by $Traces_{\mathcal{S}}(C)$, and $\mathbb{R}_{\geq 0} = \{k \in \mathbb{R} \mid k \geq 0\} \cup \{\infty\}$. For a non-empty set $M \subseteq \mathbb{R}_{\geq 0}$, $sup\ M$ is the least upper bound of M and $inf\ M$ is the greatest lower bound of M. The worst-case cost of an initial configuration C is the cost of the most expensive complete trace starting from C and the best-case cost is the less expensive complete trace.

Definition 1 (worst- and best-case cost). *Let $\mathcal{S}$ be a TS. Its worst-case cost function $wc_{\mathcal{S}} : \mathcal{C} \to \mathbb{R}_{\geq 0}$ is $wc_{\mathcal{S}}(C) = sup\ \{len(t) \mid t \in Traces_{\mathcal{S}}(C)\}$ and its best-case cost function $bc_{\mathcal{S}} : \mathcal{C} \to \mathbb{R}_{\geq 0}$ is $bc_{\mathcal{S}}(C) = inf\ \{len(t) \mid t \in Traces_{\mathcal{S}}(C)\}$.*

Clearly, $wc_{\mathcal{S}}$ and $bc_{\mathcal{S}}$ are not computable. Our goal in this paper is to automatically find a lower-bound function $\rho : \mathbb{Z}^n \to \mathbb{R}_{\geq 0}$ such that for any initial configuration $C = (\ell_0, \sigma)$ we have $wc_{\mathcal{S}}(C) \geq \rho(\sigma(\bar{x}))$, i.e., it is an LB^w. An LB^b would be a function $\rho : \mathbb{Z}^n \to \mathbb{R}_{\geq 0}$ that ensures that $bc_{\mathcal{S}}(C) \geq \rho(\bar{x})$ for any initial configuration $C = (\ell_0, \sigma)$. In what follows, for a function $\rho(\bar{x})$, we let $\|\rho(\bar{x})\| = \lceil \max(0, \rho(x)) \rceil$ to map all negative valuations of ρ to zero.

Example 2. Consider the TS $\mathcal{S} = \langle \{x\}, \{u\}, \{\ell_0, \ell_1, \ell_e\}, \mathcal{T}, true \rangle$ with transitions:

$$\begin{aligned} \mathcal{T} \equiv \{\ & \tau_1 = (\ell_0, \ell_1, x \geq 0), \\ & \tau_2 = (\ell_1, \ell_1, x > 0 \wedge x' = x - u \wedge u \geq 1 \wedge u \leq 2), \\ & \tau_3 = (\ell_1, \ell_e, x \leq 0 \wedge x' = x)\ \} \end{aligned}$$

$\mathcal{S}$ contains a loop at ℓ_1 where variable x is non-deterministically decreased by 1 or 2. From any initial configuration $C_0 = (\ell_0, \sigma_0)$, the longest possible complete trace decreases x by 1 in every iteration with τ_2, therefore $wc_{\mathcal{S}}(C_0) = \|\sigma_0(x)\| + 2$ because of the $\|\sigma_0(x)\|$ iterations in ℓ_1 plus the cost of τ_1 and τ_3. The most precise lower bound for $wc_{\mathcal{S}}$ is $\rho(x) = \|x\| + 2$, although $\rho(x) = \|x\|$ or $\rho(x) = \|x - 2\|$ are also valid lower bounds. The shortest complete trace from C_0 decreases x by 2 in every iteration, so $bc_{\mathcal{S}}(C_0) = \|\frac{\sigma_0(x)}{2}\| + 2$. There are several valid lower bounds for $bc_{\mathcal{S}}(C_0)$ like $\rho(x) = \|\frac{x}{2}\| + 2$, $\rho(x) = \|\frac{x}{2}\|$, or $\rho(x) = 2$.

3 Local Lower-Bound Functions

Focus on Local Bounds. Existing techniques and tools for cost analysis (e.g., [1, 12]) work by inferring *local* (iteration) bounds for those parts of the TS that correspond to loops, and then combining these bounds by propagating them "backwards" to the entry point in order to obtain a *global* bound. For example, suppose that our program consists of the following two loops:

```
assert(x>0 && z>0);
while (z > 0) { x=x+z; z--; }
while (x > 0) x--;
```

where the second loop makes x iterations (when considering the value of x just before executing the loop), and the first loop makes z iterations and increments x by z in each iteration. We are interested in inferring a global function that describes the total number of iterations of both loops, in terms of the input values x_0 and z_0. While both loops have linear complexity locally, i.e., iteration bounds z and x, the second one has quadratic complexity w.r.t the initial values. This can be inferred automatically from the local bounds z and x by inferring how the value of x changes in the first loop, and then rewriting x in terms of the initial values to $e = x_0 + \frac{z_0 \cdot (z_0 - 1)}{2}$ (e.g., by solving corresponding recurrence relations). Now the global cost would be e plus the cost of the first loop z_0. Rewriting the loop bound x as above is done by propagating it backwards to the entry point, and there are several techniques in the literature for this purpose that can be directly adopted in our setting to produce global bounds. These techniques can infer global bounds for nested-loops as well, given the iteration bounds of each loop. Thus, we focus on inferring local lower-bounds on the number of iterations that non-nested loops (more precisely, parts of the TS that correspond to loops) can make, and assume that they can be rewritten to global bounds by adopting the existing techniques of [1,12] (our implementation indeed could be used as a black-box which provides local lower-bounds to these tools). Namely, we aim at inferring, for each non-nested loop, a function $\|\rho(\bar{x})\| = \lceil \max(0, \rho(x)) \rceil$ that is a (local) LB^w on its number of iterations, i.e., whenever the loop is reached with values $\bar{v}$ for the variables $\bar{x}$, it is possible to make at least $\|\rho(\bar{v})\|$ iterations.

Loops and TSs. For ease of presentation, we first consider a special case of TSs in which all locations, except the initial and exit ones define loops, and Sect. 3.6 explains how the techniques can be used for the general case. In particular, we consider that each non-trivial SCC consists of a single location ℓ and at least one transition, and we call it *loop* ℓ. Transitions from ℓ to ℓ are called *loop transitions* and their guards are called *loop guards*, and transitions from ℓ to $\ell' \neq \ell$ are called *exit transitions*. The number of iterations of a loop ℓ in a trace t is defined as the number of transitions from ℓ to ℓ, which we refer to as the cost of loop ℓ as well (since we are assuming that the cost of transitions is always 1, see Sect. 2.2). The notions of best-case and worst-case cost in Definition 1 naturally extend to the cost of a loop ℓ, i.e., we can ask what is the best-case and worst-case number of iterations of a given loop.

Overview of the Section. The overall idea of our approach is to *specialize* each loop ℓ, by restricting the initial values and/or adding constraints to its transitions, such that it becomes possible to obtain a metering function for the specialized loop. A function that is a LB^b of the specialized loop is by definition a LB^w of loop ℓ, as it does not necessarily hold for all execution traces but rather for the class of restricted ones. Technically, inferring a LB^b of a (specialized) loop is done by inferring a metering function ρ [13], such that whenever the (specialized) loop is reached with a state σ, it is guaranteed to make at least $\|\rho(\sigma(\bar{x}))\|$ iterations. Besides, specialization is done in such away that the TS obtained by putting all specialized loops together is non-blocking, i.e., there is an execution that is either non-terminating or reaches the exit location, and thus the cost of this execution is, roughly, the sum of the costs of all (specialized) loops that are traversed. The rest of this section is organized as follows. In Sect. 3.1 we generalize the basic definition of metering function for simple loops from [12] to general types of loops and explore its limitations. Then, in the following 3 sections, we explain how to overcome these limitations by means of the following specializations: using quasi-invariants to narrow the set of input values (Sect. 3.2); narrowing loop guards to make loop transitions mutually exclusive and force some execution order between them (Sect. 3.3); and narrowing the space of non-deterministic choices to force longer executions (Sect. 3.4). Sect. 3.5 states the conditions, to be satisfied when specializing loops, in order to guarantee that the TS obtained by putting all specialized loops together is non-blocking.

3.1 Metering Functions

Metering functions were introduced by [13], as a tool for inferring a lower-bound on the number of iterations that a given loop can make. The definition is analogue to that of (linear) ranking function which is often used to infer upper-bounds on the number of iterations. The definition as given in [13] considers a loop with a single transition, and assumes that the exit condition is the negation of its guard. We start by generalizing it to our notion of loop.

Definition 2 (Metering function). *We say that a function ρ_ℓ is a metering function for a loop $\ell \in \mathcal{L}$, if the following conditions are satisfied*

$$\forall \bar{x}, \bar{u}, \bar{x}'.\ \mathcal{R} \rightarrow \rho_\ell(\bar{x}) - \rho_\ell(\bar{x}') \leq 1 \qquad \textbf{for each } (\ell, \ell, \mathcal{R}) \in \mathcal{T} \qquad (3)$$

$$\forall \bar{x}, \bar{u}, \bar{x}'.\ \mathcal{R} \rightarrow \rho_\ell(\bar{x}) \leq 0 \qquad \textbf{for each } (\ell, \ell', \mathcal{R}) \in \mathcal{T} \qquad (4)$$

Intuitively, Condition (3) *requires ρ_ℓ to decrease at most by* 1 *in each iteration, and Condition* (4) *requires ρ_ℓ to be non-positive when leaving the loop.*

Assuming (ℓ, σ) is a reachable configuration in $\mathcal{S}$, it is easy to see that loop ℓ will make at least $\|\rho_\ell(\sigma(\bar{x}))\|$ iterations when starting from (ℓ, σ). We require (ℓ, σ) to be reachable in $\mathcal{S}$ since we are interested only in non-blocking executions. Typically, we are interested in linear metering functions, i.e., of the form $\rho_\ell(\bar{x}) = \bar{a} \cdot \bar{x} + a_0$, since they are easier to infer and cover most loops in practice. Non-linear lower-bound functions will be obtained when rewriting these local linear lower-bounds in terms of the initial input at location ℓ_0 (see beginning of Sect. 3) and by composing nested loops (see Sect. 3.6).

Example 3 (Metering function). Consider the following loop on location ℓ_1 that decreases x (τ_1) until it takes non-positive values and exits to ℓ_2 (τ_2):

$$\tau_1 = (\ell_1, \ell_1, x \geq 0 \land x' = x - 1) \qquad \tau_2 = (\ell_1, \ell_2, x < 0 \land x' = x)$$

The function $\rho_{\ell_1}(x) = x + 1$ is a valid metering function because it decreases by exactly 1 in τ_1 and becomes non-positive when τ_2 is applicable ($x < 0 \rightarrow x+1 \leq 0$, Condition (3) of Definition 2). The function $\rho'_{\ell_1}(x) = \frac{x}{2}$ is also metering because its value decreases by less than 1 when applying τ_1 ($\frac{x}{2} - \frac{x-1}{2} = \frac{1}{2} \leq 1$) and becomes non-positive in τ_2. Even a function as $\rho''_{\ell_1}(x) = 0$ is trivially metering, as it satisfies (3) and (4). Although all of them are valid metering functions, $\rho_{\ell_1}(x)$ is preferable as it is more accurate (i.e., larger) and thus captures more precisely the number of iterations of the loop. Note that functions like $\rho^*_{\ell_1}(x) = 2x$ or $\rho^{**}_{\ell_1}(x) = x + 5$ are not metering because they do not verify (3) (because $2x - 2(x-1) = 2 \not\leq 1$ for $\rho^*_{\ell_1}$) or (4) (because $x < 0 \not\rightarrow x + 5 \leq 0$ for $\rho^{**}_{\ell_1}$).

3.2 Narrowing the Set of Input Values Using Quasi-Invariants

Metering functions typically exist for loops with simple loop guards. However, when guards involve more than one inequality they usually do not exist in a simple (linear) form. This is because such loops often include several exit transitions with unrelated conditions, where each one corresponds to the negation of an inequality of the guard. It is unlikely then that a non-trivial (linear) function satisfies (4) for all exit transitions. This is illustrated in the next example.

Example 4. Consider the following loop that iterates on ℓ_1 if $x \geq 0 \land y > 0$, and exits when $x < 0$ or $y \leq 0$:

$$\begin{aligned}
\tau_1 &= (\ell_1, \ell_1, x \geq 0 \land y > 0 \land x' = x - 1 \land y' = y) \\
\tau_2 &= (\ell_1, \ell_2, x < 0 \land x' = x \land y' = y) \\
\tau_3 &= (\ell_1, \ell_2, y \leq 0 \land x' = x \land y' = y)
\end{aligned}$$

Intuitively, this loop executes $x + 1$ transitions, but $\rho_{\ell_1}(x, y) = x + 1$ is not a valid metering function because it does not satisfy (4) for τ_3: $y \leq 0 \not\rightarrow x+1 \leq 0$. Moreover, no other function depending on x (e.g., $\frac{x}{2}$, $x - 2$, etc.) will be a valid metering function, as it will be impossible to prove (4) for τ_3 only from the information $y \leq 0$ on its guard. The only valid metering function for this loop will be the trivial one $\rho_{\ell_1}(x, y) = c$ with $c \leq 0$, which does not provide any information about the number of iterations of the loop.

Our proposal to overcome the imprecision discussed above is to consider only a subset of the input values s.t. conditions (3,4) hold in the context of the corresponding reachable states. For example, the reachable states might exclude some of the exit transitions, i.e., it is guaranteed that they are never used, and then (4) is not required to hold for them. A metering function in this context is a LB^b of the loop when starting from that specific input, and thus it is a LB^w (i.e., not necessarily best-case) of the loop when the input values are not restricted.

Technically, our analysis materializes the above idea by relying on quasi-invariants [17]. A quasi-invariant for a loop ℓ is a formula $\mathcal{Q}_\ell$ over $\bar{x}$ such that

$$\forall \bar{x}, \bar{u}, \bar{x}'.\ \mathcal{Q}_\ell(\bar{x}) \wedge \mathcal{R} \rightarrow \mathcal{Q}_\ell(\bar{x}') \qquad \textbf{for each } (\ell, \ell, \mathcal{R}) \in \mathcal{T} \tag{5}$$

$$\exists \bar{x}.\ \mathcal{Q}_\ell(\bar{x}) \tag{6}$$

Intuitively, $\mathcal{Q}_\ell$ is similar to an inductive invariant but without requiring it to hold on the initial states, i.e., once $\mathcal{Q}_\ell$ holds it will hold during all subsequent visits to ℓ. This also means that for executions that start in states within $\mathcal{Q}_\ell$, it is guaranteed that $\mathcal{Q}_\ell$ is an over-approximation of the reachable states. Condition (6) is used to avoid quasi-invariants that are *false*. Given a quasi-invariant $\mathcal{Q}_\ell$ for ℓ, we say that ρ_ℓ is a metering function for ℓ if the following holds

$$\forall \bar{x}, \bar{u}, \bar{x}'.\ \mathcal{Q}_\ell(\bar{x}) \wedge \mathcal{R} \rightarrow \rho_\ell(\bar{x}) - \rho_\ell(\bar{x}') \leq 1 \qquad \textbf{for each } (\ell, \ell, \mathcal{R}) \in \mathcal{T} \tag{7}$$

$$\forall \bar{x}, \bar{u}, \bar{x}'.\ \mathcal{Q}_\ell(\bar{x}) \wedge \mathcal{R} \rightarrow \rho_\ell(\bar{x}) \leq 0 \qquad \textbf{for each } (\ell, \ell', \mathcal{R}) \in \mathcal{T} \tag{8}$$

Intuitively, these conditions state that (3,4) hold in the context of the states induced by $\mathcal{Q}_\ell$. Assuming that (ℓ, σ) is reachable in $\mathcal{S}$ and that $\sigma \models \mathcal{Q}_\ell$, loop ℓ will make at least $\|\rho_\ell(\sigma(\bar{x}))\|$ iterations in any execution that starts in (ℓ, σ).

Example 5. Recall that the loop in Example 4 only admitted trivial metering functions because of the exit transition τ_3. It is easy to see that $\mathcal{Q}_{\ell_1} \equiv x < y$ verifies (5,6), because y is not modified in τ_1 and x decreases, and thus it is a quasi-invariant. In the context of $\mathcal{Q}_{\ell_1}$, function $\rho_{\ell_1}(x, y) = x + 1$ is metering because when taking τ_3 the value of x is guaranteed to be negative, i.e., τ_3 satisfies (8) because $x < y \wedge y \leq 0 \rightarrow x + 1 \leq 0$. Notice that $\rho_{\ell_1}(x, y) = x + 1$ will still be a valid metering function considering other quasi-invariants of the form $\mathcal{Q}'_{\ell_1} \equiv y > c$ with $c \geq 0$, as they would completely disable transition τ_3.

3.3 Narrowing Guards

The loops that we have considered so far consist of a single loop transition, what makes easier to find a metering function. This is because there is only one way to modify the program variables (with some degree of non-determinism induced by the non-deterministic variables). However, when we allow several loop transitions, we can have loops for which a non-trivial metering function does not exist even when narrowing the set of input values.

Example 6. Consider the extension of the loop in Example 4 with a new transition τ_4 that decrements y (it corresponds to the example in Sect. 1):

$$\begin{aligned}
\tau_1 &= (\ell_1, \ell_1, x \geq 0 \wedge y > 0 \wedge x' = x - 1 \wedge y' = y) \\
\tau_4 &= (\ell_1, \ell_1, x \geq 0 \wedge y > 0 \wedge x' = x \wedge y' = y - 1) \\
\tau_2 &= (\ell_1, \ell_2, x < 0 \wedge x' = x \wedge y' = y) \\
\tau_3 &= (\ell_1, \ell_2, y \leq 0 \wedge x' = x \wedge y' = y)
\end{aligned}$$

The most precise LB^w of this loop is $\|\rho_{\ell_1}(x, y)\|$ where $\rho_{\ell_1}(x, y) = x + y$. As mentioned, this corresponds, e.g., to an execution that uses τ_1 until $x = 0$, i.e., x times, and then τ_4 until $y = 0$, i.e., y times. It is easy to see that if we start from a state that satisfies $x \geq 0 \wedge x \leq y$, then it will be satisfied during the particular execution that we just described. Moreover, assuming that $\mathcal{Q}_{\ell_1} \equiv x \geq 0 \wedge x \leq y$ is a quasi-invariant, it is easy to show that together with ρ_{ℓ_1} we can verify (7,8), and thus ρ_{ℓ_1} will be a metering function. However, unfortunately, $\mathcal{Q}_{\ell_1}$ is not a quasi-invariant since the above loop can make executions other than the one described above (e.g., decreasing y to 1 first and then x to 0).

Our idea to overcome this imprecision is to narrow the set of states for which loop transitions are enabled, i.e., strengthening loop guards by additional inequalities. This, in principle, reduces the number of possible executions, and thus it is more likely to find a metering function (or a better quasi-invariant), because now they have to be valid for fewer executions. For example, this might force an execution order between the different paths, or even disable some transitions by narrowing their guard to *false*. Again, a metering function for the specialized loop is not a valid LB^b of the original loop, but rather its a valid LB^w that is what we are interested in. Next, we state the requirements that such narrowing should satisfy. The choice of a narrowing that leads to longer executions is discussed in Sect. 4.

A *guard narrowing* for a loop transition $\tau \in \mathcal{T}$ is a formula $\mathcal{G}_\tau(\bar{x})$, over variables $\bar{x}$. A specialization of a loop is obtained simply by adding these formulas to the corresponding transitions. Conditions (5)-(8) can be specialized to hold only for executions that use the specialized loop as follows. Suppose that for a loop $\ell \in \mathcal{L}$ we are given a narrowing $\mathcal{G}_\tau$ for each loop transition τ, then $\mathcal{Q}_\ell$ and ρ_ℓ are quasi-invariant and metering function resp. for the corresponding specialized loop if the following conditions hold

$$\forall \bar{x}, \bar{u}, \bar{x}'.\ \mathcal{Q}_\ell(\bar{x}) \wedge \mathcal{G}_\tau(\bar{x}) \wedge \mathcal{R} \rightarrow \mathcal{Q}_\ell(\bar{x}') \qquad \textbf{for each } (\ell, \ell, \mathcal{R}) \in \mathcal{T} \qquad (9)$$

$$\exists \bar{x}.\ \mathcal{Q}_\ell(\bar{x}) \qquad (10)$$

$$\forall \bar{x}, \bar{u}, \bar{x}'.\ \mathcal{Q}_\ell(\bar{x}) \wedge \mathcal{G}_\tau(\bar{x}) \wedge \mathcal{R} \rightarrow \rho_\ell(\bar{x}) - \rho_\ell(\bar{x}') \leq 1 \qquad \textbf{for each } (\ell, \ell, \mathcal{R}) \in \mathcal{T} \qquad (11)$$

$$\forall \bar{x}.\ \mathcal{Q}_\ell(\bar{x}) \wedge \mathcal{R}(\bar{x}) \rightarrow \rho_\ell(\bar{x}) \leq 0 \qquad \textbf{for each } (\ell, \ell', \mathcal{R}) \in \mathcal{T} \qquad (12)$$

Conditions (9,10) guarantee that $\mathcal{Q}_\ell$ is a non-empty quasi-invariant for the specialized loop, and conditions (11,12) guarantee that ρ_ℓ is a metering function for the specialized loop in the context of $\mathcal{Q}_\ell$. However, in this case, function ρ_ℓ induces a lower-bound on the number of iterations only if the specialized loop is non-blocking for states in $\mathcal{Q}_\ell$. This is illustrated in the following example.

Example 7. Consider the loop from Example 3 where we have specialized the guard of τ_1 by adding $x \geq 5$:

$$\tau_1 = (\ell_1, \ell_1, x \geq 0 \wedge x \geq 5 \wedge x' = x - 1) \qquad \tau_2 = (\ell_1, \ell_2, x < 0 \wedge x' = x)$$

With this specialized guard and considering $\mathcal{Q}_{\ell_1} \equiv \mathit{true}$, the metering function $\rho_{\ell_1}(x) = x + 1$ still satisfies (11,12), and $\mathcal{Q}_{\ell_1}$ trivially satisfies (9,10). However, ρ_{ℓ_1} is not a valid measure of the number of transitions executed because the loop gets blocked whenever x takes values $0 \leq x \leq 5$, and thus it will never execute $x + 1$ transitions.

To guarantee that the specialized loop is non-blocking for states in $\mathcal{Q}_\ell$, it is enough to require the following condition to hold

$$\forall \bar{x}.\ \mathcal{Q}_\ell(\bar{x}) \rightarrow \bigvee_{\tau=(\ell,\ell,\mathcal{R})\in\mathcal{T}} (\mathcal{R}(\bar{x}) \wedge \mathcal{G}_\tau(\bar{x})) \bigvee_{\tau=(\ell,\ell',\mathcal{R})\in\mathcal{T}} \mathcal{R}(\bar{x}) \qquad (13)$$

Intuitively, it states that from any state in $\mathcal{Q}_\ell$ we can make progress, either by making a loop iteration or exiting the loop. Assuming that (ℓ,σ) is reachable in $\mathcal{S}$ and that $\sigma \models \mathcal{Q}_\ell$, the specialized loop ℓ will make at least $\|\rho_\ell(\sigma(\bar{x}))\|$ iterations in any execution that starts in (ℓ,σ). This also means that the original loop *can* make at least $\|\rho_\ell(\sigma(\bar{x}))\|$ iterations in any execution that starts in (ℓ,σ).

Example 8. In Example 6, we have seen that if $\mathcal{Q}_{\ell_1} \equiv x \leq y \wedge x \geq 0$ was a quasi-invariant, then function $\rho_{\ell_1}(x,y) = x+y$ becomes metering. We can make $\mathcal{Q}_{\ell_1}$ a quasi-invariant by specializing the guards of the loop in transitions τ_1 and τ_4 to force the following execution with $x+y$ iterations: first use τ_1 until $x=0$ (x iterations) and then use τ_4 until $y=0$ (y iterations). This behavior can be forced by taking $\mathcal{G}_{\tau_1} \equiv x > 0$ and $\mathcal{G}_{\tau_4} \equiv x \leq 0$. With $\mathcal{G}_{\tau_1}$ we assure that x stops decreasing when $x=0$, and with $\mathcal{G}_{\tau_4}$ we assure that τ_4 is used only when $x=0$. Now, $\mathcal{Q}_{\ell_1} \equiv x \leq y \wedge x \geq 0$ and $\rho_{\ell_1}(x,y) = x+y$ are valid quasi-invariant and metering, resp. Function ρ_{ℓ_1} decreases by exactly 1 in τ_1 and τ_4, is trivially non-positive in τ_2 because that transition is indeed disabled ($x \geq 0$ from $\mathcal{Q}_{\ell_1}$ and $x < 0$ from the guard) and is non-positive in τ_3 ($x \leq y \wedge y \leq 0 \rightarrow x+y \leq 0$). Regarding $\mathcal{Q}_{\ell_1}$, it verifies (9,10), and more importantly, the loop in ℓ_1 is non-blocking w.r.t $\mathcal{Q}_{\ell_1}$, $\mathcal{G}_{\tau_1}$, and $\mathcal{G}_{\tau_4}$, i.e., Condition (13) holds.

3.4 Narrowing Non-deterministic Choices

Loop transitions that involve non-deterministic variables, might give rise to executions of different lengths when starting from the same input values. Since we are interested in LB^w, we are clearly searching for longer executions. However, since our approach is based on inferring LB^b, we have to take all executions into account which might result in less precise, or even trivial, LB^w.

Example 9. Consider a modification of the loop in Example 6 in which the variable x in τ_1 is decreased by a non-deterministic positive quantity u:

$$\tau_1 = (\ell_1, \ell_1, x \geq 0 \wedge y > 0 \wedge x' = x - u \wedge u \geq 1 \wedge y' = y)$$

The effect of this non-deterministic variable u is that τ_1 can be applied x times if we always take $u=1$, $\lceil \frac{x}{2} \rceil$ times if we always take $u=2$ or even only once if we take $u > x$. As a consequence, $\rho_{\ell_1}(x,y) = x+y$ is no longer a valid metering function because x can decrease by more than 1 in τ_1. Moreover, $\mathcal{Q}_{\ell_1} \equiv x \leq y \wedge x \geq 0$ is not a quasi-invariant anymore since $x' = x - u \wedge u \geq 1$ does not entail $x' \geq 0$. In fact, no metering function involving x will be valid in τ_1 because x can decrease by any positive amount.

To handle this complex situation, we propose narrowing the space of non-deterministic choices, and thus metering functions should be valid *wrt.* fewer executions and more likely be found and be more precise. Next we state the requirements that such narrowing should satisfy. The choice of a narrowing that leads to longer executions is discussed in Sect. 4.

A *non-deterministic variables narrowing* for a loop transition $\tau \in \mathcal{T}$ is a formula $\mathcal{U}_\tau(\bar{x}, \bar{u})$, over variables $\bar{x}$ and $\bar{u}$, that is added to τ to restrict the choices for variables $\bar{u}$. A specialized loop is now obtained by adding both $\mathcal{G}_\tau$ and $\mathcal{U}_\tau$ to the corresponding transitions. Suppose that for loop $\ell \in \mathcal{L}$, in addition to $\mathcal{G}_\tau$, we are also given $\mathcal{U}_\tau$ for each of its loop transitions τ. For $\mathcal{Q}_\ell$ and ρ_ℓ to be quasi-invariant and metering function for the specialized loop ℓ, we require conditions (9)-(13) to hold but after adding $\mathcal{U}_\tau$ to the left-hand side of the implications in (9) and (11). Besides, unlike narrowing of guards, narrowing of non-deterministic choices might make a transition invalid, i.e., not satisfying Condition (1), and thus $\|\rho_\ell(\bar{x})\|$ cannot be used as a lower-bound on the number of iterations. To guarantee that specialized transitions are valid we require, in addition, the following condition to hold

$$\forall \bar{x} \exists \bar{u}.\ \mathcal{Q}_\ell(\bar{x}) \wedge \mathcal{R}(\bar{x}) \wedge \mathcal{G}_\tau(\bar{x}) \rightarrow \mathcal{R}(\bar{x}, \bar{u}) \wedge \mathcal{U}_\tau(\bar{x}, \bar{u}) \qquad \textbf{for each } (\ell, \ell, \mathcal{R}) \in \mathcal{T} \tag{14}$$

This condition is basically (1) taking into account the inequalities introduced by the corresponding narrowings. Assuming that (ℓ, σ) is reachable in $\mathcal{S}$ and that $\sigma \models \mathcal{Q}_\ell$, the specialized loop ℓ will make at least $\|\rho_\ell(\sigma(\bar{x}))\|$ iterations in any execution that starts in (ℓ, σ), which also means, as before, that the original loop *can* make at least $\|\rho_\ell(\sigma(\bar{x}))\|$ iterations in any execution that starts in (ℓ, σ).

Example 10. To solve the problems shown in Example 9 we need to narrow the non-deterministic variable u to take bounded values that reflect the worst-case execution of the loop. Concretely, we need to take $\mathcal{U}_{\tau_1} \equiv u \leq 1$, which combined with $u \geq 1$ entails $u = 1$ so x decreases by exactly 1 in τ_1. Considering the narrowing $\mathcal{U}_{\tau_1}$, the resulting loop is equivalent to the one presented in Example 8 so we could obtain the precise metering function $\rho_{\ell_1}(x, y) = x + y$ with the quasi-invariant $\mathcal{Q}_{\ell_1} \equiv x \leq y \wedge x \geq 0$. Note that (14) holds for τ_1 because $u = 1$ makes the consequent true for every value of x and y: $\forall \bar{x} \exists \bar{u}.\ (x \leq y \wedge x \geq 0) \wedge (x \geq 0 \wedge y > 0) \wedge x > 0 \rightarrow u \geq 1 \wedge u \leq 1$

3.5 Ensuring the Feasibility of the Specialized Loops

In order to enable the propagation of the local lower-bounds back to the input location (as we have discussed at the beginning of Sect. 3), we have to ensure that there is actually an execution that starts in ℓ_0 and passes through the specialized loop. In other words, we have to guarantee that when putting all specialized loops together, they still form a non-blocking TS for some set of input values. We achieve this by requiring that the quasi-invariants of the preceding loops ensure that the considered quasi-invariant for this loop also holds on initialization (i.e., it is an invariant for the considered context). Technically, we require, in addition to (9)-(14), the following conditions to hold for each loop ℓ:

$$\forall \bar{x}, \bar{u}, \bar{x}'.\ \mathcal{Q}_{\ell'}(\bar{x}) \wedge \mathcal{R} \rightarrow \mathcal{Q}_\ell(\bar{x}') \qquad \textbf{for each } (\ell', \ell, \mathcal{R}) \in \mathcal{T} \tag{15}$$

$$\forall \bar{x}.\ \mathcal{Q}_{\ell_0} \rightarrow \Theta \tag{16}$$

Condition (15) means that transitions entering loop ℓ, strengthened with the quasi-invariant of the preceding location ℓ', must lead to states within the quasi-invariant $\mathcal{Q}_\ell$. Condition (16) guarantees that $\mathcal{Q}_{\ell_0}$ defines valid input values, i.e., within the initial condition Θ.

Theorem 1 (soundness). *Given $\mathcal{Q}_\ell$ for each non-exit location $\ell \in \mathcal{L}$, narrowings $\mathcal{G}_\tau$ and $\mathcal{U}_\tau$ for each loop transition $\tau \in \mathcal{T}$, and function ρ_ℓ for each loop location ℓ, such that (9)-(16) are satisfied, it holds:*

1. *The TS $\mathcal{S}'$ obtained from $\mathcal{S}$ by adding $\mathcal{G}_\tau$ and $\mathcal{U}_\tau$ to the corresponding transitions, and changing the initial condition to $\mathcal{Q}_{\ell_0}$, is non-blocking.*
2. *For any complete trace t of $\mathcal{S}'$, if $C = (\ell, \sigma)$ is a configuration in t, then t includes at least $\|\rho_\ell(\sigma(\bar{x}))\|$ visits to ℓ after C (i.e., $\|\rho_\ell(\bar{x})\|$ is a lower-bound function on the number of iterations of the loop defined by location ℓ).*

The proof of this soundness result is straightforward: it follows as a sequence of facts using the definitions of the conditions (9)-(16) given in this section.

We note that when there is an unbounded overlap between the guards of the loop transitions and the guards of exit transitions, it is likely that a non-trivial metering function does not exist because it must be non-positive on the overlapping states. To overcome this limitation, instead of using the exit transitions in (12), we can use ones that correspond to the negation of the guards of loop transitions, and thus it is ensured that they do not overlap. However, we should require (13) to hold for the original exit transitions as well in order to ensure that the non-blocking property holds. Another way to overcome this limitation is to simply strengthen the exit transitions by the negation of the guards.

As a final comment, we note that it is not needed to assume that the TS $\mathcal{S}$ that we start with is non-blocking (even though we have done so in Sect. 2.1 for clarity). This is because our formalization above finds a subset of $\mathcal{S}$ ($\mathcal{S}'$ in Theorem 1) that is non-blocking, which is enough to ensure the feasibility of the local lower-bounds. This is useful not only for enlarging the set of TSs that we accept as input, but also allows us to start the analysis from any subset of $\mathcal{S}$ that includes a path from ℓ_0 to the exit location. For example, it can be used to remove trivial execution paths from $\mathcal{S}$, or concentrate on ones that include more sequences of loops (since we are interested in LB^w).

3.6 Handling General TSs

So far we have considered a special case of TSs in which all locations, except the entry and exit ones, are multi-path loops. Next we explain how to handle the general case. It is easy to see that we can allow locations that correspond to trivial SCCs. These correspond to paths that connect loops and might include branching as well. For such locations, there is no need to infer metering functions or apply any specialization, we only need to assign them quasi-invariants that satisfy (15) to guarantee that the overall specialized TS is non-blocking.

The more elaborated case is when the TS includes non-trivial SCCs that do not form a multi-path loop. In such case, if a SCC has a single cut-point, we can unfold its edges and transform it into a multi-path following the techniques of [1]. It is important to note that when merging two transitions, the cost of the new one is the sum of their costs. In this case the number of iterations is still a lower-bound on the cost of the loop, however, we might get a better one by multiplying it by the minimal cost of its transitions.

If a SCC cannot be transformed into a multi-path loop by unfolding its transitions, then it might correspond to a nested loop, and, in such case, we can recover the nesting structure and consider them as separated TSs that are "called" from the outer one using loop extraction techniques [25]. Each inner-loop is then analyzed separately, and replaced (in the original TS, where is "called") by a single edge with its lower-bound as cost for that edge, and then the outer is analyzed taking that cost into account. Besides, to guarantee that the specialized program corresponds to a valid execution, we require the quasi-invariant of the inner loop to hold in the context of the quasi-invariant of the outer loop. This approach is rather standard in cost analysis of structured programs [1,3,12].

Another issue is how to compose the (local) lower-bounds of the specialized loops into a global-lower bound. For this, we can rely on the techniques [1,3] that rewrite the local lower-bounds in terms of the input values by relying on invariant generation and recurrence relations solving.

4 Inference Using Max-SMT

This section presents how metering functions and narrowings can be inferred automatically using Max-SMT, namely how to automatically infer all $\mathcal{G}_\tau$, $\mathcal{U}_\tau$, $\mathcal{Q}_\ell$, and ρ_ℓ such that (9)-(16) are satisfied. We do it in a modular way, i.e., we seek $\mathcal{G}_\tau$, $\mathcal{U}_\tau$, $\mathcal{Q}_\ell$, and ρ_ℓ for one loop at a time following a (reversed) topological order of the SCCs, as we describe next. Recall that (16) is required only for loops connected directly to ℓ_0, and w.l.o.g. we assume there is only one such loop.

4.1 A Template-Based Verification Approach

We first show how the template-based approach of [6,17] can be used to find $\mathcal{G}_\tau$, $\mathcal{U}_\tau$, and $\mathcal{Q}_\ell$ by representing them as template constraint systems, i.e., each is a conjunction of linear constraints where coefficients and constants are unknowns. Also, ρ_ℓ is represented as a linear template function $\bar{a} \cdot \bar{x} + a_0$ where $(a_0, \bar{a})$ are unknowns. Then, the problem is to find concrete values for the unknowns such that all formulas generated by (9)-(16) are satisfied:

– Each $\forall$-formula generated by (9)-(16), except those of (14) that we handle below, can be viewed as an $\exists\forall$ problem where the $\exists$ is over the unknowns of the templates and the $\forall$ is over (some of) the program variables. It is well-known that solving such an $\exists\forall$ problem, i.e., finding values for the unknowns, can be done by translating it into a corresponding $\exists$ problem over the existentially quantified variables (i.e., the unknowns) using Farkas' lemma [20], which can then be solved using an off-the-shelf SMT solver.

– To handle (14) we follow [17], and eliminate $\exists \bar{u}$ using the skolemization $u_i = \bar{a} \cdot \bar{x} + a_0$ where $(a_0, \bar{a})$ are fresh unknowns (different for each u_i). This allows handling it using Farkas' lemma as well. However, in addition, when solving the corresponding $\exists$ problem we require all $(a_0, \bar{a})$ to be integer. This is because the domain of program variables is the integers, and picking integer values for all $(a_0, \bar{a})$ guarantees that the values of any x'_i that depends on $\bar{u}$ will be integer as well[1].

The size of templates for $\mathcal{G}_\tau$, $\mathcal{U}_\tau$, and $\mathcal{Q}_\ell$, i.e., the number of inequalities, is crucial for precision and performance. The larger the size is, the more likely that we get a solution if one exists, but also the worse the performance is (as the corresponding SMT problem will include more constraints and variables). In practice, one typically starts with templates of size 1, and iteratively increases it by 1 when failing to find values for the unknowns, until a solution is found or the bound on the size is reached.

Alternatively, we can use the approach of [17] to construct $\mathcal{G}_\tau$, $\mathcal{U}_\tau$, and $\mathcal{Q}_\ell$ incrementally. This starts with templates of size 1, but instead of requiring all (9)-(16) to hold, the conditions generated by (12) are marked as soft constraints (i.e., we accept solutions in which they do not hold) and use Max-SMT to get a solution that satisfies as many of such soft conditions as possible. If all are satisfied, we are done, if not, we use the current solution to instantiate the templates, and then add another template inequality to each of them and repeat the process again. This means that at any given moment, each template will include at most one inequality with unknowns. Finally, to guarantee progress from one iteration to another, soft conditions that hold at some iteration are required to hold at the next one, i.e., they become hard.

The use of (12) as soft constraint is based on the observation [12] that when seeking a metering function, the problematic part is often to guarantee that it is negative on exit transitions, which is normally achieved by adding quasi-invariants that are incrementally inferred. By requiring (12) to be soft we handle more exit transitions as the quasi-invariant gets stronger until all are covered.

4.2 Better Quality Solutions

The precision can also be affected by the quality of the solution picked by the SMT solver for the corresponding $\exists$ problem. Since there might be many metering functions that satisfy (9)-(16), we are interested in narrowing the search space of the SMT solver in order to find more accurate ones, i.e., lead to longer executions. Next we present some techniques for this purpose.

Enabling More Loop Transitions. We are interested in guard narrowings that keep as many loop transitions as possible, since such narrowings are more likely

[1] Because we assumed that constraints involving primed variables are of the form $x'_i = \bar{a} \cdot \bar{x} + \bar{b} \cdot \bar{u} + c$.

to generate longer executions. This can be done by requiring the following to hold

$$\exists \bar{x}. \bigvee_{\tau=(\ell,\ell,\mathcal{R})\in\mathcal{T}} (\mathcal{Q}_\ell(\bar{x}) \wedge \mathcal{R}(\bar{x}) \wedge \mathcal{G}_\tau(\bar{x})) \tag{17}$$

We also use Max-SMT to require a solution that satisfies as many disjuncts as possible and thus eliminating less loop transitions (if $\mathcal{Q}_\ell(\bar{x}) \wedge \mathcal{R}(\bar{x}) \wedge \mathcal{G}_\tau(\bar{x})$ is *false* for a transition τ, then it is actually disabled). Note that this condition can be used instead of (10) that requires the quasi-invariant to be non-empty.

Larger Metering Functions. We are interested in metering functions that lead to longer executions. One way to achieve this is to require metering functions to be ranking as well, i.e., in addition to (11) we require the following to hold

$$\forall \bar{x}, \bar{u}, \bar{x}'.\mathcal{Q}_\ell(\bar{x}) \wedge \mathcal{G}_\tau(\bar{x}) \wedge \mathcal{U}_\tau(\bar{x}, \bar{u}) \wedge \mathcal{R} \rightarrow \rho_\ell(\bar{x}) - \rho_\ell(\bar{x}') \geq 1 \quad \textbf{for each } (\ell, \ell, \mathcal{R}) \in \mathcal{T} \tag{18}$$

$$\forall \bar{x}, \bar{u}.\mathcal{Q}_\ell(\bar{x}) \wedge \mathcal{G}_\tau(\bar{x}) \wedge \mathcal{R}(\bar{x}) \rightarrow \rho_\ell(\bar{x}) \geq 0 \quad \textbf{for each } (\ell, \ell, \mathcal{R}) \in \mathcal{T} \tag{19}$$

These new conditions are added as soft constraints, and we use Max-SMT to ask for a solution that satisfies as many conditions as possible.

Unbounded Metric Functions. We are interested in metering functions that do not have an upper bound, since otherwise they will lead to constant lower-bound functions. For example, for a loop with a transition $x \geq 0 \wedge x' = x - 1$, we want to avoid quasi-invariants like $x \leq 5$ which would make the metering function x bounded by 5. For this, we rely on the following lemma.

Lemma 1. *A function $\rho(\bar{x}) = \bar{a} \cdot \bar{x} + a_0$ is unbounded over a polyhedron $\mathcal{P}$, iff $\bar{a} \cdot \bar{y}$ is positive on at least one ray $\bar{y}$ of the* recession cone *of $\mathcal{P}$.*

It is known that for a polyhedron $\mathcal{P}$ given in constraints representation, its recession cone $\texttt{cone}(\mathcal{P})$ is the set specified by the constraints of $\mathcal{P}$ after removing all free constants. Now we can use the above lemma to require that the metering function $\rho_\ell(\bar{x}) = \bar{a} \cdot \bar{x} + \bar{a}_0$ is unbounded in the quasi-invariant $\mathcal{Q}_\ell$ by requiring the following condition to hold

$$\exists \bar{x}.\ \texttt{cone}(\mathcal{Q}_\ell) \wedge \bar{a} \cdot \bar{x} > 0 \tag{20}$$

where $\texttt{cone}(\mathcal{Q}_\ell)$ is obtained from the template of $\mathcal{Q}_\ell$ by removing all (unknowns corresponding to) free constants, i.e., it is the *recession cone* of $\mathcal{Q}_\ell$.

Note that all encodings discussed in this section generate non-linear SMT problems, because they either correspond to $\exists\forall$ problems that include templates on the left-hand side of implications, or to $\exists$ problems over templates that include both program variables and unknowns.

Finally, it is important to note that the optimizations described provide theoretical guarantees to get better lower bounds: the one that adds (18,19) leads to a bound that corresponds exactly to the worst-case execution (of the specialized program), and the one that uses (20) is essential to avoid constant bounds.

5 Implementation and Experimental Evaluation

We have implemented a *LO*wer-*B*ound synthesiz*ER*, named LOBER, that can be used from an online web interface at http://costa.fdi.ucm.es/lober. LOBER is built as a pipeline with the following processes: (1) it first reads a KoAT file [5] and generates a corresponding set of multi-path loops, by extracting parts of the TS that correspond to loops [25], applying unfolding, and inferring loop summaries to be used in the calling context of nested loops, as explained in Sect. 3.6; (2) it then encodes in SMT the conditions (9)–(13) defined through the paper, for each loop separately, by using template generation, a process that involves several non-trivial implementations using Farkas' lemma (this part is implemented in Java and uses Z3 [8] for simple (linear) satisfiability checks when producing the Max-SMT encoding); (3) the problem is solved using the SMT solver Barcelogic [4], as it allows us to use non-linear arithmetic and Max-SMT capabilities in order to assert soft conditions and implement the solutions described in Sect. 4; (4) in order to guarantee the correctness of our system results, we have added to the pipeline an additional checker that proves that the obtained metering function and quasi-invariants verify conditions (9)–(13) by using Z3. To empirically evaluate the results of our approach, we have used benchmarks from the Termination Problem Data Base (TPDB), namely those from the category *Complexity_ITS* that contains Integer Transition Systems. We have removed non-terminating TSs and terminating TSs whose cost is unbounded (i.e., the cost depends on some non-deterministic variables and can be arbitrarily high) or non-linear, because they are outside the scope of our approach. In total, we have considered a set of 473 multi-path loops from which we have excluded 13 that were non-linear. Analyzing these 473 programs took 199 min, an average of 25 sec by program, approximately. For 255 of them, it took less than 1 s.

Table 1 illustrates our results and compares them to those obtained by the LoAT [12,13] system, which also outputs a pair $(\rho, \mathcal{Q})$ of a lower-bound function ρ and initial conditions $\mathcal{Q}$ on the input for which ρ is a valid lower-bound. In order to automatically compare the results obtained by the two systems, we have implemented a comparator that first expresses costs as functions $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$ over a single variable n and then checks which function is greater. To obtain this unary cost function from the results $(\rho, \mathcal{Q})$, we use convex polyhedra manipulation libraries to maximize the obtained cost ρ wrt. $\mathcal{Q} \wedge \overline{-n \leq x_i \leq n}$, where x_i are the TS variables, and express that maximized expressions in terms of n. Therefore, $f(n)$ represents the maximum cost when the variables are bounded by $|x_i| \leq n$ and satisfy the corresponding initial condition $\mathcal{Q}$, a notion very similar to the runtime complexity used in [12,13]. Once we have both unary linear costs $f_1(n) = k_1 n + d_1$ and $f_2(n) = k_2 n + d_2$, we compare them in $n \geq 0$ by inspecting k_1 and k_2.

Each row of the table contains the number of loops for which both tools obtain the same result (=), the number of loops where LOBER is better than LoAT (>) and the number of loops where LoAT is better than LOBER (<). The subcategories are obtained directly from the name of the innermost folder, except for the cases in which this folder contains too few examples that we merge them

Table 1. Results of the experiments.

Benchmark set	Total	=	>	<
Brockschmidt_16				
c-examples/ABC	33	33	0	0
c-examples/SPEED	29	25	4	0
c-examples/WTC	45	39	4	2
c-examples/Misc	9	9	0	0
costa	6	5	1	0
FGPSF09/Beerendonk	28	24	4	0
FGPSF09/patrs	18	16	2	0
FGPSF09/Misc	20	16	3	1
KoAT-2013	10	10	0	0
KoAT-2014	14	14	0	0
SAS10	46	40	1	5
Flores-Montoya_16	176	158	16	2
Hark_20				
Ben_Amram_Genaim	10	7	2	1
Nils_2019	16	16	0	0

all in a Misc folder in the parent directory. The total number of loops that are considered in each subcategory appears in column **Total**. Brockschmidt_16 and Hark_20 have their first row empty as all their results are contained in their subcategories. Globally, both tools behave the same in 412 programs (column "="), obtaining equivalent linear lower bounds in 376 of them and a constant lower bound in the remaining ones. Our tool LOBER achieves a better accuracy in 37 programs (column ">"), while LoAT is more precise in 11 programs (column "<"). Let us discuss the two sets of programs in which both tools differ. As regards the 37 examples for which we get better results, we have that LoAT crashes in 4 cases and it can only find a constant lower bound in 1 example while our tool is able to find a path of linear length by introducing the necessary quasi-invariants. For the remaining 32 loops, both tools get a linear bound, but LOBER finds one that leads to an unboundedly longer execution: 18 of these loops correspond to cases that have implicit relations between the different execution paths (like our running examples) and require semantic reasoning; for the remaining 14, we get a better set of quasi-invariants. The following techniques have been needed to get such results in these 37 better cases (note that (i) is not mutually exclusive with the others):

(i) 1 needs narrowing non-deterministic choices,
(ii) 5 do not need quasi-invariants nor guard narrowing,
(iii) 14 need quasi-invariants only,
(iv) 18 need both quasi-invariants and guard narrowing (in 3 of them this is only used to disable transitions).

Therefore, this shows experimentally the relevance of all components within our framework and its practical applicability thanks to the good performance of the Max-SMT solver on non-linear arithmetic problems. In general, for all the set of programs, we can solve 308 examples without quasi-invariants and 444 without guard-narrowing. The intersection of these two sets is: 298 examples (63% of the programs), that leaves 175 programs that need the use of some of the proposed techniques to be solved.

As regards the 11 examples for which we get worse results than LoAT, we have two situations: (1) In 6 cases, the SMT-solver is not able to find a solution.

We noticed that too many quasi-invariants were required, what made the SMT problem too hard. To improve our results, we could start, as a preprocessing step, from a quasi-invariant that includes all invariant inequalities that syntactically appear in the loop transitions, something similar to what is done by LoAT when inferring what they call conditional metering function [12]. This is left for future experimentation. (2) In the other 5 cases, our tool finds a linear bound but with a worse set of quasi-invariants, which makes the LoAT bound provide unboundedly longer executions. We are investigating whether this can be improved by adding new soft constraints that guide the solver to find these better solutions. Finally, let us mention that, for the 13 problems that LoAT gives a non-linear bound and have been excluded from our benchmarks as justified above, we get a linear bound for the 12 that have a polynomial bound (of degree 2 or more), and a constant bound for the additional one that has a logarithmic lower bound. This is the best we can obtain as our approach focuses on the inference of precise local linear bounds, as they constitute the most common type of loops.

All in all, we argue that our experimental results are promising: we triple LoAT in the number of benchmarks for which we get more accurate results and, besides, many of those examples correspond to complex loops that lead to worse results when disconnecting transitions. Besides, we see room for further improvement, as most examples for which LoAT outperforms us could be handled as accurately as them with better quasi-invariants (that is somehow a black-box component in our framework). Syntactic strategies that use invariant inequalities that appear in the transitions, like those used in LoAT, would help, as well as further improvements in SMT non-linear arithmetic.

Application Domains. The accuracy gains obtained by LOBER have applications in several domains in which knowing the precise cost can be fundamental. This is the case for predicting the gas usage [26] of executing *smart contracts*, where gas cost amounts to monetary fees. The caller of a transaction needs to include a gas limit to run it. Giving a too low gas limit can end in an "out of gas" exception and giving a too high gas limit can end in a "not enough eth (money)" error. Therefore having a tighter prediction is needed to be safe on both sides. Also, when the UB is equal to the LB, we have an exact estimation, e.g., we would know precisely the runtime or memory consumption of the most costly executions. This can be crucial in safety-critical applications and has been used as well to detect potential vulnerabilities such as denial-of-service attacks. In https://apps.dtic.mil/sti/pdfs/AD1097796.pdf, vulnerabilities are detected in situations in which both bounds do not coincide. For instance, in password verification programs, if the UB and LB differ due to a difference on the delays associated to how many characters are right in the guessed password, this is identified as a potential attack.

6 Related Work and Conclusions

We have proposed a novel approach to synthesize precise lower-bounds from integer non-deterministic programs. The main novelties are on the use of loop specialization to facilitate the task of finding a (precise) metering function and on the Max-SMT encoding to find larger (better) solutions. Our work is related to two lines of research: (1) non-termination analysis and (2) LB inference. In both kinds of analysis, one aims at finding classes of inputs for which the program features a non-terminating behavior (1) or a cost-expensive behavior (2). Therefore, techniques developed for non-termination might provide a good basis for developing a LB analysis. In this sense, our work exploits ideas from the Max-SMT approach to non-termination in [17]. The main idea borrowed from [17] has been the use of quasi-invariants to specialize loops towards the desired behavior: in our case towards the search of a metering function, while in theirs towards the search of a non-termination proof. However, there are fundamental differences since we have proposed other new forms of loop specialization (see a more detailed comparison in Sect. 1) and have been able to adapt the use of Max-SMT to accurately solve our problem (i.e., find larger bounds). As mentioned in Sect. 1, our loop specialization technique can be used to gain precision in non-termination analysis [17]. For instance, in this loop: "while (x>=0 and y>=0) {if (∗) {x++; y−−;} else {x−−;y++;}}" no sub SCC (considering only one of the transitions) is non-terminating and no quasi-invariant can be found to ensure we will stay in the loop (when considering both transitions), hence cannot be handled by [17]. Instead if we narrow the transitions by adding $y >= x$ in the if-condition (and hence $x > y$ in the else), we can prove that $x >= 0 \wedge y >= 0 \wedge x + y = 1$ is quasi-invariant, which allow us to prove non-termination in the way of [17] (as we will stay in the loop forever).

As regards LB inference, the current state-of-the-art is the work by Frohn et al. [12,13] that introduces the notion of metering function and acceleration. Our work indeed tries to recover the semantic loss in [12,13] due to defining metering functions for simple loops and combining them in a later stage using acceleration. Technically, we only share with this work the basic definition of metering function in Sect. 3.1. Indeed, the definition in conditions (3) and (4) already generalizes the one in [12,13] since it is not restricted to simple loops. This definition is improved in the following sections with several loop specializations. While [12,13] relies on pure SMT to solve the problem, we propose to gain precision using Max-SMT. We believe that similar ideas could be adapted by [12,13]. Due to the different technical approaches underlying both frameworks, their accuracy and efficiency must be compared experimentally wrt. the LoAT system that implements the ideas in [12,13]. We argue that the results in Sect. 5 justify the important gains of using our new framework and prove experimentally that, the fact that we do not lose semantic relations in the search of metering functions is key to infer LB for challenging cases in which [12,13] fails. Originally, the LoAT [12,13] system only accelerated simple loops by using metering functions, so the overall precision of the lower bound relied on obtaining valid and precise metering functions. However, the framework in [12,13] is independent of the accelerating technique applied. In order to increase the number of simple loops that can be accelerated, Frohn [11] proposes a calculus to combine different conditional acceleration techniques (monotonic increase/decrease, eventual increase/decrease, and metering functions). These conditional acceleration techniques assume that all the iterations of the loop verify some condition φ, and the calculus applies the techniques in order and extract those conditions φ from fragments of the loop guard. Although more precise and powerful, the combined acceleration calculus considers only simple loops, so it does not solve the precision loss when the loop cost involves several interleaved transitions. Moreover, the techniques in [11] are integrated into LoAT, so the experimental evaluation in Sect. 5 compares our approach to the framework in [12,13] extended with several techniques to accelerate loops (not only metering functions).

Finally, our approach presents similarities to the CTL* verification for ITS in [7] as both extend transition guards of the original ITS. The difference is that in [7] the added constraints only contain newly created *prophecy variables* and the transitions to modify are detected directly using graph algorithms; whereas our SMT-based approach adds constraints only over existing variables to satisfy the properties that characterize a good metering function. Additionally, both approaches differ both in the goal (CTL* verification vs. inference of lower-bounds) and the technologies applied (CTL model checkers vs. Max-SMT solvers).

References

1. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Automatic inference of upper bounds for recurrence relations in cost analysis. In: Alpuente, M., Vidal, G. (eds.) SAS 2008. LNCS, vol. 5079, pp. 221–237. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69166-2_15
2. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost analysis of java bytecode. In: De Nicola, Rocco (ed.) ESOP 2007. LNCS, vol. 4421, pp. 157–172. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71316-6_12
3. Albert, E., Genaim, S., Masud, A.N.: On the inference of resource usage upper and lower bounds. ACM Trans. Comput. Logic **14**(3), 1–35 (2013)
4. Bofill, M., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A.: The barcelogic SMT solver. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 294–298. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70545-1_27
5. Brockschmidt, M., et al.: Analyzing runtime and size complexity of integer programs. ACM Trans. Program. Lang. Syst. **38**(4), 1–50 (2016)
6. Colón, M.A., Sankaranarayanan, S., Sipma, H.B.: Linear invariant generation using non-linear constraint solving. In: Hunt, W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 420–432. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45069-6_39
7. Cook, B., Khlaaf, H., Piterman, N.: On automation of CTL* verification for infinite-state systems. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 13–29. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_2
8. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24

9. Debray, S.K., López-García, P., Hermenegildo, M., Lin, N.-W.: Lower Bound Cost Estimation for Logic Programs. The MIT Press, In Logic Programming (1997)
10. Flores-Montoya, A.: Upper and lower amortized cost bounds of programs expressed as cost relations. In: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (eds.) FM 2016. LNCS, vol. 9995, pp. 254–273. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48989-6_16
11. Frohn., F.: A calculus for modular loop acceleration. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 58–76. Springer International Publishing (2020)
12. Frohn, F., Naaf, M., Brockschmidt, M., Giesl, J.: Inferring lower runtime bounds for integer programs. ACM Trans. Program. Lang. Syst. **42**(3), 1–50 (2020)
13. Frohn, F., Naaf, M., Hensel, J., Brockschmidt, M., Giesl, J.: Lower runtime bounds for integer programs. In: Olivetti, N., Tiwari, A. (eds.) IJCAR 2016. LNCS (LNAI), vol. 9706, pp. 550–567. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40229-1_37
14. Gulwani, S., Mehra, K.K., Chilimbi, T.: SPEED. In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL 2009. ACM Press (2008)
15. Hark, M., Kaminski, B.L., Giesl, J., Katoen, J.-P.: Aiming low is harder: induction for lower bounds in probabilistic program verification. In: Proceedings of the ACM on Programming Languages, 4(POPL), pp. 1–28, January 2020
16. Hoffmann, J., Shao, Z.: Automatic static cost analysis for parallel programs. In: Vitek, J. (ed.) ESOP 2015. LNCS, vol. 9032, pp. 132–157. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46669-8_6
17. Larraz, D., Nimkar, K., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A.: Proving non-termination using Max-SMT. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 779–796. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_52
18. Ngo, V.C., Carbonneaux, Q., Hoffmann, J.: Bounded expectations: resource analysis for probabilistic programs. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, June 2018
19. Otto, C., Brockschmidt, M., Von Essen, C., Giesl, J.: Automated termination analysis of java bytecode by term rewriting. In: Lynch, C. (eds.) RTA, vol. 6 of LIPIcs, pp. 259–276. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2010)
20. Schrijver, A.: Theory of Linear and Integer Programming. Wiley, Chichester (1986)
21. Sinn, M., Zuleger, F., Veith, H.: A simple and scalable static analysis for bound analysis and amortized complexity analysis. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 745–761. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_50
22. Sinn, M., Zuleger, F., Veith, H.: Complexity and resource bound analysis of imperative programs using difference constraints. J. Autom. Reasoning **59**(1), 3–45 (2017). https://doi.org/10.1007/s10817-016-9402-4
23. Srivastava, S., Gulwani, S.: Program verification using templates over predicate abstraction. ACM SIGPLAN Not. **44**(6), 223–234 (2009)
24. Wegbreit, B.: Mechanical program analysis. Commun. ACM **18**(9), 528–539 (1975)
25. Wei, Tao, Mao, Jian, Zou, Wei, Chen, Yu.: A new algorithm for identifying loops in decompilation. In: Nielson, Hanne Riis, Filé, Gilberto (eds.) SAS 2007. LNCS, vol. 4634, pp. 170–183. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74061-2_11
26. Wood, G.: Ethereum: a secure decentralised generalised transaction ledger (2014)

Fast Computation of Strong Control Dependencies

Marek Chalupa(✉), David Klaška, Jan Strejček, and Lukáš Tomovič

Masaryk University, Brno, Czech Republic
{chalupa,strejcek}@fi.muni.cz,
{david.klaska,tomovic}@mail.muni.cz

Abstract. We introduce new algorithms for computing *non-termination sensitive control dependence* (NTSCD) and *decisive order dependence* (DOD). These relations on vertices of a control flow graph have many applications including program slicing and compiler optimizations. Our algorithms are asymptotically faster than the current algorithms. We also show that the original algorithms for computing NTSCD and DOD may produce incorrect results. We implemented the new as well as fixed versions of the original algorithms for the computation of NTSCD and DOD. Experimental evaluation shows that our algorithms dramatically outperform the original ones.

1 Introduction

Control dependencies between program statements are studied since 70's. They have important applications in compiler optimizations [12,14,16], program analysis [9,19,36], and program transformations, especially program slicing [1,9,22,26,37]. Slicing is used in many areas including testing, debugging, parallelization, reverse engineering, program analysis and verification [17,28].

Informally, two statements in a program are control dependent if one directly controls the execution of the other in some way. This is typically the case for **if** statements and their bodies. Control dependencies are nowadays classified as *weak (non-termination insensitive)* if they assume that a given program always terminates, or as *strong (non-termination sensitive)* if they do not have this assumption [13]. We illustrate the difference on the control flow graph in Fig. 1. Node a controls whether b or c (and then d) is going to be executed, so b, c, and d are control dependent on a (the convention is to display dependence as edges in the "controls" direction). Similarly, b controls the execution of c and d, as these nodes may be bypassed by going from b to e. Note also that d controls whether d is going to be executed in the future and thus is control dependent on itself. However, c does not control d as any path from c hits d. All dependencies mentioned so far are weak, namely *standard control dependencies* as defined by Ferrante et al. [16]. Weak control dependence assumes that the program always terminates, in particular, that the loop over d cannot iterate forever. As a result,

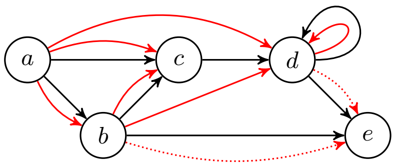


Fig. 1. An example of a control flow graph and control dependencies (red edges). The dotted dependencies are additional non-termination sensitive control dependencies. (Color figure online)

e is reached by all executions and thus it is not weakly control dependent on any node. However, e is strongly control dependent on b and d. Indeed, if we assume that some executions can loop over d forever, then reaching e is controlled clearly by d and also by b as it can send the execution directly to e.

This paper is concerned with the computation of two prominent strong control dependencies introduced by Ranganath et al. [32,33], namely *non-termination sensitive control dependence (NTSCD)* and *decisive order dependence (DOD)*. NTSCD is studied in Sect. 3, which follows after preliminaries in Sect. 2. We first recall the definition of NTSCD and the algorithm of Ranganath et al. [33] for its computation. Then we show a flaw in the algorithm and suggest a fix. Finally, we introduce a new algorithm for the computation of NTSCD. Given a control flow graph with $|V|$ nodes, the new algorithm runs in time $O(|V|^2)$, while the algorithm of Ranganath et al. runs in time $O(|V|^4 \cdot \log |V|)$ and its fixed version in time $O(|V|^5)$. We show a NTSCD relation of size $\Theta(|V|^2)$, which means that our algorithm is asymptotically optimal.

The DOD relation captures the cases when one node controls the execution order of two other nodes. Roughly speaking, nodes $\{b, c\}$ are DOD on a whenever all executions passing through a eventually reach both b and c and a controls which is reached first. Ranganath et al. [33] proved that the relation is empty for *reducible* graphs [21], i.e., graphs where every cycle has a single entry point. Control flow graphs of structured programs are reducible, but irreducible graphs may arise for example in the following situations [11,33,35]:

- unstructured coding by a human, which is rather rare nowadays,
- compilation into unstructured code representation like JVM bytecode,
- tail call recursion optimization during compilation,
- when the control flow graph is interprocedural – in this case, irreducibility may be introduced by recursion or exceptions handling,
- by reversing a control flow graph containing, for example, break statements
- when the control flow graph is not generated from program, but, e.g., from a finite state machine.

The DOD relation is important (together with NTSCD) when we want to slice possibly non-terminating programs with irreducible control flow graphs and preserve their termination properties as well as data integrity [1,33]. This is a common requirement when slicing is used as a preprocessing step before program verification [9,23,26], worst-case execution time analysis [29], information flow analysis [18,19], analysis of concurrent programs [18] with busy-waiting synchronization or synchronization where possible spurious wake-ups of threads are guarded by loops (e.g., programs using the *pthread* library), and analysis of reactive systems and generic state-based models [2,24,33].

The DOD relation is studied in Sect. 4, where we recall its definition, discuss the Ranganath et al.'s algorithm for DOD [33], and show that this algorithm also contains a flaw. Fortunately, this flaw can be easily fixed without changing the complexity of the algorithm. Further, we develop a theory that underpins our new algorithm for the computation of DOD. Due to the space limitations, proofs of theorems can be found only in the extended version of this paper [8]. The new algorithm, presented at the end of the section, computes DOD in time $O(|V|^3)$, while the original as well as the fixed version of the Ranganath et al.'s algorithm runs in $O(|V|^5 \cdot \log |V|)$. We show a DOD relation of size $\Theta(|V|^3)$, which means that our algorithm is again asymptotically optimal.

Section 5 focuses on *control closures (CC)* introduced by Danicic et al. [33], which generalize control dependence to arbitrary directed graphs. It is known that the *strong* (i.e., non-termination sensitive) control closure for a set of nodes containing the starting node is equivalent to the closure under NTSCD and DOD relations. Hence, our algorithms for NTSCD and DOD can be used to compute strong CC in time $O(|V|^3)$ on control flow graphs, while the original algorithm by Danicic et al. [13] runs in $O(|V|^4)$.

Our theoretical contribution to computation of strong control dependencies is summarized in Table 1. Section 6 presents experimental evaluation showing that our algorithms are indeed dramatically faster than the original ones. The paper is concluded with Sect. 7.

1.1 Related Work

The first paper concerned with control dependence is due to Denning and Denning [15], who used control dependence to certify that flow of information in a program is secure. Weiser [37], Ottenstein and Ottenstein [30], and Ferrante et al. [16] used control dependence in program slicing, which is also the motivation for the most of the latter research in this area. These "classical" papers study control dependence in terminating programs with a unique exit node eventually reached by every execution. These restrictions have been gradually removed.

Table 1. Overview of discussed algorithms and their complexities on CFGs

Relation/closure	Algorithm	Complexity
NTSCD	Original algorithm by Ranganath et al. [33]	$O(\|V\|^4 \cdot \log \|V\|)$
(Sect. 3)	Fixed algorithm by Ranganath et al. [33]	$O(\|V\|^5)$
	New algorithm	$O(\|V\|^2)$
DOD	Original algorithm by Ranganath et al. [33]	$O(\|V\|^5 \cdot \log \|V\|)$
(Sect. 4)	Fixed algorithm by Ranganath et al. [33]	$O(\|V\|^5 \cdot \log \|V\|)$
	New algorithm	$O(\|V\|^3)$
Strong CC	Original algorithm by Danicic et al. [13]	$O(\|V\|^4)$
(Sect. 5)	New NTSCD-and-DOD-based algorithm	$O(\|V\|^3)$

Podgurski and Clarke [31] defined the first strong control dependence that does not assume termination of the program.[1] However, their definitions and algorithms still require programs to have a unique exit node.

Bilardi and Pingal [5] introduced a framework that uses generalized dominance relation on graphs. In their framework, they are able to compute Podgurski and Clarke's control dependence in $O(|E|+|V|^2)$ time for a directed graph (V, E) with a unique exit node. In theory, NTSCD could be computed in their framework. However, computing *augmented post-dominator tree* – the central data structure of their framework – requires the unique exit node as it starts with post-dominator tree and, mainly, is much more complicated compared to our algorithm for NTSCD [5].

Chen and Rosu [10] introduced a parametric approach where loops can be annotated with information about termination. The resulting control dependence is somewhere between the classical and Podgurski and Clarke's control dependence, the two being the extremes.

The notion of NTSCD and DOD was founded in works of Ranganath et al. [32,33] in order to slice reactive systems, e.g., operating systems or controllers of embedded devices. They generalized also classical (*non-termination insensitive*) control dependence to graphs without the unique exit point (further investigated, e.g., by Androutsopoulos et al. [3]) and provided several relaxed versions of DOD.

Danicic et al. [13] introduced *weak* and *strong control closures (CC)* that generalize weak and strong control dependence (thus also NTSCD) to arbitrary graphs. They provide algorithms for the computation of minimal closures that run in $O(|V|^3)$ (weak CC) and $O(|V|^4)$ (strong CC) on graph with $|V|$ nodes.

An orthogonal study of control dependence that arises between statements in different procedures (e.g., due to calls to `exit()`) was carried out by Loyall and Mathisen [27], Harrold et al. [20], and Sinha et al. [34].

2 Preliminaries

A *finite directed graph* is a pair $G = (V, E)$, where V is a finite set of *nodes* and $E \subseteq V \times V$ is a set of *edges*. If there is an edge $(m, n) \in E$, then n is called a *successor* of m, m is a *predecessor* of n, and the edge is an *outgoing edge* of m. Given a node n, *Successors*(n) and *Predecessors*(n) denote the sets of all its successors and predecessors, respectively. A *path* from a node n_1 is a nonempty finite or infinite sequence $n_1 n_2 \ldots \in V^+ \cup V^\omega$ of nodes such that there is an edge $(n_i, n_{i+1}) \in E$ for each pair n_i, n_{i+1} of adjacent nodes in the sequence. A path is called *maximal* if it cannot be prolonged, i.e., it is infinite or the last node of the path has no outgoing edge. A node m is *reachable* from a node n if there exists a finite path such that its first node is n and its last node is m.

We say that a graph is a *cycle*, if it is isomorphic to a graph (V, E) where $V = \{n_1, \ldots, n_k\}$ for some $k > 0$ and $E = \{(n_1, n_2), (n_2, n_3), \ldots, (n_{k-1}, n_k),$

[1] Podgurski and Clarke [31] called their control dependence *weak control dependence* as it is a superset of classical control dependence. Nowadays, we use the terms *weak* and *strong* precisely in the opposite meaning [13].

$(n_k, n_1)\}$. A cycle *unfolding* is a path in the cycle that contains each node precisely once.

In this paper, we consider programs represented by control flow graphs, where nodes correspond to program statements and edges model the flow of control between the statements. As control dependence reflects only the program structure, our definition of a control flow graph does not contain any statements. Our definition also does not contain any start or exit nodes as these are not important for the problems we study in this paper.

Definition 1 (Control flow graph, CFG). *A* control flow graph (CFG) *is a finite directed graph $G = (V, E)$ where each node $v \in V$ has at most two outgoing edges. Nodes with exactly two outgoing edges are called* predicate nodes *or simply* predicates*. The set of all predicates of a CFG G is denoted by $Predicates(G)$.*

3 Non-termination Sensitive Control Dependence

This section recalls the definition of NTSC by Ranganath et al. [32] and their algorithm for computing NTSCD. Then we show that the algorithm can produce incorrect results and introduce a new algorithm that is asymptotically faster.

Definition 2 (Non-termination sensitive control dependence, NTSCD). *Given a CFG $G = (V, E)$, a node $n \in V$ is* non-termination sensitive control dependent (NTSCD) *on a predicate node $p \in Predicates(G)$, written $p \xrightarrow{NTSCD} n$, if p has two successors s_1 and s_2 such that*

- *all maximal paths from s_1 contain n, and*
- *there exists a maximal path from s_2 that does not contain n.*

3.1 Algorithm of Ranganath et al. [33] for NTSCD

The algorithm is presented in Algorithm 1. Its central data structure is a two-dimensional array S where for each node n and for each predicate node p with successors r and s, $S[n, p]$ always contains a subset of $\{t_{pr}, t_{ps}\}$. Intuitively, t_{pr} should be added to $S[n, p]$ if n appears on all maximal paths from p that start with the prefix pr. The *workbag* holds the set of nodes n for which some $S[n, p]$ value has been changed and this change should be propagated. The first part of the algorithm initializes the array S with the information that each successor r of a predicate node p is on all maximal paths from p starting with pr. The main part of the algorithm then spreads the information about the reachability on all maximal paths in the forward manner. Finally, the last part computes the NTSCD relation according to Definition 2 and with use of the information in S.

The algorithm runs in time $O(|E| \cdot |V|^3 \cdot \log |V|)$ [33] for a CFG $G = (V, E)$. The $\log |V|$ factor comes from set operations. Since every node in CFG has at most 2 outgoing edges, we can simplify the complexity to $O(|V|^4 \cdot \log |V|)$.

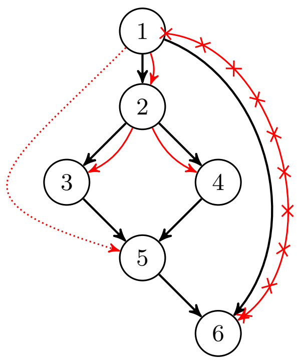
Although the correctness of the algorithm has been proved [32, Theorem 7], Fig. 2 presents an example where the algorithm provides an incorrect answer.

Algorithm 1: The NTSCD algorithm by Ranganath et al. [33]

```
   Input: a CFG $G = (V, E)$
   Output: a potentially incorrect NTSCD relation stored in $ntscd$

 1 Set $S[n,p] = \emptyset$ for all $n \in V$ and $p \in Predicates(G)$      // Initialization
 2 $workbag \leftarrow \emptyset$
 3 for $p \in Predicates(G)$ do
 4     for $r \in Successors(p)$ do
 5         $S[r,p] \leftarrow \{t_{pr}\}$
 6         $workbag \leftarrow workbag \cup \{r\}$
 7
 8 while $workbag \neq \emptyset$ do                                         // Computation of $S$
 9     $n \leftarrow$ pop from $workbag$
10     if $Successors(n) = \{s\}$ for some $s \neq n$ then                  // One successor case
11         for $p \in Predicates(G)$ do
12             if $S[n,p] \smallsetminus S[s,p] \neq \emptyset$ then
13                 $S[s,p] \leftarrow S[s,p] \cup S[n,p]$
14                 $workbag \leftarrow workbag \cup \{s\}$
15     if $|Successors(n)| > 1$ then                                        // Multiple successors case
16         for $m \in V$ do
17             if $|S[m,n]| = |Successors(n)|$ then
18                 for $p \in Predicates(G) \smallsetminus \{n\}$ do
19                     if $S[n,p] \smallsetminus S[m,p] \neq \emptyset$ then
20                         $S[m,p] \leftarrow S[m,p] \cup S[n,p]$
21                         $workbag \leftarrow workbag \cup \{m\}$
22
23 $ntscd \leftarrow \emptyset$                                             // Computation of NTSCD
24 for $n \in V$ do
25     for $p \in Predicates(G)$ do
26         if $0 < |S[n,p]| < |Successors(p)|$ then
27             $ntscd \leftarrow ntscd \cup \{p \xrightarrow{\text{NTSCD}} n\}$
```

The first part of the algorithm initializes S as shown in the figure and sets *workbag* to $\{2, 6, 3, 4\}$. Then any node from *workbag* can be popped and processed. Let us apply the policy used for queues: always pop the oldest element in *workbag*. Hence, we pop 2 and nothing happens as the condition on line 17 is not satisfied for any m. This also means that the symbol t_{12} is not propagated any further. Next we pop 6, which has no effect as 6 has no successor. By processing 3 and 4, t_{23} and t_{24} are propagated to $S[5, 2]$ and 5 is added to the *workbag*. Finally, we process 5 and set $S[6, 2]$ to $\{t_{23}, t_{24}\}$. The final content of S is provided in the figure. Unfortunately, the information in S is sound but incomplete. In other words, if $t_{pr} \in S[n, p]$, then n is indeed on all maximal paths from p starting with pr, but the opposite implication does not hold. In particular, t_{12} is missing in $S[5, 1]$ and $S[6, 1]$. Consequently, the last part of the algorithm computes an incorrect NTSCD relation: it correctly identifies $1 \xrightarrow{\text{NTSCD}} 2$, $2 \xrightarrow{\text{NTSCD}} 3$, and $2 \xrightarrow{\text{NTSCD}} 4$, but it also incorrectly produces $1 \xrightarrow{\text{NTSCD}} 6$ and misses $1 \xrightarrow{\text{NTSCD}} 5$.



S after initialization
$S[2,1] = \{t_{12}\}$
$S[6,1] = \{t_{16}\}$
$S[3,2] = \{t_{23}\}$
$S[4,2] = \{t_{24}\}$

final S when nodes are popped in order	
$2, 6, 3, 4, 5$ (oldest first)	$3, 4, 2, 5, 6$ (correct)
$S[2,1] = \{t_{12}\}$	$S[2,1] = \{t_{12}\}$
$S[6,1] = \{t_{16}\}$	$S[3,2] = \{t_{23}\}$
$S[3,2] = \{t_{23}\}$	$S[4,2] = \{t_{24}\}$
$S[5,2] = \{t_{23}, t_{24}\}$	$S[5,1] = \{t_{12}\}$
$S[6,2] = \{t_{23}, t_{24}\}$	$S[6,1] = \{t_{12}, t_{16}\}$
	$S[6,2] = \{t_{23}, t_{24}\}$

Fig. 2. An example that shows the incorrectness of the NTSCD algorithm by Ranganath et al. [33]. Solid red edges depict the dependencies computed by the algorithm when it always pops the oldest element in *workbag*. The crossed dependence is incorrect. The dotted dependence is missing in the result.

A necessary condition to get the correct result is to process 2 only after $3, 4$ are processed and $S[5,6] = \{t_{23}, t_{24}\}$. For example, one obtains the correct S (also shown in the figure) when the nodes are processed in the order $3, 4, 2, 5, 6$.

The algorithm is clearly sensitive to the order of popping nodes from *workbag*. We are currently not sure whether for each *CFG* there exists an order that leads to the correct result. An easy way to fix the algorithm is to ignore the *workbag* and repeatedly execute the body of the **while** loop (lines 10–21) for all $n \in V$ until the array S reaches a fixpoint. However, this modification would slow down the algorithm substantially. Computing the fixpoint needs $O(|V|^3)$ iterations over the loop body (lines 10–21 excluding lines 14 and 21 handling the *workbag*) and one iteration of this loop body needs $O(|V|^2)$. Hence, the overall time complexity of the fixed version is $O(|V|^5)$.

3.2 New Algorithm for NTSCD

We have designed and implemented a new algorithm computing NTSCD. Our algorithm is correct, significantly simpler and asymptotically faster than the original algorithm of Ranganath et al. [33].

The new algorithm calls for each node n a procedure that identifies all NTSCD dependencies of n on predicate nodes. The procedure works in the following steps.

1. Color n red.
2. Pick an uncolored node such that it has some successors and they all are red. Color the node red. Repeat this step until no such node exists.
3. For each predicate node p that has a red successor and an uncolored one, output $p \xrightarrow{\text{NTSCD}} n$.

Algorithm 2: The new NTSCD algorithm

```
Input: a CFG $G = (V, E)$
Output: the NTSCD relation stored in $ntscd$

Procedure VISIT($n$)                                  // Auxiliary procedure
    $n.counter \leftarrow n.counter - 1$
    if $n.counter = 0 \ \wedge \ n.color \neq red$ then
        $n.color \leftarrow red$
        for $m \in Predecessors(n)$ do
            VISIT($m$)

Procedure COMPUTE($n$)              // Coloring the graph red for a given $n$
    for $m \in V$ do
        $m.color \leftarrow uncolored$
        $m.counter \leftarrow |Successors(m)|$
    $n.color \leftarrow red$
    for $m \in Predecessors(n)$ do
        VISIT($m$)

$ntscd \leftarrow \emptyset$                                        // Computation of NTSCD
for $n \in V$ do
    COMPUTE($n$)
    for $p \in Predicates(G)$ do
        if $p$ has a red successor and an uncolored successor then
            $ntscd \leftarrow ntscd \cup \{p \xrightarrow{\text{NTSCD}} n\}$
```

Unlike the Ranganath et al.'s algorithm which works in a forward manner, our algorithm spreads the information about the reachability of n on all maximal paths in the backward direction starting from n.

The algorithm is presented in Algorithm 2. The procedure COMPUTE(n) implements the first two steps mentioned above. In the second step, it does not search over all nodes to pick the next node to color. Instead, it maintains the count of uncolored successors for each node. Once the count drops to 0 for a node, the node is colored red and the search continues with predecessors of this node. The third step is implemented directly in the main loop of the algorithm.

To prove that the algorithm is correct, we basically need to show that when COMPUTE(n) finishes, a node m is red iff all maximal paths from m contain n. We start with a simple observation.

Lemma 1. *After* COMPUTE(n) *finishes, a node m is red if and only if $m = n$ or m has a positive number of successors and all of them are red.*

Proof. For each node m, the counter is initialized to the number of its successors and it is decreased by calls to VISIT(m) each time a successor of m gets red. When the counter drops to 0 (i.e., all successors of the node are red), the node is colored red. Therefore, if m is red, it got red either on line 12 and $m = n$, or $m \neq n$ and m is red because all its successors got red (it must have a positive number of successors, otherwise the counter could not be 0 after its decrement). In the other direction, if $m = n$, it gets red on line 12. If it has a positive number of successors which all get red, the node is colored red by the argument above. □

Theorem 1. *After* COMPUTE(n) *finishes, for each node m it holds that m is red if and only if all maximal paths from m contain n.*

Proof. ("$\Longleftarrow$") We prove this implication by contraposition. Assume that m is an uncolored node. Lemma 1 implies that each uncolored node has an uncolored successor (if it has any). Hence, we can construct a maximal path from m containing only uncolored nodes simply by always going to an uncolored successor, either up to infinity or up to a node with no successors. This uncolored maximal path cannot contain n which is red.

("$\Longrightarrow$") For the sake of contradiction, assume that there is a red node m and a maximal path from m that does not contain n. Lemma 1 implies that all nodes on this path are red. If the maximal path is finite, it has to end with a node without any successor. Lemma 1 says that such a node can be red if and only if it is n, which is a contradiction. If the maximal path is infinite, it must contain a cycle since the graph is finite. Let r be the node on this cycle that has been colored red as the first one. Let s be the successor of r on the cycle. Recall that $r \neq n$ as the maximal path does not contain n. Hence, node r could be colored red only when all its successors including s were already red. This contradicts the fact that r was colored red as the first node on the cycle. □

To determine the complexity of our algorithm on a CFG (V, E), we first analyze the complexity of one run of COMPUTE(n). The lines 9–11 iterate over all nodes. The crucial observation is that the procedure VISIT is called at most once for each edge $(m, m') \in E$ of the graph: to decrease the counter of m when m' gets red. Hence, the procedure COMPUTE(n) runs in $O(|V| + |E|)$. This procedure is called on line 18 for each node n. Finally, lines 20–21 are executed for each pair of node n and predicate node p. This gives us the overall complexity $O((|V| + |E|) \cdot |V| + |V|^2) = O((|V| + |E|) \cdot |V|)$. Since in control flow graphs it holds $|E| \leq 2|V|$, the complexity can be simplified to $O(|V|^2)$.

Note that our algorithm is asymptotically optimal as there are CFGs with NTSCD relations of size $\Theta(|V|^2)$. For example, the CFG in Fig. 3 has $|V| = 2k+1$ nodes and the corresponding NTSCD relation

$$\{n_i \xrightarrow{\mathsf{NTSCD}} m_j \mid i, j \in \{1, \ldots, k\}\} \cup \{n_i \xrightarrow{\mathsf{NTSCD}} n_{i+1} \mid i \in \{1, \ldots, k-1\}\}$$

is of size $k^2 + k - 1 \in \Theta(|V|^2)$.

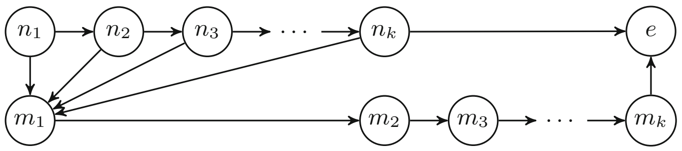


Fig. 3. A CFG with $|V|$ nodes that has the NTSCD relation of size $\Theta(|V|^2)$.

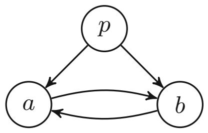


Fig. 4. An example of an irreducible CFG. There are no NTSCD dependencies, but a and b are DOD on p.

4 Decisive Order Dependence

There are control dependencies not captured by NTSCD. For example, consider the CFG in Fig. 4. Nodes a and b are not NTSCD on p as they lie on all maximal paths from p. However, p controls which of a and b is executed first. Ranganath et al. [33] introduced the DOD relation to capture such dependencies.

Definition 3 (Decisive order dependence, DOD). *Let $G = (V, E)$ be a CFG and $p, a, b \in V$ be three distinct nodes such that p is a predicate node with successors s_1 and s_2. Nodes a, b are* decisive order-dependent (DOD) *on p, written $p \xrightarrow{\text{DOD}} \{a, b\}$, if*

- *all maximal paths from p contain both a and b,*
- *all maximal paths from s_1 contain a before any occurrence of b, and*
- *all maximal paths from s_2 contain b before any occurrence of a.*

The importance of DOD for slicing of irreducible programs is discussed in the introduction.

4.1 Algorithm of Ranganath et al. [33] for DOD

Ranganath et al. provided an algorithm that computes the DOD relation for a given CFG $G = (V, E)$ in time $O(|V|^4 \cdot |E| \cdot \log |V|)$ which amounts to $O(|V|^5 \cdot \log |V|)$ on CFGs [33, Fig. 7]. The algorithm contains one unclear point. For each triple of nodes $p, a, b \in V$ such that $p \in \mathit{Predicates}(G)$ and $a \neq b$, the algorithm executes the following check and if it succeeds, then $p \xrightarrow{\text{DOD}} \{a, b\}$ is reported:

$$\textsc{reachable}(a, b, G) \wedge \textsc{reachable}(b, a, G) \wedge \textsc{dependence}(p, a, b, G) \qquad (1)$$

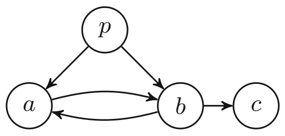


Fig. 5. An example that shows the incorrectness of the DOD algorithm by Ranganath et al. [33]

The procedure DEPENDENCE(p, a, b, G) returns *true* iff a is on all maximal paths from one successor of p before any occurrence of b and b is on all maximal paths from the other successor of p before any occurrence of a. The procedure REACHABLE is specified only by words [33, description of Fig. 7] as follows:

REACHABLE(a, b, G) returns *true* if b is reachable from a in the graph G.

Unfortunately, this algorithm can provide incorrect results. For example, consider the CFG in Fig. 5. Nodes p, a, b satisfy the formula (1): a appears on all maximal paths from one successor of p (namely a) before any occurrence of b, and b appears on all maximal paths from the other successor of p (which is b) before any occurrence of a. At the same time, a and b are reachable from each other. However, it is not true that $p \xrightarrow{\text{DOD}} \{a, b\}$, because a and b do not lie on all maximal paths from p (the first condition of Definition 3 is violated).

The algorithm can be fixed by modifying the procedure REACHABLE(a, b, G) to return *true* if b is on all maximal paths from a. The modified procedure can be implemented with use of the procedure COMPUTE(b) of Algorithm 2. As the procedure COMPUTE(b) runs in $O(|V| + |E|)$, the modification does not increase the overall complexity of the algorithm. By comparing the fixed and the original version of REACHABLE(a, b, G), one can readily confirm that the original version produces supersets of DOD relations.

4.2 New Algorithm for DOD: Crucial Observations

As in the case of NTSCD, we have designed a new algorithm for the computation of DOD, which is relatively simple and asymptotically faster than the DOD algorithm of Ranganath et al. [33].

Given a CFG, our algorithm first computes for each predicate p the set V_p of nodes that are on all maximal paths from p. The definition of DOD implies that only pairs of nodes in V_p can be DOD on p. For every predicate p we build an auxiliary graph A_p with nodes V_p and from this graph we get all pairs of nodes that are DOD on p. The graph A_p is defined as follows.

Definition 4 (V'-interval [13]). *Given a CFG $G = (V, E)$ and a subset $V' \subseteq V$, a path $n_1 \dots n_k$ such that $k \geq 2$, $n_1, n_k \in V'$, and $\forall 1 < i < k : n_i \notin V'$ is called a V'-*interval *from n_1 to n_k in G.*

In other words, a V'-interval is a finite path with at least one edge that has the first and the last node in V' but no other node on the path is in V'.

Definition 5 (Graph A_p[2]). *Given a CFG $G = (V, E)$, a predicate node $p \in$ Predicates(G) and the subset $V_p \subseteq V$ of nodes that are on all maximal paths from p, the $A_p = (V_p, E_p)$ is the graph where*

$$E_p = \{(x, y) \mid \textit{there exists a } V_p\textit{-interval from } x \textit{ to } y \textit{ in } G\}.$$

In this subsection, we describe the connections between these graphs and DOD that underpin our algorithm. The proofs of the theorems can be found in the extended version of this paper [8].

Given a predicate p of a CFG G, the graph A_p does not have to be a CFG as nodes in A_p can have more than two successors. However, A_p preserves exactly all possible orders of the first occurrences of nodes in V_p on maximal paths in G starting from p. More precisely, for each maximal path from p in G, there exists a maximal path from p in A_p with the same order of the first occurrences of all nodes in V_p, and vice versa. Further, it turns out that there are no nodes DOD on p unless A_p has the *right shape.*

Definition 6 (Right shape of A_p). *Given a CFG G, a predicate node $p \in$ Predicates(G) and the graph $A_p = (V_p, E_p)$, we say that A_p has the* right shape *if it consists only of a cycle and the node p with at least two edges going to some nodes on the cycle (i.e., the nodes of $V_p \setminus \{p\}$ can be labeled $n_1, \ldots, n_k$ such that $E_p = \{(n_1, n_2), (n_2, n_3), \ldots, (n_{k-1}, n_k), (n_k, n_1)\} \cup \{(p, n_i) \mid i \in I\}$ for some $I \subseteq \{1, \ldots, k\}$ with $|I| \geq 2$).*

Figure 6 depicts an A_p which has the right shape. In the following text, we work only with A_p graphs in the right shape.

Let s_1 and s_2 be the two successors of p in G. Note that s_1 and s_2 may, but do not have to be in A_p. To compute the pairs of nodes that are DOD on p, we need to know all possible orders of the first occurrences of nodes in V_p on the maximal paths in G starting in s_1 and s_2. Hence, for each successor s_i we compute the set S_i of nodes that appear as the first node of V_p on some maximal path from s_i in G. Formally, for $i \in \{1, 2\}$, we define

$$S_i = \{n \in V_p \mid \text{there exists a path } s_i \ldots n \in (V \setminus V_p)^*.V_p \text{ in } G\}.$$

The nodes in $S_1 \cup S_2$ are exactly all the successors of p in A_p. Further, the maximal paths from the nodes of S_i in A_p reflect exactly all possible orders of the first occurrences of nodes in V_p on maximal paths in G starting in s_i. If S_1 and S_2 are not disjoint, then there exist two maximal paths in G, one starting in s_1 and the other in s_2, that differ only in prefixes of nodes outside V_p. The definition of DOD implies that there are no nodes DOD on p in this case. Therefore we assume that S_1 and S_2 are disjoint.

The nodes in S_i divide the cycle of A_p into s_i*-strips*, which are parts of the cycle starting with a node from S_i and ending before the next node of S_i.

[2] Graph A_p can be defined as the graph induced by V_p in terms of Danicic et al. [13].

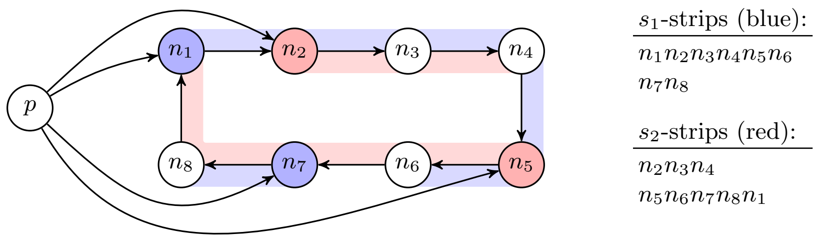


Fig. 6. An example of A_p in the right shape. Strips are computed for $S_1 = \{n_1, n_7\}$ (blue nodes) and $S_2 = \{n_2, n_5\}$ (red nodes). (Color figure online)

Definition 7 (s_i-strip). *Let $i \in \{1, 2\}$. An* s_i-strip *is a path $n \dots m \in S_i.(V_p \setminus S_i)^*$ in A_p such that the successor of m in A_p is a node in S_i.*

An example of A_p with s_i-strips is in Fig. 6. The s_i-strips directly say which pairs of nodes of V_p are in the same order on all maximal paths from s_i in G. In particular, a node a is before any occurrence of node b on all maximal paths from a successor s of p in G if and only if there is an s-strip containing both a and b where a is before b. As a corollary, we get the following theorem:

Theorem 2. *Let p be a predicate with successors s_1, s_2 such that A_p has the right shape and $S_1 \cap S_2 = \emptyset$. Then nodes $a, b \in V_p$ are DOD on p if and only if*

- *there exists an s_1-strip in A_p that contains a before b and*
- *there exists an s_2-strip in A_p that contains b before a.*

Consider again the A_p in Fig. 6. The theorem implies that nodes n_1, n_5 are DOD on p as they appear in s_1-strip $n_1n_2n_3n_4n_5n_6$ and in s_2-strip $n_5n_6n_7n_8n_1$ in the opposite order. Nodes n_1, n_6 are DOD on p for the same reason.

With use of the previous theorem, we can find a regular language over V_p such that there exist nodes a, b DOD on p iff some unfolding of the cycle in A_p is in the language.

Theorem 3. *Let p be a predicate with successors s_1, s_2 such that A_p has the right shape and $S_1 \cap S_2 = \emptyset$. Further, let $U = V_p \setminus (S_1 \cup S_2)$. There are some nodes a, b DOD on p if and only if the cycle in A_p has an unfolding of the form $S_1.U^*.(S_2.U^*)^*.S_2.U^*.(S_1.U^*)^*$.*

Finally, an unfolding of the mentioned form can be directly used for the computation of nodes that are DOD on p.

Theorem 4. *Let p be a predicate with successors s_1, s_2 such that A_p has the right shape and $S_1 \cap S_2 = \emptyset$. Further, let A_p have an unfolding of the form $S_1.U^*.(S_2.U^*)^*.S_2.U^*.(S_1.U^*)^*$ where $U = V_p \setminus (S_1 \cup S_2)$. Then there is exactly one path $m_1 \dots m_i \in S_1.U^*.S_2$ and exactly one path $o_1 \dots o_j \in S_2.U^*.S_1$ on the cycle. Moreover, $p \xrightarrow{DOD} \{a, b\}$ if and only if $m_1 \dots m_{i-1}$ contains a and $o_1 \dots o_{j-1}$ contains b (or the other way round).*

Algorithm 3: The algorithm computing V_n for all nodes n

```
Input: a CFG $G = (V, E)$
Output: $V_n = \{m \in V \mid m$ is on all maximal paths from $n\}$ for all $n \in V$

Procedure VISIT$(n, r)$                                  // Auxiliary procedure
    $n.counter \leftarrow n.counter - 1$
    if $n.counter = 0 \;\wedge\; r \notin V_n$ then
        $V_n \leftarrow V_n \cup \{r\}$
        for $m \in Predecessors(n)$ do
            VISIT$(m, r)$

Procedure COMPUTE$(n)$    // 'Coloring the graph red' for a given $n$
    for $m \in V$ do
        $m.counter \leftarrow |Successors(m)|$
    $V_n \leftarrow V_n \cup \{n\}$
    for $m \in Predecessors(n)$ do
        VISIT$(m, n)$

Procedure COMPUTE$V_p$S   // Computation of sets $V_n$ for all nodes $n$
    for $n \in V$ do
        $V_n \leftarrow \emptyset$
    for $n \in V$ do
        COMPUTE$(n)$
```

4.3 New Algorithm for DOD: Pseudocode and Complexity

Our DOD algorithm is shown in Algorithms 3 and 4. As nearly all applications of DOD need also NTSCD, we present the algorithm with a simple extension (gray lines with asterisks) that simultaneously computes NTSCD.

The DOD algorithm starts at line 20 of Algorithm 4. The first step is to compute the sets V_p for all predicate nodes p of a given CFG G. The computation of predicate nodes can be found in Algorithm 3. It is a slightly modified version of Algorithm 2. Recall that the procedure COMPUTE(n) of Algorithm 2 marks red every node such that all maximal paths from the node contain n. The procedure COMPUTE(n) of Algorithm 3 does in principle the same, but instead of the red color it marks the nodes with the identifier of the node n. Every node m collects these marks in set V_m. After we run COMPUTE(n) for all the nodes n in the graph, each node m has in its set V_m precisely all nodes that are on all maximal paths from m. For the computation of DOD, only the sets V_p for predicate nodes p are needed, but the extension computing NTSCD may use all these sets.

When the sets V_p are calculated, we compute DOD (and NTSCD) dependencies for each predicate node separately by procedures COMPUTEDOD(p) and COMPUTENTSCD(p). The procedure COMPUTEDOD(p) first constructs the graph A_p with the use of BUILD$A_p(p)$. Nodes of the graph are these of V_p. To compute edges, we trigger depth-first search in G from each $n \in V_p$. If we find a node $m \in V_p$, we add the edge (n, m) to the graph A_p and stop the search on

Algorithm 4: The new DOD algorithm which computes also NTSCD if the gray lines are included (COMPUTEV_pS is given in Algorithm 3)

```
Input: a CFG $G = (V, E)$
Output: the DOD relation stored in $dod$ (and NTSCD stored in $ntscd$)

1   Procedure COMPUTEDOD($p$) // Computation of DOD for predicate $p$
2       $A_p \leftarrow$ BUILD$A_p(p)$                          // Get the graph $A_p$
3       if $A_p$ does not have the right shape then
4           return $\emptyset$
5       $S_1, S_2 \leftarrow$ COMPUTE$S_1S_2(p)$                 // Get the sets $S_1, S_2$
6       if $S_1 \cap S_2 \neq \emptyset$ then
7           return $\emptyset$
8       $n_1n_2 \dots n_t \leftarrow$ UNFOLDCYCLE$(A_p, S_1)$      // Unfold the cycle of $A_p$
9       $U \leftarrow V_p \smallsetminus (S_1 \cup S_2)$
10      if $n_1n_2 \dots n_t \notin (S_1.U^*)^+.(S_2.U^*)^+.(S_1.U^*)^*$ then   // Apply Thm. 3
11          return $\emptyset$
12      $m_1 \dots m_i \leftarrow$ EXTRACT$(n_1n_2 \dots n_t, S_1.U^*.S_2)$     // Apply Thm. 4
13      $o_1 \dots o_j \leftarrow$ EXTRACT$(n_1n_2 \dots n_t, S_2.U^*.S_1)$
14      return $\{p \xrightarrow{\text{DOD}} \{a, b\} \mid a \in \{m_1, \dots, m_{i-1}\}, b \in \{o_1, \dots, o_{j-1}\}\}$
15
*16 Procedure COMPUTENTSCD($p$)          // Computation of NTSCD for
    predicate $p$
*17     $\{s_1, s_2\} \leftarrow Successors(p)$
*18     return $\{p \xrightarrow{\text{NTSCD}} n \mid n \in (V_{s_1} \smallsetminus V_{s_2}) \cup (V_{s_2} \smallsetminus V_{s_1})\}$
19
20  COMPUTE$V_p$S          // Computation of DOD and NTSCD for all nodes
21  $dod \leftarrow \emptyset$
*22 $ntscd \leftarrow \emptyset$
23  for $p \in Predicates(G)$ do
24      $dod \leftarrow dod \cup$ COMPUTEDOD$(p)$
*25     $ntscd \leftarrow ntscd \cup$ COMPUTENTSCD$(p)$
```

this path. When the graph A_p is constructed, we check whether it has the right shape. If not, we return $\emptyset$ as there are no nodes DOD on p in this case.

The next step is to compute the sets S_1 and S_2. Again, we apply a similar depth-first search as in the construction of A_p described above. If the sets S_1, S_2 are not disjoint, we return $\emptyset$ as there are no nodes DOD on p.

Then we unfold the cycle in A_p from an arbitrary node in S_1, compute the set U, and check whether the unfolding matches $(S_1.U^*)^+.(S_2.U^*)^+.(S_1.U^*)^*$. Note that any unfolding starting in S_1 matches this language iff the cycle has an unfolding of the form $S_1.U^*.(S_2.U^*)^*.S_2.U^*.(S_1.U^*)^*$ of Theorem 3. Hence, we return $\emptyset$ if the check fails.

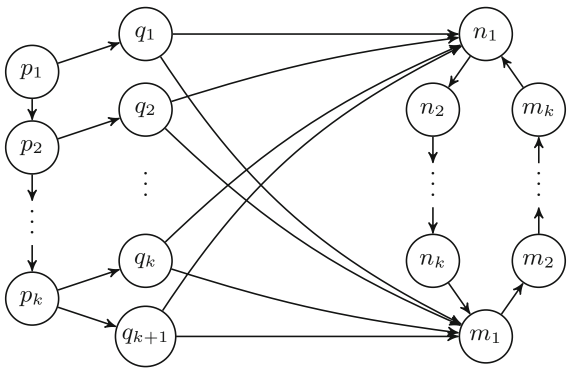


Fig. 7. A CFG with $|V|$ nodes that has the DOD relation of size $\Theta(|V|^3)$.

Finally, we extract the paths of the form $S_1.U^*.S_2$ and $S_2.U^*.S_1$ from the unfolding. Note that the last node of the latter path can be the first node of the unfolding. Finally, we compute the DOD dependencies according to Theorem 4.

The procedure COMPUTENTSCD(p) used for the computation of NTSCD simply follows Definition 2: it makes dependent on p each node that is on all maximal paths from the successor s_1 but not on all maximal paths from the successor s_2 or symmetrically for s_2 and s_1.

As the correctness of our algorithm comes directly from the observations made in the previous subsection, it remains only to analyze its complexity. The procedure COMPUTEV_pS consists of two cycles in sequence. The first cycle runs in $O(|V|)$. The second cycle calls $O(|V|)$-times the procedure COMPUTE(n). This procedure is essentially identical to the procedure of the same name in Algorithm 2 and so is its time complexity, namely $O(|V| + |E|)$. Note that sets can be represented by bitvectors and therefore adding an element and checking the presence of an element in a set are constant-time. Overall, the procedure COMPUTEV_pS runs in $O(|V| \cdot (|V| + |E|))$, which is $O(|V|^2)$ for CFGs.

Now we discuss the complexity of the procedure COMPUTEDOD(p). Creating the graph A_p requires calling depth-first search $O(|V|)$ times, which yields $O(|V| \cdot |E|)$ in total. Computation of S_1, S_2 requires another two calls of depth-first search, which is in $O(|E|)$. When sets are represented as bitvectors, checking that S_1 and S_2 are disjoint is in $O(|V|)$. Unfolding the cycle, matching the unfolding to the language (line 10), and the procedure EXTRACT run also in $O(|V|)$. The construction of the DOD relation on line 14 is in $O(|V|^2)$. Altogether, COMPUTEDOD(p) runs in $O(|V| \cdot |E| + |V|^2)$ which simplifies to $O(|V|^2)$ for CFGs.

COMPUTEDOD is called $O(|V|)$ times, so the overall complexity of computing DOD for a CFG $G = (V, E)$ is $O(|V|^3)$. If we compute also NTSCD, we make $O(|V|)$ extra calls to COMPUTENTSCD(p), where one call takes $O(|V|)$ time. Therefore, the asymptotic complexity of computing NTSCD with DOD does not change from computing DOD only.

Our algorithm running in time $O(|V|^3)$ is asymptotically optimal as there exist graphs with DOD relations of size $\Theta(|V|^3)$. For example, the CFG in Fig. 7 has $|V| = 4k + 1$ nodes and the corresponding DOD relation

$$\{q_i \xrightarrow{\text{DOD}} \{n_j, m_l\} \mid i \in \{1, \ldots, k+1\}, j, l \in \{1, \ldots, k\}\}$$

is of size $k^3 + k^2 \in \Theta(|V|^3)$.

5 Comparison to Control Closures

In 2011, Danicic et al. [13] introduced *control closures (CC)* that generalize control dependence from CFGs to arbitrary graphs. In particular, *strong control closure*, which is sensitive to non-termination, generalizes strong control dependence including NTSCD and DOD.

Definition 8 (Strongly control-closed set). *Let $G = (V, E)$ be a CFG and let $U \subseteq V$. The set U is* strongly control-closed[3] *in G if and only if for every node $v \in V \smallsetminus U$ that is reachable in G from a node in U, one of these holds:*

- *there is no node in U reachable from v or*
- *there exists a node $u \in U$ such that all maximal paths from v contain u and it is the first node from U on all these paths.*

In other words, whenever we leave a strongly control-closed set, we either cannot return back or we have to return back to the set in a certain node.

Definition 9 (Strong control closure, strong CC). *Let $G = (V, E)$ be a CFG and $V' \subseteq V$. A* strong control closure (strong CC) *of V' is a strongly control-closed set $U \supseteq V'$ such that there is no strongly control-closed set U' satisfying $U \supsetneq U' \supseteq V'$.*

Danicic et al. present an algorithm for the computation of strong control closures running in $O(|V|^4)$ [13, Theorem 66]. In fact, the algorithm uses a procedure Γ that is very similar to our procedure COMPUTE(n) of Algorithm 2.

We can also define the closure of a set of nodes under NTSCD and DOD.

Definition 10 (NTSCD and DOD closure). *Let $G = (V, E)$ be a CFG. A* NTSCD and DOD closure *of a set $V' \subseteq V$ is the smallest set $U \supseteq V'$ satisfying*

$$(n \in U \wedge p \xrightarrow{\text{NTSCD}} n) \implies p \in U \quad \textit{and} \quad (a, b \in U \wedge p \xrightarrow{\text{DOD}} \{a, b\}) \implies p \in U.$$

Definition 10 directly provides an algorithm computing the NTSCD and DOD closure of a given set $V' \subseteq V$. Roughly speaking, if we represent the NTSCD relation with edges and the DOD relation with hyperedges in a directed hypergraph with nodes V, the closure computation amounts to gathering backward reachable nodes from V'.

[3] We adjusted the definition to the fact that predicates in our CFGs always have two outgoing edges (i.e., they are *complete* in terms of Danicic et al. [13]). The original definition [13] works with CFGs where each predicate has at most two successors and considers also paths that may end in a predicate with less than two successors.

Danicic et al. [13, Lemmas 93 and 94] proved that for a CFG $G = (V, E)$ with a distinguished *start* node from which all nodes in V are reachable and a subset $U \subseteq V$ such that $start \in U$, the set U is strongly control-closed iff it is closed under NTSCD and DOD. Hence, on graphs with such a *start* node, the strong CC of a set V' containing the *start* node can be computed also by computing its NTSCD and DOD closure. Computation of the NTSCD and DOD closure runs in $O(|V|^3)$ as the backward reachability is dominated by the computation of NTSCD and DOD relations.

A substantial difference between the algorithm for strong CC by Danicic et al. [13] and our algorithm is that we are able to compute DOD and NTSCD separately, whereas the former is not. Moreover, our algorithm for NTSCD and DOD closure is asymptotically faster.

6 Experimental Evaluation

We implemented our algorithms for the computation of NTSCD, DOD, and the NTSCD and DOD closure in C++ on top of the LLVM [25] infrastructure. The implementation is a part of the library for program analysis and slicing called DG [6], which is used for example in the verification and test generation tool Symbiotic [7]. We also implemented the original Ranganath et al.'s algorithms for NTSCD and DOD, the fixed versions of these algorithms from Subsects. 3.1 and 4.1, and the algorithm for the computation of strong CC by Danicic et al.

In the implementation of the strong CC algorithm by Danicic et al. [13], we use our procedure COMPUTE(n) of Algorithm 2 to implement the function Γ. This should have only a positive effect as this procedure is more efficient than iterating over all edges in a copy of the graph and removing them [13].

In our experiments, we use CFGs of functions (where nodes of the CFG represent basic blocks of the function) obtained in the following way. We took all benchmarks from the *Competition on Software Verification (SV-COMP)* 2020.[4] These benchmarks contain many artificial or generated code, but also a lot of real-life code, e.g., from the Linux project. Each source code file was compiled with CLANG into LLVM and preprocessed by the `-lowerswitch` pass to ensure that every basic block has at most two successors. Then we extracted individual functions and removed those with less than 100 basic blocks, as the computation of control dependence runs swiftly on small graphs. Because it is possible that one function is present in multiple benchmarks, the next step was to remove these duplicate functions. For every function, we computed the number of nodes and edges in its CFG, and performed DFS on the CFG to obtain the number of tree, forward, cross and back edges, and the depth of the DFS tree. If two or more functions shared the name and all the computed numbers, we kept only one such function. Note that this process may have removed also a function that was not a duplicate of some other, but only with a low probability. At the end, we were left with 2440 functions. The biggest function has 27851 basic blocks. Table 2 shows the distribution of the sizes of the generated CFGs.

[4] https://github.com/sosy-lab/sv-benchmarks, tag `svcomp20`.

Table 2. The *numbers* of considered CFGs by their *sizes*. The size of a CFG is the number of its nodes, which is the number of basic blocks of the corresponding function.

size	number	size	number	size	number
100 – 199	1713	500 – 599	35	900 – 999	3
200 – 299	355	600 – 699	29	1000 – 1999	23
300 – 399	159	700 – 799	18	2000 – 9999	22
400 – 499	73	800 – 899	7	≥ 10000	3

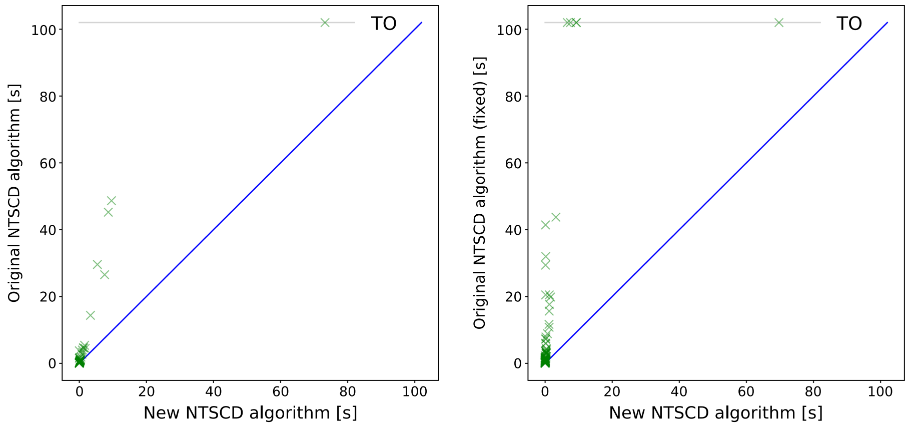


Fig. 8. Comparison of the running times of the new NTSCD algorithm and the incorrect (left) and the fixed (right) versions of the original NTSCD algorithm. TO stands for timeout.

The experiments were run on machines with *AMD EPYC* CPU with the frequency 3.1 GHz. Each benchmark run was constrained to 1 core and 8 GB of RAM. We used the tool *Benchexec* [4] to enforce resources isolation and to measure their usage. All presented times are CPU times. We set the timeout to 100 s for each algorithm run.

In the following, *original* algorithms refers to the algorithms of Ranganath et al. (we distinguish between the incorrect and the fixed versions when needed) and *new* algorithms refers to the algorithms introduced in this paper.

NTSCD Algorithms. In the first set of experiments, we compared the new algorithm for NTSCD against the incorrect and the fixed version of the original NTSCD algorithm. Although it seems that comparing to the incorrect version is meaningless, we did not want to compare only to the fixed version as the provided fix slows down the algorithm.

The results are depicted in Fig. 8. On the left scatter plot, there is the comparison of the new algorithm to the incorrect original algorithm and on the right scatter plot we compare to the fixed original algorithm. As we can see,

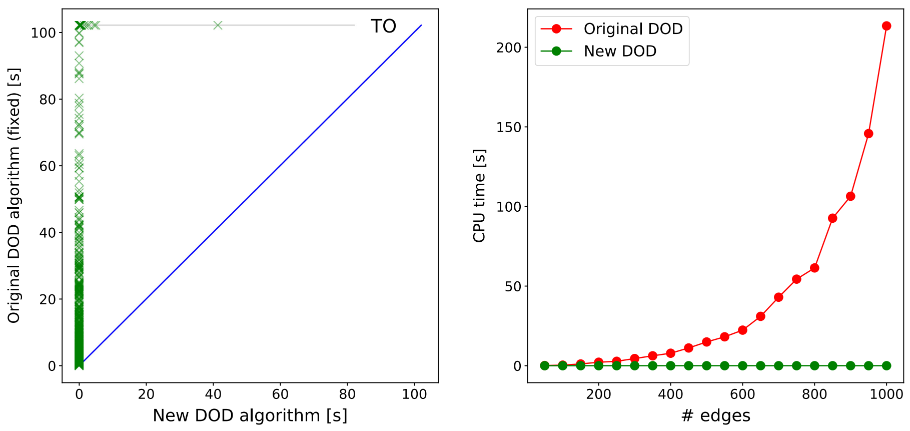


Fig. 9. Comparison of the running times of the new and the (fixed) original DOD algorithm. We use the considered benchmarks (left) and random graphs with 500 nodes and the number of edges specified by the x-axis (right).

the new algorithm outperforms the original algorithm significantly. The incorrect original algorithm produced a wrong NTSCD relation in 98.6 % of the considered benchmarks. The fixed version of the original algorithm returned precisely the same NTSCD relations as the new algorithm. We can also see that the scatter plot on the right contains more timeouts of the original algorithm. It supports the claim that the fix slows down the original algorithm.

DOD Algorithms. We compared the new DOD algorithm to the fixed version of the original DOD algorithm. As the fix does not change the asymptotic complexity of the original algorithm, we do not compare the new algorithm with the incorrect version of the original algorithm. The results of the experiments are displayed in Fig. 9 (left). We can see that the new algorithm is again very fast. In fact, the results resemble the results of the pure NTSCD algorithm, which is basically the part of the DOD algorithm that computes V_p sets. It benefits from early checks that detect predicate nodes with no DOD dependencies.

As mentioned in the introduction, DOD is empty for structured programs as their CFGs are reducible. We do not know precisely how many of the 2440 considered functions have irreducible CFGs, but we know that 2373 of them use **goto** statements. DOD relations for 12 functions was non-empty, which means that CFGs of these functions are irreducible. Note that there may have been other irreducible CFGs with empty DOD relation.

Additionally, we tested the DOD algorithms on randomly generated graphs, where we can expect that irreducible graphs emerge more often. Figure 9 (right) shows the results for graphs that have 500 nodes and 50, 100, 150, ... randomly distributed edges (such that every node has at most two successors). Each presented running time is in fact an average of 10 measurements with different random graphs. We can see that the new algorithm is agnostic to the number of

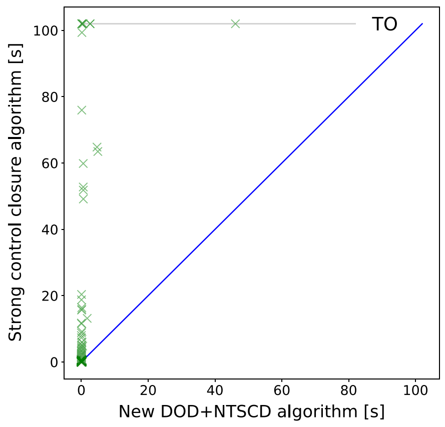


Fig. 10. Comparison of the running times of the strong CC algorithm by Danicic et al. [13] and our algorithm for the NTSCD and DOD closure.

edges. Its running time in this experiment ranges from $4.12 \cdot 10^{-3}$ to $8.89 \cdot 10^{-3}$ seconds. The original DOD algorithm does not scale well with the increasing number of edges.

Strong CC Algorithm. We also compare the strong CC algorithm of Danicic et al. [13] against our NTSCD and DOD closure algorithm on sets of nodes containing a distinguished *start* node, where these two algorithms produce equivalent results. For these experiments, we need a starting set that is going to be closed. We decided to run these experiments on the considered functions that have at least two exit points. The starting set consists of the node representing the entry point and the node representing one of the exit points. The closure of this set contains all nodes that may influence getting to the other exit points. The results are shown on the scatter plot in Fig. 10. Our algorithm clearly outperforms the strong CC algorithm.

7 Conclusion

We studied algorithms for the computation of strong control dependence, namely non-termination sensitive control dependence (NTSCD) and decisive order dependence (DOD) by Ranganath et al. [33] and strong control closures (strong CC) by Danicic et al. [13] on control flow graphs where each branching statement has two successors. We have demonstrated flaws in the original algorithms for computation of NTSCD and DOD and we have suggested corrections. Moreover, we have introduced new algorithms for NTSCD, DOD, and strong CC that are asymptotically faster. All the mentioned algorithms have been implemented and our experiments confirm dramatically better performance of the new algorithms.

References

1. Amtoft, T.: Slicing for modern program structures: a theory for eliminating irrelevant loops. Inf. Process. Lett. **106**(2), 45–51 (2008). https://doi.org/10.1016/j.ipl.2007.10.002
2. Androutsopoulos, K., Clark, D., Harman, M., Krinke, J., Tratt, L.: State-based model slicing: a survey. ACM Comput. Surv. **45**(4), 53:1-53:36 (2013). https://doi.org/10.1145/2501654.2501667
3. Androutsopoulos, K., Clark, D., Harman, M., Li, Z., Tratt, L.: Control dependence for extended finite state machines. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 216–230. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00593-0_15
4. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: requirements and solutions. Int. J. Softw. Tools Technol. Transf. **21**(1), 1–29 (2019). https://doi.org/10.1007/s10009-017-0469-y
5. Bilardi, G., Pingali, K.: A framework for generalized control dependence. In: PLDI 1996, pp. 291–300. ACM (1996). https://doi.org/10.1145/231379.231435
6. Chalupa, M.: DG: analysis and slicing of LLVM bitcode. In: Hung, D.V., Sokolsky, O. (eds.) ATVA 2020. LNCS, vol. 12302, pp. 557–563. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-59152-6_33
7. Chalupa, M., Jašek, T., Novák, J., Řechtáčková, A., Šoková, V., Strejček, J.: Symbiotic 8: beyond symbolic execution. In: TACAS 2021. LNCS, vol. 12652, pp. 453–457. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72013-1_31
8. Chalupa, M., Klaška, D., Strejček, J., Tomovič, L.: Fast computation of strong control dependencies. CoRR abs/2011.01564 (2020). https://arxiv.org/abs/2011.01564
9. Chalupa, M., Strejček, J.: Evaluation of program slicing in software verification. In: Ahrendt, W., Tapia Tarifa, S.L. (eds.) IFM 2019. LNCS, vol. 11918, pp. 101–119. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-34968-4_6
10. Chen, F., Roşu, G.: Parametric and termination-sensitive control dependence. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 387–404. Springer, Heidelberg (2006). https://doi.org/10.1007/11823230_25
11. Cooper, K., Harvey, T., Kennedy, K.: Iterative data- ow analysis, revisited (2002)
12. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. **13**(4), 451–490 (1991). https://doi.org/10.1145/115372.115320
13. Danicic, S., Barraclough, R.W., Harman, M., Howroyd, J., Kiss, Á., Laurence, M.R.: A unifying theory of control dependence and its application to arbitrary program structures. Theor. Comput. Sci. **412**(49), 6809–6842 (2011). https://doi.org/10.1016/j.tcs.2011.08.033
14. Darte, A., Silber, G.-A.: Temporary arrays for distribution of loops with control dependences. In: Bode, A., Ludwig, T., Karl, W., Wismüller, R. (eds.) Euro-Par 2000. LNCS, vol. 1900, pp. 357–367. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44520-X_47
15. Denning, D.E., Denning, P.J.: Certification of programs for secure information ow. Commun. ACM **20**(7), 504–513 (1977). https://doi.org/10.1145/359636.359712
16. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. **9**(3), 319–349 (1987). https://doi.org/10.1145/24039.24041

17. Gallagher, K., Binkley, D.: Program slicing. In: FoSM 2008, pp. 58–67 (2008). https://doi.org/10.1109/FOSM.2008.4659249
18. Giffhorn, D.: Slicing of concurrent programs and its application to information flow control. Ph.D. thesis, Karlsruhe Institute of Technology (2012). http://digbib.ubka.uni-karlsruhe.de/volltexte/1000028814
19. Hammer, C., Snelting, G.: Flow-sensitive, context-sensitive, and object-sensitive information ow control based on program dependence graphs. Int. J. Inf. Sec. **8**(6), 399–422 (2009). https://doi.org/10.1007/s10207-009-0086-1
20. Harrold, M.J., Rothermel, G., Sinha, S.: Computation of interprocedural control dependence. In: ISSTA 1998, pp. 11–20. ACM (1998). https://doi.org/10.1145/271771.271780
21. Hecht, M.S., Ullman, J.D.: Characterizations of reducible ow graphs. J. ACM **21**(3), 367–375 (1974). https://doi.org/10.1145/321832.321835
22. Horwitz, S., Reps, T.W., Binkley, D.W.: Interprocedural slicing using dependence graphs. ACM Trans. Program. Lang. Syst. **12**(1), 26–60 (1990). https://doi.org/10.1145/77606.77608
23. Khanfar, H., Lisper, B., Masud, A.N.: Static backward program slicing for safety-critical systems. In: de la Puente, J.A., Vardanega, T. (eds.) Ada-Europe 2015. LNCS, vol. 9111, pp. 50–65. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19584-1_4
24. Labbé, S., Gallois, J.: Slicing communicating automata specifications: polynomial algorithms for model reduction. Formal Aspects Comput. **20**(6), 563–595 (2008). https://doi.org/10.1007/s00165-008-0086-3
25. Lattner, C., Adve, V.S.: LLVM: a compilation framework for lifelong program analysis & transformation. In: CGO 2004, pp. 75–88. IEEE Computer Society (2004). https://doi.org/10.1109/CGO.2004.1281665
26. Léchenet, J., Kosmatov, N., Gall, P.L.: Cut branches before looking for bugs: certifiably sound verification on relaxed slices. Formal Asp. Comput. **30**(1), 107–131 (2018). https://doi.org/10.1007/s00165-017-0439-x
27. Loyall, J.P., Mathisen, S.A.: Using dependence analysis to support the software maintenance process. In: ICSM 1993, pp. 282–291. IEEE Computer Society (1993). https://doi.org/10.1109/ICSM.1993.366934
28. Lucia, A.D.: Program slicing: methods and applications. In: SCAM 2001, pp. 144–151. IEEE Computer Society (2001). https://doi.org/10.1109/SCAM.2001.972675
29. Metta, R., Becker, M., Bokil, P., Chakraborty, S., Venkatesh, R.: TIC: a scalable model checking based approach to WCET estimation. In: LCTES 2016, pp. 72–81. ACM (2016). https://doi.org/10.1145/2907950.2907961
30. Ottenstein, K.J., Ottenstein, L.M.: The program dependence graph in a software development environment. In: FSE 1984, pp. 177–184. ACM (1984). https://doi.org/10.1145/800020.808263
31. Podgurski, A., Clarke, L.A.: A formal model of program dependences and its implications for software testing, debugging, and maintenance. IEEE Trans. Software Eng. **16**(9), 965–979 (1990). https://doi.org/10.1109/32.58784
32. Ranganath, V.P., Amtoft, T., Banerjee, A., Dwyer, M.B., Hatcliff, J.: A new foundation for control-dependence and slicing for modern program structures. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 77–93. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31987-0_7
33. Ranganath, V.P., Amtoft, T., Banerjee, A., Hatcliff, J., Dwyer, M.B.: A new foundation for control dependence and slicing for modern program structures. ACM Trans. Program. Lang. Syst. **29**(5), 27 (2007). https://doi.org/10.1145/1275497.1275502

34. Sinha, S., Harrold, M.J., Rothermel, G.: Interprocedural control dependence. ACM Trans. Softw. Eng. Methodol. **10**(2), 209–254 (2001). https://doi.org/10.1145/367008.367022
35. Stanier, J., Watson, D.: A study of irreducibility in C programs. Softw. Pract. Exp. **42**(1), 117–130 (2012). https://doi.org/10.1002/spe.1059
36. Tšahhirov, I., Laud, P.: Application of dependency graphs to security protocol analysis. In: Barthe, G., Fournet, C. (eds.) TGC 2007. LNCS, vol. 4912, pp. 294–311. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78663-4_20
37. Weiser, M.: Program slicing. IEEE Trans. Softw. Eng. **10**(4), 352–357 (1984). https://doi.org/10.1109/TSE.1984.5010248

Diffy: Inductive Reasoning of Array Programs Using Difference Invariants

Supratik Chakraborty[1], Ashutosh Gupta[1], and Divyesh Unadkat[1,2](✉)

[1] Indian Institute of Technology Bombay, Mumbai, India
{supratik,akg}@cse.iitb.ac.in
[2] TCS Research, Pune, India
divyesh.unadkat@tcs.com

Abstract. We present a novel verification technique to prove properties of a class of array programs with a symbolic parameter N denoting the size of arrays. The technique relies on constructing two slightly different versions of the same program. It infers difference relations between the corresponding variables at key control points of the joint control-flow graph of the two program versions. The desired post-condition is then proved by inducting on the program parameter N, wherein the difference invariants are crucially used in the inductive step. This contrasts with classical techniques that rely on finding potentially complex loop invaraints for each loop in the program. Our synergistic combination of inductive reasoning and finding simple difference invariants helps prove properties of programs that cannot be proved even by the winner of Arrays sub-category in SV-COMP 2021. We have implemented a prototype tool called Diffy to demonstrate these ideas. We present results comparing the performance of Diffy with that of state-of-the-art tools.

1 Introduction

Software used in a wide range of applications use arrays to store and update data, often using loops to read and write arrays. Verifying correctness properties of such array programs is important, yet challenging. A variety of techniques have been proposed in the literature to address this problem, including inference of quantified loop invariants [20]. However, it is often difficult to automatically infer such invariants, especially when programs have loops that are sequentially composed and/or nested within each other, and have complex control flows. This has spurred recent interest in mathematical induction-based techniques for verifying parametric properties of array manipulating programs [11,12,42,44]. While induction-based techniques are efficient and quite powerful, their Achilles heel is the automation of the inductive argument. Indeed, this often becomes the limiting step in applications of induction-based techniques. Automating the induction step and expanding the class of array manipulating programs to which induction-based techniques can be applied forms the primary motivation for our work. Rather than being a stand-alone technique, we envisage our work being used as part of a portfolio of techniques in a modern program verification tool.

We propose a novel and practically efficient induction-based technique that advances the state-of-the-art in automating the inductive step when reasoning about array manipulating programs. This allows us to automatically verify interesting properties of a large class of array manipulating programs that are beyond the reach of state-of-the-art induction-based techniques, viz. [12,42]. The work that comes closest to us is VAJRA [12], which is part of the portfolio of techniques in VERIABS [1] – the winner of SV-COMP 2021 in the Arrays Reach sub-category. Our work addresses several key limitations of the technique implemented in VAJRA, thereby making it possible to analyze a much larger class of array manipulating programs than can be done by VERIABS. Significantly, this includes programs with nested loops that have hitherto been beyond the reach of automated techniques that use mathematical induction [12,42,44].

A key innovation in our approach is the construction of two slightly different versions of a given program that have identical control flow structures but slightly different data operations. We automatically identify simple relations, called *difference invariants*, between corresponding variables in the two versions of a program at key control flow points. Interestingly, these relations often turn out to be significantly simpler than inductive invariants required to prove the property directly. This is not entirely surprising, since the difference invariants depend less on what individual statements in the programs are doing, and more on the difference between what they are doing in the two versions of the program. We show how the two versions of a given program can be automatically constructed, and how differences in individual statements can be analyzed to infer simple difference invariants. Finally, we show how these difference invariants can be used to simplify the reasoning in the inductive step of our technique.

We consider programs with (possibly nested) loops manipulating arrays, where the size of each array is a symbolic integer parameter N (> 0)[1]. We verify (a sub-class of) quantified and quantifier-free properties that may depend on the symbolic parameter N. Like in [12], we view the verification problem as one of proving the validity of a parameterized Hoare triple $\{\varphi(N)\}$ P_N $\{\psi(N)\}$ for all values of N (> 0), where arrays are of size N in the program P_N, and N is a free variable in $\varphi(\cdot)$ and $\psi(\cdot)$.

To illustrate the kind of programs that are amenable to our technique, consider the program shown in Fig. 1(a), adapted from an SV-COMP benchmark. This program has a couple of sequentially composed loops that update arrays and scalars. The scalars `S` and `F` are initialized to 0 and 1 respectively before the first loop starts iterating. Subsequently, the first loop computes a recurrence in variable `S` and initializes elements of the array `B` to 1 if the corresponding elements of array `A` have non-negative values, and to 0 otherwise. The outermost branch condition in the body of the second loop evaluates to true only if the program parameter N and the variable `S` have same values. The value of `F` is reset based on some conditions depending on corresponding entries of arrays `A` and `B`. The pre-condition of this program is `true`; the post-condition asserts that `F` is never reset in the second loop.

[1] For a more general class of programs supported by our technique, please see [13].

```
// assume(true)
S = 0; F = 1;
for(i = 0; i< N; i++) {
 S = S + 1;
 if ( A[i] >= 0 ) B[i] = 1;
 else B[i] = 0;
}
for(j = 0; j< N; j++) {
 if(S == N) {
  if ( A[j] >= 0 && !B[j] ) F = 0;
  if ( A[j] < 0 && B[j] ) F = 0;
}
}
// assert(F == 1)
```

(a)

```
// assume(true)
S = 0;
for(i=0; i<N; i++) A[i] = 0;
for(j=0; j<N; j++) S = S + 1;
for(k=0; k<N; k++) {
 for(l=0; l<N; l++) A[l] = A[l] + 1;
 A[k] = A[k] + S;
}
// assert(forall x in [0,N), A[x]==2*N)
```

(b)

Fig. 1. Motivating examples

State-of-the-art techniques find it difficult to prove the assertion in this program. Specifically, Vajra [12] is unable to prove the property, since it cannot reason about the branch condition (in the second loop) whose value depends on the program parameter N. VeriAbs [1], which employs a sequence of techniques such as loop shrinking, loop pruning, and inductive reasoning using [12] is also unable to verify the assertion shown in this program. Indeed, the loops in this program cannot be merged as the final value of `S` computed by the first loop is required in the second loop; hence loop shrinking does not help. Also, loop pruning does not work due to the complex dependencies in the program and the fact that the exact value of the recurrence variable `S` is required to verify the program. Subsequent abstractions and techniques applied by VeriAbs from its portfolio are also unable to verify the given post-condition. VIAP [42] translates the program to a quantified first-order logic formula in the theory of equality and uninterpreted functions [32]. It applies a sequence of tactics to simplify and prove the generated formula. These tactics include computing closed forms of recurrences, induction over array indices and the like to prove the property. However, its sequence of tactics is unable to verify this example within our time limit of 1 min.

Benchmarks with nested loops are a long standing challenge for most verifiers. Consider the program shown in Fig. 1(b) with a nested loop in addition to sequentially composed loops. The first loop initializes entries in array `A` to 0. The second loop aggregates a constant value in the scalar `S`. The third loop is a nested loop that updates array `A` based on the value of `S`. The entries of `A` are updated in the inner as well as outer loop. The property asserts that on termination, each array element equals twice the value of the parameter N.

While the inductive reasoning of Vajra and the tactics in VIAP do not support nested loops, the sequence of techniques used by VeriAbs is also unable to prove the given post-condition in this program. In sharp contrast, our prototype tool Diffy is able to verify the assertions in both these programs automatically within a few seconds. This illustrates the power of the inductive technique proposed in this paper.

The technical contributions of the paper can be summarized as follows:

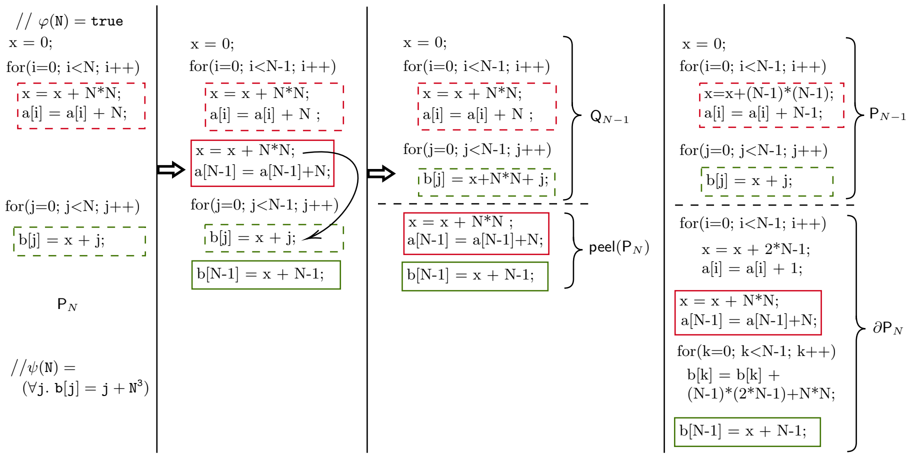
– We present a novel technique based on mathematical induction to prove interesting properties of a class of programs that manipulate arrays. The crucial inductive step in our technique uses difference invariants from two slightly different versions of the same program, and differs significantly from other induction-based techniques proposed in the literature [11,12,42,44].
– We describe algorithms to transform the input program for use in our inductive verification technique. We also present techniques to infer simple difference invariants from the two slightly different program versions, and to complete the inductive step using these difference invariants.
– We describe a prototype tool Diffy that implements our algorithms.
– We compare Diffy vis-a-vis state-of-the-art tools for verification of C programs that manipulate arrays on a large set of benchmarks. We demonstrate that Diffy significantly outperforms the winners of SV-COMP 2019, 2020 and 2021 in the Array Reach sub-category.

2 Overview and Relation to Earlier Work

In this section, we provide an overview of the main ideas underlying our technique. We also highlight how our technique differs from [12], which comes closest to our work. To keep the exposition simple, we consider the program P_N, shown in the first column of Fig. 2, where N is a symbolic parameter denoting the sizes of arrays `a` and `b`. We assume that we are given a parameterized pre-condition $\varphi(N)$, and our goal is to establish the parameterized post-condition $\psi(N)$, for all $N > 0$. In [12,44], techniques based on mathematical induction (on N) were proposed to solve this class of problems. As with any induction-based technique, these approaches consist of three steps. First, they check if the *base case* holds, i.e. if the Hoare triple $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$ holds for small values of N, say $1 \leq N \leq M$, for some $M > 0$. Next, they assume that the *inductive hypothesis* $\{\varphi(N-1)\}\ \mathsf{P}_{N-1}\ \{\psi(N-1)\}$ holds for some $N \geq M+1$. Finally, in the *inductive step*, they show that if the inductive hypothesis holds, so does $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$. It is not hard to see that the inductive step is the most crucial step in this style of reasoning. It is also often the limiting step, since not all programs and properties allow for efficient inferencing of $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$ from $\{\varphi(N-1)\}\ \mathsf{P}_{N-1}\ \{\psi(N-1)\}$.

Like in [12,44], our technique uses induction on N to prove the Hoare triple $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$ for all $N > 0$. Hence, our base case and inductive hypothesis are the same as those in [12,44]. However, our reasoning in the crucial inductive step is significantly different from that in [12,44], and this is where our primary contribution lies. As we show later, not only does this allow a much larger class of programs to be efficiently verified compared to [12,44], it also permits reasoning about classes of programs with nested loops, that are beyond the reach of [12,44]. Since the work of [12] significantly generalizes that of [44], henceforth, we only refer to [12] when talking of earlier work that uses induction on N.

In order to better understand our contribution and its difference vis-a-vis the work of [12], a quick recap of the inductive step used in [12] is essential. The



```
// φ(N) = true
x = 0;
for(i=0; i<N; i++)
  x = x + N*N;
  a[i] = a[i] + N;

for(j=0; j<N; j++)
  b[j] = x + j;

//ψ(N) =
  (∀j. b[j] = j + N^3)
```

```
x = 0;
for(i=0; i<N-1; i++)
  x = x + N*N;
  a[i] = a[i] + N ;
x = x + N*N;
a[N-1] = a[N-1]+N;
for(j=0; j<N-1; j++)
  b[j] = x + j;
b[N-1] = x + N-1;
```

```
x = 0;
for(i=0; i<N-1; i++)
  x = x + N*N;
  a[i] = a[i] + N ;
for(j=0; j<N-1; j++)
  b[j] = x+N*N+ j;
- - - - - - - - - - - -
x = x + N*N ;
a[N-1] = a[N-1]+N;
b[N-1] = x + N-1;
```

```
x = 0;
for(i=0; i<N-1; i++)
  x=x+(N-1)*(N-1);
  a[i] = a[i] + N-1;
for(j=0; j<N-1; j++)
  b[j] = x + j;
- - - - - - - - - - - -
for(i=0; i<N-1; i++)
  x = x + 2*N-1;
  a[i] = a[i] + 1;
x = x + N*N;
a[N-1] = a[N-1]+N;
for(k=0; k<N-1; k++)
  b[k] = b[k] +
  (N-1)*(2*N-1)+N*N;
b[N-1] = x + N-1;
```

Fig. 2. Pictorial depiction of our program transformations

inductive step in [12] crucially relies on finding a "difference program" $\partial \mathsf{P}_N$ and a "difference pre-condition" $\partial\varphi(N)$ such that: (i) P_N is semantically equivalent to $\mathsf{P}_{N-1}; \partial\mathsf{P}_N$, where ';' denotes sequential composition of programs[2], (ii) $\varphi(N) \Rightarrow \varphi(N-1) \wedge \partial\varphi(N)$, and (iii) no variable/array element in $\partial\varphi(N)$ is modified by P_{N-1}. As shown in [12], once $\partial\mathsf{P}_N$ and $\partial\varphi(N)$ satisfying these conditions are obtained, the problem of proving $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$ can be reduced to that of proving $\{\psi(N-1) \wedge \partial\varphi(N)\}\ \partial\mathsf{P}_N\ \{\psi(N)\}$. This approach can be very effective if (i) $\partial\mathsf{P}_N$ is "simpler" (e.g. has fewer loops or strictly less deeply nested loops) than P_N and can be computed efficiently, and (ii) a formula $\partial\varphi(N)$ satisfying the conditions mentioned above exists and can be computed efficiently.

The requirement of P_N being semantically equivalent to $\mathsf{P}_{N-1}; \partial\mathsf{P}_N$ is a very stringent one, and finding such a program $\partial\mathsf{P}_N$ is non-trivial in general. In fact, the authors of [12] simply provide a set of syntax-guided conditionally sound heuristics for computing $\partial\mathsf{P}_N$. Unfortunately, when these conditions are violated (we have found many simple programs where they are violated), there are no known algorithmic techniques to generate $\partial\mathsf{P}_N$ in a sound manner. Even if a program $\partial\mathsf{P}_N$ were to be found in an ad-hoc manner, it may be as "complex" as P_N itself. This makes the approach of [12] ineffective for analyzing such programs. As an example, the fourth column of Fig. 2 shows P_{N-1} followed by one possible $\partial\mathsf{P}_N$ that ensures P_N (shown in the first column of the same figure) is semantically equivalent to $\mathsf{P}_{N-1}; \partial\mathsf{P}_N$. Notice that $\partial\mathsf{P}_N$ in this example has two sequentially composed loops, just like P_N had. In addition, the assignment statement in the body of the second loop uses a more complex expression than that present in the corresponding loop of P_N. Proving $\{\psi(N-1) \wedge \partial\varphi(N)\}\ \partial\mathsf{P}_N\ \{\psi(N)\}$

[2] Although the authors of [12] mention that it suffices to find a $\partial\mathsf{P}_N$ that satisfies $\{\varphi(N)\}\ \mathsf{P}_{N-1}; \partial\mathsf{P}_N\ \{\psi(N)\}$, they do not discuss any technique that takes $\varphi(N)$ or $\psi(N)$ into account when generating $\partial\mathsf{P}_N$.

may therefore not be any simpler (perhaps even more difficult) than proving $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$.

In addition to the difficulty of computing $\partial\mathsf{P}_N$, it may be impossible to find a formula $\partial\varphi(N)$ such that $\varphi(N) \Rightarrow \varphi(N-1) \wedge \partial\varphi(N)$, as required by [12]. This can happen even for fairly routine pre-conditions, such as $\varphi(N) \equiv \big(\bigwedge_{i=0}^{N-1} A[i] = N\big)$. Notice that there is no $\partial\varphi(N)$ that satisfies $\varphi(N) \Rightarrow \varphi(N-1) \wedge \partial\varphi(N)$ in this case. In such cases, the technique of [12] cannot be used at all, even if P_N, $\varphi(N)$ and $\psi(N)$ are such that there exists a trivial proof of $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$.

The inductive step proposed in this paper largely mitigates the above problems, thereby making it possible to efficiently reason about a much larger class of programs than that possible using the technique of [12]. Our inductive step proceeds as follows. Given P_N, we first algorithmically construct two programs Q_{N-1} and $\mathsf{peel}(\mathsf{P}_N)$, such that P_N is semantically equivalent to $\mathsf{Q}_{N-1}; \mathsf{peel}(\mathsf{P}_N)$. Intuitively, Q_{N-1} is the same as P_N, but with all loop bounds that depend on N now modified to depend on $N-1$ instead. Note that this is different from P_{N-1}, which is obtained by replacing *all uses* (not just in loop bounds) of N in P_N by $N-1$. As we will see, this simple difference makes the generation of $\mathsf{peel}(\mathsf{P}_N)$ significantly simpler than generation of $\partial\mathsf{P}_N$, as in [12]. While generating Q_{N-1} and $\mathsf{peel}(P_N)$ may sound similar to generating P_{N-1} and $\partial\mathsf{P}_N$ [12], there are fundamental differences between the two approaches. First, as noted above, P_{N-1} is semantically different from Q_{N-1}. Similarly, $\mathsf{peel}(\mathsf{P}_N)$ is also semantically different from $\partial\mathsf{P}_N$. Second, we provide an algorithm for generating Q_{N-1} and $\mathsf{peel}(\mathsf{P}_N)$ that works for a significantly larger class of programs than that for which the technique of [12] works. Specifically, our algorithm works for all programs amenable to the technique of [12], and also for programs that violate the restrictions imposed by the grammar and conditional heuristics in [12]. For example, we can algorithmically generate Q_{N-1} and $\mathsf{peel}(\mathsf{P}_N)$ even for a class of programs with arbitrarily nested loops – a program feature explicitly disallowed by the grammar in [12]. Third, we guarantee that $\mathsf{peel}(\mathsf{P}_N)$ is "simpler" than P_N in the sense that the maximum nesting depth of loops in $\mathsf{peel}(\mathsf{P}_N)$ is *strictly less* than that in P_N. Thus, if P_N has no nested loops (all programs amenable to analysis by [12] belong to this class), $\mathsf{peel}(\mathsf{P}_N)$ is guaranteed to be loop-free. As demonstrated by the fourth column of Fig. 2, no such guarantees can be given for $\partial\mathsf{P}_N$ generated by the technique of [12]. This is a significant difference, since it greatly simplifies the analysis of $\mathsf{peel}(\mathsf{P}_N)$ vis-a-vis that of $\partial\mathsf{P}_N$.

We had mentioned earlier that some pre-conditions $\varphi(N)$ do not admit any $\partial\varphi(N)$ such that $\varphi(N) \Rightarrow \varphi(N-1) \wedge \partial\varphi(N)$. It is, however, often easy to compute formulas $\varphi'(N-1)$ and $\Delta\varphi'(N)$ in such cases such that $\varphi(N) \Rightarrow \varphi'(N-1) \wedge \Delta\varphi'(N)$, and the variables/array elements in $\Delta\varphi'(N)$ are not modified by either P_{N-1} or Q_{N-1}. For example, if we were to consider a (new) pre-condition $\varphi(N) \equiv \big(\bigwedge_{i=0}^{N-1} A[i] = N\big)$ for the program P_N shown in the first column of Fig. 2, then we have $\varphi'(N-1) \equiv \big(\bigwedge_{i=0}^{N-2} A[i] = N\big)$ and $\Delta\varphi'(N) \equiv \big(A[N-1] = N\big)$. We assume the availability of such a $\varphi'(N-1)$ and $\Delta\varphi'(N)$ for the given $\varphi(N)$. This significantly relaxes the requirement on pre-conditions and allows a much larger class of Hoare triples to be proved using our technique vis-a-vis that of [12].

The third column of Fig. 2 shows Q_{N-1} and $\mathsf{peel}(\mathsf{P}_N)$ generated by our algorithm for the program P_N in the first column of the figure. It is illustrative to compare these with P_{N-1} and $\partial\mathsf{P}_N$ shown in the fourth column of Fig. 2. Notice that Q_{N-1} has the same control flow structure as P_{N-1}, but is not semantically equivalent to P_{N-1}. In fact, Q_{N-1} and P_{N-1} may be viewed as closely related versions of the same program. Let V_{Q} and V_{P} denote the set of variables of Q_{N-1} and P_{N-1} respectively. We assume V_{Q} is disjoint from V_{P}, and analyze the joint execution of Q_{N-1} starting from a state satisfying the pre-condition $\varphi'(N-1)$, and P_{N-1} starting from a state satisfying $\varphi(N-1)$. The purpose of this analysis is to compute a difference predicate $D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1)$ that relates corresponding variables in Q_{N-1} and P_{N-1} at the end of their joint execution. The above problem is reminiscent of (yet, different from) translation validation [4,17,24,40,46,48,49], and indeed, our calculation of $D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1)$ is motivated by techniques from the translation validation literature. An important finding of our study is that corresponding variables in Q_{N-1} and P_{N-1} are often related by simple expressions on N, regardless of the complexity of P_N, $\varphi(N)$ or $\psi(N)$. Indeed, in all our experiments, we didn't need to go beyond quadratic expressions on N to compute $D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1)$.

Once the steps described above are completed, we have $\Delta\varphi'(N)$, $\mathsf{peel}(\mathsf{P}_N)$ and $D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1)$. It can now be shown that if the inductive hypothesis, i.e. $\{\varphi(N-1)\}\ \mathsf{P}_{N-1}\ \{\psi(N-1)\}$ holds, then proving $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$ reduces to proving $\{\Delta\varphi'(N) \wedge \psi'(N-1)\}\ \mathsf{peel}(\mathsf{P}_N)\ \{\psi(N)\}$, where $\psi'(N-1) \equiv \exists V_{\mathsf{P}}\big(\psi(N-1) \wedge D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1)\big)$. A few points are worth emphasizing here. First, if $D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1)$ is obtained as a set of equalities, the existential quantifier in the formula $\psi'(N-1)$ can often be eliminated simply by substitution. We can also use quantifier elimination capabilities of modern SMT solvers, viz. Z3 [39], to eliminate the quantifier, if needed. Second, recall that unlike $\partial\mathsf{P}_N$ generated by the technique of [12], $\mathsf{peel}(\mathsf{P}_N)$ is guaranteed to be "simpler" than P_N, and is indeed loop-free if P_N has no nested loops. Therefore, proving $\{\Delta\varphi'(N) \wedge \psi'(N-1)\}\ \mathsf{peel}(\mathsf{P}_N)\ \{\psi(N)\}$ is typically significantly simpler than proving $\{\psi(N-1) \wedge \partial\varphi(N)\}\ \partial\mathsf{P}_N\ \{\psi(N)\}$. Finally, it may happen that the pre-condition in $\{\Delta\varphi'(N) \wedge \psi'(N-1)\}\ \mathsf{peel}(\mathsf{P}_N)\ \{\psi(N)\}$ is not strong enough to yield a proof of the Hoare triple. In such cases, we need to strengthen the existing pre-condition by a formula, say $\xi'(N-1)$, such that the strengthened pre-condition implies the weakest pre-condition of $\psi(N)$ under $\mathsf{peel}(\mathsf{P}_N)$. Having a simple structure for $\mathsf{peel}(\mathsf{P}_N)$ (e.g., loop-free for the entire class of programs for which [12] works) makes it significantly easier to compute the weakest pre-condition. Note that $\xi'(N-1)$ is defined over the variables in V_{Q}. In order to ensure that the inductive proof goes through, we need to strengthen the post-condition of the original program by $\xi(N)$ such that $\xi(N-1) \wedge D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1) \Rightarrow \xi'(N-1)$. Computing $\xi(N-1)$ requires a special form of logical abduction that ensures that $\xi(N-1)$ refers only to variables in V_P. However, if $D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1)$ is given as a set of equalities (as is often the case), $\xi(N-1)$ can be computed from $\xi'(N-1)$ simply by substitution. This process of strengthening the pre-condition and post-condition may need to iterate a few times until a fixed point is reached, similar to what happens in the inductive step of [12]. Note that the fixed point iterations may not always converge (verification is undecidable in general). However, in our experiments, convergence always happened within a few iterations. If $\xi'(N-1)$ denotes the formula obtained on reaching the fixed point, the final Hoare triple to be proved is $\{\xi'(N-1) \wedge \Delta\varphi'(N) \wedge \psi'(N-1)\}$ $\mathsf{peel}(\mathsf{P}_N)$ $\{\xi(N) \wedge \psi(N)\}$, where $\psi'(N-1) \equiv \exists V_{\mathsf{P}}\big(\psi(N-1) \wedge D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1)\big)$. Having a simple (often loop-free) $\mathsf{peel}(\mathsf{P}_N)$ significantly simplifies the above process.

We conclude this section by giving an overview of how Q_{N-1} and $\mathsf{peel}(\mathsf{P}_N)$ are computed for the program P_N shown in the first column of Fig. 2. The second column of this figure shows the program obtained from P_N by peeling the last iteration of each loop of the program. Clearly, the programs in the first and second columns are semantically equivalent. Since there are no nested loops in P_N, the peels (shown in solid boxes) in the second column are loop-free program fragments. For each such peel, we identify variables/array elements modified in the peel and used in subsequent non-peeled parts of the program. For example, the variable `x` is modified in the peel of the first loop and used in the body of the second loop, as shown by the arrow in the second column of Fig. 2. We replace all such uses (if needed, transitively) by expressions on the right-hand side of assignments in the peel until no variable/array element modified in the peel is used in any subsequent non-peeled part of the program. Thus, the use of `x` in the body of the second loop is replaced by the expression `x + N * N` in the third column of Fig. 2. The peeled iteration of the first loop can now be moved to the end of the program, since the variables modified in this peel are no longer used in any subsequent non-peeled part of the program. Repeating the above steps for the peeled iteration of the second loop, we get the program shown in the third column of Fig. 2. This effectively gives a transformed program that can be divided into two parts: (i) a program Q_{N-1} that differs from P_N only in that all loops are truncated to iterate $N-1$ (instead of N) times, and (ii) a program $\mathsf{peel}(\mathsf{P}_N)$ that is obtained by concatenating the peels of loops in P_N in the same order in which the loops appeared in P_N. It is not hard to see that P_N, shown in the first column of Fig. 2, is semantically equivalent to $\mathsf{Q}_{N-1}; \mathsf{peel}(\mathsf{P}_N)$. Notice that the construction of Q_{N-1} and $\mathsf{peel}(\mathsf{P}_N)$ was fairly straightforward, and did not require any complex reasoning. In sharp contrast, construction of $\partial\mathsf{P}_N$, as shown in the bottom half of fourth column of Fig. 2, requires non-trivial reasoning, and produces a program with two sequentially composed loops.

3 Preliminaries and Notation

We consider programs generated by the grammar shown below:

$$
\begin{aligned}
\mathsf{PB} &::= \mathsf{St}\\
\mathsf{St} &::= \mathsf{St}\ ;\ \mathsf{St} \mid\ v := \mathsf{E} \mid\ A[\mathsf{E}] := \mathsf{E} \mid\ \mathbf{if}(\mathsf{BoolE})\ \mathbf{then}\ \mathsf{St}\ \mathbf{else}\ \mathsf{St} \mid\\
&\qquad \mathbf{for}\ (\ell := 0;\ \ell < \mathsf{UB};\ \ell := \ell{+}1)\ \ \{\mathsf{St}\}\\
\mathsf{E} &::= \mathsf{E}\ \mathsf{op}\ \ \mathsf{E} \mid\ A[\mathsf{E}]\ \mid\ v\ \mid\ \ell\ \mid\ \mathsf{c}\ \mid\ N\\
\mathsf{op} &::= +\ \mid\ \text{-}\ \mid\ *\ \mid\ /\\
\mathsf{UB} &::= \mathsf{UB}\ \mathsf{op}\ \ \mathsf{UB} \mid\ \ell\ \mid\ \mathsf{c}\ \mid\ N\\
\mathsf{BoolE} &::= \mathsf{E}\ \mathsf{relop}\ \mathsf{E} \mid\ \ \mathsf{BoolE\ AND\ BoolE}\ \mid\ \ \mathsf{NOT\ BoolE}\ \mid\ \ \mathsf{BoolE\ OR\ BoolE}
\end{aligned}
$$

Formally, we consider a program P_N to be a tuple $(\mathcal{V}, \mathcal{L}, \mathcal{A}, \mathsf{PB}, N)$, where $\mathcal{V}$ is a set of scalar variables, $\mathcal{L} \subseteq \mathcal{V}$ is a set of scalar loop counter variables, $\mathcal{A}$ is a set of array variables, PB is the program body, and N is a special symbol denoting a positive integer parameter of the program. In the grammar shown above, we assume that $A \in \mathcal{A}$, $v \in \mathcal{V} \setminus \mathcal{L}$, $\ell \in \mathcal{L}$ and $\mathsf{c} \in \mathbb{Z}$. We also assume that each loop L has a unique loop counter variable ℓ that is initialized at the beginning of L and is incremented by 1 at the end of each iteration. We assume that the assignments in the body of L do not update ℓ. For each loop L with termination condition $\ell < \mathsf{UB}$, we require that UB is an expression in terms of N, variables in $\mathcal{L}$ representing loop counters of loops that nest L, and constants as shown in the grammar. Our grammar allows a large class of programs (with nested loops) to be analyzed using our technique, and that are beyond the reach of state-of-the-art tools like [1,12,42].

We verify Hoare triples of the form $\{\varphi(N)\}$ P_N $\{\psi(N)\}$, where the formulas $\varphi(N)$ and $\psi(N)$ are either universally quantified formulas of the form $\forall I\ (\alpha(I, N) \Rightarrow \beta(\mathcal{A}, \mathcal{V}, I, N))$ or quantifier-free formulas of the form $\eta(\mathcal{A}, \mathcal{V}, N)$. In these formulas, I is a sequence of array index variables, α is a quantifier-free formula in the theory of arithmetic over integers, and β and η are quantifier-free formulas in the combined theory of arrays and arithmetic over integers.

For technical reasons, we rename all scalar and array variables in the program in a pre-processing step as follows. We rename each scalar variable using the well-known Static Single Assignment (SSA) [43] technique, such that the variable is written at (at most) one location in the program. We also rename arrays in the program such that each loop updates its own version of an array and multiple writes to an array element within the same loop are performed on different versions of that array. We use techniques for array SSA [30] renaming studied earlier in the context of compilers, for this purpose. In the subsequent exposition, we assume that scalar and array variables in the program are already SSA renamed, and that all array and scalar variables referred to in the pre- and post-conditions are also expressed in terms of SSA renamed arrays and scalars.

4 Verification Using Difference Invariants

The key steps in the application of our technique, as discussed in Sect. 2, are

A1: Generation of Q_{N-1} and $\mathsf{peel}(\mathsf{P}_N)$ from a given P_N.
A2: Generation of $\varphi'(N-1)$ and $\Delta\varphi'(N)$ from a given $\varphi(N)$.
A3: Generation of the difference invariant $D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1)$, given $\varphi(N-1)$, $\varphi'(N-1)$, Q_{N-1} and P_{N-1}.
A4: Proving $\{\Delta\varphi'(N) \wedge \exists V_{\mathsf{P}}\big(\psi(N-1) \wedge D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1)\big)\}$ $\mathsf{peel}(\mathsf{P}_N)$ $\{\psi(N)\}$, possibly by generation of $\xi'(N-1)$ and $\xi(N)$ to strengthen the pre- and post-conditions, respectively.

We now discuss techniques for solving each of these sub-problems.

4.1 Generating Q_{N-1} and $\mathsf{peel}(\mathsf{P}_N)$

The procedure illustrated in Fig. 2 (going from the first column to the third column) is fairly straightforward if none of the loops have any nested loops within them. It is easy to extend this to arbitrary sequential compositions of non-nested loops. Having all variables and arrays in SSA-renamed forms makes it particularly easy to carry out the substitution exemplified by the arrow shown in the second column of Fig. 2. Hence, we don't discuss any further the generation of Q_{N-1} and $\mathsf{peel}(\mathsf{P}_N)$ when all loops are non-nested.

The case of nested loops is, however, challenging and requires additional discussion. Before we present an algorithm for handling this case, we discuss the intuition using an abstract example. Consider a pair of nested loops, L_1 and L_2, as shown in Fig. 3. Suppose that `B1` and `B3` are loop-free code fragments in the body of L_1 that precede and succeed the nested loop L_2. Suppose further that the loop body, `B2`, of L_2 is loop-free. To focus on the key aspects of computing peels of nested loops, we make two simplifying assumptions: (i) no scalar variable or array element modified in `B2` is used subsequently (including transitively) in either `B3` or `B1`, and (ii) every scalar variable or array element that is modified in `B1` and used subsequently in `B2`, is not modified again in either `B1`, `B2` or `B3`. Note that these assumptions are made primarily to simplify the exposition. For a detailed discussion on how our technique can be used even with some relaxations of these assumptions, the reader is referred to [13]. The peel of the abstract loops L_1 and L_2 is as shown in Fig. 4. The first loop in the peel includes the last iteration of L_2 in each of the $N-1$ iterations of L_1, that was missed in Q_{N-1}. The subsequent code includes the last iteration of L_1 that was missed in Q_{N-1}.

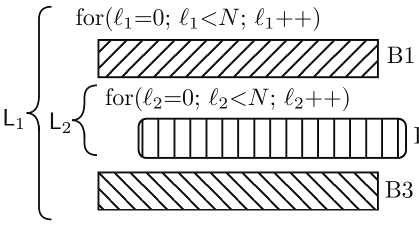

for(ℓ_1=0; ℓ_1<N; ℓ_1++)
B1
L_1
L_2
for(ℓ_2=0; ℓ_2<N; ℓ_2++)
B2
B3

Fig. 3. A generic nested loop

Formally, we use the notation `L`$_1$`(N)` to denote a loop L_1 that has no nested loops within it, and its loop counter, say ℓ_1, increases from 0 to an upper bound that is given by an expression in N. Similarly, we use `L`$_1$`(N, L`$_2$`(N))` to denote a loop L_1 that has another loop L_2 nested within it. The loop counter ℓ_1 of L_1 increases from 0 to an upper bound expression in N, while the loop counter ℓ_2 of L_2 increases from 0 to an upper bound expression in ℓ_1 and N. Using this notation, `L`$_1$`(N, L`$_2$`(N, L`$_3$`(N)))` represents three nested loops, and so on. Notice that the upper bound expression for a nested loop can depend not only on N but also on the loop counters of other loops nesting it. For notational clarity, we also use `LPeel(L`$_i$`, a, b)` to denote the peel of loop L_i

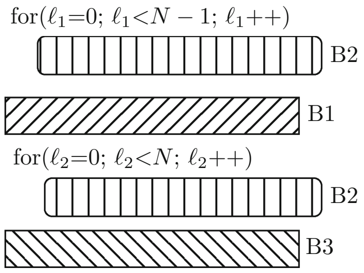

for(ℓ_1=0; ℓ_1<$N-1$; ℓ_1++)
B2
B1
for(ℓ_2=0; ℓ_2<N; ℓ_2++)
B2
B3

Fig. 4. Peel of the nested loop

consisting of all iterations of L_i where the value of ℓ_i ranges from `a` to `b-1`, both inclusive. Note that if `b-a` is a constant, this corresponds to the concatenation of `(b-a)` peels of L_i.

We will now try to see how we can implement the transformation from the first column to the second column of Fig. 2 for a nested loop L_1`(N,` L_2`(N))`. The first step is to truncate all loops to use $N-1$ instead of N in the upper bound expressions. Using the notation introduced above, this gives the loop L_1`(N-1,` L_2`(N-1))`. Note that all uses of N other than in loop upper bound expressions stay unchanged as we go from L_1`(N,` L_2`(N))` to L_1`(N-1,` L_2`(N-1))`. We now ask: *Which are the loop iterations of* L_1`(N,` L_2`(N))` *that have been missed (or skipped) in going to* L_1`(N-1,` L_2`(N-1))`*?* Let the upper bound expression of L_1 in L_1`(N,` L_2`(N))` be $U_{\mathsf{L}_1}(N)$, and that of L_2 be $U_{\mathsf{L}_2}(\ell_1, N)$. It is not hard to see that in every iteration ℓ_1 of L_1, where $0 \le \ell_1 < U_{\mathsf{L}_1}(N-1)$, the iterations corresponding to $\ell_2 \in \{U_{\mathsf{L}_2}(\ell_1, N-1), \ldots, U_{\mathsf{L}_2}(\ell_1, N)-1\}$ have been missed. In addition, all iterations of L_1 corresponding to $\ell_1 \in \{U_{\mathsf{L}_1}(N-1), \ldots, U_{\mathsf{L}_1}(N)-1\}$ have also been missed. This implies that the "peel" of L_1`(N,` L_2`(N))` must include all the above missed iterations. This peel therefore is the program fragment shown in Fig. 5.

```
for(ℓ₁=0; ℓ₁<U_L₁(N-1); ℓ₁++)
  LPeel(L₂, U_L₂(ℓ₁,N-1), U_L₂(ℓ₁,N))
LPeel(L₁, U_L₁(N-1), U_L₁(N))
```

Fig. 5. Peel of L_1`(N,` L_2`(N))`

Notice that if U_{L_2} `(`ℓ_1 `,N) -` U_{L_2} `(`ℓ_1 `,N-1)` is a constant (as is the case if U_{L_2}`(`ℓ_1`,N)` is any linear function of ℓ_1 and N), then the peel does not have any loop with nesting depth 2. Hence, the maximum nesting depth of loops in the peel is strictly less than that in L_1`(N,` L_2`(N))`, yielding a peel that is "simpler" than the original program. This argument can be easily generalized to loops with arbitrarily large nesting depths. The peel of L_1`(N,` L_2`(N,` L_3`(N)))` is as shown in Fig. 6.

```
for(ℓ₁=0; ℓ₁<U_L₁(N-1); ℓ₁++) {
  for(ℓ₂=0; ℓ₂<U_L₂(ℓ₁,N-1); ℓ₂++)
    LPeel(L₃, U_L₃(ℓ₁,ℓ₂,N-1), U_L₃(ℓ₁,ℓ₂,N))
  LPeel(L₂, U_L₂(ℓ₁,N-1), U_L₂(ℓ₁,N))
}
LPeel(L₁, U_L₁(N-1), U_L₁(N))
```

Fig. 6. Peel of L_1`(N,` L_2`(N,` L_3`(N)))`

As an illustrative example, let us consider the program in Fig. 7(a), and suppose we wish to compute the peel of this program containing nested loops. In this case, the upper bounds of the loops are $U_{\mathsf{L}_1}(N) = U_{\mathsf{L}_2}(N) = N$. The peel is shown in Fig. 7(b) and consists of two sequentially composed non-nested loops. The first loop takes into account the missed iterations of the inner loop (a single iteration in this example) that are executed in P_N but are missed in Q_{N-1}. The

```
for(i=0; i<N; i++)     for(i=0; i<N-1; i++)
  for(j=0; j<N; j++)     A[i][N-1] = N;
    A[i][j] = N;       for(j=0; j<N; j++)
                         A[N-1][j] = N;
        (a)                    (b)
```

Fig. 7. (a) Nested Loop & (b) Peel

Algorithm 1. GenQandPeel(P_N: program)

```
1: Let sequentially composed loops in $\mathsf{P}_N$ be in the order $\mathsf{L}_1$, $\mathsf{L}_2$, ..., $\mathsf{L}_m$;
2: for each loop $\mathsf{L}_i \in$ TopLevelLoops($\mathsf{P}_N$) do
3:     $\langle \mathsf{Q}_{\mathsf{L}_i}, \mathsf{R}_{\mathsf{L}_i} \rangle \leftarrow$ GenQandPeelForLoop($\mathsf{L}_i$);
4: while $\exists v.use(v) \in \mathsf{Q}_{\mathsf{L}_i} \wedge def(v) \in \mathsf{R}_{\mathsf{L}_j}$, for some $1 \le j < i \le N$ do    ▷ $v$ is var/array element
5:     Substitute rhs expression for $v$ from $\mathsf{R}_{\mathsf{L}_j}$ in $\mathsf{Q}_{\mathsf{L}_i}$;    ▷ If $\mathsf{R}_{\mathsf{L}_j}$ is a loop, abort
6: $\mathsf{Q}_{N-1} \leftarrow \mathsf{Q}_{\mathsf{L}_1}; \mathsf{Q}_{\mathsf{L}_2}; \ldots; \mathsf{Q}_{\mathsf{L}_m}$;
7: $\mathsf{peel}(\mathsf{P}_N) \leftarrow \mathsf{R}_{\mathsf{L}_1}; \mathsf{R}_{\mathsf{L}_2}; \ldots; \mathsf{R}_{\mathsf{L}_m}$;
8: return $\langle \mathsf{Q}_{N-1}, \mathsf{peel}(\mathsf{P}_N) \rangle$;

9: procedure GenQandPeelForLoop(L: loop)
10:     Let $U_{\mathsf{L}}(N)$ be the UB expression of loop L;
11:     $\mathsf{Q}_{\mathsf{L}} \leftarrow$ L with $N-1$ substituted for $N$ in all UB expressions (including for nested loops);
12:     if L has subloops then
13:         $t \leftarrow$ nesting depth of inner-most nested loop in L;
14:         $\mathsf{R}_{t+1} \leftarrow$ empty program with no statements;
15:         for $k = t; k \ge 2; k$-- do
16:             for each subloop $SL_j$ in $\mathsf{L}_i$ at nesting depth $k$ do    ▷ Ordered $SL_1$, $SL_2$, ..., $SL_j$
17:                 $\mathsf{R}_{SL_j} \leftarrow$ LPeel($SL_j$, $U_{SL_j}(\ell_1, \ldots, \ell_{k-1}, N-1)$, $U_{SL_j}(\ell_1, \ldots, \ell_{k-1}, N)$);
18:             $\mathsf{R}_k \leftarrow$ for (i=0; i<$U_{\mathsf{L}_{k-1}}(N-1)$; i++) { $\mathsf{R}_{k+1}$;$\mathsf{R}_{SL_1}$;$\mathsf{R}_{SL_2}$;...;$\mathsf{R}_{SL_j}$ };
19:         $\mathsf{R}_{\mathsf{L}} \leftarrow \mathsf{R}_2$;LPeel(L, $U_{\mathsf{L}}(N-1), U_{\mathsf{L}}(N)$);
20:     else
21:         $\mathsf{R}_{\mathsf{L}} \leftarrow$ LPeel(L, $U_{\mathsf{L}}(N-1), U_{\mathsf{L}}(N)$);
22:     return $\langle \mathsf{Q}_{\mathsf{L}}, \mathsf{R}_{\mathsf{L}} \rangle$;
```

second loop takes into account the missed iterations of the outer loop in Q_{N-1} compared to P_N.

Generalizing the above intuition, Algorithm 1 presents function GenQandPeel for computing Q_{N-1} and $\mathsf{peel}(\mathsf{P}_N)$ for a given P_N that has sequentially composed loops with potentially nested loops. Due to the grammar of our programs, our loops are well nested. The method works by traversing over the structure of loops in the program. In this algorithm $\mathsf{Q}_{\mathsf{L}_i}$ and $\mathsf{R}_{\mathsf{L}_i}$ represent the counterparts of Q_{N-1} and $\mathsf{peel}(\mathsf{P}_N)$ for loop L_i. We create the program Q_{N-1} by peeling each loop in the program and then propagating these peels across subsequent loops. We identify the missed iterations of each loop in the program P_N from the upper bound expression UB. Recall that the upper bound of each loop L_k at nesting depth k, denoted by U_{L_k} is in terms of the loop counters ℓ of outer loops and the program parameter N. We need to peel $U_{\mathsf{L}_k}(\ell_1, \ell_2, \ldots, \ell_{k-1}, N) - U_{\mathsf{L}_k}(\ell_1, \ell_2, \ldots, \ell_{k-1}, N-1)$ number of iterations from each loop, where $\ell_1 \le \ell_2 \le \ldots \le \ell_{k-1}$ are counters of the outer nesting loops. As discussed above, whenever this difference is a constant value, we are guaranteed that the loop nesting depth reduces by one. It may so happen that there are multiple sequentially composed loops SL_j at nesting depth k and not just a single loop L_k. At line 2, we iterate over top level loops and call function GenQandPeelForLoop(L_i) for each sequentially composed loop L_i in P_N. At line 11 we construct Q_{L} for loop L. If the loop L has no nested loops, then the peel is the last iterations computed using the upper bound in line 21 For nested loops, the loop at line 15 builds the peel for all loops inside L following the above intuition. The peels of all sub-loops are collected and inserted in the peel of L at line 19. Since all the peeled iterations are moved after Q_{L} of each loop, we need to repair expressions appearing in Q_{L}. The repairs are applied by the loop at line 4. In the repair step, we identify the right hand side expressions for all the variables and array elements assigned in the peeled iterations. Subsequently, the uses of the variables and arrays in $\mathsf{Q}_{\mathsf{L}_i}$ that are assigned in $\mathsf{R}_{\mathsf{L}_j}$ are replaced with the assigned expressions whenever $j < i$. If $\mathsf{R}_{\mathsf{L}_j}$ is a loop, this step is more involved and hence currently not considered. Finally at line 8, the peels and Qs of all top level loops are stitched and returned.

Note that lines 4 and 5 of Algorithm 1 implement the substitution represented by the arrow in the second column of Fig. 2. This is necessary in order to move the peel of a loop to the end of the program. If either of the loops L_i or L_j use array elements as index to other arrays then it can be difficult to identify what expression to use in $\mathsf{Q}_{\mathsf{L}_i}$ for the substitution. However, such scenarios are observed less often, and hence, they hardly impact the effectiveness of the technique on programs seen in practice. The peel $\mathsf{R}_{\mathsf{L}_j}$, from which the expression to be substituted in $\mathsf{Q}_{\mathsf{L}_i}$ has to be taken, itself may have a loop. In such cases, it can be significantly more challenging to identify what expression to use in $\mathsf{Q}_{\mathsf{L}_i}$. We use several optimizations to transform the peeled loop before trying to identify such an expression. If the modified values in the peel can be summarized as closed form expressions, then we can replace the loop in the peel with its summary. For example, consider the peeled loop, `for (` ℓ_1 `=0;` $\ell_1 <$ `N;` ℓ_1 `++) {` `S = S + 1; }`. This loop is summarized as `S = S + N;` before it can be moved across subsequent code. If the variables modified in the peel of a nested loop are not used later, then the peel can be trivially moved. In many cases, the loop in the peel can also be substituted with its conservative over-approximation. We have implemented some of these optimizations in our tool and are able to verify several benchmarks with sequentially composed nested loops. It may not always be possible to move the peel of a nested loop across subsequent loops but we have observed that these optimizations suffice for many programs seen in practice.

Theorem 1. *Let* Q_{N-1} *and* $\mathsf{peel}(\mathsf{P}_N)$ *be generated by application of function* GenQandPeel *from Algorithm 1 on program* P_N*. Then* P_N *is semantically equivalent to* $\mathsf{Q}_{N-1}; \mathsf{peel}(\mathsf{P}_N)$*.*

Lemma 1. *Suppose the following conditions hold;*

- *Program* P_N *satisfies our syntactic restrictions (see Sect. 3).*
- *The upper bound expressions of all loops are linear expressions in* N *and in the loop counters of outer nesting loops.*

Then, the max nesting depth of loops in $\mathsf{peel}(\mathsf{P}_N)$ *is strictly less than that in* P_N*.*

Proof. Let $U_{\mathsf{L}_k}(\ell_1, \ldots, \ell_{k-1}, N)$ be the upper bound expression of a loop L_k at nesting depth k. Suppose $U_{\mathsf{L}_k} = c_1.\ell_1 + \cdots c_{k-1}.\ell_{k-1} + C.N + D$, where $c_1, \ldots c_{k-1}, C$ and D are constants. Then $U_{\mathsf{L}_k}(\ell_1, \ldots, \ell_{k-1}, N) - U_{\mathsf{L}_k}(\ell_1, \ldots \ell_{k-1}, N-1) = C$, i.e. a constant. Now, recalling the discussion in Sect. 4.1, we see that `LPeel(`L_k`,` U_k`(`$\ell_1, \ldots, \ell_{k-1}$`,N − 1),` U_k`(`$\ell_1, \ldots, \ell_{k-1}$`,N))` simply results in concatenating a constant number of peels of the loop L_k. Hence, the maximum nesting depth of loops in `LPeel(` L_k, U_k`(`$\ell_1, \ldots, \ell_{k-1}$`,N` $-$ `1),` U_k`(`$\ell_1, \ldots, \ell_{k-1}$`,N))` is strictly less than the maximum nesting depth of loops in L_k.

Suppose loop L with nested loops (having maximum nesting depth t) is passed as the argument of function GenQandPeelForLoop (see Algorithm 1). In line 15 of function GenQandPeelForLoop, we iterate over all loops at nesting depth 2 and above within L. Let L_k be a loop at nesting depth k, where $2 \leq k \leq t$. Clearly, L_k can have at most $t - k$ nested levels of loops within it. Therefore, when `LPeel` is invoked on such a loop, the maximum nesting depth of loops in the peel generated for L_k can be at most $t - k - 1$. From lines 18 and 19 of function GenQandPeelForLoop, we also know that this `LPeel` can itself appear at nesting depth k of the overall peel R_L. Hence, the maximum nesting depth of loops in R_L can be $t - k - 1 + k$, i.e. $t - 1$. This is strictly less than the maximum nesting depth of loops in L. □

Corollary 1. *If* P_N *has no nested loops, then* $\mathsf{peel}(\mathsf{P}_N)$ *is loop-free.*

4.2 Generating $\varphi'(N-1)$ and $\Delta\varphi'(N)$

Given $\varphi(N)$, we check if it is of the form $\bigwedge_{i=0}^{N-1} \rho_i$, where ρ_i is a formula on the i^{th} elements of one or more arrays, and scalars used in P_N. If so, we infer $\varphi'(N-1)$ to be $\bigwedge_{i=0}^{N-2} \rho_i$ and $\Delta\varphi'(N)$ to be ρ_{N-1} (assuming variables/array elements in ρ_{N-1} are not modified by Q_{N-1}). Note that all uses of N in ρ_i are retained as is (i.e. not changed to $N-1$) in $\varphi'(N-1)$. In general, when deriving $\varphi'(N-1)$, we do not replace any use of N in $\varphi(N)$ by $N-1$ unless it is the limit of an iterated conjunct as discussed above. Specifically, if $\varphi(N)$ doesn't contain an iterated conjunct as above, then we consider $\varphi'(N-1)$ to be the same as $\varphi(N)$ and $\Delta\varphi'(N)$ to be True. Thus, our generation of $\varphi'(N-1)$ and $\Delta\varphi'(N)$ differs from that of [12]. As discussed earlier, this makes it possible to reason about a much larger class of pre-conditions than that admissible by the technique of [12].

4.3 Inferring Inductive Difference Invariants

Once we have P_{N-1}, Q_{N-1}, $\varphi(N-1)$ and $\varphi'(N-1)$, we infer *difference invariants*. We construct the standard cross-product of programs Q_{N-1} and P_{N-1}, denoted as $\mathsf{Q}_{N-1} \times \mathsf{P}_{N-1}$, and infer difference invariants at key control points. Note that P_{N-1} and Q_{N-1} are guaranteed to have synchronized iterations of corresponding loops (both are obtained by restricting the upper bounds of all loops to use $N-1$ instead of N). However, the conditional statements within the loop body may not be synchronized. Thus, whenever we can infer that the corresponding conditions are equivalent, we synchronize the branches of the conditional statement. Otherwise, we consider all four possibilities of the branch conditions. It can be seen that the net effect of the cross-product is executing the programs P_{N-1} and Q_{N-1} one after the other.

We run a dataflow analysis pass over the constructed product graph to infer difference invariants at loop head, loop exit and at each branch condition. The only dataflow values of interest are differences between corresponding variables in Q_{N-1} and P_{N-1}. Indeed, since structure and variables of Q_{N-1} and P_{N-1} are similar, we can create the correspondence map between the variables. We start the difference invariant generation by considering relations between corresponding variables/array elements appearing in pre-conditions of the two programs. We apply static analysis that can track equality expressions (including disjunctions over equality expressions) over variables as we traverse the program. These equality expressions are our difference invariants.

We observed in our experiments the most of the inferred equality expressions are simple expressions of N (atmost quadratic in N). This not totally surprising and similar observations have also been independently made in [4,15,24]. Note that the difference invariants may not always be equalities. We can easily extend our analysis to learn inequalities using interval domains in static analysis. We can also use a library of expressions to infer difference invariants using a guess-and-check framework. Moreover, guessing difference invariants can be easy as in many cases the difference expressions may be independent of the program constructs, for example, the equality expression $\mathtt{v} = \mathtt{v}'$ where $\mathtt{v} \in \mathsf{P}_{N-1}$ and $\mathtt{v}' \in \mathsf{Q}_{N-1}$ does not depend on any other variable from the two programs.

For the example in Fig. 2, the difference invariant at the head of the first loop of $Q_{N-1} \times P_{N-1}$ is $D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1) \equiv (\mathtt{x}' - \mathtt{x} = \mathtt{i} \times (2 \times \mathtt{N} - 1) \wedge \forall \mathtt{i} \in [0, \mathtt{N}-1),\ \mathtt{a}'[\mathtt{i}] - \mathtt{a}[\mathtt{i}] = 1)$, where $\mathtt{x}, \mathtt{a} \in V_{\mathsf{P}}$ and $\mathtt{x}', \mathtt{a}' \in V_{\mathsf{Q}}$. Given this, we easily get $\mathtt{x}' - \mathtt{x} = (\mathtt{N}-1) \times (2 \times \mathtt{N} - 1)$ when the first loop terminates. For the second loop, $D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1) \equiv (\forall \mathtt{j} \in [0, \mathtt{N}-1),\ \mathtt{b}'[\mathtt{j}] - \mathtt{b}[\mathtt{j}] = (\mathtt{x}' - \mathtt{x}) + \mathtt{N}^2 = (\mathtt{N}-1) \times\ (2 \times \mathtt{N} - 1) + \mathtt{N}^2)$.

Note that the difference invariants and its computation are agnostic of the given post-condition. Hence, our technique does not need to re-run this analysis for proving a different post-condition for the same program.

4.4 Verification Using Inductive Difference Invariants

We present our method DIFFY for verification of programs using inductive difference invariants in Algorithm 2. It takes a Hoare triple $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$ as input, where $\varphi(N)$ and $\psi(N)$ are pre- and post-condition formulas. We check the base in line 1 to verify the Hoare triple for $N = 1$. If this check fails, we report a counterexample. Subsequently, we compute Q_{N-1} and $\mathsf{peel}(\mathsf{P}_N)$ as described in Sect. 4.1 using the function GENQANDPEEL from Algorithm 1. At line 4, we compute the formulas $\varphi'(N-1)$ and $\Delta\varphi'(N)$ as described in Sect. 4.2. For automation, we analyze the quantifiers appearing in $\varphi(N)$ and modify the quantifier ranges such that the conditions in Sect. 4.2 hold. We infer difference invariants $D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1)$ on line 5 using the method described in Sect. 4.3, wherein V_{Q} and V_{P} are sets of variables from Q_{N-1} and P_{N-1} respectively. At line 6, we compute $\psi'(N-1)$ by eliminating variables V_{P} from P_{N-1} from $\psi(N-1) \wedge D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1)$. At line 7, we check the inductive step of our analysis. If the inductive step succeeds, then we conclude that the assertion holds.

Algorithm 2. DIFFY($\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$)

```
1: if $\{\varphi(1)\}\ \mathsf{P}_1\ \{\psi(1)\}$ fails then    ▷ Base case for N=1
2:     return “Counterexample found!”;

3: $\langle \mathsf{Q}_{N-1}, \mathsf{peel}(\mathsf{P}_N)\rangle \leftarrow$ GenQAndPeel($\mathsf{P}_N$);
4: $\langle \varphi'(N-1), \Delta\varphi'(N)\rangle \leftarrow$ FormulaDiff($\varphi(N)$);    ▷ $\varphi(N) \Rightarrow \varphi'(N-1) \wedge \Delta\varphi'(N)$
5: $D(V_\mathsf{Q}, V_\mathsf{P}, N-1) \leftarrow$ InferDiffInvs($\mathsf{Q}_{N-1}, \mathsf{P}_{N-1}, \varphi'(N-1), \varphi(N-1)$);
6: $\psi'(N-1) \leftarrow \mathrm{QE}(V_\mathsf{P}, \psi(N-1) \wedge D(V_\mathsf{Q}, V_\mathsf{P}, N-1))$;
7: if $\{\psi'(N-1) \wedge \Delta\varphi'(N)\}\ \mathsf{peel}(\mathsf{P}_N)\ \{\psi(N)\}$ then
8:     return True;    ▷ Verification Successful
9: else
10:    return Strengthen($\mathsf{P}_N, \mathsf{peel}(\mathsf{P}_N), \varphi(N), \psi(N), \psi'(N-1), \Delta\varphi'(N), D(V_\mathsf{Q}, V_\mathsf{P}, N)$);

11: procedure Strengthen($\mathsf{P}_N$, $\mathsf{peel}(\mathsf{P}_N)$, $\varphi(N)$, $\psi(N)$, $\psi'(N-1)$, $\Delta\varphi'(N)$, $D(V_\mathsf{Q}, V_\mathsf{P}, N)$)
12:     $\chi(N) \leftarrow \psi(N)$;
13:     $\xi(N) \leftarrow$ True;
14:     $\xi'(N-1) \leftarrow$ True;
15:     repeat
16:         $\chi'(N-1) \leftarrow \mathrm{WP}(\chi(N), \mathsf{peel}(\mathsf{P}_N))$;    ▷ Dijkstra’s WP for loop free code
17:         if $\chi'(N-1) = \emptyset$ then
18:             if $\mathsf{peel}(\mathsf{P}_N)$ has a loop then
19:                 return Diffy($\{\xi'(N-1) \wedge \Delta\varphi'(N) \wedge \psi'(N-1)\}\ \mathsf{peel}(\mathsf{P}_N)\ \{\xi(N) \wedge \psi(N)\}$);
20:             else
21:                 return False;    ▷ Unable to prove
22:         $\chi(N) \leftarrow \mathrm{QE}(V_\mathsf{Q}, \chi'(N) \wedge D(V_\mathsf{Q}, V_\mathsf{P}, N))$;
23:         $\xi(N) \leftarrow \xi(N) \wedge \chi(N)$;
24:         $\xi'(N-1) \leftarrow \xi'(N-1) \wedge \chi'(N-1)$;
25:         if $\{\varphi(1)\}\ \mathsf{P}_1\ \{\xi(1)\}$ fails then
26:             return False;    ▷ Unable to prove
27:         if $\{\xi'(N-1) \wedge \Delta\varphi'(N) \wedge \psi'(N-1)\}\ \mathsf{peel}(\mathsf{P}_N)\ \{\xi(N) \wedge \psi(N)\}$ holds then
28:             return True;    ▷ Verification Successful
29:     until timeout;
30:     return False;
```

If that is not the case then, we try to iteratively strengthen both the pre- and post-condition of $\mathsf{peel}(\mathsf{P}_N)$ simultaneously by invoking Strengthen.

The function Strengthen first initializes the formula $\chi(N)$ with $\psi(N)$ and the formulas $\xi(N)$ and $\xi'(N-1)$ to True. To strengthen the pre-condition of $\mathsf{peel}(\mathsf{P}_N)$, we infer a formula $\chi'(N-1)$ using Dijkstra’s weakest pre-condition computation of $\chi(N)$ over the $\mathsf{peel}(\mathsf{P}_N)$ in line 16. It may happen that we are unable to infer such a formula. In such a case, if the program $\mathsf{peel}(\mathsf{P}_N)$ has loops then we recursively invoke Diffy at line 19 to further simplify the program. Otherwise, we abandon the verification effort (line 21). We use quantifier elimination to infer $\chi(N-1)$ from $\chi'(N-1)$ and $D(V_\mathsf{Q}, V_\mathsf{P}, N-1))$ at line 6.

The inferred pre-conditions $\chi(N)$ and $\chi'(N-1)$ are accumulated in $\xi(N)$ and $\xi'(N-1)$, which strengthen the post-conditions of P_N and Q_{N-1} respectively in lines 23–24. We again check the base case for the inferred formulas in $\xi(N)$ at line 25. If the check fails we abandon the verification attempt at line 26. If the base case succeeds, we then proceed to the inductive step. When the inductive step succeeds, we conclude that the assertion is verified. Otherwise, we continue in the loop and try to infer more pre-conditions untill we run out of time.

The pre-condition in Fig. 2 is $\phi(N) \equiv \mathsf{True}$ and the post-condition is $\psi(N) \equiv \forall \mathtt{j} \in [\mathtt{0}, \mathtt{N}),\ \mathtt{b[j]} = \mathtt{j} + \mathtt{N}^3)$. At line 4, $\phi'(N-1)$ and $\Delta\phi'(N-1)$ are computed to be True. $D(V_\mathsf{Q}, V_\mathsf{P}, N-1)$ is the formula computed in Sect. 4.3. At line 6,

Table 1. Summary of the experimental results. S is successful result. U is inconclusive result. TO is timeout.

Program Category		Diffy			Vajra		VeriAbs		VIAP		
		S	U	TO	S	U	S	TO	S	U	TO
Safe C1	110	110	0	0	110	0	96	14	16	1	93
Safe C2	24	21	0	3	0	24	5	19	4	0	20
Safe C3	23	20	3	0	0	23	9	14	0	23	0
Total	157	151	3	3	110	47	110	47	20	24	113
Unsafe C1	99	98	1	0	98	1	84	15	98	0	1
Unsafe C2	24	24	0	0	17	7	19	5	22	0	2
Unsafe C3	23	20	3	0	0	23	22	1	0	23	0
Total	146	142	4	0	115	31	125	21	120	23	3

$\psi'(N-1) \equiv (\forall \mathtt{j} \in [\mathtt{0}, \mathtt{N}-\mathtt{1}),\ \mathtt{b}'[\mathtt{j}] = \mathtt{j} + (\mathtt{N}-\mathtt{1})^3 + (\mathtt{N}-\mathtt{1}) \times (\mathtt{2} \times \mathtt{N} - \mathtt{1}) + \mathtt{N}^2 = \mathtt{j} + \mathtt{N}^3)$. The algortihm then invokes Strengthen at line 10 which infers the formulas $\chi'(N-1) \equiv (\mathtt{x}' = (\mathtt{N}-\mathtt{1})^3)$ at line 16 and $\chi(N) \equiv (\mathtt{x} = \mathtt{N}^3)$ at line 22. These are accumulated in $\xi'(N-1)$ and $\xi(N)$, simultaneosuly strengthening the pre- and post-condition. Verification succeeds after this strengthening iteration.

The following theorem guarantees the soundness of our technique.

Theorem 2. *Suppose there exist formulas $\xi'(N)$ and $\xi(N)$ and an integer $M > 0$ such that the following hold*

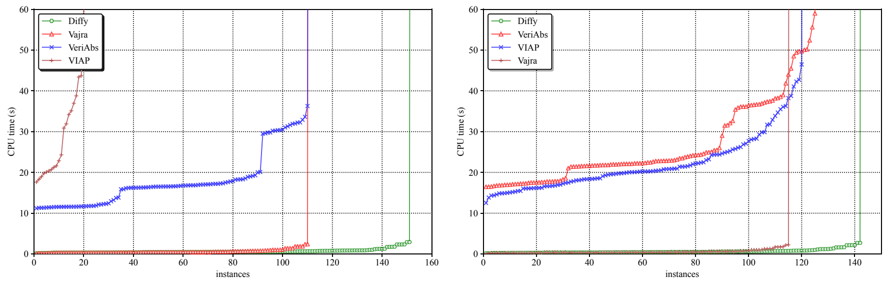
- $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N) \wedge \xi(N)\}$ *holds for* $1 \leq N \leq M$*, for some* $M > 0$.
- $\xi(N) \wedge D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N) \Rightarrow \xi'(N)$ *for all* $N > 0$.
- $\{\xi'(N-1) \wedge \Delta\varphi'(N) \wedge \psi'(N-1)\}\ \mathsf{peel}(\mathsf{P}_N)\ \{\xi(N) \wedge \psi(N)\}$ *holds for all* $N \geq M$*, where* $\psi'(N-1) \equiv \exists V_{\mathsf{P}}\big(\psi(N-1) \wedge D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1)\big)$.

Then $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$ *holds for all* $N > 0$.

5 Experimental Evaluation

We have instantiated our technique in a prototype tool called Diffy. It is written in `C++` and is built using the LLVM(v6.0.0) [31] compiler. We use the SMT solver Z3(v4.8.7) [39] for proving Hoare triples of loop-free programs. Diffy and the supporting data to replicate the experiments are openly available at [14].

Setup. All experiments were performed on a machine with Intel i7-6500U CPU, 16 GB RAM, running at 2.5 GHz, and Ubuntu 18.04.5 LTS operating system. We have compared the results obtained from Diffy with Vajra(v1.0) [12], VIAP(v1.1) [42] and VeriAbs(v1.4.1-12) [1]. We choose Vajra which also employs inductive reasoning for proving array programs and verify the benchmarks in its test-suite. We compared with VeriAbs as it is the winner of the arrays sub-category in SV-COMP 2020 [6] and 2021 [7]. VeriAbs applies a



(a) (b)

Fig. 8. Cactus Plots (a) All Safe Benchmarks (b) All Unsafe Benchmarks

sequence of techniques from its portfolio to verify array programs. We compared with VIAP which was the winner in arrays sub-category in SV-COMP 2019 [5]. VIAP also employs a sequence of tactics, implemented for proving a variety of array programs. DIFFY does not use multiple techniques, however we choose to compare it with these portfolio verifiers to show that it performs well on a class of programs and can be a part of their portfolio. All tools take C programs in the SV-COMP format as input. Timeout of 60 s was set for each tool. A summary of the results is presented in Table 1.

Benchmarks. We have evaluated DIFFY on a set of 303 array benchmarks, comprising of the entire test-suite of [12], enhanced with challenging benchmarks to test the efficacy of our approach. These benchmarks take a symbolic parameter N which specifies the size of each array. Assertions are (in-)equalities over array elements, scalars and (non-)linear polynomial terms over N. We have divided both the safe and unsafe benchmarks in three categories. Benchmarks in C1 category have standard array operations such as min, max, init, copy, compare as well as benchmarks that compute polynomials. In these benchmarks, branch conditions are not affected by the value of N, operations such as modulo and nested loops are not present. There are 110 safe and 99 unsafe programs in the C1 category in our test-suite. In C2 category, the branch conditions are affected by change in the program parameter N and operations such as modulo are used in these benchmarks. These benchmarks do not have nested loops in them. There are 24 safe and unsafe benchmarks in the C2 category. Benchmarks in category C3 are programs with atleast one nested loop in them. There are 23 safe and unsafe programs in category C3 in our test-suite. The test-suite has a total of 157 safe and 146 unsafe programs.

Analysis. DIFFY verified 151 safe benchmarks, compared to 110 verified by VAJRA as well as VERIABS and 20 verified by VIAP. DIFFY was unable to verify 6 safe benchmarks. In 3 cases, the smt solver timed out while trying to prove the induction step since the formulated query had a modulus operation and in 3 cases it was unable to compute the predicates needed to prove the assertions. VAJRA was unable to verify 47 programs from categories C2 and

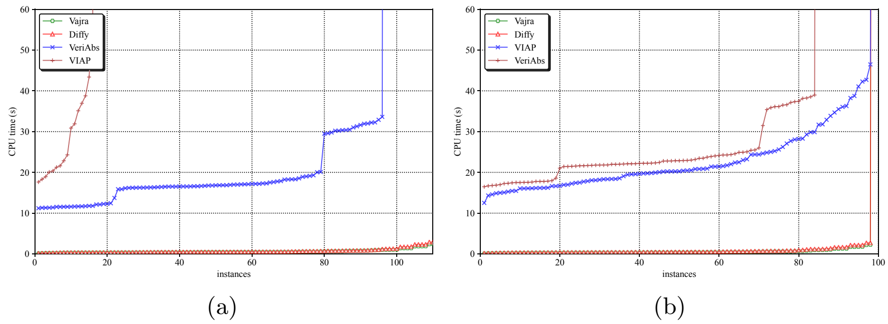


Fig. 9. Cactus plots (a) Safe C1 benchmarks (b) Unsafe C1 benchmarks

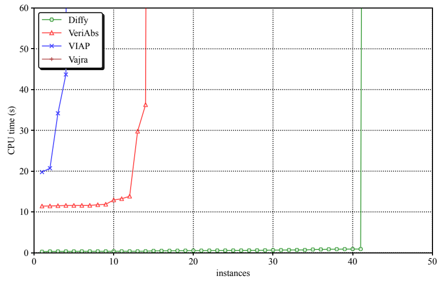
C3. These are programs with nested loops, branch conditions affected by N, and cases where it could not compute the difference program. The sequence of techniques employed by VeriAbs, ran out of time on 47 programs while trying to prove the given assertion. VeriAbs proved 2 benchmarks in category C2 and 3 benchmarks in category C3 where Diffy was inconclusive or timed out. VeriAbs spends considerable amount of time on different techniques in its portfolio before it resorts to Vajra and hence it could not verify 14 programs that Vajra was able to prove efficiently. VIAP was inconclusive on 24 programs which had nested loops or constructs that could not be handled by the tool. It ran out of time on 113 benchmarks as the initial tactics in its sequence took up the allotted time but could not verify the benchmarks. Diffy was able to verify all programs that VIAP and Vajra were able to verify within the specified time limit.

The cactus plot in Fig. 8(a) shows the performance of each tool on all safe benchmarks. Diffy was able to prove most of the programs within three seconds. The cactus plot in Fig. 9(a) shows the performance of each tool on safe benchmarks in C1 category. Vajra and Diffy perform equally well in the C1 category. This is due to the fact that both tools perform efficient inductive reasoning. Diffy outperforms VeriAbs and VIAP in this category. The cactus plot in Fig. 10(a) shows the performance of each tool on safe benchmarks in the combined categories C2 and C3, that are difficult for Vajra as most of these programs are not within its scope. Diffy out performs all other tools in categories C2 and C3. VeriAbs was an order of magnitude slower on programs it was able to verify, as compared to Diffy. VeriAbs spends significant amount of time in trying techniques from its portfolio, including Vajra, before one of them succeeds in verifying the assertion or takes up the entire time allotted to it. VIAP took 70 seconds more on an average as compared to Diffy to verify the given benchmark. VIAP also spends a large portion of time in trying different tactics implemented in the tool and solving the recurrence relations in programs.

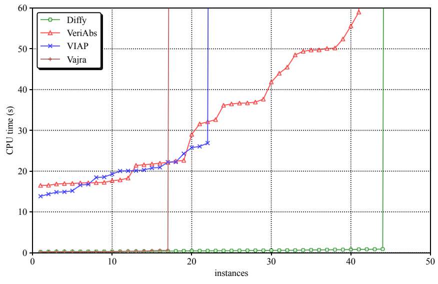
Our technique reports property violations when the base case of the analysis fails for small fixed values of N. While the focus of our work is on proving assertions, we report results on unsafe versions of the safe benchmarks from our test-suite. Diffy was able to detect a property violation in 142 unsafe programs and was inconclusive on 4 benchmarks. Vajra detected violations in 115 programs and was inconclusive on 31 programs. VeriAbs reported 125 programs as unsafe and ran out of time on 21 programs. VIAP reported property violation in 120 programs, was inconclusive on 23 programs and timed out on 3 programs.

The cactus plot in Fig. 8(b) shows the performance of each tool on all unsafe benchmarks. Diffy was able to detect a violation faster than all other tools and on more benchmarks from the test-suite. Figure 9(b) and Fig. 10(b) give a finer glimpse of the performance of these tools on the categories that we have defined. In the C1 category, Diffy and Vajra have comparable performance and Diffy disproves the same number of benchmarks as Vajra and VIAP. In C2 and C3 categories, we are able to detect property violations in more benchmarks than other tools in less time.

To observe any changes in the performance of these, we also ran them with an increased time out of 100 seconds (Fig. 11). Performance remains unchanged for Diffy, Vajra and VeriAbs on both safe and unsafe benchmarks, and of VIAP on unsafe benchmarks. VIAP was able to additionally verify 89 safe programs in categories C1 and C2 with the increased time limit.

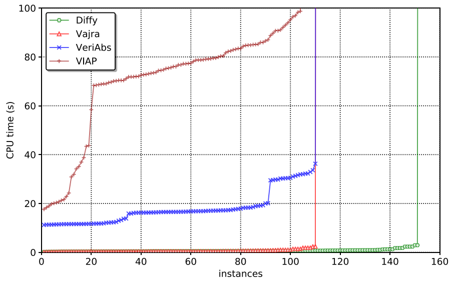


(a)

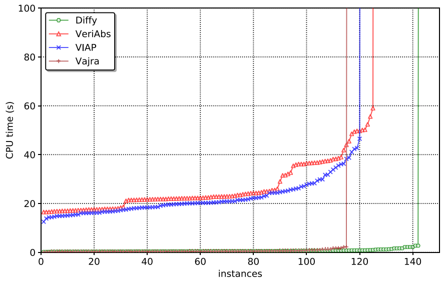


(b)

Fig. 10. Cactus plots (a) Safe C2 & C3 benchmarks (b) Unsafe C2 & C3 benchmarks



(a)



(b)

Fig. 11. Cactus plots. TO = 100 s. (a) Safe benchmarks (b) Unsafe benchmarks

6 Related Work

Techniques Based on Induction. Our work is related to several efforts that apply inductive reasoning to verify properties of array programs. Our work subsumes the full-program induction technique in [12] that works by inducting on the entire program via a program parameter N. We propose a principled method for computation and use of difference invariants, instead of computing difference programs which is more challenging. An approach to construct safety proofs by automatically synthesizing squeezing functions that shrink program traces is proposed in [27]. Such functions are not easy to synthesize, whereas difference invariants are relatively easy to infer. In [11], the post-condition is inductively established by identifying a tiling relation between the loop counter and array indices used in the program. Our technique can verify programs from [11], when supplied with the *tiling* relation. [44] identifies recurrent program fragments for induction using the loop counter. They require restrictive data dependencies, called *commutativity of statements*, to move peeled iterations across subsequent loops. Unfortunately, these restrictions are not satisfied by a large class of programs in practice, where our technique succeeds.

Difference Computation. Computing differences of program expressions has been studied for incremental computation of expensive expressions [35,41], optimizing programs with arrays [34], and checking data-structure invariants [45]. These differences are not always well suited for verifying properties, in contrast with the difference invariants which enable inductive reasoning in our case.

Logic Based Reasoning. In [21], trace logic that implicitly captures inductive loop invariants is described. They use theorem provers to introduce and prove lemmas at arbitrary control locations in the program. Unlike their technique, we focus primarily on universally quantified and quantifier-free properties, although a restricted class of existentially quantified properties can be handled by our technique (see [13] for more details). VIAP [42] translates the program to an quantified first-order logic formula using the scheme proposed in [32]. It uses a portfolio of tactics to simplify and prove the generated formulas. Dedicated solvers for recurrences are used whereas our technique adapts induction for handling recurrences.

Invariant Generation. Several techniques generate invariants for array programs. QUIC3 [25], FreqHorn [9,19] infer universally quantified invariants over arrays for Constrained Horn Clauses (CHCs). Template-based techniques [8,23,47] search for inductive quantified invariants by instantiating parameters of a fixed set of templates. We generate relational invariants, which are often easier to infer compared to inductive quantified invariants for each loop.

Abstraction-Based Techniques. Counterexample-guided abstraction refinement using prophecy variables for programs with arrays is proposed in [36]. Veri-Abs [1] uses a portfolio of techniques, specifically to identify loops that can be soundly abstracted by a bounded number of iterations. Vaphor [38] transforms array programs to array-free Horn formulas to track bounded number of array cells. Booster [3] combines lazy abstraction based interpolation [2] and acceleration [10,28] for array programs. Abstractions in [16,18,22,26,29,33,37] implicitly or explicitly partition the range array indices to infer and prove facts on array segments. In contrast, our method does not rely on abstractions.

7 Conclusion

We presented a novel verification technique that combines generation of difference invariants and inductive reasoning. Difference invariants relate corresponding variables and arrays from the two versions of a program and are often easy to infer and prove. We have instantiated these techniques in our prototype Diffy. Experiments shows that Diffy out-performs the tools that won the Arrays sub-category in SV-COMP 2019, 2020 and 2021. Although we have focused on universal and quantifier-free properties in this paper, the technique applies to some classes of existential properties as well. The interested reader is referred to [13] for more details. Investigations in using synthesis techniques for automatic generation of difference invariants to verify properties of array manipulating programs is a part of future work.

References

1. Afzal, M., et al.: Veriabs: verification by abstraction and test generation (competition contribution). In: TACAS 2020. LNCS, vol. 12079, pp. 383–387. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45237-7_25
2. Alberti, F., Bruttomesso, R., Ghilardi, S., Ranise, S., Sharygina, N.: Lazy abstraction with interpolants for arrays. In: Bjørner, N., Voronkov, A. (eds.) LPAR 2012. LNCS, vol. 7180, pp. 46–61. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28717-6_7
3. Alberti, F., Ghilardi, S., Sharygina, N.: Booster: an acceleration-based verification framework for array programs. In: Cassez, F., Raskin, J.-F. (eds.) ATVA 2014. LNCS, vol. 8837, pp. 18–23. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11936-6_2
4. Barthe, G., Crespo, J.M., Kunz, C.: Relational verification using product programs. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 200–214. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21437-0_17
5. Beyer, D.: Competition on software verification (SV-COMP) (2019). http://sv-comp.sosy-lab.org/2019/
6. Beyer, D.: Competition on software verification (SV-COMP) (2020). http://sv-comp.sosy-lab.org/2020/
7. Beyer, D.: Competition on software verification (SV-COMP) (2021). http://sv-comp.sosy-lab.org/2021/
8. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Invariant synthesis for combined theories. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 378–394. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-69738-1_27
9. Bjørner, N., McMillan, K., Rybalchenko, A.: On solving universally quantified horn clauses. In: Logozzo, F., Fähndrich, M. (eds.) SAS 2013. LNCS, vol. 7935, pp. 105–125. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38856-9_8

10. Bozga, M., Iosif, R., Konečný, F.: Fast acceleration of ultimately periodic relations. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 227–242. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_23
11. Chakraborty, S., Gupta, A., Unadkat, D.: Verifying array manipulating programs by tiling. In: Ranzato, F. (ed.) SAS 2017. LNCS, vol. 10422, pp. 428–449. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66706-5_21
12. Chakraborty, S., Gupta, A., Unadkat, D.: Verifying array manipulating programs with full-program induction. In: TACAS 2020. LNCS, vol. 12078, pp. 22–39. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45190-5_2
13. Chakraborty, S., Gupta, A., Unadkat, D.: Diffy: inductive reasoning of array programs using difference invariants (2021). https://arxiv.org/abs/2105.14748
14. Chakraborty, S., Gupta, A., Unadkat, D.: Diffy: inductive reasoning of array programs using difference invariants, April 2021. https://doi.org/10.6084/m9.figshare.14509467
15. Churchill, B., Padon, O., Sharma, R., Aiken, A.: Semantic program alignment for equivalence checking. In: Proceedings of PLDI, pp. 1027–1040 (2019)
16. Cousot, P., Cousot, R., Logozzo, F.: A parametric segmentation functor for fully automatic and scalable array content analysis. In: Proceedings of POPL, pp. 105–118 (2011)
17. Dahiya, M., Bansal, S.: Black-box equivalence checking across compiler optimizations. In: Chang, B.-Y.E. (ed.) APLAS 2017. LNCS, vol. 10695, pp. 127–147. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-71237-6_7
18. Dillig, I., Dillig, T., Aiken, A.: Fluid updates: beyond strong vs. weak updates. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 246–266. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11957-6_14
19. Fedyukovich, G., Prabhu, S., Madhukar, K., Gupta, A.: Quantified invariants via syntax-guided synthesis. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 259–277. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_14
20. Flanagan, C., Qadeer, S.: Predicate abstraction for software verification. In: Proceedings of POPL, pp. 191–202 (2002)
21. Georgiou, P., Gleiss, B., Kovács, L.: Trace logic for inductive loop reasoning. In: Proceedings of FMCAD, pp. 255–263 (2020)
22. Gopan, D., Reps, T.W., Sagiv, S.: A framework for numeric analysis of array operations. In: Proceedings of POPL, pp. 338–350 (2005)
23. Gulwani, S., McCloskey, B., Tiwari, A.: Lifting abstract interpreters to quantified logical domains. In: Proceedings of POPL, pp. 235–246 (2008)
24. Gupta, S., Rose, A., Bansal, S.: Counterexample-guided correlation algorithm for translation validation. Proc. OOPSLA **4**, 1–29 (2020)
25. Gurfinkel, A., Shoham, S., Vizel, Y.: Quantifiers on demand. In: Lahiri, S.K., Wang, C. (eds.) ATVA 2018. LNCS, vol. 11138, pp. 248–266. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01090-4_15
26. Halbwachs, N., Péron, M.: Discovering properties about arrays in simple programs. In: Proceedings of PLDI, pp. 339–348 (2008)
27. Ish-Shalom, O., Itzhaky, S., Rinetzky, N., Shoham, S.: Putting the squeeze on array programs: loop verification via inductive rank reduction. In: Proceedings of VMCAI, pp. 112–135 (2020)
28. Jeannet, B., Schrammel, P., Sankaranarayanan, S.: Abstract acceleration of general linear loops. In: Proceedings of POPL, pp. 529–540 (2014)

29. Jhala, R., McMillan, K.L.: Array abstractions from proofs. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 193–206. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73368-3_23
30. Knobe, K., Sarkar, V.: Array ssa form and its use in parallelization. In: Proceedings of POPL, pp. 107–120 (1998)
31. Lattner, C.: LLVM and clang: next generation compiler technology. In: The BSD Conference, pp. 1–2 (2008)
32. Lin, F.: A formalization of programs in first-order logic with a discrete linear order. Artif. Intell. **235**, 1–25 (2016)
33. Liu, J., Rival, X.: Abstraction of arrays based on non contiguous partitions. In: D'Souza, D., Lal, A., Larsen, K.G. (eds.) VMCAI 2015. LNCS, vol. 8931, pp. 282–299. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46081-8_16
34. Liu, Y.A., Stoller, S.D., Li, N., Rothamel, T.: Optimizing aggregate array computations in loops. TOPLAS **27**(1), 91–125 (2005)
35. Liu, Y.A., Stoller, S.D., Teitelbaum, T.: Static caching for incremental computation. TOPLAS **20**(3), 546–585 (1998)
36. Mann, M., Irfan, A., Griggio, A., Padon, O., Barrett, C.: Counterexample-guided prophecy for model checking modulo the theory of arrays. In: TACAS 2021. LNCS, vol. 12651, pp. 113–132. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72016-2_7
37. Monniaux, D., Alberti, F.: A simple abstraction of arrays and maps by program translation. In: Blazy, S., Jensen, T. (eds.) SAS 2015. LNCS, vol. 9291, pp. 217–234. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48288-9_13
38. Monniaux, D., Gonnord, L.: Cell morphing: from array programs to array-free horn clauses. In: Rival, X. (ed.) SAS 2016. LNCS, vol. 9837, pp. 361–382. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53413-7_18
39. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
40. Necula, G.C.: Translation validation for an optimizing compiler. In: Proceedings of PLDI, pp. 83–94 (2000)
41. Paige, R., Koenig, S.: Finite differencing of computable expressions. TOPLAS **4**(3), 402–454 (1982)
42. Rajkhowa, P., Lin, F.: Extending VIAP to handle array programs. In: Piskac, R., Rümmer, P. (eds.) VSTTE 2018. LNCS, vol. 11294, pp. 38–49. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03592-1_3
43. Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Global value numbers and redundant computations. In: Proceedings of POPL, pp. 12–27 (1988)
44. Seghir, M.N., Brain, M.: Simplifying the verification of quantified array assertions via code transformation. In: Albert, E. (ed.) LOPSTR 2012. LNCS, vol. 7844, pp. 194–212. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38197-3_13
45. Shankar, A., Bodik, R.: Ditto: automatic incrementalization of data structure invariant checks (in Java). ACM SIGPLAN Not. **42**(6), 310–319 (2007)
46. Sharma, R., Schkufza, E., Churchill, B., Aiken, A.: Data-driven equivalence checking. In: Proceedings of OOPSLA, pp. 391–406 (2013)
47. Srivastava, S., Gulwani, S.: Program verification using templates over predicate abstraction. ACM SIGPLAN Not. **44**(6), 223–234 (2009)

48. Zaks, A., Pnueli, A.: CoVaC: compiler validation by program analysis of the cross-product. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 35–51. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68237-0_5
49. Zuck, L., Pnueli, A., Fang, Y., Goldberg, B.: VOC: a translation validator for optimizing compilers. ENTCS **65**(2), 2–18 (2002)

Author Index