UC San Diego

UC San Diego Electronic Theses and Dissertations

Title

Sequentialization and Synchronization for Distributed Programs

Permalink

https://escholarship.org/uc/item/0hh6k2zn

Author

Bakst, Alexander Goldberg

Publication Date

2017

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA, SAN DIEGO

Sequentialization and Synchronization for Distributed Programs

A dissertation submitted in partial satisfaction of the
requirements for the degree of Doctor of Philosophy

in

Computer Science

by

Alexander Goldberg Bakst

Committee in charge:

Professor Ranjit Jhala, Chair
Professor Sorin Lerner
Professor Todd Millstein
Professor Jeffrey Remmel
Professor Geoffrey Voelker

2017

Copyright

Alexander Goldberg Bakst, 2017

All rights reserved.

The Dissertation of Alexander Goldberg Bakst is approved and is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California, San Diego

2017

DEDICATION

For Mom, Dad, and Sarey.

EPIGRAPH

The key to understanding complicated things is knowing what not to look at.

Gerald Jay Sussman

TABLE OF CONTENTS

Signature Page iii

Dedication iv

Epigraph v

Table of Contents vi

List of Figures viii

List of Tables x

Acknowledgements xi

Vita xiii

Abstract of the Dissertation xiv

Chapter 1 Introduction 1
1.1 Verification of Distributed Systems And Protocols 2
1.2 Toward Automated Verification 4
1.2.1 Iterating over a list 5
1.2.2 Verifying Distributed Programs 9
1.2.3 Discussion 10
1.3 Related Approaches to Automated Verification 11
1.4 Contributions 15

Chapter 2 Message Passing Programs 17
2.1 Overview 17
2.2 Syntax and Semantics 18
2.2.1 Syntax 18
2.2.2 Semantics 20
2.3 Tagged Programs 23
2.4 Reductions for I_S 24
2.5 Summary 27

Chapter 3 Canonical Sequentializations 29
3.1 Overview 31
3.1.1 A Task Distribution Service 31
3.1.2 Main Ideas 36
3.1.3 Expressiveness 40
3.2 Symmetric Non-determinism 43
3.3 Rewrite Rules 44
3.3.1 Basic Rules 45
3.3.2 Loops, Unfolding and Conditionals 46

3.3.3 Residual Processes 52
3.3.4 Correctness 55
3.4 Canonical Sequentialization of I_S Programs 56
3.5 From HASKELL to I_S 57
3.5.1 Syntax and Semantics of λ_m 57
3.5.2 Translation to I_S 60
3.5.3 Correctness of the Translation 66
3.6 Evaluation 69
3.7 Summary 73

Chapter 4 Pretend Synchrony 75
4.1 Overview 77
4.2 Verification with I_S 82
4.3 Rewrite Rules 84
4.3.1 Rewriting loop-free programs 85
4.3.2 Communication between a Single Process and a Group 86
4.3.3 Communication between Groups of Processes 90
4.3.4 Rewrite Rule Correctness 92
4.3.5 Extensions 94
4.4 Generating Verification Conditions 95
4.5 Case Studies: Synchronous Invariants for Distributed Protocols 100
4.5.1 Raft Leader Election 101
4.5.2 Single-decree Paxos 104
4.6 Summary 107

Chapter 5 Conclusion 109
5.1 Future Directions 110

Appendix A Translation from λ_m to I_S 113

Appendix B Sequentialization 120

Bibliography 130

LIST OF FIGURES

Figure 1.1. Program ManyPings: A server collects "ping" messages from a set of clients. . . . 4

Figure 1.2. Printing a list of messages. . . . 6

Figure 1.3. Closing a list of files with heap summaries corresponding to labeled program points. . . . 7

Figure 1.4. Simplification of ManyPings. . . . 8

Figure 2.1. I_S Syntax. . . . 18

Figure 2.2. Definition of $A : \checkmark$. . . . 19

Figure 2.3. Definition of $\mathsf{Procs}(A)$ 19

Figure 2.4. Definition of $A \ltimes B$. . . . 20

Figure 2.5. Congruence relation $A \equiv B$ 22

Figure 2.6. I_S Semantic Definitions 22

Figure 2.7. I_S Semantics: $\mathcal{C} \xrightarrow{T}_{I_S} \mathcal{C}$ 28

Figure 3.1. A task distribution service. . . . 34

Figure 3.2. Canonical Sequentialization of the task distribution system in Figure 3.1 as computed by BRISK. . . . 35

Figure 3.3. Syntax for context Γ. . . . 44

Figure 3.4. Rewrite Rules (Basic Statements) 47

Figure 3.5. Rewrite Rules (Iteration over sets of processes). . . . 47

Figure 3.6. Rewrite Rules (Iteration over sets of indices). . . . 53

Figure 3.7. Rewrite Rules (Unfold) 53

Figure 3.8. Rewrite Rules (Residue) 54

Figure 3.9. Rewrite Rules (Branch) 54

Figure 3.10. λ_m syntax 58

Figure 3.11. Semantics of λ_m terms 59

Figure 3.12. MapReduce implemented in HASKELL 61

Figure 3.13. Transformation of λ_m to I_S 63

Figure 3.14. Well-formed λ_m terms 65

Figure 3.15. Tagged I_S model of MAPREDUCE program from Figure 3.12 66

Figure 3.16. THEQUE messages (top) and remote procedure calls (bottom). An arrow from A to B indicates that A sends B messages. The dashed line surrounds the services comprising THEQUE. 71

Figure 4.1. Two phase commit. Arrows denote the flow of messages during each logical phase of the protocol. The boxes of code at either end of an arrow will be synchronized by our method. Each `for` loop is annotated with an invariant {I}. 78

Figure 4.2. Synchronization of two phase commit. 81

Figure 4.3. I_S Syntax. Changes with respect to Figure 2.1 hilighted in gray. ... 83

Figure 4.4. Syntax for context Γ. Changes with respect to Figure 3.3 highlighted in gray. 84

Figure 4.5. Rewrite Rules (Basic Statements) 88

Figure 4.6. Rewrite Rules (Iteration over sets of processes and unfolding) 88

Figure 4.7. Selected rules establishing control flow independence. 91

Figure 4.8. Rewrite Rules (case splitting on a single process) 93

Figure 4.9. Reactive while loop. 95

Figure 4.10. Weakest precondition computation. $V(A)$ denotes the variables that may be assigned in A. 96

Figure 4.11. Simplified Raft leader election algorithm (asynchronous version) .. 102

Figure 4.12. Single-decree Paxos (asynchronous version) 105

LIST OF TABLES

Table 3.1. BRISK and SPIN results. The first grouping is a set of micro-benchmarks, the second are ported from related work, and THEQUE is our own case study. It takes at most 100 ms for BRISK to rewrite a program into its canonical sequentialization. 70

Table 4.1. CHAI benchmark results. 101

ACKNOWLEDGEMENTS

By a series of fortunate events, the past 6 years have graced me with the profound pleasures that are wise mentors, distinguished colleagues, and dear friends.

First, great thanks are due to my advisor, Ranjit Jhala. With his particular blend of patience, frankness, and enthusiasm, Ranjit has above all else instilled in me an invaluable sense of taste by which to understand and evaluate problems and their solutions. Thanks are due to Sorin Lerner, not only for his mentorship and cheap laughs at his expense, but also for stubbornly refusing to hang up the phone until I agreed to visit UCSD. I am additionally thankful to Todd Millstein, Jeffrey Remmel, and Geoff Voelker for serving on my committee. I had the distinct pleasure of working with Chris Hawblitzel, who offered precious tutelage early in my graduate career.

A source of nightmares is imagining a second go at graduate school, but without the great collaborators and mentors. Thank you to Pat Rondon and Ming Kawaguchi, not only for deftly bringing me up to speed on all things Liquid Types, but also for nurturing a light-hearted approach to curmodgeonery. Thanks to Ravi Chugh for patiently watching me stare at the whiteboard, vexed by my own mess, and for removing all obstacles to giving yoga a shot. To Zach Tatlock, thank you for all of the goofs and spoofs, nurturing my physical dependence on caffeine, and your sometimes downright frustrating committment to honest and logical dissection of my offhand remarks. To Klaus Gleissenthall and Rami Kıcı, thank you for collaborating on the bulk of the work in this dissertation.

The UCSD CSE community is a treasure trove of wonderful people whom I am fortunate to call friends. In particular, Dimitar Bounov, Neha Chachra, Alan Leung, Valentin Robert, Eric Seidel, Niki Vazou, and Panagiotis Vekris all made graduate school more than bearable.

It is impossible to overstate the overwhelming support of Mom, Dad, and Sis. Finally, thank you to Maya, who refuses to let my brain rest for even a moment, and positively ruins all of my bad moods.

Published Works Adapted in this Dissertation

Chapter 2, in part, is adapted from the material as it appears in Alexander Bakst, Klaus v. Gleissenthall, Rami Gökhan Kıcı, and Ranjit Jhala. Verifying Distributed Programs via Canonical Sequentialization. *Accepted for publication in the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2017*. The dissertation author was the primary investigator and author of this paper.

Chapter 3, in full, is adapted from the material as it appears in Alexander Bakst, Klaus v. Gleissenthall, Rami Gökhan Kıcı, and Ranjit Jhala. Verifying Distributed Programs via Canonical Sequentialization. *Accepted for publication in the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2017*. The dissertation author was the primary investigator and author of this paper.

Chapter 4, in full, is adapted from the material as it appears in Klaus v. Gleissenthall, Alexander Bakst, Rami Gökhan Kıcı, and Ranjit Jhala. Pretend Synchrony. *In submission to the ACM SIGPLAN International Conference on Principles of Programming Languages, POPL 2018*. The dissertation author was the primary investigator and author of this paper.

VITA

2008	Bachelor of Science, Massachusetts Institute of Technology
2009	Master of Engineering, Massachusetts Institute of Technology
2011–2017	Research Assistant, University of California, San Diego Diego
2017	Doctor of Philosophy, University of California, San Diego

PUBLICATIONS

Alexander Bakst, Klaus v. Gleissenthall, Rami Gökhan Kıcı, and Ranjit Jhala. Verifying Distributed Programs via Canonical Sequentialization. *Accepted for publication in the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2017*

Alexander Bakst and Ranjit Jhala. Predicate abstraction for linked data structures. In *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings*, pages 65–84, 2016

Niki Vazou, Alexander Bakst, and Ranjit Jhala. Bounded refinement types. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, pages 48–61, 2015

Patrick Maxim Rondon, Alexander Bakst, Ming Kawaguchi, and Ranjit Jhala. Csolve: Verifying C with liquid types. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, pages 744–750, 2012

Ming Kawaguchi, Patrick Maxim Rondon, Alexander Bakst, and Ranjit Jhala. Deterministic parallelism via liquid effects. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 45–54, 2012

ABSTRACT OF THE DISSERTATION

Sequentialization and Synchronization for Distributed Programs

by

Alexander Goldberg Bakst

Doctor of Philosophy in Computer Science

University of California, San Diego, 2017

Professor Ranjit Jhala, Chair

Distributed systems are essential for building services that can handle the ever increasing number of people and devices connected to the internet as well as the associated growth in data accumulation. However, building distributed programs is hard, and building confidence in the correctness of an algorithm or implementation is harder still. One fundamental reason is the highly asynchronous nature of distributed execution. Timing differences caused by network delays and variation in compute power can trigger behaviors that were unanticipated by the programmer.

Unfortunately, techniques for building confidence are all up against the same problem: the combinatorial explosion in the number of behaviors of a distributed system.

Testing and model checking techniques can not hope to weed out all behaviors when the state space is infinite. At the other end of the spectrum, constructing proofs by hand is a daunting task, as inductive invariants must simultaneously account not only for the algorithm-specific properties, but also the asynchronous nature of distributed computation.

In this dissertation, we take the view that many distributed programs are actually designed in such a way that the developer can safely reason about a small number of representative, *synchronous* executions. To support this claim, we first identify a property of message passing programs called *symmetric non-determinism*. Intuitively, the observable behavior of a program with symmetric non-determinism is insensitive to underlying network and processor non-determinism. Second, we develop an algorithm for transforming programs with symmetric non-determinism into an equivalent *sequential* program, called its canonical sequentialization. Our experiments show that canonical sequentializations can be computed quickly enough for eventual use in the design-implement-check cycle. Finally, we generalize the previous approach and present an algorithm for proving the correctness of distributed programs by automatically transforming them into an equivalent *synchronized* program. In this setting, rather than reasoning about message buffers, the developer writes loop invariants as *synchronous assertions*. We have implemented both of these algorithms as tools that demonstrate a reduction in the manual annotation burden for distributed systems.

Chapter 1

Introduction

Asynchronous distributed systems are notoriously hard to implement correctly as timing differences caused by network delays or variations in execution time may trigger behaviours that were neither intended nor anticipated by the programmer. In case the design of algorithms and protocols for concurrent, distributed execution was not already tricky business, ensuring the correctness of *implementations* tailored to a particular application adds even more complexity. One approach to mitigate the challenge of developing correct protocols and implementations is to construct workloads and stress tests to coax out corner cases hiding behind unanticipated schedules [19, 83, 37]. Unfortunately, these efforts are doomed to come up short. Due to infinite data domains and the fact that programs are often parameterized, *i.e.*, the number of participating nodes is not known at compile time, the number of possible schedules is infinite, making their enumeration a hopeless endeavor. As a result, programmers are left to guess, without any guarantee that their tests actually cover all errors.

A principled alternative is to construct mathematical proofs of correctness, in which one formally verifies that a system adheres to its specification. However, formal verification of parameterized systems is an entirely different kettle of fish. Verifying a system amounts to (1) defining a good set of executions, known as the *specification*, and then (2) proving that *all* of the executions of the system are contained within the specification. Often this takes the shape of an inductive argument, in which the program verifier (an algorithm or a human) attempts to discover a program *invariant*, which is a

set of states closed under all of the state transitions that the program can make. If the invariant contains the initial state of the program *and* is contained in the specification, then the system is guaranteed to satisfy its specification. Due to the *asynchronous* nature of communication, invariants must often explicitly enumerate possible schedules by case splitting over the joint global states of all participants. This makes the invariants both unwieldy and hard to come up with. Even worse, asynchronous invariants must reason about the contents of (unbounded) message buffers which renders even *checking* the correctness of candidate invariants undecidable.

1.1 Verification of Distributed Systems And Protocols

Specification in proof assistants. Specifying distributed algorithms requires extreme care. Machine-checkable proofs of correctness are essential for improving confidence in the correctness of a system, as even meticulously hand-checked proofs can be incorrect [84]. Thus, a number of approaches have been developed to aid in developing protocols and algorithms, and in particular proving their correctness. For example, PVS [57] has been used to verify Paxos [41], and and TLA+ [42] and TLAPS [14] have been used to specify and prove the correctness of Multi-Paxos [9]. This strategy imposes relatively few restrictions on the properties that can be specified and hence, verified. The trade-off is that the proof must be painstakingly constructed by hand: the specification in [9] is only 158 lines but the proof is 1136 lines. To mitigate this problem somewhat, Ivy [58] allows users to discover inductive invariants by examining counterexamples produced by bounded checking of algorithms specified in a custom modeling language. The language, RML, is restricted in order to guarantee that the verification conditions for (loop-free) programs fall into the effectively propositional fragment and are hence decidable [60].

Frameworks for verifying distributed systems. Another approach is to build custom program logics and frameworks specifically for reasoning about *and* constructing distributed systems, and then discharge proof obligations through prover backends.

The Verdi framework [81], which uses Coq [49] as a proof assistant, has been used to prove linearizability for Raft [82]. EventML [62], using the NuPRL proof assistant [13], has been used to verify the correctness of a Paxos implementation [67]. Recently, a type theoretic approach based on Hoare Type Theory [80] was proposed, which enables the user to *compose* different protocols and hence make correctness proofs modular. Related deductive techniques exploit advances in automated theorem proving to reduce the manual proof burden. Typically, the user provides an inductive invariant, which the system then uses to construct a proof of correctness. For example, the Dafny [44] language was used to build and verify a replicated state machine library based on Paxos in [31]. Dafny automatically proves the system correct, given substantial annotation by the user (about 9 lines of annotation for each line of code).

Programming model simplification. One of the principal obstacles to proving a system correct is the asynchronous nature of communication and execution. The techniques thus far require the programmer to reason about message buffers or to manually prove refinement of intermediate layers of abstraction between the specification and implementation. Thus, one alternative strategy is to design programming models specifically purposed for simplifying reasoning. The Heard-Of (HO) model, proposed in [11], introduced a new model of distributed computation in which communication is divided into *rounds*. Messages that arrive late are considered lost, and so the model effectively induces synchronous communication. The authors argue that this leads to simpler proofs. A logic for reasoning about the HO model is presented in [22] and a domain-specific language for developing and verifying programs in the HO model is described in [23]. The SCons language described in [46] is designed to be expressive enough to implement consensus algorithms such as Paxos while obtaining small *cutoff bounds*: finite systems whose correctness implies the correctness of the *parameterized* system. Finally, the work in [40, 38, 39] focuses on threshold-based fault tolerant distributed algorithms, which are algorithms that use *threshold guards* (*i.e.*, requirements that a message be received from at least t distinct processes). This approach models algorithms as *threshold automata*, which

```
process server(pid_set ps) {
  string x;

  for (p : ps) {
    x = recv();
  }
}

process client(pid server) {
  send(server, "ping");
}
```

Figure 1.1. Program `ManyPings`: A server collects `"ping"` messages from a set of clients.

use integer counters to track the number of processes in a given state, and hence can represent parameterized systems. These approaches reduce verifying a threshold automata to checking a finite set of representative executions that can be checked automatically using a satisfiability-modulo-theory (SMT) solver.

1.2 Toward Automated Verification

Ultimately, the challenge is to devise how to manage the complexity of the number of possible program executions in an asynchronous, concurrent system. One approach asks the user to provide summaries of the executions in the form of program invariants, the other restricts the problem domain to (small) finite systems. In either case, verification crucially depends on reasoning about the protocol structure, *i.e.*, which messages are destined for which processes, and the *contents* of messages. Thus, the verifier must reason simultaneously about message contents *and* messaging *structure*. Doing both at the same time complicates reasoning when proving even shallow properties.

Consider the program `ManyPings` in Figure 1.1, in which each member of an *unbounded* set `ps` of client processes asychronously send a `"ping"` message to a `server` process. The server uses a blocking `recv()` statement to retrieve the first available

message, binding it to `x`. How do we prove that the composition of the server and clients does not deadlock – that is, that the server does not get "stuck" at a particular `recv()`?

For even small instantiations of `ps`, the state space can grow quite large. For example, for a single client `p` in `ps`, either (a) `p` has not sent a message to the server yet, (b) it *has* and the message has not been received, or (c) the server has received `p`'s message. To make matters worse, in the general case we must explore the states resulting from the *order* in which messages queue up at the server. However, these details are irrelevant as the behavior of the server does not depend on the order in which it receives messages from the clients.

The intuition is as follows. We can partition the clients into two *disjoint* sets: those whose messages the server has yet to receive and those who the server has heard from. Initially, the latter set is empty. On each iteration of the loop, a single process "moves" from the first set to the second set, since the union of the two sets must equal `ps` and the server could not have received a message from the latter set.

The ability to partition the clients in this way is a consequence of exploiting the structure of the (implicit) messaging protocol, in which the clients only send messages to the server (and not to each other, for example). Having partitioned the clients in this way, we then summarize the interaction of the server with an *unbounded* number of clients by considering an arbitrary process in isolation. Remarkably, this strategy resembles techniques for automated reasoning about heap-manipulating programs, which we call next.

1.2.1 Iterating over a list

Consider the short program in Figure 1.2. The program is a function `print_msgs` that takes a pointer to a linked list of tasks, each a message to be printed only once. The function asserts that the message at the head of the list has not been printed, and then proceeds to print the message string with `printf()`. To mark that the task has been processed, the `done` field is set to `1`. Finally, the remaining messages are printed by a

```
struct list {
  int done;
  char *msg;
  struct list *next;
}

void print_msgs(list *p) {
  if (p != null) {
    assert(!p->done);
    printf(p->msg);
    p->done = 1;
    print_msgs(p->next);
  }
  return;
}
```

Figure 1.2. Printing a list of messages.

recursive call of `print_msgs()` on `p->next`, the tail of the list.

Specification. We would like to ensure that on return, every node reachable from `p` has been processed (*i.e.*, its `done` field is set to `1`). As the program will crash if the function encounters an element whose `done` field is set to `1` *before* the message is printed, the specification should additionally *require* that on entry to `print_msgs()`, every node reachable from `p` has its `done` field set to `0`.

Verification. To verify the correctness of `print_msgs()`, we first devise a finite representation or *abstraction* of the heap, as the input to `print_msgs()` is a list of unbounded size. One family of approaches (*e.g.*, [4, 79]) abstracts heaps by a composition of *disjoint* sub-heaps which are either (a) a precise description of values stored at a finite set of addresses or (b) a *summary* of an unbounded number of memory locations. These abstractions factor out the so-called "physical" description of the heap, describing the internal pointers of a data structure, and the "logical" description of the *values* stored within the data structure.

We simulate the effect of each statement in `print_msgs()` by manipulating the

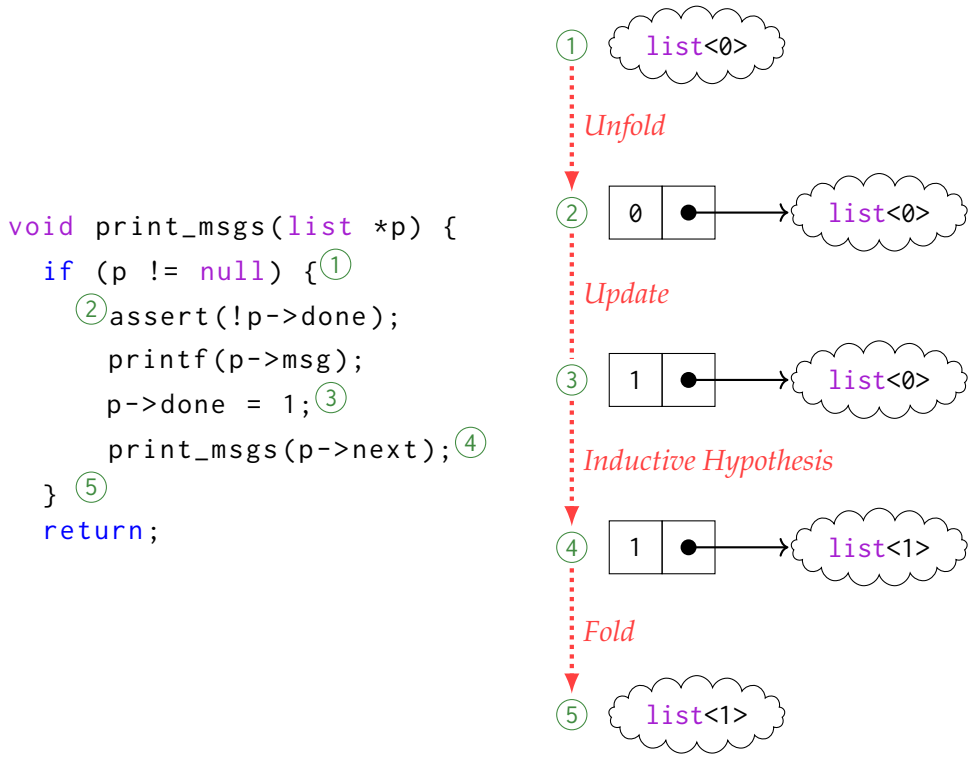
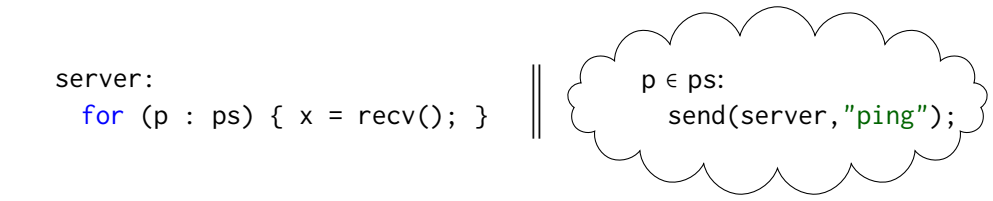


Figure 1.3. Closing a list of files with heap summaries corresponding to labeled program points.

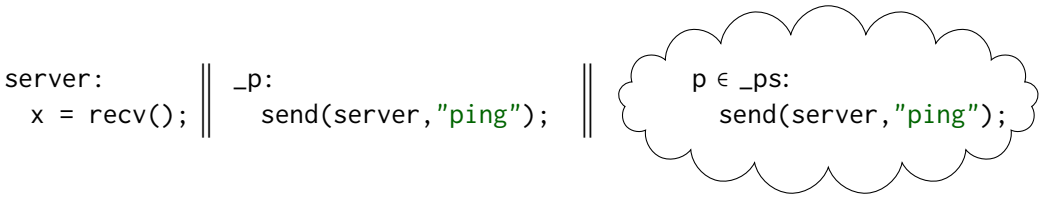
abstract heap accordingly. Figure 1.3 contains the `close_files()` function as before, in addition to several labeled program points. On the right is the abstraction of the heap rooted at `p` corresponding to each program point. On entry to the function, when `p` is not `null`, we assume that `p` points to a list of unprocessed tasks. We represent this by the abstract heap corresponding to program point 1, which denotes a collection of list nodes whose data values are equal to `0`.

Before the first dereference of `p` at the assertion at program point 2, we *unfold* the summary to reveal the head element of the list (represented by a box with two cells to denote contiguous addresses in memory), which is *disjoint* from the remainder of the list. Hence, after the update, the abstract heap at program point 3 records the fact that the *first* task is complete, but the remainder are *not*, as there is no danger of

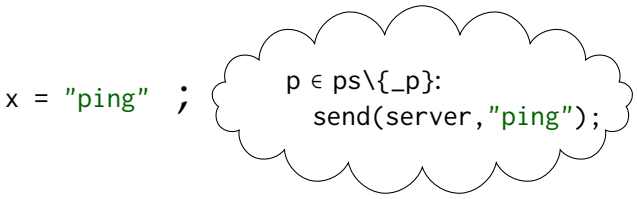
aliasing. Assuming inductively that `print_msgs()` obeys its specification, and since the heap rooted at `p->next` is a list of unprocessed tasks, the heap after the recursive call at point 4 is updated to contain a list of completed tasks. Finally, before exiting the `if` statement, we compute a new summary by *folding* the head node back into the rest of the list. Since the value of `done` at the head and all of the nodes in the tail is `0`, the new summary indicates that the list is completely processed.



(a) Initial summary.



(b) Unfold _p from _ps, where _ps ⊆ ps.



(c) Same final states as (b).

```
for (p:ps) { x = "ping"; }
```

(d) Same final states as (a) by generalizing the step from (b) to (c).

Figure 1.4. Simplification of `ManyPings`. The initial summary (a) says that each process is at its starting position. To simplify the loop, we consider an arbitrary subset of processes and unfold an arbitrary element in (b). The interaction between the server and _p is reaches the same states as the program in (c). Generalizing this reasoning produces (d).

1.2.2 Verifying Distributed Programs

The verification of `print_msgs()` considers a step in the recursive traversal of `p` corresponding to some *arbitrary* element of the list and generalizes the effect (marking the element "done") to the entire list. Let us now see how to verify `ManyPings` by employing a similar style of symmetric reasoning.

In distributed systems with no shared memory, most pairs of actions *commute* with each other, that is, given an initial state they result in the same state when executed in either order. A particular client's message is independent of all others: by an application of Lipton's theory of movers [45] we can consider only the program traces in which the message is received immediately after it is sent. However, this reasoning breaks down when there are *multiple* possible messages that could be received at a given program point. Our key insight is that, although there is non-determinism with respect to whose message the server will receive as its first message, for example, the states resulting from picking a particular client are merely permutations of each other [34]. This observation leads to a property of programs called *symmetric non-determinism*, which allows us to treat a single message exchange as representative of a set of possible exchanges. Without delving into the details of symmetric non-determinism, let us see how symmetric reasoning allows us to verify `ManyPings` in the same way as `print_msgs()`.

Specification. The specification of `ManyPings` is that it does not deadlock. In each iteration of its loop, the `server` should receive a message from some client. To verify that `ManyPings` meets its specification, we must show that in every halted state, every process has terminated successfully.

Verification. For the verification task we adopt the following strategy. We will iteratively apply proof rules that *rewrite* the program, transforming it into a simpler program – that is, a program with fewer behaviors. Each rewriting step preserves the *halted* states of the program We denote the parameterized program as the composition of the concrete server process and a group of client processes.

The initial proof state is shown in Figure 1.4a, which is the composition of the

server program and the set `ps` of client processes. The parallel bars (‖) separate distinct groups of processes. While there are many possible transitions in the concrete program, in the proof we focus first on the server's loop.

To rewrite the server's loop, we consider an execution of the iteration corresponding to some arbitrary process `_p` in the *subset* of the client processes that the server has not yet interacted with, `_ps`. *Unfolding* `_p` from `_ps` produces the new program in Figure 1.4b. Focusing on only the single `server` loop iteration and the process `_p`, we conclude that the program in Figure 1.4c deadlocks if Figure 1.4b does, since we can execute the server's receive immediately after the message is sent by `_p`. Symmetric non-determinism is critical here, as it allows us to argue that `_p` summarizes the possible behaviors of *all* clients.

We can then *generalize* the step from Figure 1.4b to Figure 1.4c to *every* iteration of the loop, producing the final program in Figure 1.4d. Finally, it is easy to see that Figure 1.4d does not get stuck as it contains no blocking operations. Because Figure 1.4d halts in the same states as the initial program, we conclude that the program is deadlock free.

1.2.3 Discussion

In the presence of aliasing (in the verification of `print_msgs()`), it would be unsound to perform the step labeled *Update* in Figure 1.3, in which only the description of the head tail is updated. The heap abstraction, together with the unfold and fold operations, maintains the critical invariant that, for any two nodes n_1 and n_2 belonging to a list summary, n_1`.next` $\neq$ n_2`.next`. Thus, the guaranteed separation between the head and the tail enable us to perform *local reasoning*, reasoning about updates to the head and the tail in isolation.

Likewise, there are many possible transitions from a given state in a concurrent system. In general, visiting *all* of the reachable states would involve *backtracking* in order to consider all of the enabled transitions from an initial state. Symmetric non-

determinism allows us to avoid backtracking in the step from Figure 1.4b to Figure 1.4c, and hence permits a form of local reasoning between a finite set of processes (as with the server and `_p`). The dual folding and unfolding steps preserve the symmetric non-determinism of the input program.

In both settings, the complexity of verification is managed by factoring out structure from the property to be proved. In the case of `print_msgs()`, the structure is the linked list itself. The nodes summarized as a `list<n>` are always organized as a proper linked list (there is no aliasing). The heap abstraction is updated only at locations corresponding to a single *concrete* memory address, allowing us to avoid the precision lost in the presence of aliasing. As for `ManyPings`, representing all of the possible senders by a *single* (albeit arbitrary) sender reduces an model with an unbounded number of a processes to a finite instance, allowing us to produce a simpler program by symbolic execution. We will see that symmetric non-determinism is the property that enables this reasoning.

1.3 Related Approaches to Automated Verification

While we have already discussed important work relating to distributed systems in, we now discuss how the ideas described in the previous section relate to, and borrow from, existing work which addresses message passing and model checking in general.

Session Types. There is tension between automation and strength of guarantee – this is by no means unique to the problem of distributed systems. However, the problem is certainly exacerbated by the complexities of the general execution model. While (mainstream) type systems are not expressive enough to prove full functional correctness, they do provide *guarantees* that certain classes of errors are absent at run time. *Session Types* [32, 33, 18, 10, 17] aim to bring these guarantees to developers of distributed systems. A session type is essentially a linearly typed resource denoting the protocol (or protocol fragment) executed by the corresponding typed term [78]. In general, the developer provides a *global type*, which corresponds to a specification of the

protocol. The relevant pieces of the specification are then *projected* out as types for each participant's program. If each process typechecks with respect to this projection, then the system as a whole is guaranteed to implement the protocol (and hence never get "stuck"). Session types have been implemented for mainstream programming languages such as Haskell [56], Go [54], and MPI [48].

Reduction. The ideas discussed in the previous section are based on Lipton's theory of movers [45], which designates certain program actions as either *left* or *right* movers, depending on how they commute with actions performed by concurrently executing threads. Left and right movers can be composed to produce a *single* action, resulting in a program with fewer possible interleavings of actions. This idea was further developed in [24] to build a system in which the user iteratively applies mover reasoning to *rewrite* a program into a simpler variant with fewer atomic actions. In a given trace, a receive commutes to the left of every action *except* the matching send. Thus, we can not reduce a program (by replacing the matching send and receive pair with an assignment) unless we know that (a) the receive will eventually be enabled (b) the matching send is the *unique* matching send.

Model Checking. Model checkers, which explore the reachable states of a system either exhaustively or up to some bound, are effective at finding unanticipated corner cases. Because they explore every reachable state, exhaustive coverage (and hence verification) is only possible with finite, (and in practice, small) systems. Nevertheless, model checking techniques offer a good tradeoff to developers by improving confidence in the correctness of a system while offering high degrees of automation. The Mace language [37] features a model checker and has been used to implement well known distributed systems such as Chord [72] and Pastry [64]. In contrast, MODIST [83] does not require modification to existing systems, and has been used to test systems such as Berkeley DB and MPS. Amazon recently published a report [53] on experiences applying TLA+ to some of its core services and uncovering design bugs using the TLC model checker.

Partial-Order Reduction. The verification problem amounts to checking if the set of program executions (or traces) is contained within the specification (a set of "good" traces). One way to simplify the task, then, is to *reduce* the set of program executions under consideration by arguing that certain traces are representative

For example, model checkers a combinatorial explosion in the number of reachable states. As a result, typically only instantiations with small values are suitable for model checking, and as such do not provide a proof of correctness. Mitigating the so-called *state-space explosion problem* has been a topic of copious research. One of the most well-known family of techniques is *partial-order reduction* [29, 28, 3]. Partial-order reduction methods explore representative traces in acyclic state spaces. These methods are based on the idea that often a concurrent or distributed system only defines a *partial ordering* of events. Thus, whenever there is no ordering between two given events, it is sufficient to explore an arbitrary representative one. This idea is embodied in the concept of *persistent sets* [29]. A persistent set, for a given state, is a *subset* of all enabled transitions (*i.e.*, those that can be executed in this state) such that (1) the transition cannot be disabled by other transitions and (2) it *commutes* with all transitions reachable through transitions that are *not* in the set. These techniques do not directly translate to the parameterized (or infinite-state) setting, as persistent sets would be unbounded in size. Although some work has addressed the challenge of infinite state systems [12, 77], these techniques focus on systems with a bounded number of processes.

Synchronization. As synchronous systems have fewer behaviors than their asynchronous counterparts, they are, in principle, easier to verify. Verification results for a program assuming synchronous communication do not necessarily carry over to the asynchronous mode in the general case. However, substantial research has targeted this problem to determine which properties can be verified, and under what circumstances, by considering only synchronized communication. A condition for transforming asynchronous finite-state programs into synchronous programs is presented in [6]. However, this condition excludes even our `ManyPings` example. Similarly, the reduction proposed

by [19] explores only system configurations where the message buffers are small, by eagerly executing receives. However, this is not guaranteed to visit all global state configurations, and can thus not be used to prove the absence of deadlocks, for example.

This problem has also been considered in the more specific context of (finite-state models of) MPI programs [68, 69, 71, 70]. If each message receive point has only one possible sender, then it is possible to explore all of the halted states of the program by only considering *synchronous* communication. Our approach incorporates this kind of reasoning. Crucially, symmetric non-determinism allows a process to receive from multiple sources.

Parameterized Verification. The model checking techniques discussed thus far are restricted to systems with a finite number of processes. In order to handle *parameterized* models, one generally constructs an *abstract model* that approximates the behavior of the concrete system. Each of the (finite number of) abstract states denotes a subset of the concrete program's state space. Counter abstractions [61] are a classic way to exploit symmetry by *counting* the number of processes in a given state. Recent work focuses on automatically inferring counting arguments in the form of counting automata [27] or by synthesizing descriptions of sets and referring to their cardinalities [76]. For verifying distributed systems, however, we often require tracking the *contents* of messages (*e.g.*, when they contain process identifiers that are used as arguments in later sends), which is challenging for counter-based approaches, which is challenging for counter-based approaches.

The notion of unfolding is related the idea of data type reductions [50] (itself an instance of abstract interpretation [15]). Here, the data domain is collapsed into a distinguished representative and the "rest," which summarize the remainder of the domain. Verification typically requires stating and proving lemmas to reclaim precision lost by creating these summaries.

There has been much work on inferring universally quantified invariants [7, 52, 66, 76]; despite recent progress, reliably synthesising both quantified invariant and

auxiliary state remains a challenge.

1.4 Contributions

The thesis of this dissertation is that the verification of distributed systems is simplified by automatically eliminating the non-determinism of the messaging system. To that end, the dissertation makes the following contributions.

1. We formalize a property of message passing programs called *Symmetric Non-determinism*, which enables *local* reasoning about actions;

2. We then describe *Canonical Sequentialization*, an algorithm for transforming a message passing program into its canonical sequentialization, which is a *sequential program* that preserves the final states of its input program. We evaluate canonical sequentialization by implementing it as a tool, BRISK, which compiles a HASKELL program to a I_S program and then computes its canonical sequentialization.

3. Finally, we present *Pretend Synchrony*, which generalizes the idea of canonical sequentialization to transform a message passing program into a *synchronized* program with coarse-grained concurrency. Pretend synchrony allows the user to treat an asynchronous program as if it were synchronous, which simplifies the annotations required to prove a program correct. We evaluate the approach on several case studies, including the classic two-phase commit protocol, Raft leader election, and single-decree Paxos.

We will see that canonical sequentizalizations can be computed fast enough to be integrated tightly into the design-implement-check cycle for modern, high-level programming languages. Moreover, by again exploiting symmetric non-determinism, we will see how to drastically simplify proofs of correctness by first automatically (and efficiently) computing the *synchronization* of a system, should a sequentialization not exist. We first describe a language, I_S, in Chapter 2 for formalizing *message passing programs*. In Chapter 3 we describe *Canonical Sequentialization* and in Chapter 4 we

discuss *Pretend Synchrony.* In Chapter 5 we propose ideas for future research based on the sequentialization and synchronization techniques discussed in the preceding chapters.

Chapter 2

Message Passing Programs

2.1 Overview

In the following chapters we will be discussing approaches to verifying properties of distributed systems. Programmers have at their disposal a rich set of languages and tools for constructing these systems. While they make programming convenient, features like (higher-order) procedures and dynamic process creation complicate analysis by obscuring both control flow and process structure.

To make progress, in this chapter we formalize $\mathrm{I_S}$, a core Imperative language for Symmetric reasoning. $\mathrm{I_S}$ programs are procedure-less (but have unbounded iteration) and feature explicit parallel composition rather than dynamic process creation. Crucially, $\mathrm{I_S}$ explicitly supports symmetric by allowing programs to be parameterized by *scalar sets* of values (such as process identifiers) [34], and including forms for iterating parallel and sequential composition. We shall see in Chapter 3 how this simplifies reasoning about message passing programs.

In this chapter, we present the syntax and semantics of $\mathrm{I_S}$, and then prove an essential theorem about reducing the complexity of exploring the *halted* states of $\mathrm{I_S}$ programs. In this chapter and those that follow we make heavy (ab)use of mathematical notation. $A[e/x]$ substitutes the variable x with the expression e. We let $\varnothing$ denote an empty map (in addition to the empty set) and write $m[x \leftarrow y]$ to denote the map m' that is equivalent to m at every point except x which is mapped to y.

Expressions	e	::=	c	*Constants*
		|	$i.x$	*Variables*
		|	X	*Parameters*
		|	$\texttt{f}(\overline{e})$	*Local Computation*
Statements	s	::=	$i.x \coloneqq e$	*Assignment*
		|	$i.x \coloneqq \texttt{recv}(t,\texttt{w})$	*Message Receive*
		|	$\texttt{send}(t,e,e)$	*Message Send*
		|	`break`	*Loop Exit*
Programs	A	::=	`skip`	*No Behavior*
		|	$[s]_x$	*Singleton Program*
		|	$A\,;A$	*Sequential Composition*
		|	$\texttt{if}\ e\ \{\ A\ \}\ \texttt{else}\ \{\ A\ \}$	*If Else*
		|	$\texttt{for}\ (x : X)\ \{\ A\ \}$	*Iteration*
		|	$\texttt{while true}\ \{\ A\ \}$	*Looping*
		|	$A \parallel A$	*Parallel Composition*
		|	$\bigsqcap_{i:e} A$	*Iterated Parallel Composition*
		|	$\bigsqcup_{i:e} A$	*Nondeterministic Value*

$\texttt{w} \in \{e, *\}$ $\quad i, x \in \mathsf{Identifiers}$ $\quad X \in \mathsf{ParamIdentifiers}$ $\quad t \in \mathsf{TypeIdentifiers}$

Figure 2.1. $\mathrm{I_S}$ Syntax.

2.2 Syntax and Semantics

2.2.1 Syntax

Expressions. $\mathrm{I_S}$ is designed to be parametric in the language of purely local computations. We assume that the local process state at least includes integers and process identifiers (PIDs). Primitive expressions in $\mathrm{I_S}$ are constants c and variables $i.x$. In $\mathrm{I_S}$, each process is associated with a unique process identifier (PID), which serves as an address for sending messages. Variables are qualified by the PID of their owning process, *i.e.*, $i.x$ refers to i's variable x. When this is clear from context or unimport, we often drop the prefix and just write x. We assume disjoint sets of identifiers to name program variables, parameters, and types. The language of expressions includes $\texttt{f}(\overline{e})$, which denotes a purely local computation – that is, a computation that does not send or receive any messages.

$$[s]_p : \checkmark \qquad \texttt{skip} : \checkmark$$

$$\frac{A : \checkmark \qquad B : \checkmark \qquad |\mathsf{Procs}(A) \cup \mathsf{Procs}(B)| \leq 1 \qquad \star \in \{;, \square\}}{A \star B : \checkmark}$$

$$\frac{A : \checkmark}{\texttt{for (}x : X\texttt{) \{ }A\texttt{ \}} : \checkmark} \qquad \frac{A : \checkmark}{\texttt{while true \{ }A\texttt{ \}} : \checkmark}$$

$$\frac{A : \checkmark \qquad B : \checkmark \qquad \mathsf{Procs}(A) \cap \mathsf{Procs}(B) = \varnothing}{A \parallel B : \checkmark}$$

$$\frac{A : \checkmark}{\bigsqcap_{x:X} A : \checkmark} \qquad \frac{A : \checkmark}{\bigsqcup_{x:X} A : \checkmark}$$

Figure 2.2. Definition of $A : \checkmark$.

$$\mathsf{Procs}(A) \triangleq \begin{cases} \varnothing & A = \texttt{skip} \\ \{p\} & A = [s]_p \\ \mathsf{Procs}(B) \cup \mathsf{Procs}(C) & A = B; C \text{ or } A = B \square C \\ \mathsf{Procs}(B) & A = \texttt{for (}x : X\texttt{) \{ }B\texttt{ \}} \text{ or } A = \texttt{while true \{ }B\texttt{ \}} \\ \mathsf{Procs}(B) & A = \bigsqcap_{x:X} B \text{ or } A = \bigsqcup_{x:X} B \end{cases}$$

Figure 2.3. Definition of $\mathsf{Procs}(A)$

Programs. We use `skip` to denote the empty process, which does nothing. The process $[s]_p$ is a singleton that denotes the process with PID p executing statement s. Processes are further constructed by sequential ($P; Q$) and parallel composition ($P \parallel Q$), and when a single process executes an entire program, we abuse notation and abbreviate, *e.g.*, $[s_1]_p ; [s_2]_p$ as $[s_1; s_2]_p$ or `for (`$x : X$`) {` $[s_1]_p ; [s_2]_p$ `}` as $[\texttt{for (}x : X\texttt{) \{ }s\texttt{ \}}]_p$. We write `for (`$i : I$`) {` A `}` and $\bigsqcap_{i:I} A$ to denote the sequential and parallel instantiations of A to the values in I, respectively of A to the values in I, respectively.

Normal Form. A program is in normal form, written $A : \checkmark$, if it consists of parallel compositions of sequences of statements from distinct processes. The definition

of $A : \checkmark$ is found in Figure 2.2.

Example 2.1. Consider A and B defined below.

$$A \triangleq [s_1]_p \parallel \prod_{q:Q} [s_2]_q \qquad\qquad B \triangleq [s_3]_p \parallel \prod_{q:Q} [s_4]_q \parallel [s_5]_m$$

A and B are both in normal form, but $A; B$ and $A \parallel B$ are not.

Composition. We define $A \ltimes B$ to be the process-wise composition of A and B.

$$A \ltimes B \triangleq \begin{cases} \prod_{x:X} C; D & A = \prod_{x:X} C,\ B = \prod_{x:X} D, \textit{ and } \mathsf{Procs}(C) = \mathsf{Procs}(D) = \{x\} \\ A; B & \mathsf{Procs}(A) \cap \mathsf{Procs}(B) \neq \varnothing \\ A \parallel B & \textit{otherwise} \end{cases}$$

Figure 2.4. Definition of $A \ltimes B$.

Intuitively, $A \ltimes B$ sequences the statements in B after the corresponding statements in A, assuming $A : \checkmark$ and $B : \checkmark$. The definition is found in Figure 2.4.

Example 2.2. Recall A and B defined in Example 2.1. Their composition is given by

$$A \ltimes B = [s_1; s_3]_p \parallel \prod_{q:Q} [s_2; s_4]_q \parallel [s_5]_m$$

2.2.2 Semantics

Next we discuss the semantics of $\mathrm{I_S}$. The operational semantics are shown in Figure 2.7. First we discuss the semantics at a high level, and then formally define the properties that will be of interest in this dissertation.

Message Channels and Ordering. In $\mathrm{I_S}$, processes communicate by sending and receiving messages over *typed* channels: There is a separate channel for each (ordered) pair of processes, and each channel is split into sub-channels for different message types. Messages on the same sub-channel are delivered in order, whereas there are no

guarantees for messages sent on separate sub-channels. Message delivery can be delayed arbitrarily, but messages are never dropped or duplicated. These semantics are modeled after the guarantees provided by high-level languages such as ERLANG.

Sends and Receives. The process $[\texttt{send}(t, q, e)_\ell]_p$ asynchronously enqueues message e in the sub-channel (p, q, t). The tag ℓ is a unique identifier for the *syntactic* send statement. We denote a set of tags as $\overline{\ell}$. Sets of tags are used to annotate receive statements, but have no effect on the semantics. Dually, if the sub-channel (p, q, t) is not empty, then $\left[q.x \coloneqq \texttt{recv}(t, p)_{\overline{\ell}}\right]_q$ dequeues the first message on the channel and binds it to x_q; otherwise, the process blocks. $\left[q.x \coloneqq \texttt{recv}(t, I)_{\overline{\ell}}\right]_q$ blocks until there is some non-empty sub-channel (i, q, t) and then behaves as $\left[q.x \coloneqq \texttt{recv}(t, i)_{\overline{\ell}}\right]_q$. The form $\left[q.x \coloneqq \texttt{recv}(t, I)_{\overline{\ell}}\right]_q$ behaves similarly except the sub-channel is constrained to be an element of I.

Example 2.3. Consider the processes A and B defined below:

$$A \triangleq [\texttt{send}(t, r, v_0); \texttt{send}(t, r, v_1)]_p \parallel [r.x \coloneqq \texttt{recv}(t, *); r.y \coloneqq \texttt{recv}(t, *)]_r$$
$$B \triangleq [\texttt{send}(t, r, v_0)]_p \parallel [\texttt{send}(t, r, v_1)]_q \parallel [r.x \coloneqq \texttt{recv}(t, *); r.y \coloneqq \texttt{recv}(t, *)]_r$$

In A, both sends are executed by the process p, and hence the messages are sent on the same sub-channel. They are therefore delivered to r in order, and on termination we always have $r.x = v_0$ and $r.y = v_1$. On the other hand, in B, the sends are executed by *different* processes and hence the messages may be delivered in any order. On termination, *either* $r.x = v_0$ and $r.y = v_1$ *or* $r.x = v_1$ and $r.y = v_0$.

Operational Semantics. Definitions required for the semantics of $\mathrm{I_s}$ are shown in Figure 2.6, and the semantics itself is in Figure 2.7. A configuration $\mathcal{C}$ is either a triple of a program store, buffer, and program (σ, μ, P) or the distinguished `crash` denoting an error state. The semantics is defined by the relation $\mathcal{C} \xrightarrow{T}_{\mathrm{I_s}} \mathcal{C}'$ where T is a trace (which we also call a *transition*). The relation $A \equiv B$, shown in Figure 2.5 is used to perform trivial reductions: if $A \xrightarrow{T}_{\mathrm{I_s}} B$ and $B \equiv B'$, then $A \xrightarrow{T}_{\mathrm{I_s}} B'$ (and similarly if $B \xrightarrow{T}_{\mathrm{I_s}} A$). Finally, we

$$\texttt{skip}; A \equiv A \qquad A; \texttt{skip} \equiv A \qquad \prod_{p:P} \texttt{skip} \equiv \texttt{skip}$$

$$\prod_{p:P} A; \texttt{skip} \equiv A \qquad A \parallel B \equiv B \parallel A \qquad A \parallel (B \parallel C) \equiv (A \parallel B) \parallel C$$

Figure 2.5. Congruence relation $A \equiv B$

Stores	σ	$\in$	$(PID \times \mathsf{Identifiers}) \uplus \mathsf{ParamIdentifiers} \rightharpoonup \mathsf{Value}$
Buffers	μ	$\in$	$PID \times PID \times \mathsf{TypeIdentifiers} \rightharpoonup \overline{\mathsf{Value}}$
Configurations	$\mathcal{C}$	$\in$	$(\mathsf{Stores} \times \mathsf{Buffers} \times \mathsf{Programs}) \uplus \texttt{crash}$
Traces	T	$::=$	$\varepsilon \mid [s]_p ; T$

Figure 2.6. $\mathrm{I_S}$ Semantic Definitions

will be discussing reachable configurations and enabled transitions in the remainder of the dissertation.

Definition 2.1 (Enabled)**.** A transition T is *enabled* in configuration $\mathcal{C}$ if and only if there exists $\mathcal{C}'$ such that $\mathcal{C} \xrightarrow{T}_{\mathrm{I_S}} \mathcal{C}'$.

Definition 2.2 (Reach Set)**.** We define $\mathsf{Reach}(\mathcal{C})$ to be the reachable configurations of $\mathcal{C}$:

$$\mathsf{Reach}(\mathcal{C}) \triangleq \left\{ \mathcal{C}' \mid \exists T.\, \mathcal{C} \xrightarrow{T}_{\mathrm{I_S}} \mathcal{C}' \right\}$$

Parameterized Programs. Programs are parameterized by disjoint sets of PIDs, denoting finite but unbounded collections of process identifiers. To discuss properties of these programs, we first discuss valid *instantiations* of parameterized programs. An instantiation of a program A is a program store σ that gives a valuation to all of the PID and PID set-valued variables in A: we call these *initial stores*.

Definition 2.3 (Initial Stores)**.** The store σ_0 is an initial store of A if

1. For all $X \in \mathsf{PidSetParams}(A)$, $|\sigma(X)| > 0$;

2. For all distinct $x, y \in \mathsf{PidParams}(A)$, $\sigma(x) \neq \sigma(y)$; and

3. For all distinct $X, Y \in \mathsf{PidSetParams}(A)$, $\sigma(X) \cap \sigma(Y) = \varnothing$.

We lift the definition of initial stores to define the initial configurations of A:

$$\mathsf{Initial}(A) \triangleq \{(\sigma, \varnothing, A) \mid \sigma \textit{ is an initial store of } A\}$$

Properties. A property φ is a formula over a program store: a store σ satisfies φ (we write $\sigma \vDash \varphi$) if and only if φ is true when its free variables are given their valuation from σ. Satisfaction is lifted to configurations in the obvious way.

Halting properties. In this dissertations we are concerned with verifying *halting properties* of programs, that is, properties over halted states. Intuitively, the halting property of A is the set of halted configurations reachable from the initial configurations of A, where a configuration is halted if its reachable set is empty.

Definition 2.4 (Halted Configurations)**.** We define the halted configurations of A, written $\mathsf{Halted}(A)$, as

$$\mathsf{Halted}(A) \triangleq \{\mathcal{C} \mid \mathcal{C} \in \mathsf{Initial}(A) \textit{ and } \mathsf{Reach}(\mathcal{C}) = \varnothing\}$$

Examples of halting properties range from local assertion safety to global properties such as deadlock freedom or agreement as in the two-phase commit protocol. We write $P \vDash_H \varphi$ to denote that φ holds in all halted configurations reachable from the initial configuration.

Definition 2.5 (Halting Properties)**.** We say that φ is a halting property of A, written $A \vDash_H \varphi$ if

$$\forall \mathcal{C} \in \mathsf{Halted}(A).\ \mathcal{C} \vDash \varphi$$

2.3 Tagged Programs

As described in Section 2.2.1, sends and receives are annotated with tags and sets of tags, respectively. The intention is that a receive's tag set corresponds to the set of

syntactic send statements whose messages may be received.

Definition 2.6. A program A is *tagged consistently* if for all initial stores σ_0, whenever

$$(\sigma_0, \varnothing, A) \xrightarrow{T}_{\mathrm{I_S}} (\sigma, \mu, B) \xrightarrow{\left[\mathtt{recv}(t,p)_{\overline{\ell}}\right]_q}_{\mathrm{I_S}} (\sigma', \mu', C)$$

then, letting $\mu(p, q, t) = v^{\ell} \cdot v'$, $\ell \in \overline{\ell}$.

We will see in Chapter 3 that being able to determine the possible senders for a given receive is essential for reducing the complexity of message passing programs.

2.4 Reductions for $\mathbf{I_S}$

Ultimately, reasoning about the behavior of concurrent systems is complicated because the *behavior* program, that is, its set of program executions or traces, by definition includes *all possible* interleavings of actions by each of constituent threads or processes. However, in practice, we are often interested in only a subset of the program's behaviors, *e.g.*, a property of the program's *halting* executions. Recalling Example 2.3, the non-determinism of the message delivery system is observed as different final values for the variables x and y, depending on whether p or q's message is delivered first. On the other hand, consider C below, modified from B by a deft change of sub-channels:

$$C \triangleq \left[\mathtt{send}(t_0, r, v_0)\right]_p \parallel \left[\mathtt{send}(t_1, r, v_1)\right]_q \parallel \left[r.x \coloneqq \mathtt{recv}(t_0, *); r.y \coloneqq \mathtt{recv}(t_1, *)\right]_r$$

The traces of B' exhibit multiple interleavings of the statements in p, q, r: for example, r's receive may execute as soon as p has sent its message or r may wait until both messages have been sent.

On the other hand, B' has only a single *halted* configuration, in which $x_r = v_0$ and $y_r = v_1$. This is because the first receive in r *commutes* with q's send: given an initial state, executing both statements in either order results in equivalent states. Considering only halted configurations, then, we would not be able to distinguish C from D (which is not

in normal form):

$$D \triangleq [\texttt{send}(t_0, r, v_0)]_p ; [\texttt{send}(t_1, r, v_1)]_q ; [r.x \coloneqq \texttt{recv}(t_0, *)]_r ; [r.y \coloneqq \texttt{recv}(t_1, *)]_r$$

or indeed even from E:

$$E \triangleq [\texttt{send}(t_0, r, v_0)]_p ; [r.x \coloneqq \texttt{recv}(t_0, *)]_r ; [\texttt{send}(t_1, r, v_1)]_q ; [r.y \coloneqq \texttt{recv}(t_1, *)]_r .$$

If we focus our attention on halted configurations, then it is impossible to distinguish two executions that start in some configuration $\mathcal{C}$ and terminate in the configuration $\mathcal{C}'$. For example if $\mathcal{C} \xrightarrow{t_0;t_1}_{\mathrm{I_s}} \mathcal{C}'$ and $\mathcal{C} \xrightarrow{t_1;t_0}_{\mathrm{I_s}} \mathcal{C}'$, then t_0 and t_1 *commute* with respect to each other. We can exploit the fact that some transitions commute in order to build *simpler* programs (*i.e.*,, have *fewer* traces) that have the same *observational* behavior. We reduce complexity by *eliminating* program interleavings, *i.e.*, by replacing parallel composition with *sequential* composition. Intuitively, if two statements s_1 and s_2 in different processes commute, then we can consider only those executions in which, say, s_1 occurs before s_2. We formalize this intuition by defining *left movers*, after [24, 45].

We define a notion of *left movers*. Intuitively, a left mover *in a configuration* is an enabled transition T_λ that, given a sequence T of transitions, (1) remains enabled after T, (2) commutes to the left of T, and (3) if T leads to a crash, then running T_λ *first* still leads to a crash (note that T should therefore not contain T_λ).

Our definition closely resembles that of *ample* or *persistent sets* [29, 59], however we additionally require that T_λ can be preprended to a trace that results in a crash.

Definition 2.7 (Left Movers)**.** Let $\mathcal{C}$ be a configuration and $[s]_p$ be a transition enabled in $\mathcal{C}$. Then $[s]_p$ is a *left mover in* $\mathcal{C}$ if:

1. For all $\mathcal{C}' \neq \texttt{crash}$, $[s]_p$, if $\mathcal{C} \xrightarrow{T;[s]_p}_{\mathrm{I_s}} \mathcal{C}'$ then $\mathcal{C} \xrightarrow{[s]_p;T}_{\mathrm{I_s}} \mathcal{C}'$; and
2. For all T not containing $[s]_p$, if $\mathcal{C} \xrightarrow{T}_{\mathrm{I_s}} \mathcal{C}'$ and $\mathcal{C}'$ is halted, then either $[s]_p$ is enabled in $\mathcal{C}'$ or $\mathcal{C}' = \texttt{crash}$

3. If $\mathcal{C} \xrightarrow{T}_{\mathrm{I_s}} \texttt{crash}$ and T does not contain $[s]_p$, then $\mathcal{C} \xrightarrow{[s]_p;T}_{\mathrm{I_s}} \texttt{crash}$.

Theorem 2.1 says that halted states are preserved by executing left movers in a particular configuration. This theorem is a straightforward consequence of our definition (indeed, this result is the reason for our particular definition).

Theorem 2.1 (Left Mover Reduction)**.** *Assume $[s]_p$ is a left mover in configuration $\mathcal{C}$. Then*

$$\mathsf{Halted}(\mathcal{C}) = \bigcup \left\{ \mathsf{Halted}(\mathcal{C}_s) \mid \mathcal{C} \xrightarrow{[s]_p}_{\mathrm{I_s}} \mathcal{C}_s \right\}$$

Proof. The theorem is a consequence of Definition 2.7. Suppose $\mathcal{C} \xrightarrow{T}_{\mathrm{I_s}} \mathcal{C}'$ where $\mathcal{C}'$ is halted. Further suppose that $[s]_p$ is enabled in $\mathcal{C}$. There are two cases to consider, depending on if T contains $[s]_p$.

1. Case $T = T_0; [s]_p; T_1$ for some T_0 and T_1:
   Then, by Definition 2.7, $\mathcal{C} \xrightarrow{[s]_p;T_0;T_1}_{\mathrm{I_s}} \mathcal{C}'$, and hence $\mathcal{C} \xrightarrow{[s]_p}_{\mathrm{I_s}} \mathcal{C}''$ and $\xrightarrow{T_0;T_1}_{\mathrm{I_s}} \mathcal{C}'$ by inversion. Thus, $\mathcal{C}' \in \mathsf{Halted}(\mathcal{C}'')$.

2. Case $T \neq T_0; [s]_p; T_1$ for all T_0 and T_1:
   Then, $\mathcal{C}' = \texttt{crash}$ since otherwise $[s]_p$ would be enabled and hence $\mathcal{C}'$ wouldn not be halted. Then, by Definition 2.7, $\mathcal{C} \xrightarrow{[s]_p;T}_{\mathrm{I_s}} \mathcal{C}'$ and hence $\mathcal{C} \xrightarrow{[s]_p}_{\mathrm{I_s}} \mathcal{C}'' \xrightarrow{T}_{\mathrm{I_s}} \mathcal{C}'$ Thus, $\mathcal{C}' \in \mathsf{Halted}(\mathcal{C}'')$.

□

As a consequence of Theorem 2.1, it is easy to see that sends and enabled receives, for whom there is only one possible sender, are left movers. To explore reachable halted states, it suffices to execute these statements first, without backtracking to explore the states reachable from co-enabled statements.

Fact 2.1. *Given $\mathcal{C} = (\sigma, \mu, A \parallel [\,\texttt{send}(t,q,v)\,]_p; s)$, if $A \parallel [\,\texttt{send}(t,q,m)\,]_p; s : \checkmark$ then*

$$\mathsf{Halted}(\mathcal{C}) = \mathsf{Halted}((\sigma, \mu[(p,q,t) \leftarrow \mu(p,q,t) \cdot v], A \parallel s))$$

Fact 2.2. *Given* $\mathcal{C} = (\sigma, \mu, A \parallel [p.x \coloneqq \texttt{recv}(t,q)]_p ; s)$*, if* $A \parallel [p.x \coloneqq \texttt{recv}(t,q)]_p ; s : \checkmark$ *and* $\mu(q,p,t) = v \cdot v'$*, then*

$$\mathsf{Halted}(\mathcal{C}) = \mathsf{Halted}((\sigma[(p,x) \leftarrow v], \mu[(q,p,t) \leftarrow v'], A \parallel s))$$

2.5 Summary

In this chapter we have described $\mathrm{I_S}$, a first order language which will serve as the intermediate verification language in the remainder of the dissertation. Importantly, we have given a $\mathrm{I_S}$ a semantics that allows us to connect in-flight messages to their syntactic point of origin, which is crucial to the approaches describeed later. Finally, we have defined our own notion of left movers. Because left movers commute to the left in any trace in which they appear, to explore the halted states reachable from a given configuration, one need only explore the halted states reachable *after* executing the left movers enabled in the current configuration. This observation will be exploited in the two chapters that follow.

Acknowledgments

Chapter 2, in part, is adapted from the material as it appears in Alexander Bakst, Klaus v. Gleissenthall, Rami Gökhan Kıcı, and Ranjit Jhala. Verifying Distributed Programs via Canonical Sequentialization. *Accepted for publication in the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2017*. The dissertation author was the primary investigator and author of this paper.

ASSIGNMENT

$$\frac{\mathsf{eval}(\sigma, e) = v}{(\sigma, \mu, [i.x := e]_j) \xrightarrow{[i.x:=e]_j}_{\mathrm{I_s}} (\sigma[i.x \leftarrow v], \mu, \mathtt{skip})}$$

RECEIVE

$$\frac{\mu[(p,q,t)] = v^\ell \cdot v' \qquad \mathsf{w} = * \textit{ or } \mathsf{eval}(\sigma, \mathsf{w}) = p \textit{ or } p \in \mathsf{eval}(\sigma, \mathsf{w})}{(\sigma, \mu, [i.x := \mathtt{recv}(t, \mathsf{w})_{\overline{\ell}}]_q) \xrightarrow{[\mathtt{recv}(t,p)_{\overline{\ell}}]_q}_{\mathrm{I_s}} (\sigma, \mu[(p,q,t) \leftarrow v'], [i.x := v]_q)}$$

SEND

$$\frac{\mathsf{eval}(\sigma, e_0) = q \qquad \mathsf{eval}(\sigma, e_1) = m}{(\sigma, \mu, [\mathtt{send}(t, e_0, e_1)_\ell]_p) \xrightarrow{[\mathtt{send}(t,e_0,e_1)_\ell]_p}_{\mathrm{I_s}} (\sigma, \mu[(p,q,t) \leftarrow \mu[(p,q,t)] \cdot m^\ell], \mathtt{skip})}$$

CHOICE

$$\frac{R = P \textit{ or } R = Q}{(\sigma, \mu, P \,\square\, Q) \xrightarrow{\varepsilon}_{\mathrm{I_s}} (\sigma, \mu, R)}$$

SEQUENCE

$$\frac{(\sigma, \mu, A) \xrightarrow{T}_{\mathrm{I_s}} (\sigma', \mu', \mathtt{skip})}{(\sigma, \mu, A\,;B) \xrightarrow{T}_{\mathrm{I_s}} (\sigma', \mu', B)}$$

ITERATION

$$\frac{\overline{v_i} \in \mathsf{Permute}(\mathsf{eval}(\sigma, e))}{(\sigma, \mu, \mathtt{for}\ (i : e)\ \{\ A(i)\ \}) \xrightarrow{\varepsilon}_{\mathrm{I_s}} (\sigma, \mu, A(v_0)\,;\cdots;A(v_i)\,;\cdots)}$$

PARALLEL COMPOSITION

$$\frac{(\sigma, \mu, A) \xrightarrow{T}_{\mathrm{I_s}} (\sigma', \mu', A')}{(\sigma, \mu, A \parallel B) \xrightarrow{T}_{\mathrm{I_s}} (\sigma', \mu', A' \parallel B)}$$

ITERATED COMPOSITION

$$\frac{\overline{v_i} = \mathsf{eval}(\sigma, e)}{(\sigma, \mu, \bigsqcap_{i:e} A(i)) \xrightarrow{\varepsilon}_{\mathrm{I_s}} (\sigma, \mu, \cdots \parallel A(v_i) \parallel \cdots)}$$

WHILE-EXPAND

$$\frac{}{(\sigma, \mu, \mathtt{while\ true\ \{}\ A\ \mathtt{\}}) \xrightarrow{\varepsilon}_{\mathrm{I_s}} (\sigma', \mu', A; \mathtt{while\ true\ \{}\ A\ \mathtt{\}})}$$

WHILE-EXIT

$$\frac{}{(\sigma, \mu, \mathtt{break; while\ true\ \{}\ A\ \mathtt{\}}) \xrightarrow{\varepsilon}_{\mathrm{I_s}} (\sigma', \mu', \mathtt{skip})}$$

NONDETERMINISM

$$\frac{v \in \mathsf{eval}(\sigma, e)}{(\sigma, \mu, \bigsqcup_{i:e} A(i)) \xrightarrow{\varepsilon}_{\mathrm{I_s}} (\sigma, \mu, A(v))}$$

ASSERT-TRUE

$$\frac{\mathsf{eval}(\sigma, e) = \mathtt{true}}{(\sigma, \mu, [\mathtt{assert}(e)]_p) \xrightarrow{\varepsilon}_{\mathrm{I_s}} (\sigma, \mu, \mathtt{skip})}$$

ASSERT-FALSE

$$\frac{\mathsf{eval}(\sigma, e) = \mathtt{false}}{(\sigma, \mu, [\mathtt{assert}(e)]_p \parallel P) \xrightarrow{\varepsilon}_{\mathrm{I_s}} \mathtt{crash}}$$

ASSUME

$$\frac{\mathsf{eval}(\sigma, e) = \mathtt{true}}{(\sigma, \mu, [\mathtt{assume}(e)]_p) \xrightarrow{\varepsilon}_{\mathrm{I_s}} (\sigma, \mu, \mathtt{skip})}$$

TRANSITIVE

$$\frac{\mathcal{C}_1 \xrightarrow{T_1}_{\mathrm{I_s}} \mathcal{C} \qquad \mathcal{C} \xrightarrow{T_2}_{\mathrm{I_s}} \mathcal{C}_2}{\mathcal{C}_1 \xrightarrow{T_1;T_2}_{\mathrm{I_s}} \mathcal{C}_2}$$

Figure 2.7. $\mathrm{I_s}$ Semantics: $\mathcal{C} \xrightarrow{T}_{\mathrm{I_s}} \mathcal{C}$

Chapter 3

Canonical Sequentializations

Concurrent and distributed message passing programs have remained viciously difficult to implement, as the programmer gets little feedback about the correctness of their system *at development time*. For classical single process applications, modern type systems and IDEs can provide instantaneous feedback about whether or not the individual parts compose correctly. The distributed systems developer on the other hand is bereft of development- and compile-time tools that guarantee the absence of concurrency errors.

In this chapter, we describe *canonical sequentialization*, a new approach to verifying concurrency properties of unbounded, asynchronous, message-passing programs at compile-time. Our approach builds upon the following observation: due the combinatorial explosion in complexity, programmers do not reason about their systems by case-splitting over all the possible execution orders. Instead, correct programs tend to be *well-structured* so that the programmer can reason about a small number of representative executions which we call the program's *canonical sequentialization*.

Consider an implementation of MapReduce [16] which consists of (1) a number of *worker* processes that perform map/reduce tasks, (2) a *queue* process that distributes work, and (3) a *master* process that orchestrates the entire computation. Workers query the queue for assignments, perform the assigned task, and then send the results to the master; the queue waits for a request and answers with a work assignment; the master waits for results (see Figure 3.12).

The apparently highly concurrent MapReduce protocol has a canonical sequentialization. All the races are symmetric: even though worker threads compete for work assignments, the states that result from picking a particular worker are equivalent modulo a shuffling of process identifiers. Similarly, even though the order in which results reach the master is non-deterministic, the resulting states are symmetric. Thus, instead of reasoning about the original distributed program, we can reason about its canonical sequentialization, which first sequentially assigns all tasks to the workers, and then passes the results to the master.

In this chapter we describe the following contributions.

1. **Symmetric Non-Determinism & Canonical Sequentialization.** Our first contribution is to identify and formalize a property of message passing programs called *symmetric non-determinism* which serves as a pre-requisite for sequentialization (Section 3.2). We formalize canonical sequentialization as a set of *local rewriting rules* (Section 3.3) on I_S programs. Each rewriting step produces a new program that consists of a *sequential prefix* and remainder term that still needs to be rewritten. We show that each rewrite preserves the halting states of all processes. This allows us to use the sequentialization to not only prove local safety properties, but also *global* properties (*e.g.*, deadlock freedom) of the original program.

2. **Synthesizing Sequentializations.** Our second contribution is to demonstrate that our rewriting rules can be turned into an method to *automatically synthesize* a canonical sequentialization from a symmetric non-deterministic program (Section 3.4). We use this synthesis algorithm to implement a distributed systems verification tool called BRISK. BRISK first compiles HASKELL programs that use the CLOUD HASKELL library [25] into our core language (Section 3.5). BRISK verifies that its input has only symmetric races, and computes its canonical sequentialization thereby checking absence of deadlocks and assertion failures.

3. **Evaluation.** Our third contribution is an evaluation of our approach on a diverse

range of benchmarks including distributed programs taken from the literature [33, 21], a concurrent programming textbook [47], well known protocols such as two-phase-commit [43], MapReduce [16] and implementations of a key-value store, and a distributed file-system (Section 3.6). We show that unlike model checking, which gets prohibitively slow — *i.e.*, times out at one minute even with just 10 processes — BRISK verifies the *unbounded* versions of the benchmarks in tens of *milliseconds*, yielding the first concurrency verification tool that is fast enough to be integrated into a design-implement-check cycle.

3.1 Overview

Before discussing the technical details behind canonical sequentializations, we first give an overview of how BRISK lets us write and verify a concurrent task distribution service in HASKELL by synthesizing its canonical sequentialization (Section 3.1.1). Next, we explain the main ideas underlying canonical sequentialization with a series of small examples (Section 3.1.2) designed to give an intuition for the reasoning that BRISK performs. Finally, we make some remarks on its expressiveness (Section 3.1.3).

3.1.1 A Task Distribution Service

Figure 3.1 shows the implementation of a task distribution service. The program consists of n `clients`, each of which requests a task assignment from the server (line 12). Upon receiving the assignment (line 14), a client executes the assigned task and finally either sends an acknowledgment to the master (line 16), or fails in case no work was provided (line 18). The `server` uses the higher-order combinator `foldM` to repeat the function `serverLoop` once for each client. In each iteration, the server waits for a client to request a work item, (line 23), computes the next assignment (line 25), and finally sends the assignment to the client process (line 27). Finally, the `master` waits for acknowledgements from each client (line 34).

Verification Goals. We want to show that (1) the program is deadlock free

and (2) the clients never execute the fail statement in line 18. Even though these two properties seem obvious on inspection, proving them is far from trivial. First, since the program contains unboundedly many threads, a proof needs to track the number of processes that are yet to execute the sends in line 12 and 17 in order to ensure that, any time, there are still enough processes left to make sure that the receives in line 23 and 34 do not deadlock. Second and more problematically, the proof needs to reason about the messages that are being sent and received by the processes, of which there may be unboundedly many. This difficulty is often avoided by reasoning about the *number* of messages sent rather than their *content* [38].

This, however, is not enough for our example as the correctness of the program relies on 1) clients sending their own PID in line 12 as well as the server sending the master's PID in line 27 (if they sent some arbitrary value, the messages might vanish into the ether leaving their intended receives deadlocked) and 2) the server always sending a work item (if it sent `None`, the program would execute the failure branch in line 18). Thus, in general, reasoning about programs like our task service requires complex invariants that are *universally quantified* over the set of participating processes (*e.g.*, to track the contents of messages) and require auxiliary state (*e.g.*, to track how many processes have sent a certain type of message). Despite recent progress, automatic synthesis of such invariants remains a difficult challenge [27, 76, 7] and thus, we are not aware of any automated verification method that an handle this simple example.

Verification via Canonical Sequentialization. In this chapter, we propose a new approach: rather than verifying the original distributed program, we synthesize and verify its canonical sequentialization. Writing a verified program in BRISK starts by importing the BRISK library (line 1) which provides primitives for sending and receiving messages (`send` and `receive` which are built on top of CLOUD HASKELL), process creation, and iterating over sets of processes. When BRISK is invoked to verify the program, it compiles (Section 3.5) the higher-order HASKELL source into the first-order I_S in order to make explicit the control flow and process structure of the input

program. BRISK then synthesizes the program's canonical sequentialization (Section 3.3) which we show in Figure 3.2. The canonical sequentialization consists of two `for`-loops over the clients `cs`. In each iteration of the first loop, the client issues a request, the server assigns a task to the respective client, and the client processes the tasks or fails. In the second loop, the master receives the clients' acknowledgements.

Correctness. In the canonical sequentialization, proving both deadlock-freedom and safety becomes straightforward. As the sequentialization contains neither sends nor receives, the program cannot deadlock. Similarly, since the clients assign `Tasks` to `msg`, the failure branch cannot execute – a fact that can be easily proved (BRISK verifies this assertion automatically). Since our method guarantees that the original program and its canonical sequentialization are equivalent (or in general, the canonical sequentialization *over-approximates* the original program Section 3.3.4), we can conclude that the original program is correct.

```
1   import Brisk
2   data Msg = Request ProcessId | Ack
3   data Work = Task ProcessId Item | None
4
5   main n = do self ← getSelfPid
6               cs   ← spawnMany n (client self)
7               m    ← spawn (master cs)
8               server m cs
9
10  client sv = do self ← getSelfPid
11                 -- request a work item from server
12                 send sv (Request self)
13                 -- block until an item is assigned
14                 msg  ← receive
15                 case msg of
16                    Task m task → do process task
17                                     send m Ack
18                    None → fail
19  server m cs = foldM serverLoop () cs
20    where
21      serverLoop _ _ = do
22        -- wait for a request
23        Request p  ← receive
24        -- compute next item
25        item ← nextItem
26        -- send item to p
27        send p (Task m item)
28        return ()
29
30  master cs = foldM masterLoop () cs
31     where
32       masterLoop _ _ = do
33         -- wait for Ack
34         Ack ← receive
35         return ()
```

Figure 3.1. A task distribution service.

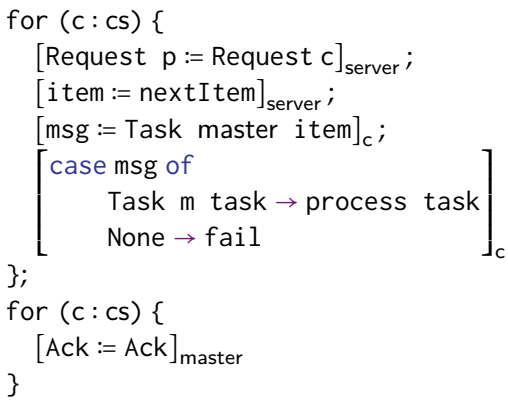


Figure 3.2. Canonical Sequentialization of the task distribution system in Figure 3.1 as computed by BRISK.

3.1.2 Main Ideas

Symmetric Nondeterminism. The crux of our method, and thereby the reason why we can soundly represent a program through its canonical sequentialization, lies in the following observation: for any given trace, we can always move a receive up to its *matching send*. This transformation is an application of Lipton's theory of movers [45]. *Statically* moving a receive up to a matching send is only sound if either a *unique matching send* exists, or all matching sends are *symmetric*, *i.e.*, picking an arbitrary one will result in equivalent states up to permutations of PIDs. To make sure this requirement is met, BRISK checks that the program satisfies the following condition, which we call *symmetric non-determinism*: every receive in a given program location can only receive messages from either *i)* a single process, only, or *ii)* a set of symmetric processes (*i.e.*, processes running the same code) at the same program location.

Canonical Sequentialization. EX1 below is written in our core language $\mathrm{I_S}$, in which two processes p and q exchange messages. In the interest of clarity, in this section we omit the type arguments to send and receive statements when there is no room for confusion.

$$\text{EX1} \triangleq \left[\begin{array}{l} \texttt{send}(q,\texttt{ping}); \\ w \coloneqq \texttt{recv}(*) \end{array}\right]_p \parallel \left[\begin{array}{l} v \coloneqq \texttt{recv}(*); \\ \texttt{send}(p,\texttt{pong}) \end{array}\right]_q$$

Process p *asynchronously* sends a `ping` message to q and then waits for a reply. Process q waits for a message and, upon receipt, sends a `pong` message to p. Since the sends and receives can only be executed in a *single order*, we can rewrite the above concurrent program into its canonical sequentialization:

$$[v \coloneqq \texttt{ping}]_q \,;\, [w \coloneqq \texttt{pong}]_p$$

Crucially, both programs are equivalent in the sense that they terminate in the same state.

Parametric Programs. Next, consider program EX2, which contains a (statically) unbounded number of processes:

$$\text{EX2} \triangleq \left[\begin{array}{l}\texttt{for } (q : Q)\ \{\\ \quad \texttt{send}(q, \texttt{ping});\\ \quad w \coloneqq \texttt{recv}(q)\\ \}\end{array}\right]_p \parallel \bigsqcap_{q:Q} \left[\begin{array}{l} v \coloneqq \texttt{recv}(*);\\ \texttt{send}(p, \texttt{pong})\end{array}\right]_q$$

In EX2 a single process p exchanges messages with a *set* of processes Q that run the same program code. We call such processes *symmetric*. Process p executes a loop which iterates over all processes q in Q. For each q, it first sends a `ping` and subsequently waits for a reply *from* q. Each process in Q first waits for a message and, upon receipt, sends a `pong` message to p. If we fix an iteration order over Q, the sends and receives are again constrained to execute in a single order. Thus, BRISK computes the canonical sequentialization:

$$\begin{array}{l}\texttt{for } (q : Q)\ \{\\ \quad [v \coloneqq \texttt{ping}]_q;\\ \quad [w \coloneqq \texttt{pong}]_p\\ \}\end{array}.$$

Multiple Orders. Next, consider the program EX3 which allows for multiple execution orders. This program is a variant of EX2 where p's loop is split into two parts: first p sends out all `ping` messages, then it waits for the answers to arrive.

$$\text{EX3} \triangleq \left[\boxed{\texttt{for } (q : Q)\ \{\ \texttt{send}(q, \texttt{ping})_a\ \}}; \texttt{for } (q : Q)\ \{\ w \coloneqq \texttt{recv}(*)_{\{b\}}\ \}\right]_p$$
$$\parallel \bigsqcap_{q:Q} \left[\boxed{v \coloneqq \texttt{recv}(*)_{\{a\}}}; \texttt{send}(p, \texttt{pong})_b\right]_q$$

Different executions of EX3 may see messages sent and received in different orders. For example, `ping` messages sent by p can arrive at their respective processes in Q in any order, as messages to different processes may be transported at different speeds. By the

same logic, p may receive pong messages in any order due to the speed of the underlying network or differences in execution times between members of Q.

Checking Symmetric Non-Determinism. In order to rewrite EX3 into its canonical sequentialization, we first need to check that it satisfies symmetric non-determinism. For this, we annotate each syntactic occurrence of send with a unique *tag* as described in Section 2.2.1. For each receive, we then compute an *over-approximation* of the set of tags it can receive from. BRISK computes these tags using a lightweight syntax-guided method (Section 3.2). EX3 shows send-tags in red and receive-tags in blue. In order to satisfy symmetric non-determinism, we require that each receive-set contains either *i)* only tags from a *single process*, or *ii)* a *single tag* from a process in set of *symmetric processes*. The tags for EX3 satisfy this requirement.

Moving Left: Eagerly Executing Receives. We split the rewrite of EX3 into two steps. For the first step, consider a send to some process q in p's `for`-loop together with its matching receive in q. Since EX3 satisfies symmetric non-determinism, we know that there are no *additional* sends q could receive from. Moreover, the receive is *independent* of messages sent by p to other members of Q and of messages that other process might send to p. This means, we can eagerly execute it *directly after* its matching send in p without changing the behaviour of the program. We say that the receive can be *moved left*, up to its matching send. Applying the above reasoning, we can rewrite EX3 into the partially sequentialized program:

$$\texttt{for } (q : Q) \texttt{ \{ } [v \coloneqq \texttt{ping}]_q \texttt{ \};}$$
$$(\texttt{for } (q : Q) \texttt{ \{ } \boxed{[\boxed{w \coloneqq \texttt{recv(*)}}]_p} \texttt{ \} } \parallel \textstyle\prod_{q:Q} \boxed{[\boxed{\texttt{send}(p, \texttt{pong})}]_q})$$

The program consists of a *sequential prefix* which is a rewriting of the boxed statements and a *remainder term* that corresponds to the part of the program that still needs to be rewritten. For the second rewriting step, consider the boxed statements in the remainder term. In a given loop iteration, p can receive a send from *any* of the remaining processes in Q. However, each process runs the same program (with the notable exception that the

processes differ in their PIDs. This means, moving the receive up to an *arbitrary* send will result in the same final state. As a result, we can rewrite the program to its canonical sequentialization:

$$\texttt{for } (q : Q)\ \{\ [v \coloneqq \texttt{ping}]_q\ \};\texttt{for } (q : Q)\ \{\ [w \coloneqq \texttt{pong}]_p\ \}$$

Example 4: Multi-Party Communication. Finally, we turn to the example EX4, whose communication structure matches the task distribution service from Figure 3.1:

$$\text{EX4} \triangleq \left[\begin{array}{l}\texttt{for } (q : Q)\ \{\\ \quad id \coloneqq \texttt{recv}(*)_{\{b\}};\\ \quad \texttt{send}(id, \texttt{ping})_a\\ \}\end{array}\right]_p \parallel \bigsqcap_{q:Q} \left[\begin{array}{l}\texttt{send}(p, q)_b;\\ v \coloneqq \texttt{recv}(*)_{\{a\}};\\ \texttt{send}(m, \texttt{pong})_c\end{array}\right]_q \parallel \left[\begin{array}{l}\texttt{for } (q : Q)\ \{\\ \quad w \coloneqq \texttt{recv}(*)_{\{c\}}\\ \}\end{array}\right]_m \tag{3.1}$$

Process p executes a loop in which it waits for a message, and upon receipt, sends a ping message to the process it received from. Each process in Q first sends its PID to process p, then sends a pong message to process m and finally waits for a reply. Process m executes a loop in which it receives messages from processes in Q.

First, we check that EX4 is symmetrically non-deterministic: the annotated tags show that this requirement is met. Next, we rewrite EX4 into its canonical sequentialization. The rewrite is split into two steps: a first step in which BRISK rewrites the communication between process p and set Q, and a second step in which it rewrites the communication between set Q and m. For the first step, BRISK rewrites the boxed statements. This rewrite step produces a sequential prefix:

$$\begin{array}{l}\texttt{for } (q : Q)\ \{\\ \quad [id \coloneqq q]_p;\\ \quad [v \coloneqq \texttt{ping}]_q;\\ \}\end{array} \tag{3.2}$$

and an additional *residual term* contains the messages the processes in Q sent to m:

$$\bigsqcap_{q:Q} \Big[\!\Big[\; \mathsf{send}(m, \mathsf{pong}) \; \Big]\!\Big]_q \tag{3.3}$$

In the second step, BRISK rewrites the parallel composition of the remainder program with the residual term, shown below.

$$\begin{array}{l} \texttt{for } (q : Q) \; \{ \\ \quad [id \coloneqq q]_p \, ; \\ \quad [v \coloneqq \mathsf{ping}]_q \\ \} \end{array} \tag{3.4}$$

$$\bigsqcap_{q:Q} \Big[\!\Big[\; \boxed{\mathsf{send}(m, \mathsf{pong});} \; \Big]\!\Big]_q \; \| \; \Big[\!\Big[\; \boxed{\begin{array}{l} \texttt{for } (q : Q) \; \{ \\ \quad w \coloneqq \texttt{recv(*)} \\ \} \end{array}} \; \Big]\!\Big]_m$$

BRISK rewrites the boxed statements into the canonical sequentialization:

$$\begin{array}{l} \texttt{for } (q : Q) \; \{ \\ \quad [id \coloneqq q]_p \, ; \\ \quad [v \coloneqq \mathsf{ping}]_q \\ \} \\ \texttt{for } (q : Q) \; \{ \\ \quad [w \coloneqq \mathsf{pong}]_m \\ \} \end{array} \tag{3.5}$$

3.1.3 Expressiveness

Not all programs have a canonical sequentialization. We can characterize the systems where our method is applicable by describing its limits, *i.e.*, the four cases where BRISK *fails* to synthesize a sequentialization.

1. Asymmetric Non-determinism. BRISK will reject programs if it cannot prove that they only exhibit symmetric non-determinism. In this case, BRISK outputs the sends and receives that are involved in the suspected asymmetric race. In our experience, this often is either a bug or easy to remedy by restructuring the receives. That said, there are algorithms that do break symmetry and hence cannot be sequentialized by BRISK, *e.g.*, process sets with asymmetric topology such as rings.

Example 3.1. Consider program C below, which is a set of clients of a key-value store S whose API supports retrieving the value of a key k with a $\mathsf{Get}(k)$ message, and setting the value v of a key k with a $\mathsf{Set}(k, v)$ message. Each $c \in Cs$ performs a non-deterministic choice (represented by the condition $*$) to either send a $\mathsf{Get}(k)$ or a $\mathsf{Set}(k, v)$ message to s.

$$\mathrm{C} \triangleq \prod_{c:Cs} \left[\begin{array}{l} \texttt{if } * \texttt{ \{} \\ \quad \mathsf{send}(s, \mathsf{Get}(k)) \\ \texttt{\} else \{} \\ \quad \mathsf{send}(s, \mathsf{Set}(k, v)) \\ \texttt{\}} \end{array}\right]_c \qquad \mathrm{C}' \triangleq \prod_{c:Cs} \left[\begin{array}{l} \texttt{if } * \texttt{ \{} \\ \quad msg \coloneqq \mathsf{Get}(k) \\ \texttt{\} else \{} \\ \quad msg \coloneqq \mathsf{Set}(k, v) \\ \texttt{\}} \\ \mathsf{send}(s, msg) \end{array}\right]_c$$

$$\mathrm{S} \triangleq \left[\begin{array}{l} \texttt{while true \{} \\ \quad op \coloneqq \mathsf{recv}(*); \\ \quad \texttt{if } isGet(op) \texttt{ \{} \\ \quad\quad \texttt{doGetOp()} \\ \quad \texttt{\} else \{} \\ \quad\quad \texttt{doSetOp()} \\ \quad \texttt{\}} \\ \texttt{\}} \end{array}\right]_s$$

The composition $\mathrm{C} \parallel \mathrm{S}$ does not satisfy symmetric non-determinism, as there are *two* distinct sends of the same type to the store s. However, this is easily refactored to a

program C′ that first performs a non-deterministic *assignment* to a *msg* variable and then sends the contents of the variable to s. Hence, C′ ‖ S has symmetric non-determinism.

Example 3.2. Next, consider a logging process that receives messages from *every* process in a system and logs the messages it receives to disk. Except in simple instances, programs including such a logger will have asymmetric non-determinism. In this example, there is no formula for refactoring the program into a variant with symmetric non-determinism.

2. Superfluous Sends. The program must not contain any superfluous sends that *do not* have a matching receive. BRISK is unable to rewrite such programs into their sequentialization, but returns a counterexample in the form of a sequential prefix that ends in the superfluous send. While superfluous sends can be benign, (unlike superfluous receives which are deadlocks), they are dubious; we consider their detection to be a virtue of our approach.

3. Indiscriminate Communication. When iterating over an (unbounded) set of processes Q, a process p must only talk to a *single process* in Q, in any iteration. Process p may however send messages to *other* processes not in Q. This is also not an onerous requirement as in well-structured programs, loops over unbounded sets Q are used primarily to "broadcast" or "gather" both of which are amenable to sequentialization.

4. Stateful Loop Termination. Finally, `while`-loops that interact with other processes must not use loop-carried state to decide whether to loop again or exit. Sequentialization requires that the decision only depends on values computed in the *current iteration*. This requirement models a reactive pattern in which loop termination depends on external messages. We found that loops requiring loop-carried state can often be restructured into an iteration over sets (processes or otherwise) which our method supports.

Counterexamples. BRISK provides useful feedback in each of the four cases of failure. When there is a (possible) asymmetric race, the programmer is pointed to the race condition that needs to be fixed. In the remaining cases, BRISK outputs the longest

sequential prefix encountered in the failed rewrite attempt (together with the remaining, unsequentialized program) thereby pinpointing the exact conditions under which the relevant condition is violated. Since BRISK is fast enough (10s of milliseconds) to provide this feedback *during* development, we envision a use case where the coding discipline required by BRISK can nudge the developer towards well-structured programs that are easier to reason about for machines (and humans).

3.2 Symmetric Non-determinism

In this section, we make precise the definition of symmetric non-determinism as introduced in Section 3.1. Let $\mathsf{Recvs}(A)$ denote the set all of the syntactic receive statements occuring in A, and let $\mathsf{SendTags}(A)$ denote all of the tags associated with the syntactic send statements occuring in A.

Definition 3.1 (Symmetric Non-determinism)**.** Let A be a program that is tagged consistently (Definition 2.6). Then A has symmetric non-determinism if, for all $\mathtt{recv}(t, \mathtt{w})_{\overline{\ell}} \in \mathsf{Recvs}(A)$, $A \equiv A_s \parallel A_r$ where $\overline{\ell} \subseteq \mathsf{SendTags}(A_s)$, $\overline{\ell} \cap \mathsf{SendTags}(A_r) = \varnothing$ and either

1. $\mathsf{Procs}(A_s) = \{p\}$ where p is a PID variable (A_s is a concrete process); or
2. $|\overline{\ell}| = 1$ and $\mathsf{Procs}(A_s) = \{P\}$ where P is a PID set variable (A_s is a symmetric group of process).

Checking Symmetric Non-Determinism. We check symmetric non-determinism through the following effective over-approximation: for each receive, we add tags of all sends of the matching type, where we treat sends to variables as sends to *any* PID and assume processes do not send messages to themselves. This approximation suffices for the examples considered in this dissertation. The method described in the following sections is orthogonal, and could be extended to a more precise analysis if deemed necessary.

Receive Wildcard Instantiation. We replace each wildcard with the expression implied by its tags. This is possible since symmetric non-determinism ensures that each

BUFF	$\in$	$(PID \times PID \times \mathsf{TypeIdentifiers}) \rightharpoonup \overline{e}$	**Buffers**
ASRT	::=		**Assertions**
		$\varnothing$	*Empty*
	$\mid$	$\text{ASRT} \cup \{x \in X\}$	*Membership*
	$\mid$	$\text{ASRT} \cup \{\varnothing \subseteq X \subseteq X\}$	*Bounds*
	$\mid$	$\text{ASRT} \cup \{\mathsf{E}(x)\}$	*External*
Γ	::=	$(\text{BUFF}, \text{ASRT})$	**Contexts**

Figure 3.3. Syntax for context Γ.

receive is matched with a single process or a symmetric set.

3.3 Rewrite Rules

Next, we formalize canonical sequentializations through a set of rewriting rules.

Symbolic States. Each rewriting rule defines a relation between a pair of *symbolic states* consisting of the following components: A *context* (Γ); a sequential *prefix* (Δ); the program (P) to be rewritten; and a *residual* process (Ψ) comprising statements that interact with processes outside of P. A context Γ consists of a symbolic message buffer BUFF and a set of assertions ASRT. BUFF maps each channel to the sequence of pending messages on that channel. More concretely, $\text{BUFF}(p, q, t)$ returns the sequence of pending messages of type t sent from p to q. ASRT contains assertions about process identifiers. We summarize the syntax in Figure 3.3. We sometimes use Γ to refer to one of its components, for example, we write $\Gamma(p, q, t)$ to mean $\text{BUFF}(p, q, t)$. Finally, for a context $\Gamma \triangleq (\text{BUFF}, \text{ASRT})$ and an assertion a, we write $\Gamma \vdash a$ to mean $a \in \text{ASRT}$, and $\Gamma \nvdash a$ to mean $a \notin \text{ASRT}$.

Rewriting Rules. Each rewriting rule defines a judgment of the form

$$\Gamma, \Delta, P, \Psi \rightsquigarrow \Gamma', \Delta', P', \Psi'.$$

The goal of each step is to move parts of program P into the sequential prefix Δ such that eventually we can rewrite P to `skip`. We now describe the main rules of our method,

starting with basic rules, followed by rules for loops and conditionals and finally residual processes.

3.3.1 Basic Rules

Figure 3.4 contains the basic rules. We assume that programs do not contain wild-card receives, having removed them according to the method described in Section 3.2.

Sends and receives. Rule R-SEND treats sends. The rule rewrites a send from process p to some x into `skip`, if x is either a PID q, or a variable that maps to some PID q, and additionally, q is not external (*i.e.*, belongs to a residual process). We enforce the check that x corresponds to a PID through the condition $\Delta \vDash x = q$, where we write $\Delta \vDash \varphi$ to mean that formula φ is valid after executing Δ (see Section 3.4 for a discussion of $\Delta \vDash \varphi$). The rule updates context Γ by adding expression n to the end of the buffer for the respective channel, where we use $\Gamma[a \leftarrow b]$ to denote the function that returns the same values as Γ on all inputs except for a where it returns b. Rule R-RECV rewrites a receive from some x into `skip`, if x corresponds to a PID p. R-RECV takes the most recent message m from the respective channel and adds the corresponding assignment to the prefix.

Context and congruence. Rule R-CONTEXT allows rewriting individual program parts independently of program parts that occur later or in parallel. Rule R-CONGR allows rewriting into congruent programs.

Example 3.3. Consider again program EX1 from Section 3.1 where we replaced wild card receives with receives from the respective processes:

$$\left[\begin{array}{l}\texttt{send}(q,\texttt{ping});\\ w \coloneqq \texttt{recv}(q)\end{array}\right]_p \parallel \left[\begin{array}{l}v \coloneqq \texttt{recv}(p);\\ \texttt{send}(p,\texttt{pong})\end{array}\right]_q$$

We start the rewrite with the symbolic state given by $\Delta \triangleq \texttt{skip}$, $\Psi \triangleq \texttt{skip}$ and $\Gamma \triangleq (\text{BUFF}_\varnothing, \varnothing)$, where $\text{BUFF}_\varnothing$ maps every channel to the empty sequence ϵ. We also assume that there is only a single message type $\top$. In a first step, we apply the rules R-CONTEXT

and R-SEND to rewrite EX1 into the program below, where we update the buffer to $\text{BUFF}_\varnothing[(p, q, \top) \leftarrow \texttt{ping}]$:

$$\begin{array}{l} \texttt{skip}; \\ [w \coloneqq \texttt{recv}(q)]_p \end{array} \quad \| \quad \left[\begin{array}{l} v \coloneqq \texttt{recv}(p); \\ \\ \texttt{send}(p, \texttt{pong}) \end{array}\right]_q$$

Applying R-CONTEXT and R-RECV yields the following, with buffer $\text{BUFF}_\varnothing$ and prefix $\Delta = [v \coloneqq \texttt{ping}]_q$.

$$\begin{array}{l} \texttt{skip}; \\ [w \coloneqq \texttt{recv}(q)]_p \end{array} \quad \| \quad \begin{array}{l} \texttt{skip}; \\ [\texttt{send}(p, \texttt{pong})]_p \end{array}$$

Applying R-CONTEXT and R-CONGR twice yields $[w \coloneqq \texttt{recv}(q)]_p \parallel [\texttt{send}(p, \texttt{pong})]_p$. Applying the same rules again yields `skip` with prefix $\Delta = [v \coloneqq \texttt{ping}]_q \,;\, [w \coloneqq \texttt{pong}]_p$.

3.3.2 Loops, Unfolding and Conditionals

Next, we present our rules for loops. We first present our rules for iterating over sets of *processes*, then our rules for iterating over sets of *indices*, and finally our rules for `while`-loops and `if`-statements.

Iterating over identical processes. Figure 3.5 contains rule R-LOOP-UPD for rewriting the interaction between a set of identical processes Q and a process p which iterates over Q. The rewrite succeeds if we can rewrite the interaction between an *arbitrary* iteration of p and a *single process* in Q, *independently* of previous iterations and other processes. This condition is enforced through the rewrite-step that appears in the pre-condition of the rule. The rule picks a fresh PID $q^* \in Q$ (corresponding to the value of q in the chosen iteration) and a subset $\varnothing \subset Q^* \subseteq Q$ (corresponding to the set of remaining processes, in that iteration) and shows that it is possible to *unfold* a process u from Q^* (u may or may not be equal to q^*) such that u and p can be rewritten to `skip`. Explicitly unfolding process u from Q^* ensures that the iteration talks to process u, only. To ensure iterations are independent, the rule modifies the prefix by havocing all variables that may be assigned in the loop. Likewise, the rule requires the context after the rewrite to

R-CONTEXT

$$\frac{\Gamma, \Delta, A, \Psi \rightsquigarrow \Gamma', \Delta', A', \Psi'}{\Gamma, \Delta, A \circ B, \Psi \rightsquigarrow \Gamma', \Delta', A' \circ B, \Psi'}$$

R-CONGRUENCE

$$\frac{A \equiv B}{\Gamma, \Delta, A, \Psi \rightsquigarrow \Gamma, \Delta, B, \Psi}$$

R-SEND

$$\frac{\begin{array}{ll}\Delta \vDash x = q & \\ q \text{ is a PID} & \Gamma(p,q,t) = m \\ \Gamma \not\vdash \mathsf{E}(q) & \Gamma' = \Gamma[(p,q,t) \leftarrow m \cdot n]\end{array}}{\Gamma, \Delta, [\mathtt{send}(t,x,n)]_p, \Psi \rightsquigarrow \Gamma', \Delta, \mathtt{skip}, \Psi}$$

R-RECV

$$\frac{\begin{array}{ll}\Delta \vDash x = p & \\ p \text{ is a PID} & \Gamma' = \Gamma[(p,q,t) \leftarrow n] \\ \Gamma(p,q,t) = m \cdot n & \Delta' = \Delta; [y \coloneqq m]_q\end{array}}{\Gamma, \Delta, [y \coloneqq \mathtt{recv}(t,x)]_q, \Psi \rightsquigarrow \Gamma', \Delta', \mathtt{skip}, \Psi}$$

Figure 3.4. Rewrite Rules (Basic Statements)

R-LOOP-UPD

$$\frac{\begin{array}{l}q^* \text{ and } Q^* \ \mathtt{fresh} \\ \Gamma_0 \triangleq \Gamma \cup \{\varnothing \subset Q^* \subseteq Q\} \cup \{q^* \in Q^*\} \\ \Delta_0 \triangleq \mathsf{havoc}(\Delta, A, B)\end{array} \qquad \begin{array}{l}\Delta' \triangleq \Delta \,;\, \mathtt{for}\ (q : Q)\ \{\ \Delta^u[q/u]\ \} \\ \Psi' \triangleq \Psi \parallel \prod_{q:Q} [\Psi^u]_q\end{array}}{\Gamma_0, \Delta_0, [A[q^*/q]]_p \parallel \prod_{q:Q^*} [B]_q, \mathtt{skip} \rightsquigarrow \Gamma_0, (\Delta_0; \Delta^u), \left(\mathtt{skip} \parallel \prod_{q:Q^* \setminus \{u\}} [B]_q\right), [\Psi^u]_u}$$

$$\Gamma, \Delta, \left[\mathtt{for}\ (q : Q)\ \{\ A\ \}\right]_p \parallel \prod_{q:Q} \left[B; C\right]_q, \Psi \rightsquigarrow \Gamma, \Delta', \prod_{q:Q} [C]_q, \Psi'$$

Figure 3.5. Rewrite Rules (Iteration over sets of processes).

be the same as before in order to rule out superfluous sends.

Unfolding. Figure 3.7 shows rules R-SEND-UNFOLD and R-RECV-UNFOLD which unfold a single process from a set of identical processes. Rule R-SEND-UNFOLD allows unfolding a process q^* from a set Q, if there is a send to q^*, and it follows from the context that $q^* \in Q$. Rule R-RECV-UNFOLD treats a situation in which a process p can receive from *any* of the processes in a set of identical processes Q', *i.e.*, there is a race between these processes. The rule picks a fresh $q^* \in Q'$ and unfolds it from the set. It then modifies the receive such that it can only receive from the freshly chosen PID. The rule has an additional precondition requiring that the receive can be rewritten to `skip`, *i.e.*, there is in fact a matching send in A. This precondition is required to ensure that the rewrite-step does not introduce any deadlocks (by over-specializing the receive from any process in Q' to just q^*).

Example 3.4. Consider EX5 shown left below (this example is based on EX4 from Section 3.1). As before, we eliminated wild card receives and assume that there is only a single message type.

$$\text{EX5} \triangleq \left[\begin{array}{l} \texttt{for } (q : Q)\ \{ \\ \quad\ \} \\ \end{array}\begin{array}{l} \\ id \coloneqq \texttt{recv}(Q); \\ \texttt{send}(id, \texttt{ping}) \end{array}\right]_p \parallel \bigsqcap_{q:Q} \left[\begin{array}{l} \texttt{send}(p, q); \\ v \coloneqq \texttt{recv}(p) \end{array}\right]_q$$

Our goal is to apply R-LOOP-UPD to produce the sequentialization EX5_{GOAL}.

$$\text{EX5}_{\text{GOAL}} \triangleq \texttt{for } (q : Q)\ \{ \quad \}\begin{array}{l} [id \coloneqq q]_p\ ; \\ [v \coloneqq \texttt{ping}]_q \end{array}$$

In order to satisfy the precondition of rule R-LOOP-UPD, we need to rewrite the program I shown below, which corresponds to an arbitrary iteration of p's loop (Q^* is a fresh set with $\Gamma \vdash \varnothing \subset Q^* \subseteq Q$), by unfolding a process from Q^* and rewriting p and the unfolded

process to skip.

$$I \triangleq \begin{bmatrix} id \coloneqq \mathtt{recv}(Q); \\ \mathtt{send}(id, \mathtt{ping}) \end{bmatrix}_p \parallel \bigsqcap_{q:Q^*} \begin{bmatrix} \mathtt{send}(p, q); \\ v \coloneqq \mathtt{recv}(p) \end{bmatrix}_q$$

Applying R-RECV-UNFOLD yields the program I', where q^* is a fresh PID. We can rewrite I' into $\mathtt{skip} \parallel \bigsqcap_{q:Q^* \setminus \{q^*\}} [\cdots]$ with sequential prefix $\Delta \triangleq \left([id \coloneqq q^*]_p \,;\, [v \coloneqq \mathtt{ping}]_{q^*} \right)$ thereby satisfying the goal in the precondition of R-LOOP-UPD. This allows us to rewrite the entire program to skip (using an additional application of Rule R-CONGR) which produces the sequential prefix EX5_{GOAL}.

$$I' \triangleq \begin{bmatrix} id \coloneqq \mathtt{recv}(q^*); \\ \mathtt{send}(id, \mathtt{ping}) \end{bmatrix}_p \parallel \begin{bmatrix} \mathtt{send}(p, q^*); \\ v \coloneqq \mathtt{recv}(p) \end{bmatrix}_{q^*} \parallel \bigsqcap_{q:Q^* \setminus \{q^*\}} [\cdots]$$

Example 3.5. Consider example EX6, shown below.

$$\text{EX6} \triangleq \begin{bmatrix} \mathtt{for}\ (q : Q)\ \{ \\ \quad \mathtt{send}(q, \mathtt{ping})\ ; \\ \}\ \ v \coloneqq \mathtt{recv}(Q) \end{bmatrix}_p \parallel \bigsqcap_{q:Q} \begin{bmatrix} \mathtt{send}(p, \mathtt{pong}); \\ w \coloneqq \mathtt{recv}(p) \end{bmatrix}_q$$

EX6 is a variant of the previous example that is *rejected* by our method. As before, in order to satisfy the precondition of rule R-LOOP-UPD, we need to rewrite the program I corresponding to an iteration of p's loop.

$$I \triangleq \begin{bmatrix} \mathtt{send}(q^*, \mathtt{ping})\ ; \\ v \coloneqq \mathtt{recv}(Q) \end{bmatrix}_p \parallel \bigsqcap_{q:Q^*} \begin{bmatrix} \mathtt{send}(p, \mathtt{pong}); \\ w \coloneqq \mathtt{recv}(p) \end{bmatrix}_q$$

Applying rule R-SEND-UNFOLD yields the program I':

$$I' \triangleq \begin{bmatrix} \mathtt{send}(q^*, \mathtt{ping})\ ; \\ id \coloneqq \mathtt{recv}(Q) \end{bmatrix}_p \parallel \begin{bmatrix} \mathtt{send}(p, \mathtt{pong})\ ; \\ w \coloneqq \mathtt{recv}(p) \end{bmatrix}_{q^*} \parallel \cdots$$

However, our method fails to rewrite this program as this would require unfolding a *second* process to handle the receive from Q. Intuitively, this is because the receive in p can receive a `pong` message from *any* process (not just the one it just sent to), which violates the requirement that each loop iteration should only talk to a *single* process.

Iterating over sets of Indices. Figure 3.6 contains rule R-LOOP-REPEAT. The rule rewrites the interaction between a set of processes Q and a process p which iterates over a set of indices I. Again, the rule contains a precondition that requires rewriting the interaction between an arbitrary iteration of p and a single process from Q. For this, the rule picks a fresh i^* and requires showing that we can unfold a process u from Q such that p's iteration can be rewritten to `skip` while process u remains *unchanged*, *i.e.*, executable after the interaction. The rule produces a sequential prefix by repeating the prefix produced in the iteration where the unfolded process u is substituted for an arbitrary process from Q.

While Loops and Conditionals. Consider again Figure 3.9. Rule R-WHILE-REPEAT unrolls an iteration of a while loop, if the iteration, together with some prefix B of another process, can be rewritten to `skip`. Rule R-WHILE-REMOVE unrolls an iteration of a while loop, if the iteration together with some prefix B of another process can be rewritten to `break`. It then removes the while loop from the program. Rule R-IF-THEN allows to rewrite the `then`-branch, if the condition holds; R-IF-ELSE allows rewriting to the `else`-branch, if the condition does not hold. Our system additionally contains rules R-BRANCH that allows to rewrite an `if`-statement, if both branches, together with an additional context, can be rewritten to `skip`, R-NONDET-RECV which allows receiving from a non-deterministically chosen process in an identical set, and a rule for rewriting pairs of `for`-loops.

Example 3.6. Consider example EX7 shown below, in which process p interacts with a

set of processes Q.

$$\textsc{Ex}7 \triangleq \left[\begin{array}{l}\texttt{for } (i : I) \texttt{ \{} \\ \quad id \coloneqq \texttt{recv}(Q)\texttt{ ;} \\ \quad \texttt{send}(id, 0) \\ \texttt{\}}\end{array}\right]_p \parallel \bigsqcap_{q:Q} \left[\begin{array}{l}\texttt{while true \{} \\ \quad \texttt{send}(p, q)\texttt{ ;} \\ \quad stop \coloneqq \texttt{recv}(p)\texttt{ ;} \\ \quad \texttt{if } stop = 1 \texttt{ \{} \\ \quad\quad \texttt{break} \\ \quad \texttt{\} else \{} \\ \quad\quad \texttt{skip} \\ \quad \texttt{\}} \\ \texttt{\}}\end{array}\right]_q$$

p executes a loop in which it receives a PID and then sends back the value 0. Each process in Q sends its PID to p and waits for a reply. Upon receipt, it assigns the received value to *stop* and breaks from the loop, if *stop* is one. Our goal is to use rule R-Loop-Repeat to rewrite p to `skip` so that the remaining program consists only of the parallel composition over Q. Applying R-Loop-Repeat and R-Recv-Unfold yields the program shown below, where q^* is fresh.

$$\left[\begin{array}{l}id \coloneqq \texttt{recv}(q^*)\texttt{ ;} \\ \texttt{send}(id, 0)\end{array}\right]_p \parallel \left[\begin{array}{l}\texttt{while true \{} \\ \quad \texttt{send}(p, q^*)\texttt{ ;} \\ \quad stop \coloneqq \texttt{recv}(p)\texttt{ ;} \\ \quad \texttt{if } stop = 1 \texttt{ \{} \\ \quad\quad \texttt{break} \\ \quad \texttt{\} else \{} \\ \quad\quad \texttt{skip} \\ \quad \texttt{\}} \\ \texttt{\}}\end{array}\right]_{q^*} \parallel \cdots$$

Using an application of R-WHILE-REPEAT and R-IF-ELSE, we can rewrite process p to `skip`, producing sequential prefix $\Delta \triangleq [id \coloneqq q^*]_p \; ; \; [stop \coloneqq 0]_{q^*}$, yielding the final sequential prefix:

$$\texttt{for } (i : I) \texttt{ \{}$$
$$\quad \sum q{:}Q.\left([id \coloneqq q]_p \,; [stop \coloneqq 0]_q\right)$$
$$\texttt{\}}$$

3.3.3 Residual Processes

Finally, Figure 3.8 shows our rules for residual processes. Rule R-SEND-RESID moves a send of value n to process x into the residual process Ψ, if x is *external*. Since we are postponing the execution of the send, the rule takes a "snapshot" of n by inserting it into a fresh set V and adding the assignment to the sequential prefix (we assume that V is initialized to $\varnothing$). The residual process then non-deterministically sends a value from V (note that this step introduces an over-approximation). Our system contains an additional rule R-RECV-RESID for receives. Rule R-COMPOSE-RESID allows a modular rewriting of programs. It takes two parallel programs A and B and first rewrites A while treating all processes in B as *external*, meaning that all sends to processes in B ($\mathsf{Procs}(B)$) are added to the residual process Ψ. The rule then composes the residual process with B and rewrites the composition. The rule requires the composition of Ψ and B be symmetrically non-deterministic, indicated by $\mathsf{rf}(B \parallel \Psi)$.

Example 3.7. Consider again example EX4 (Equation (3.1)). We apply rule R-COMPOSE-RESID, rewriting process p and the processes in Q, treating process m as external. This produces the residual term shown in Equation (3.3), where we have removed the non-deterministic choice as the set only contains one element. Next, we compose the residual process with process m as shown in Equation (3.4). The program is race-free up to symmetry as each receive in m can only receive messages from processes in Q that are at the same program location. Rewriting the resulting program yields sequentialization shown in Equation (3.5).

R-LOOP-REPEAT

$$\frac{\begin{array}{l} i^* \ \texttt{fresh} \\ \Delta_0 \triangleq \ \mathsf{havoc}(\Delta, A, B) \end{array} \qquad \begin{array}{l} \Delta' \triangleq \Delta \,;\, \begin{bmatrix} \texttt{for}\ (i : I)\ \{ \\ \quad \sum q{:}Q.\, (\Delta^u[i/i^*][q/u]) \\ \} \end{bmatrix} \\ \Psi' \triangleq \Psi \parallel \begin{pmatrix} \texttt{for}\ (i : I)\ \{ \\ \quad \sum q{:}Q.\, [\Psi^u]_q \\ \} \end{pmatrix} \end{array}}{\Gamma, \Delta_0, \Big([A[i^*/i]]_p \parallel \textstyle\prod_{q:Q} [B]_q \Big), \texttt{skip} \rightsquigarrow \Gamma, (\Delta_0; \Delta^u), \big(\texttt{skip} \parallel [B]_u \parallel \textstyle\prod_{q:Q \setminus \{u\}} B \big), [\Psi^u]_u}$$

$$\Gamma, \Delta, \big[\texttt{for}\ (i : I)\ \{\ A\ \}\big]_p \parallel \textstyle\prod_{q:Q} [B]_q, \Psi \rightsquigarrow \Gamma, \Delta', \textstyle\prod_{q:Q} [B]_q, \Psi'$$

Figure 3.6. Rewrite Rules (Iteration over sets of indices).

R-SEND-UNFOLD

$$\frac{\Gamma \vdash q^* \in Q}{\begin{array}{l} \Gamma, \Delta, \Big(\big[\mathsf{send}(t, q^*, n)\big]_p \parallel \textstyle\prod_{q:Q} A \Big), \qquad \Psi \rightsquigarrow \\ \Gamma, \Delta, \Big(\big[\mathsf{send}(t, q^*, n)\big]_p \parallel \textstyle\prod_{q:Q \setminus \{q^*\}} A \parallel [A]_{q^*} \Big), \Psi \end{array}}$$

R-RECV-UNFOLD

$$\frac{\begin{array}{l} q^* \ \texttt{fresh} \\ \Gamma \vdash \varnothing \subset Q' \subseteq Q \\ \Gamma, \Delta, \Big(\big[x \coloneqq \mathsf{recv}(t, q^*)\big]_p \parallel [A]_{q^*} \Big), \Psi \rightsquigarrow \Gamma, \Delta', [A']_{q^*}, \Psi' \end{array}}{\begin{array}{l} \Gamma, \Delta, \Big(\big[x \coloneqq \mathsf{recv}(t, Q)\big]_p \parallel \textstyle\prod_{q:Q'} [A]_q \Big), \qquad \Psi \rightsquigarrow \\ \Gamma, \Delta, \Big(\big[x \coloneqq \mathsf{recv}(t, q^*)\big]_p \parallel \textstyle\prod_{q:Q' \setminus \{q\}} A \parallel [A]_{q^*} \Big), \Psi \end{array}}$$

Figure 3.7. Rewrite Rules (Unfold)

R-SEND-RESID

$$\frac{\begin{array}{ll} V \textit{ fresh} & q \text{ is a PID} \\ \Delta \vDash x = q & \Delta' = \left(\Delta; [V \coloneqq V \cup \{n\}]_p\right) \\ \Gamma \vdash \mathsf{E}(q) & \Psi' = \left(\Psi \circ \sum v{:}V.\, [\mathsf{send}(t,q,v)]_p\right) \end{array}}{\Gamma, \Delta, [\mathsf{send}(t,x,n)]_p, \Psi \rightsquigarrow \Gamma, \Delta', \mathsf{skip}, \Psi'}$$

R-COMPOSE-RESID

$$\frac{\begin{array}{ll} \Gamma_0 \triangleq \Gamma \cup \mathsf{E}(p) \text{ for } p \in \mathsf{Procs}(B) & \mathsf{rf}(B \parallel \Psi) \\ \Gamma_0, \Delta, A, \mathsf{skip} \rightsquigarrow \Gamma_0, \Delta', \mathsf{skip}, \Psi & \Gamma, \Delta', B \parallel \Psi, \mathsf{skip} \rightsquigarrow \Gamma, \Delta'', \mathsf{skip}, \mathsf{skip} \end{array}}{\Gamma, \Delta, A \parallel B, \mathsf{skip} \rightsquigarrow \Gamma, \Delta'', \mathsf{skip}, \mathsf{skip}}$$

Figure 3.8. Rewrite Rules (Residue)

R-WHILE-REPEAT

$$\frac{\Gamma, \Delta, [A]_p \parallel [B]_q, \Psi \rightsquigarrow \Gamma', \Delta', \mathsf{skip}, \Psi'}{\begin{array}{l} \Gamma, \Delta, [\mathtt{while\ true\ \{}\ A\ \mathtt{\}}]_p \parallel [B\,;C]_q, \Psi \rightsquigarrow \\ \Gamma', \Delta', [\mathtt{while\ true\ \{}\ A\ \mathtt{\}}]_p \parallel [C]_q, \quad \Psi' \end{array}}$$

R-WHILE-REMOVE

$$\frac{\Gamma, \Delta, [A]_p \parallel [B]_q, \Psi \rightsquigarrow \Gamma', \Delta', \mathsf{break}, \Psi'}{\Gamma, \Delta, \left[\mathtt{while\ true\ \{}\ A\ \mathtt{\}}\right]_p \parallel [B;C]_q, \Psi \rightsquigarrow \Gamma', \Delta', [C]_q, \Psi'}$$

R-IF-THEN

$$\frac{\Delta \vDash e}{\Gamma, \Delta, \mathtt{if}\ e\ \mathtt{\{}\ A\ \mathtt{\}\ else\ \{}\ B\ \mathtt{\}}, \Psi \rightsquigarrow \Gamma, \Delta, A, \Psi}$$

R-IF-ELSE

$$\frac{\Delta \vDash \neg e}{\Gamma, \Delta, \mathtt{if}\ e\ \mathtt{\{}\ A\ \mathtt{\}\ else\ \{}\ B\ \mathtt{\}}, \Psi \rightsquigarrow \Gamma, \Delta, B, \Psi}$$

Figure 3.9. Rewrite Rules (Branch)

3.3.4 Correctness

Termination. Each rewrite rule, with the exception of R-CONGRUENCE, decreases the size of the input program. Moreover, there are finitely many ways to instantiate each rule. Therefore, we guarantee termination of our rewriting by restricting the use of R-CONGRUENCE to situations where it *decreases* the size of the input program.

Rewrite Soundness. We now present our main correctness theorem. For this, we need the following additional definitions. We define an interpretation on prefixes and contexts such that $(\sigma, \mu) \in [\![\Delta, \Gamma]\!]$ when σ and μ are a store and message buffer consistent with the states reachable by executing Δ and the assumptions in Γ. Let $\sigma|_P$ denote the store whose domain is restricted to the variables of the processes in P. We denote the set of processes that are halted in a state (and will never become enabled) as $\mathsf{hprocs}(\sigma, \mu, P)$.

Theorem 3.1. *Let P be a program and assume $P : \checkmark$. If*

1. $\Gamma, \Delta, P, \Psi \rightsquigarrow \Gamma', \Delta', P', \Psi'$
2. *For some extension E, $\mathsf{rf}(P \ltimes E)$ and $(\sigma, \mu) \in [\![\Delta, \Gamma]\!]$ such that*

$$(\sigma, \mu, \Psi; P \ltimes E) \to (\sigma_Q, \mu_Q, Q)$$

Then there exists $(\sigma', \mu') \in [\![\Delta', \Gamma']\!]$ such that $(\sigma', \mu', \Psi'; P' \ltimes E) \to (\sigma_{Q'}, \mu_{Q'}, Q')$ and $\sigma_Q|_H = \sigma_{Q'}|_H$ where $H = \mathsf{hprocs}(\sigma_Q, \mu_Q, Q)$.

The inverse direction does not hold since our method over-approximates values sent by residual processes.

Proof (Sketch). The proof is by induction on the derivation of $\Gamma, \Delta, P, \Psi \rightsquigarrow \Gamma', \Delta', P', \Psi'$, splitting cases on the final step. Left-movers such as sends may be sequentialized while preserving the states of halted processes, since they commute with other actions by definition [45]. Importantly, the case for R-RECV uses the fact that receives are left movers, up to their matching send. The case for R-RECV-UNFOLD relies on introducing a prophecy variable [2] that guesses the sender of the eventually received message. □

The full proof is delegated to Appendix B. Theorem 3.1 implies that we can check reachability for any halted subset of processes, consistent with results from finite-state partial order reduction techniques (*e.g.*, [68]). Thus, if we can rewrite a program P into prefix Δ, then (1) P is deadlock free and (2) process-local safety properties of Δ are enjoyed by P.

3.4 Canonical Sequentialization of $\mathbf{I_S}$ Programs

We synthesize canonical sequentializations by implementing our rewrite rules Section 3.3 as a Prolog predicate `rewrite`, shown below. Intuitively, `rewrite`$(P, \Gamma, \Delta, \Psi, P', \Gamma', \Delta', \Psi')$ holds if $\Gamma, \Delta, P, \Psi \rightsquigarrow \Gamma', \Delta', P', \Psi'$. In order to rewrite a program P, we make the query `rewrite`$(P, (\textsc{Buff}_{\varnothing}, \varnothing), \texttt{skip}, \texttt{skip}, \texttt{skip}, (\textsc{Buff}_{\varnothing}, _), \Delta, \texttt{skip})$ which rewrites P to `skip` producing its canonical sequentialization Δ.

```
rewrite(P,Γ,Δ,Ψ,P′,Γ′,Δ′,Ψ′) :- (  P = P′,Γ = Γ′,Δ = Δ′,Ψ = Ψ′
                                ;  rewrite_step(P,Γ,Δ,Ψ,P″,Γ″,Δ″,Ψ″),!,
                                   rewrite(P″,Γ″,Δ″,Ψ″,P′,Γ′,Δ′,Ψ′)
                                ).
```

Predicate `rewrite` is defined recursively: either the rewrite is complete (*i.e.*, $P = P'$), or the predicate applies a predicate `rewrite_step` which implements a *disjunction* over all rewrite rules (*i.e.*, R-SEND, R-RECV, and so on) and recursively rewrites the results. Crucially, `rewrite` performs a *cut* (using the notation !) after each successful rewrite step: once a rewrite step succeeds, the program cannot backtrack to try alternative steps. In general, this may eliminate valid rewrite sequences unless the rules are confluent (*i.e.*, the order in which rules are applied does not matter). Unfortunately, our rules are not confluent in a limited number of cases. For example, in some situations both R-SEND and R-SEND-UNFOLD are applicable. In this situation, applying R-SEND will cause the rewrite to get stuck: R-SEND consumes the send to a member of some set, and hence it is impossible to unfold the corresponding set. We address this difficulty by fixing

an ordering on these rules. For example, our implementation will always try to apply R-SEND-UNFOLD before R-SEND. The resulting system *is* confluent: the cuts do not eliminate valid rewrites. This step is crucial for ensuring that BRISK is fast enough for interactive use.

Predicate `rewrite` can also be used to rewrite a program to something other than `skip`, *e.g.*, for programs where some reactive components (*e.g.*, servers) may not terminate and we only want to rewrite the finite interaction between clients and server. Here, we rewrite the client to `skip` and keep the server as a remainder term.

Semantic Entailment. Our proof system makes use of the entailment relation, $\Delta \vDash \varphi$, *e.g.*, in R-SEND and R-RECV. As Δ may contain loops, entailment is undecidable. Computing approximations of $\Delta \vDash \varphi$ is orthogonal to this paper, however, we found that for our benchmarks (Section 3.6), loop invariants that track constants were sufficient.

3.5 From HASKELL to I_S

I_S is designed to be a convenient *target* for reasoning about distributed programs. However, is is significantly less convenient to program in than more standard, high-level programming languages. In this chapter, we thus describe a syntax-directed translation from (a subset of) HASKELL programs to I_S programs. First we describe our model of HASKELL programs by giving the syntax and semantics of a core language Section 3.5.1. Next, we formalize our translation to I_S programs in Section 3.5.2. Finally, we argue that our translation is correct in Section 3.5.3 by showing that it is sound, in the sense that translated programs only take steps allowed by the original program, and complete, in the sense that the translated program can take steps corresponding to steps taken by the original program.

3.5.1 Syntax and Semantics of λ_m

Syntax. To formalize our approach of translating HASKELL programs, we define a core calculus called λ_m that is intended to model the relevant pieces of HASKELL. λ_m is

Constants	c	::=	`true` | `false` | `0` | `1` | ...	*Pure literals*
		|	`return` | `>>=`	*Monadic literals*
		|	`self` | `send` | `recv` | `spawn`	
Expressions	e	::=	c	*Literal*
		|	x^t	*Typed Variable*
		|	$\lambda x^t.\ e$	*Abstraction*
		|	$\Lambda\alpha.\ e$	*Type Abstraction*
		|	$e\ e$	*Application*
		|	`if` e `then` e `else` e	*Control Flow*
		|	e `@`t	*Type Application*
		|	`let` x^t `=` e `in` e	*Let Binding*
		|	`rec` $x.\ e$	*Recursive Binding*
Types	t	::=	`int` | `bool` | `pid`	*Integers, Booleans, & PIDs*
		|	$t \to t$	*Functions*
		|	`P` t	*Effectful Computations*

Figure 3.10. λ_m syntax

a lambda calculus extended with message passing primitives: the message passing (also referred to as the *impure* or *effectful*) subset is *monadic*.

Definition 3.2 (Pure and impure types)**.** We say that a type t is *impure* if $t = \mathsf{P}\ t'$ for some t' or $t = t_1 \to t_2$ and t_2 is impure. If t is not impure, then it is *pure*.

The syntax of λ_m is shown in Figure 3.10. We assume that λ_m is statically typed, but the type system itself is unimportant and we do not address its particulars here, save to say that λ_m is explicitly typed as in HASKELL's core calculus [35]: in particular binders x are annotated with their types t. When clear from context, or irrelevant, we will often omit t.

Message Passing. The main primitives for message passing are `send` `@`t p m and `recv` `@`t. These functions are parametric in the messages sent or received. Thus, λ_m requires an explicit application of the type of message: The expression `send @int` p m is a computation that sends a message m of type `int` to the process with PID p, and `recv @bool` is a process that *blocks* until it receives a message of type `bool`.

Effectful Combinators. The most primitive effectful computation is `return` `@`t e,

$$\textbf{PURE}\quad \frac{e \to_\beta e'}{\langle \mu, \mathcal{H}[e]_p \rangle \to_{\lambda_m} \langle \mu, \mathcal{H}[e']_p \rangle}$$

$$\textbf{SELF}\quad \frac{}{\langle \mu, \mathcal{H}[\texttt{self}]_p \rangle \to_{\lambda_m} \langle \mu, \mathcal{H}[\texttt{return @}t\ p]_p \rangle}$$

$$\textbf{BIND}\quad \frac{}{\langle \mu, \mathcal{H}[\texttt{return @}t\ n \texttt{ >>= } m]_p \rangle \to_{\lambda_m} \langle \mu, \mathcal{H}[m\ n]_p \rangle}$$

$$\textbf{SEND}\quad \frac{}{\langle \mu, \mathcal{H}[\texttt{send @}t\ q\ m]_p \rangle \to_{\lambda_m} \langle \mu[(p,q) \leftarrow \mu(p,q), m]), \mathcal{H}[\texttt{return true}]_p \rangle}$$

$$\textbf{RECV}\quad \frac{\mu(p,q) = m, M}{\langle \mu, \mathcal{H}[\texttt{recv}]_q \rangle \to_{\lambda_m} \langle \mu[(p,q) \leftarrow M]), \mathcal{H}[\texttt{return}\ m]_q \rangle}$$

$$\textbf{SPAWN}\quad \frac{q\ \textit{fresh}}{\langle \mu, \mathcal{H}[\texttt{spawn}\ m]_p \parallel \mathcal{P} \rangle \to_{\lambda_m} \langle \mu, \mathcal{H}[\texttt{return @}t\ q]_p \parallel m_q \parallel \mathcal{P} \rangle}$$

$$\begin{array}{rrcl} \textbf{Program Contexts} & \mathcal{H}_p & ::= & E_p \parallel \mathcal{P} \\ \textbf{Evaluation Contexts} & E & ::= & [\,] \mid E \texttt{ >>= } e \end{array}$$

Figure 3.11. Semantics of λ_m terms

which has no observable side effect and, by itself, is irreducible (assuming e can not be reduced further). Message passing computations are *sequenced* with >>=. If m is a computation returning a value of type t (such as recv @t), and $\lambda x^t.\ n$ is a *function* from values of type t to effectful computations, then their composition is written $m \texttt{ >>= } \lambda x^t.\ n$, which first executes m and then *binds* the returned value to n as the value of x.

Semantics. The semantics of λ_m is formalized in Figure 3.11, and is based on [1, 30] in style (and content). We do not give the formalization of the underlying *pure* calculus, as the details are irrelevant, save to say that there is some reduction relation, $e \to_\beta e'$, which says that e reduces to e' without side effects, *e.g.*, without any any message passing.

λ_m **States.** The operational semantics of λ_m is given by the relation $\to_{\lambda_m}$, which

is a binary relation on states. In λ_m, a *state* is a pair $\langle \mu, \mathcal{P} \rangle$ of a *message buffer* μ (as as in Chapter 2) and a program $\mathcal{P}$. Programs $\mathcal{P}$ are a series of expressions joined by $\parallel$ and subscripted by the (distinct) PID of the executing process. Therefore, the program $e^0_{p_0} \parallel e^1_{p_1} \parallel \ldots$ denotes process p_0 running expression e^0 in parallel with p_1 running e^1, *etc.* We consider states equivalent up to reordering of processes.

Program Contexts. The relation $\rightarrow_{\lambda_m}$ is defined using program *contexts*, which pick a particular subexpression in a program to execute by placing "holes," [], in place of the expression to reduce. First, evaluation contexts specify the *order* of evaluation, which is to reduce the *first* argument of >>= before using the BIND rule to eliminate the >>= application. Next, program contexts pick out a particular process. For example,

$$\mathcal{H}[\texttt{return}\ @t\ n \texttt{>>=}\ m]_p$$

matches the program

$$((\texttt{return}\ @t\ n \texttt{>>=}\ m) \texttt{>>=}\ l)_p \parallel (\texttt{send}\ @\texttt{int}\ p\ 0 \texttt{>>=}\ f)_q$$

by letting $\mathcal{H} = ([\,] \texttt{>>=}\ l)_p \parallel (\texttt{send}\ @\texttt{int}\ p\ 0 \texttt{>>=}\ f)_q$

3.5.2 Translation to $\mathbf{I_S}$

Next we describe how to compile λ_m terms to $\mathrm{I_S}$ programs.

MapReduce in λ_m. Figure 3.12 contains an implementation of MapReduce in HASKELL which we use as running example. MapReduce comprises a `reducer`, a `queue`, and a set of `mapper` processes. The program is parameterized by `k`, the amount of work, and `n`, the number of `mapper` processes. A `mapper` process (lines 4-12) consists of a loop implemented in the function `mapperLoop`. In the body of the loop, the `mapper` first sends its PID to the work queue (line 7) and then waits for a response (line 8). If the response is a unit of work, the process sends (line 10) a result (in this case, just the work unit `i` itself) to the PIDof a `reducer` process and calls `mapperLoop` to begin the next iteration of

```
1  data Message = Work Int | Term
2
3  mapper :: ProcessId → ProcessId → Process ()
4  mapper q r     = mapperLoop
5    where
6      mapperLoop = do me ← getSelfPid
7                      send q me
8                      w  ← receive
9                      case w of
10                       Work i → do send r i
11                                   mapperLoop
12                       Term   → return ()
13
14 queue :: Int → [Int] → ProcessId → Process ()
15 queue n work r    = queueLoop
16  where
17    queueLoop      = do me      ← getSelfPid
18                        workers ← spawn n (mapper me r)
19                        foldM distribute () work
20                        foldM terminate () workers
21    distribute _ i = do x ← receive
22                        send x (Work i)
23    terminate  _ _ = do x ← receive
24                        send x Term
25
26 reducer :: Int → Int → Process ()
27 reducer k n       = reducerLoop
28  where
29   reducerLoop     = do me ← getSelfPid
30                        spawn 1 (queue n work me)
31                        foldM waitForUnit () work
32                        return ()
33   waitForUnit _ _ = receive
34   work            = [1..k]
35
36 main :: Process ()
37 main = do n ← readNumWorkers
38           k ← readWorkUnits
39           reducer n k
```

Figure 3.12. MapReduce implemented in HASKELL

the loop. If the response is `Term`, the loop terminates and the process exits. The work `queue` (lines 15-24) first spawns `n` `mapper` processes (18), and then uses `foldM`[1] (line 19) to wait for `k` requests (line 21). For each request, the `queue` answers with a unit of work `Work i` (line 22). Having distributed `k` units of work, the `queue` waits (using `foldM`) for each `mapper` to send a request (line 23), and responds with a `Term` message, causing the `mapper` to exit (line 24). The `reducer` process (lines 27-33) first spawns the work `queue` and then waits to receive `k` messages from the `mappers`. The `main` function first reads in the number of workers and work units and then executes the program on line 39.

Preliminary Restrictions. In this dissertation we are considering a *subset* of λ_m programs. First, we consider *defunctionalized* (*e.g.*, [51]) terms: functions are allowed as arguments to built in combinators, but otherwise we consider first-order terms. Second, we make the simplifying assumption that all of the spawn expressions in a λ_m program occur *before* any of the other effectful computations: This means that we can consider the translation to be a flat composition of processes: the translation itself is not defined when spawns may occur in the body of a loop. Second, to ease the presentation of the translation, we assume that spawned processes do not close over their environment: this behavior is equivalent to sending an *initialization message* to each newly spawned process.

Translation. We traverse the abstract syntax tree (AST) of a λ_m term e to compute the corresponding I_S program, using the types of the child expressions to guide the translation. If the type of an expression is of the form `Process t`, then the term corresponds to a tree of expressions where each node is an application of `>>=`, which corresponds to sequencing and assignment.

In Figure 3.13, we define the function $\mathcal{T}(e, x, p)$ which maps a λ_m term e, variable x and process identifier p to a I_S program. If $(A, B) = \mathcal{T}(e, x, p)$, then A is a program corresponding to the "root" process, p, of e. A assigns to x the final value computed by p. B is a flat composition of processes spawned during the execution of E.

[1] The `foldM` combinator is a standard left-fold where the "fold function" can be effectful.

$$
\begin{array}{ll}
\mathcal{T}(\texttt{self}, x, p) & \triangleq \mathcal{T}(\texttt{return @pid}\ p, x, p) \\
\mathcal{T}(\texttt{send @}t\ p\ m, x, q) & \triangleq ([\texttt{send}(t, p, m)]_q, \varnothing) \\
\mathcal{T}(\texttt{recv @}t, x, q) & \triangleq ([x \coloneqq \texttt{recv}(t, *)]_q, \varnothing) \\
\mathcal{T}(\texttt{return}\ e, x, q) & \triangleq ([x \coloneqq e]_q, \varnothing) \\
\mathcal{T}(\texttt{spawn}\ e, x, q) & \triangleq ([x \coloneqq p]_q, A) \\
 & \quad \textit{where}\ \ A = \mathcal{T}_1(e, \mathsf{ret}_p, p) \\
 & \qquad\quad p\ \ \textit{fresh} \\
\mathcal{T}(m \texttt{>>=} \lambda y^t.\ n, x, q) & \triangleq (A_1; A_2, \varrho_1 \parallel \varrho_2) \\
 & \quad \textit{where}\ \ (A_1, \varrho_1) = \mathcal{T}(m, y, q) \\
 & \qquad\quad (A_2, \varrho_2) = \mathcal{T}(n, x, q) \\
\mathcal{T}(\texttt{let}\ y^t = m\ \texttt{in}\ n, x, q) & \triangleq \begin{cases} \mathcal{T}(\texttt{return}\ m \texttt{>>=} \lambda y^t.\ n, x, q) & \mathsf{Pure}(t) \\ \mathcal{T}(n[m/y], x, q) & \textit{otherwise} \end{cases} \\
\mathcal{T}(\texttt{if}\ e\ \texttt{then}\ m\ \texttt{else}\ n, x, q) & \triangleq (\texttt{if}\ e\ \{\ A_1\ \}\ \texttt{else}\ \{\ A_2\ \}, \varrho_1 \parallel \varrho_2) \\
 & \quad \textit{where}\ \ (A_1, \varrho_1) = \mathcal{T}(m, x, q) \\
 & \qquad\quad (A_2, \varrho_2) = \mathcal{T}(n, x, q) \\
\mathcal{T}(\texttt{foldM}\ (\lambda a^{t_a}.\ \lambda y^{t_y}.\ e)\ b\ xs, q, x) & \triangleq ([a \coloneqq b]_q; \texttt{for}\ (i : I)\ \{\ A\ \}, \varnothing) \\
 & \quad \textit{where}\ A = \mathcal{T}_1(e, a, q) \\
\mathcal{T}((\textsc{Loop}\ f\ c\ m\ n)\ (\texttt{rec}\ f.\ \lambda x^t.\ b), y, p) & \triangleq (\mathcal{L}(f, x, A); \texttt{while true}\ \{\ \mathcal{L}(f, x, B)\ \}, \varnothing) \\
 & \quad \textit{where}\ A = \mathcal{T}_1((\textsc{Loop}\ f\ c\ m\ n), y, p) \\
 & \qquad\quad B = \mathcal{T}_1(e, x, p) \\
\mathcal{T}((\texttt{rec}\ f.\ (\lambda x^t.\ e))\ m, y, q) & \triangleq ([x \coloneqq m]_q; \texttt{while true}\ \{\ \mathcal{L}(f, x, A)\ \}, \varnothing) \\
 & \quad \textit{where}\ A = \mathcal{T}_1(e, y, q)
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{L}(f, x, \texttt{skip}) & \triangleq \texttt{break} \\
\mathcal{L}(f, x, [f\ e]_p) & \triangleq [x \coloneqq e]_p \\
\mathcal{L}(f, x, [s]_p; A) & \triangleq [s]_p; \mathcal{L}(f, x, A) \\
\mathcal{L}(f, x, A_1 \,\square\, A_2) & \triangleq \mathcal{L}(f, x, A_1) \,\square\, \mathcal{L}(f, x, A_2) \\
\mathcal{L}(f, x, A) & \triangleq A
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{T}_1(e, x, p) & \triangleq A \quad \textit{where}\ (A, \varnothing) = \mathcal{T}(e, x, p) \\
(\textsc{Loop}\ f\ c\ m\ n) & \triangleq \lambda f.\ \texttt{if}\ c\ \texttt{then}\ f\ m\ \texttt{else return @}t\ n
\end{array}
$$

Figure 3.13. Transformation of λ_m to $\mathrm{I_S}$

Basic Expressions. The unit of effectful λ_m programs is the `return` $@t\ e$ expression. By itself, `return` $@t\ e$ denotes a program that evaluates e and terminates. Thus, the $\mathrm{I_S}$ program corresponding to `return` $@t\ e$ is a singleton process that assigns the value of e to its "return variable."

Basic effectful primitives for sending and receiving messages and spawning processes are transformed into their $\mathrm{I_S}$ counterparts. The explicit type applications of t in `send` $@t\ p\ m$ and `recv` $@t$ are used to generate the corresponding $\mathrm{I_S}$ primitives, which require a type (channel) identifier. Note that translated receives are always "wildcard" receives: the wildcards can be eliminated by techniques discussed previously in this dissertation. It is possible to extend the language with a primitive that receives from a particular process, which would require extending the translation.

Spawning Processes. Although our goal is not the analysis of programs that use dynamic process creation heavily, we nevertheless include a case for handling some process creation (assumed to occur before the main logic of the program under translation). Thus, the case for $\mathcal{T}(\texttt{spawn}\ e, x, q)$ computes the translation of e with a freshly chosen PID, q. The translated program corresponding to the spawn itself simply returns the new PID by assigning it to x.

Sequencing Computations. Effectful computations m and $\lambda x^t.\ n$ are sequenced in λ_m using `>>=`. Thus, the translation is built by translating m and n separately, using x as the "return variable" of m.

With an eye towards verification, we optimize the translation by translating certain `let` expressions into sequenced $\mathrm{I_S}$ programs. In particular, while `let` x `=` m `in` n is often treated as syntactic sugar for $\lambda x.\ n\ m$, we consider the case when m is a pure expression (and hence reduces to a pure value via β-reduction). We translate this case by binding the value of m to x rather than substituting m for x, which allows sharing of the value via x in the final program.

Recursive Functions. In general, the most significant mismatch between λ_m and $\mathrm{I_S}$ is that λ_m employs recursive functions while $\mathrm{I_S}$ uses `while`-loops to perform

$$\vdash_f x \quad \frac{\vdash_f e}{\vdash_f \Lambda\alpha.\, e} \quad \frac{\vdash_f e}{\vdash_f e\ @t} \quad \frac{\vdash_f e}{\vdash_f \lambda x^t.\, e} \quad \frac{\vdash_f e_1 \qquad f \notin \mathsf{fv}(e_2)}{\vdash_f e_1\ e_2}$$

$$\frac{\vdash_f m \qquad \vdash_f n \qquad f \notin \mathsf{fv}(e)}{\vdash_f \ \mathtt{if}\ e\ \mathtt{then}\ m\ \mathtt{else}\ n}$$

$$\frac{\vdash_f e_2 \qquad f \notin \mathsf{fv}(e_1)}{\vdash_f \mathtt{let}\ x^t = e_1\ \mathtt{in}\ e_2} \quad \frac{f \notin \mathsf{fv}(e)}{\vdash_f \mathtt{return}\ e} \quad \frac{\vdash_f e_2 \qquad f \notin \mathsf{fv}(e_1)}{\vdash_f e_1 \mathbin{>\!\!>\!=} e_2}$$

$$\frac{\vdash_f e}{\vdash \mathtt{rec}\ f.\, e}$$

Figure 3.14. Well-formed λ_m terms

general iteration. While it would be possible to reify the function stack by transforming λ_m terms into their continuation-passing style form, this would require higher-order reasoning about $\mathrm{I_S}$ terms. Instead, we require that *effectul* recursive λ_m functions be essentially (though not strictly speaking) tail-recursive: intuitively, the recursive call should be performed as the *last action*. This requirement is expressed as a well-formedness criterion for all recursive computations. We write $\vdash \mathtt{rec}\ f.\, e$ to say that $\mathtt{rec}\ f.\, e$ is well formed. The definition of $\vdash \mathtt{rec}\ f.\, e$ is given in Figure 3.14, and closely follows the definition of tail recursion for a function f.

Observe that if $\vdash \mathtt{rec}\ f.\, \lambda x.\, e$, then then e is *equivalent* (assuming it is an effectful computation) to a term of the form $m \mathbin{>\!\!>\!=} \lambda y.\, \mathtt{if}\ c\ \mathtt{then}\ f\ a\ \mathtt{else}\ b$ where f does not appear freely in m or b. In other words, the decision to recurse can be deferred to the last step of the computation.

Example 3.8 (MapReduce in $\mathrm{I_S}$)**.** The Haskell code for `mapper` from Figure 3.12 is thus translated to a while loop in the extracted $\mathrm{I_S}$program shown in Figure 3.15, which also shows the send and receive tags, derived from the types as described in Section 3.4. Both `m` and `q` contain receives that are tagged with a single location in the set of processes `P`, while the receive in the `P` process is tagged only with sends from the `q` process. Note that since the receive at (1) is tagged with b, which appears in the worker process, the

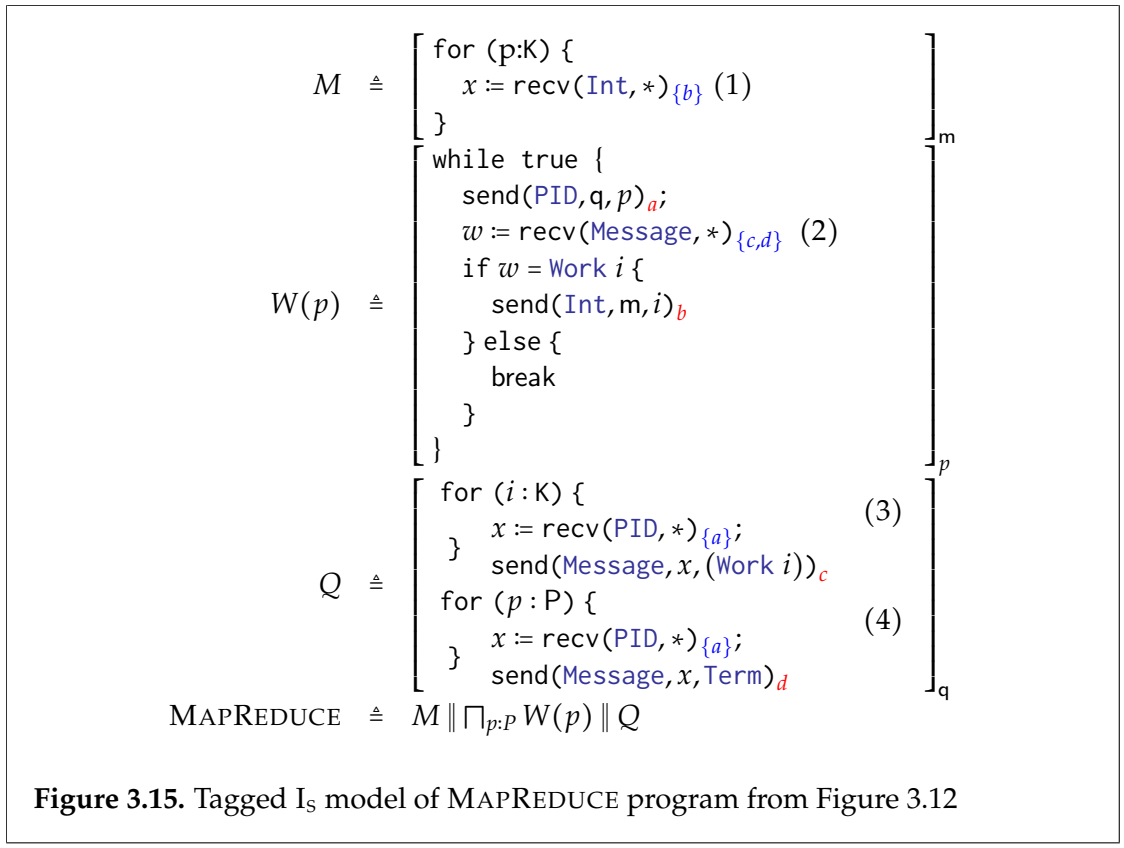


Figure 3.15. Tagged I_S model of MapReduce program from Figure 3.12

statement is equivalent to $x \coloneqq \texttt{recv(Int,P)}$. Likewise, the receive at (2) is equivalent to $w_p \coloneqq \texttt{recv(Message,q)}$ and wildcards in (3) and (4) may be instantiated to P.

3.5.3 Correctness of the Translation

The correctness criteria we are concerned with are (1) *soundness* of the translation from λ_m to I_S, in the sense that if translating e produces the program A, then the transitions of A should correspond to only *valid* transitions in e; and (2) *completeness*, in that if e can transition to e', then the translation of e can transition to the translation of e'. Intuitively, the monadic structure of λ_m makes this correspondence easy to see: in general, there is a one-to-one correspondence between the (impure) semantics λ_m and the semantics of I_S.

Lifting translations to λ_m states. In order to talk about the correspondence between executions in the two languages, we overload the function $\mathcal{T}$ to translate λ_m

program contexts. Let $\gamma : PID \to \mathsf{Identifiers}$ map PIDs to the "return variable" of each process. We lift the definition of $\mathcal{T}$ to contexts by applying $\mathcal{T}$ to each parallel-composed process.

Definition 3.3 (Translation of programs)**.** We lift the function $\mathcal{T}$ to programs by defining $\mathcal{T}(\mathcal{P}, \gamma)$ as follows

$$\mathcal{T}(\mathcal{P}, \gamma) \triangleq \begin{cases} \mathcal{T}(e, \gamma(p), p) \parallel \mathcal{T}(\mathcal{P}', \gamma) & \mathcal{P} = e_p \parallel \mathcal{P}' \\ \texttt{skip} & \mathcal{P} = \mathcal{E} \end{cases}$$

Soundness. In order to prove soundness, we want to map each execution in $\mathrm{I_S}$ to an execution in λ_m. In turn, this requires *decoding* configurations $\mathcal{C}$ from $\mathrm{I_S}$ to states in λ_m. Given $\mathcal{C} = (\sigma, \mu, A)$, we need to find all the e_p such that the composition of all $\mathcal{T}(e, x, p)$ produce A. As this may be an *open* term in general, we then produce a closed term by unloading the store σ (*i.e.*, producing $e[\sigma(p, x)/x]$ for each x in the domain of the store).

Decoding $\mathrm{I_S}$ configurations. First we need the following definitions, which handle closing open λ_m terms and producing terminated processes.

Definition 3.4 (Store Unloading)**.** We define the function $\mathsf{Apply}(\sigma, p, e)$ which unloads the store σ on the term e as

$$\mathsf{Apply}(\sigma, p, e) \triangleq e[v_0/x_0][v_1/x_1] \cdots \qquad \textit{for } \sigma(p, x_i) = v_i$$

Definition 3.5 (Closing a Term)**.** We define the function $\mathsf{Close}(\sigma, \gamma, \mathcal{P})$, which closes the program context $\mathcal{P}$ as

$$\mathsf{Close}(\sigma, \gamma, \mathcal{P}) \triangleq \begin{cases} \mathsf{Apply}(\sigma, p, e_p) \parallel \mathsf{Close}(\sigma, \gamma \setminus p, \mathcal{P}') & \mathcal{P} = e_p \parallel \mathcal{P}' \\ (\texttt{return}\ @t\ \sigma(\gamma(p)))_p \quad \textit{for } p \in \gamma & \mathcal{P} = \mathcal{E} \end{cases}$$

Terminated Processes. There is a slight impedence mismatch between processes

that have terminated in $\mathrm{I_S}$ versus those in λ_m. In particular, a halted process p in the semantics of λ_m is of the form $(\texttt{return}\ @t\ v)_p$ for some value v. A corresponding configuration in Lang is some $(\sigma, \mu, \texttt{skip})$, where $\sigma(p, \mathsf{ret}_p) = v$. Thus, when decoding such a configuration, we must produce the corresponding $(\texttt{return}\ @t\ v)_p$ term.

Loops. The other tricky part of the proof is the difference between the execution of recursive functions in λ_m versus loops in $\mathrm{I_S}$. That is, for some s,

$$\mathcal{T}((\texttt{rec}\ f.\ (\lambda x.\ e))\ m, y, p) = [x \coloneqq m]_p\,;\texttt{while true \{ } s \texttt{ \}}.$$

$[x \coloneqq m]_p\,;\texttt{while true \{ } s \texttt{ \}}$ can step to $\texttt{while true \{ } s \texttt{ \}}$, which does not correspond to the translation of a particular λ_m expression to the translation of a particular λ_m expression. Thus, it is necessary to account for these *intermediate* states reachable during the course of execution. Thus, the decoding function must first check to see if the current configuration is the result of *unrolling* a loop.

As $\mathcal{T}$ is not invertible in general, we must decode $\mathrm{I_S}$ programs as *sets* of λ_m programs. For this, we define $\mathcal{D}$ (and some helpers). $\mathcal{D}$ walks the parallel composed $A_0 \parallel A_1 \parallel \ldots$ searching for expressions that translate to the corresponding A_i.

Definition 3.6 (Decoding $\mathrm{I_S}$ Programs)**.** We *decode* $\mathrm{I_S}$ programs by defining the function $\mathcal{D}$ as follows.

$$\mathcal{D}_0(A, \gamma) \triangleq \{(e_p \parallel \mathcal{P}) \mid A = (\mathcal{T}(e, \gamma(p), p) \parallel B) \wedge \mathcal{P} \in \mathcal{D}(B, \gamma)\}$$

$$\mathcal{D}_1(A, \gamma) \triangleq \left\{ ((\texttt{rec}\ f.\ (\lambda x.\ e))\ x)_p \parallel \mathcal{P} \;\middle|\; \begin{array}{l} A = (\texttt{while true \{ } \mathcal{L}(f, x, s) \texttt{ \}} \parallel B) \\ \wedge\ e \in \mathcal{D}(s, \gamma) \wedge \mathcal{P} \in \mathcal{D}(B, \gamma) \end{array} \right\}$$

$$\mathcal{D}(A, \gamma) \triangleq \begin{cases} \{\mathcal{E}\} & A = \texttt{skip} \\ \mathcal{D}_0(A, \gamma) \cup \mathcal{D}_1(A, \gamma) & \textit{otherwise} \end{cases}$$

The helper function $\mathcal{D}_1(A, \gamma)$ accounts for the aforementioned difference between recursive functions in λ_m and `while` loops in $\mathrm{I_S}$.

We can now define how to decode $\mathrm{I_S}$ configurations:

Definition 3.7. Given a configuration $\mathcal{C} = (\sigma, \mu, A)$ and map γ, the *decoding* of $\mathcal{C}$, $\lfloor \mathcal{C} \rfloor_\gamma$, is defined:

$$\lfloor (\sigma, \mu, A) \rfloor_\gamma \triangleq \{\langle \mu, \mathsf{Close}(\sigma, \gamma, \mathcal{P}) \rangle \mid \mathcal{P} \in \mathcal{D}(A, \gamma)\}$$

Theorem 3.2 (Soundness)**.** *For all well-formed e, $(\varnothing, \varnothing, \mathcal{T}(e, \mathsf{ret}_p, p)) \rightarrow_{\mathrm{I_S}} (\sigma, \mu, \mathtt{skip}) \Rightarrow \langle \varnothing, e \rangle \twoheadrightarrow_{\lambda_m} \langle \mu, \mathsf{Close}(\sigma, , \mathcal{E}) \rangle$.*

Lemma 3.1. *Assume $\mathcal{C} \rightarrow_{\mathrm{I_S}} \mathcal{C}'$ Then, for all $\langle \mu, \mathcal{P} \rangle \in \lfloor \mathcal{C} \rfloor_\gamma$, there exists $\langle \mu', \mathcal{P}' \rangle \in \lfloor \mathcal{C}' \rfloor_\gamma$ such that $\langle \mu, \mathcal{P} \rangle \twoheadrightarrow_{\lambda_m} \langle \mu', \mathcal{P}' \rangle$.*

Completeness. The completeness of the translation execution again follows from a correspondence between states. However, the inductive lemma depends on knowing the initial σ corresponding to the starting λ_m state, as there is no function from from λ_m states to $\mathrm{I_S}$ configurations. Thus, given a σ, we consider $\mathrm{I_S}$ terms that correspond to λ_m terms closed-over by σ.

Theorem 3.3 (Completeness)**.** *For all well-formed e, when $\mathcal{T}(e, \mathsf{ret}_p, p)$ is defined, $\langle \varnothing, e \rangle \twoheadrightarrow_{\lambda_m} \langle \mu, \mathsf{Close}(\sigma, \lambda p.\mathsf{ret}_p, \mathcal{E}) \rangle \Rightarrow (\varnothing, \varnothing, \mathcal{T}(e, \mathsf{ret}_p, p)) \rightarrow_{\mathrm{I_S}} (\sigma, \mu, \mathtt{skip})$.*

Proof. The theorem follows from Lemma 3.2. □

Lemma 3.2. *For all well-formed E and σ, if $\mathcal{T}(E, \gamma)$ is defined, then there exists E'' such that $\mathcal{T}(E'', \gamma)$ is defined, $\mathsf{Close}(\sigma', \gamma, E'') = E'$ and*

$$\langle \mu, \mathsf{Close}(\sigma, \gamma, E) \rangle \twoheadrightarrow_{\lambda_m} \langle \mu', E' \rangle \Rightarrow (\sigma, \mu, \mathcal{T}(E, \gamma)) \rightarrow_{\mathrm{I_S}} (\sigma', \mu', \mathcal{T}(E'', \gamma))$$

3.6 Evaluation

To test the effectiveness of our approach, we ported programs from related work to HASKELL. Our evaluation demonstrates that a large variety of idiomatic,

Table 3.1. BRISK and SPIN results. The first grouping is a set of micro-benchmarks, the second are ported from related work, and THEQUE is our own case study. It takes at most 100 ms for BRISK to rewrite a program into its canonical sequentialization.

Name	#Param	#LOC	SPIN N	I_S #Term	BRISK time (ms)
EX2	1	14	-	69	20
EX3	1	13	11	57	20
PINGDET	1	17	13	83	20
PINGITER	1	19	11	63	20
PINGSYM	1	13	10	44	30
PINGSYM2	1	43	7	140	30
CONCDB	1	54	6	265	20
DISTDB	2	42	2	218	20
FIREWALL	1	45	9	201	30
LOCKSERVER	1	28	12	109	30
MAPREDUCE	2	64	4	205	30
PARIKH	0	35	-	173	20
REGISTRY	1	40	10	171	30
TWOBUYERS	0	59	-	332	20
2PCOMMIT	1	47	6	281	50
WORKSTEAL	2	39	5	141	40
THEQUE	3	576	3	1443	100

asynchronous message passing programs satisfy symmetric non-determinism. Even though programs satisfying symmetric nondeterminism are nontrivial to verify, BRISK can rapidly compute their canonical sequentializations, implying that our method can be easily integrated in an iterative design-implement-check loop.

Methodology. To evaluate BRISK, we recorded the time it takes to rewrite each benchmark into its canonical sequentialization. For programs that should terminate, we compute the canonical sequentialization by rewriting the entire program to `skip`. For systems with reactive components, we closed the system by adding clients that execute a finite program (*e.g.*, non-deterministically call RPCs). We then ran BRISK to compute the canonical sequentialization of the interaction of clients and the reactive components. For both classes of programs, beyond summarizing the input program's behavior, computing a canonical sequentialization serves as proof that (1) the program does not deadlock (modulo reactive components) and (2) processes do not call `fail`. We only check for simple assertion failures such as the one in Figure 3.1 by treating fail as

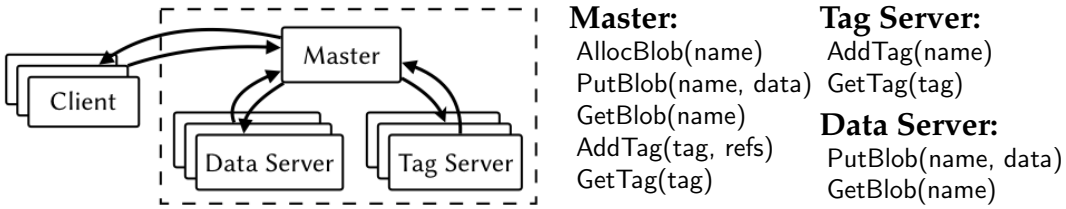


Figure 3.16. THEQUE messages (top) and remote procedure calls (bottom). An arrow from *A* to *B* indicates that *A* sends *B* messages. The dashed line surrounds the services comprising THEQUE.

a special primitive that cannot be rewritten. Checking arbitrary safety properties is an undecidable problem in general, however, one can use the computed sequentialization to check more expressive properties using off-the-shelf techniques for verifying sequential programs.

To gauge the complexity of verifying programs that satisfy symmetric nondeterminism, we have manually written a PROMELA model for each benchmark by instantiating its parameters and suitably abstracting its data values. We found the smallest instantiation such that SPIN was unable to run without consuming more than 8 GB of RAM or taking more than 60 seconds to complete. For programs containing *multiple* parameters, we instantiate each parameter with a value of the same size. We ran SPIN (optimized for safety checks with `spin -run -safety`) to to check for invalid end states. We ran our experiments on a 2.8GHz Intel® Xeon® computer.

Benchmarks. We evaluate our approach with the following benchmarks:

- EX3, EX2, PINGDET, PINGSYM, PINGITER, and PINGSYM2 are microbenchmarks from Section 3.1.
- CONCDB , PARIKH [21] and DISTDB [47] are implementations of key-value stores.
- FIREWALL [21] is a firewall mediating communication between a worker and a server.
- REGISTRY [73] contains a master process, n workers and a registry server that is used to keep track of the processes registered in the system.

- WORKSTEAL is the first phase of MAPREDUCE from Figure 3.15, where n workers compete for k jobs.
- LOCKSERVER [81] implements a lock service.
- 2PCOMMIT [43] is the classic two-phase-commit protocol for atomic commitment.
- TWOBUYERS [33] is a protocol where two "buyers" negotiate a purchase from a "seller."
- THEQUE is a prototype distributed file store that we developed, inspired by the cluster file system used in DISCO [20].

Case Study: THEQUE Cluster File System. To demonstrate that realistic systems can be developed using BRISK, we built a prototype cluster storage system, inspired by the DISCO cluster file system, called THEQUE. A THEQUE instance stores two types of data: (1) immutable, write-once *blobs* of data and (2) mutable *tags*, which are metadata. Tags are references to blobs, other tags, or both. Figure 3.16 shows the processes and remote procedure calls. Clients issue requests to a master process. Operations on blobs are forwarded to a set of data servers, tag operations are handled by a set of tag servers.

THEQUE is naturally symmetrically non-deterministic. Each type of process is responsible for a *class* of operations: the master for coordination, the data servers for data operations and the tag servers for meta-data operations. The THEQUE RPCs are partitioned into different HASKELL data types. For example, the data type `MasterAPI` defines the types of messages that the master process expects in its main event handler (*i.e.*, `AllocBlob`, `PutBlob`, `GetBlob`, `AddTag`, and `GetTag` from Figure 3.16). Likewise, `TagNodeAPI` defines the types of messages for tag nodes. Since the respective event handlers expect messages of different types, it is easy to prove THEQUE has symmetric non-determinism. For instance, tag nodes trivially cannot receive messages sent by clients, as the messages are of type `MasterAPI` which does not match the expected type `TagNodeAPI`.

In order to create a closed system, we created a module that spawns all of the necessary processes, including an unbounded number of clients that issue a non-deterministically chosen request to the Master server. We then ran BRISK on this module to verify first that its implementation is indeed non-deterministic up to symmetry and second to compute its canonical sequentialization.

Results. In Table 3.1, we compare the results of running BRISK to those of verifying deadlock freedom on finite instances with SPIN. The column labeled **#Param** indicates the number of parameterized components (sets of processes or indices). **SPIN N** indicates the *smallest* parameter instantiation such that the SPIN verification exceeds either its memory or time bound. We omit this number for the benchmarks that are not parametric, and for EX2, which is essentially deterministic. **I_S #Term** indicates the size of the intermediate I_S program (compiled from HASKELL source) measured as the number of constant and function symbol occurrences. Finally, in the column labeled **BRISK Time**, we report the time for BRISK to compute the program's canonical sequentialization. Our results show that for protocols involving asynchronous communication, the SPIN verification run suffers from the expected state space explosion for modest instantiations. In contrast, BRISK computes canonical sequentializations in less than 100 ms. In order to test how BRISK performs on faulty programs, we manually added errors such as missing sends, missing receives and sends to non-existent PID's to our benchmarks. We found that BRISK reports those errors in around the same amount of time.

3.7 Summary

In this chapter we have described *canonical sequentialization*, a new approach to verifying parameterized distributed programs. We have implemented canonical sequentialization in BRISK, which rewrites distributed programs written in Haskell. BRISK verifies the *unbounded* versions of distributed programs in tens of *milliseconds*, yielding the first concurrency verification tool that is fast enough to be integrated into a design-implement-check cycle.

Acknowledgments

Chapter 3, in full, is adapted from the material as it appears in Alexander Bakst, Klaus v. Gleissenthall, Rami Gökhan Kıcı, and Ranjit Jhala. Verifying Distributed Programs via Canonical Sequentialization. *Accepted for publication in the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2017*. The dissertation author was the primary investigator and author of this paper.

Chapter 4

Pretend Synchrony

In Chapter 3 we saw that it is possible to reduce certain kinds of distributed system to single threaded programs. In principle, this should mean that verifying the correctness of implementations should be made simpler by producing canonical sequentializations, for an appropriate notion of correctness. Indeed, for some generic properties such as deadlock-freedom, the (derivation of a) sequentialization itself serves as a certificate to the correctness of the program. However, the question of to what extent sequentialization may be employed to simplify proofs of functional correctness remains murky.

In this chapter we consider a rewriting system in the vein of Chapter 3 to study this question. We formalize an approach that lets the user annotate the program with invariants that assume a *synchronous* execution model, and hence need not consider the status of undelivered messages. The rewriting logic described in this chapter propagates the user annotations to the rewritten program, at which point a standard algorithm can be used to generate and check the verification condition for the program.

The work is motivated by considering the problem of achieving consensus in a distributed system. Noting that canonical sequentializations may not exist for well known consensus protocols such as Raft [55] or Paxos [41], the rewriting logic presented in this chapter can be viewed as a *generalization* of the approach in Chapter 3. Rather than constraining the logic to *only* produce sequential programs, we describe a logic that (like canonical sequentializations) synchronizes the communication between processes

but (unlike canonical sequentializations) may produce a coarse-grained *parallel* program, rather than a sequentialization.

In this chapter we describe the following contributions:

1. **Synchronization.** Our first contribution is an algorithm to automatically compute synchronizations (Section 4.3), Our algorithm proceeds by iteratively applying a number of *local rewrite rules* each of which transforms the input program into a new program comprising a *synchronized prefix* and a suffix that still needs to be rewritten. The algorithm is sound in the sense that the rewritten program preserves the halting states of the program (Theorem 4.1).

2. **Verification.** Our second contribution is a rely-guarantee based method for generating verification conditions for the synchronized program, that traverses the program in a syntax-directed fashion, and composes the code with the synchronization invariants to emit a logical formula whose validity implies the correctness of the original source (Section 4.4).

3. **Evaluation.** Our third contribution is an implementation and empirical evaluation of pretend synchrony in a tool called CHAI. We apply CHAI to four challenging case studies: the classic two phase commit protocol, a distributed key-value store, the Raft leader election protocol and single decree Paxos (Section 4.5). We show that all these protocols have simple and intuitive synchronous invariants that CHAI checks automatically. Our evaluation demonstrates that pretend synchrony significantly reduces proof effort. While other approaches relying directly on asynchronous invariants have annotations burdens between 9x [31] and 90x [82], pretending synchrony shrinks the proof burden to just around 0.3x, thereby drastically easing verification by more than an order of magnitude.

The chapter is organized as follows. We give an overview of our approach in Section 4.1. In Section 4.2 we briefly discuss extensions to I_S. In Section 4.3 we formalize our approach to pretending synchrony as a set of rewrite rules. In Section 4.4 we discuss

verification condition generation for synchronized programs. In Section 4.5 we evaluate our approach on a set of case studies. Finally, we make concluding remarks in Section 4.6.

4.1 Overview

We begin with an overview of our approach. First, we motivate pretend synchrony at a high-level. Next, we describe the individual ingredients that make our approach feasible.

Two-Phase Commit. Figure 4.1 shows the classic two phase commit protocol [43]. In this protocol, a coordinator node *c* tries to commit a value to a number of database nodes *dbs*. The protocol is made up of two rounds. In the first round, the coordinator loops over all nodes to send them its proposal value. Each database node then nondeterministically chooses to either commit or abort the transaction and sends its choice to the coordinator. If at least one of the nodes chose to abort, the coordinator aborts the entire transaction by setting the appropriate flag. In the protocol's second round, the coordinator broadcasts its decision to either commit or abort to the database nodes which reply with an acknowledgement. Finally, if the coordinator decided to commit the transaction, each database node sets its value to the previously received proposal.

Correctness. In order to prove correctness of our implementation of two phase commit, we want to show that, if the protocol finished and the coordinator decided to commit the transaction, all database nodes have indeed chosen the proposed value. More formally, we require the following assertion to hold after the protocol execution.

$$\forall \texttt{p} \in \texttt{dbs} : \texttt{c.committed} = \texttt{true} \Rightarrow \texttt{p.value} = \texttt{c.proposal} \tag{4.1}$$

Asynchronous Invariants: Complicated. Let us first consider how to prove this property in an *asynchronous* setting. We follow the proof[1] from [80] . Consider the

[1] For the full invariant, see https://github.com/DistributedComponents/disel/blob/master/Examples/TwoPhaseCommit/TwoPhaseInductiveInv.vlines 56-198.

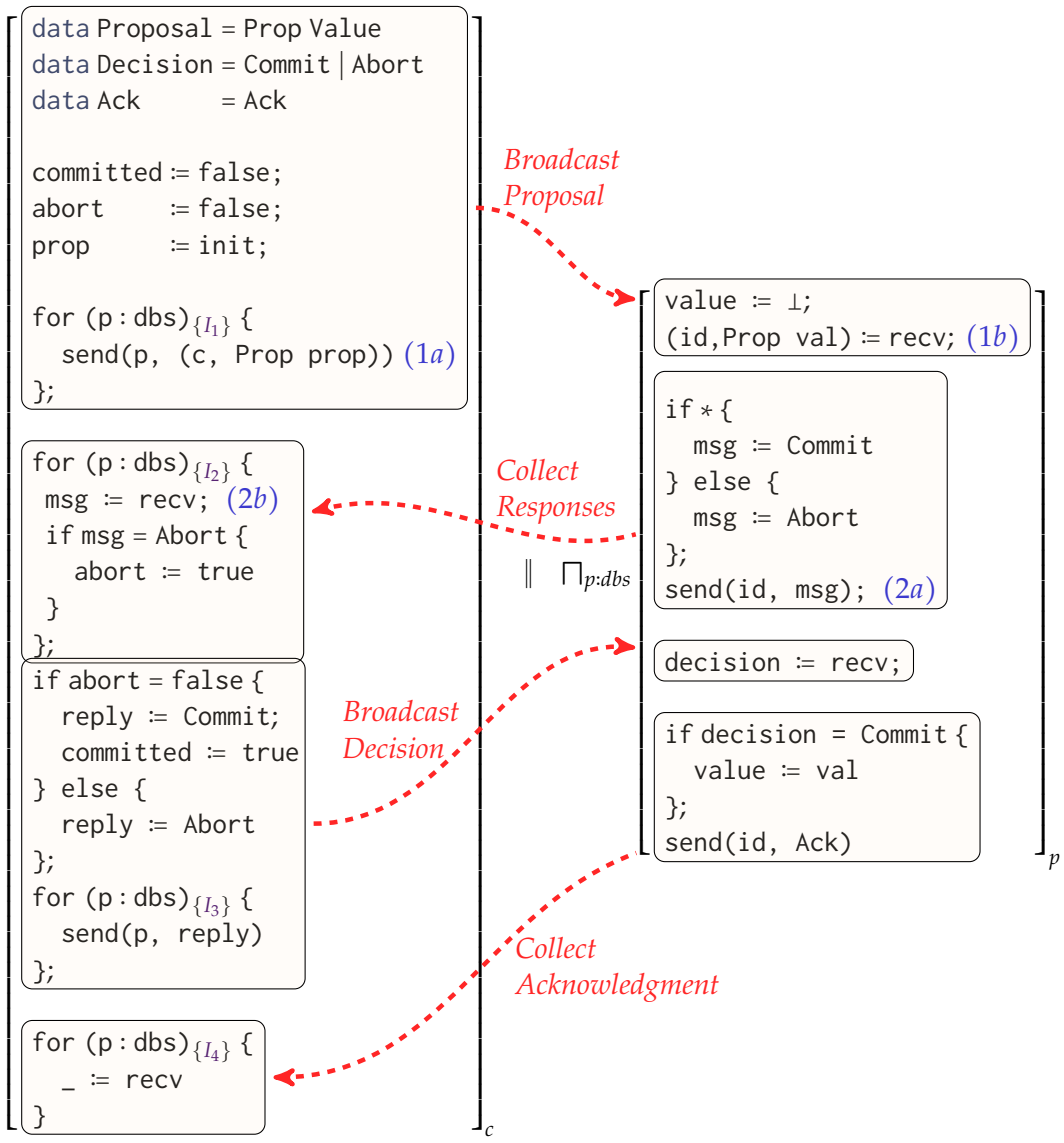


Figure 4.1. Two phase commit. Arrows denote the flow of messages during each logical phase of the protocol. The boxes of code at either end of an arrow will be synchronized by our method. Each `for` loop is annotated with an invariant {I}.

coordinator's first loop in Figure 4.1 and let *done* denote the set of all database nodes for which the send at location (1a) has been executed, so far. In order to rule out messages appearing "out of thin air", we need to assert that whenever $p \notin done$, then there are no messages from c to p, and p has not yet executed the corresponding receive at location (1b). If, $p \in done$, we need to case split over the location of p due to the asynchronous nature of communication. Either

1. There is an in-flight message from c to p that contains c's process id and proposal value and p is waiting to receive the message, or
2. The process p received c's message, set its `val` variable to `proposal` and decided to either commit or abort the transaction, but did not respond yet, or
3. The process p responded, relaying its decision to c.

We need a similar case split for c's second loop: if $p \notin done$ then either

1. There is a pending message from c to p containing c's process id and proposal, or
2. The process p has chosen to commit or abort but has not yet sent a response, or
3. The process p has sent its response consisting of either a commit or abort message.

Finally, if $p \in done$, then p must have finished the first part of the protocol and `val` must be set to c's proposal. The invariant for the second part of the protocol consists of a similar case split.

Asynchrony Makes Verification Undecidable. While avoiding such case splits — over the joint state of the coordinator, the database nodes and the message buffer — would be desirable in itself, there is a more fundamental issue with proving correctness in an asynchronous setting: directly including the message buffer into the system state by modeling it as an array requires *nested array reads* [2] which makes even *checking* a candidate invariant undecidable [8]

[2] A message buffer is a function (`processID` → `processID` → `Type` → `Int`) → `Value` that maps a typed channel (*i.e.*, messages of a particular type between two process) and a message counter for that channel to a value.

Pretend Synchrony. Even though the program behaves asynchronously when *executed*, we can soundly pretend that communication is synchronous when *reasoning* about the program. Consider, for example, the send marked with (1a) in Figure 4.1 and assume that in the current iteration coordinator c sends to some process p. Recalling the reasoning from Chapter 3, the message can only be received at a *single receive*, namely the receive of p at the location marked with (1b) in Figure 4.1 (the following receive is expecting a value of type `Decision`). Moreover, the send is *independent* of all sends between c and *other* processes in `dbs`.

Synchronous Invariants: Simple. Our method exploits this insight in the following way: instead of writing invariants for the asynchronous program, the user writes invariants *as if the program were synchronous*, effectively treating matching send and receive pairs as assignments. The synchronous invariants are given as annotations to for loops. Consider again Figure 4.1. For the first loop, we need the following invariant I_1 stating that, if the loop was executed for some process p (indicated by $p \in done$), then p must have been assigned c's proposal value.

$$I_1 \triangleq \lambda done. \forall \texttt{p.p} \in done \Rightarrow \texttt{p.val} = \texttt{c.proposal}$$

Invariants I_2 and I_3 are trivial, as the respective loops do not modify any relevant values, and need not be supplied by the user. Finally, invariant I_4 states that whenever the loop has been executed for a process p, process p's `value` variable must be set to the proposal it received in the first round.

$$I_4 \triangleq \lambda done.\ \forall p.p \in done \wedge \texttt{c.committed} = \texttt{true} \Rightarrow \texttt{p.value} = \texttt{p.val}$$

Together, these simple synchronous invariants – free of the case-splits needed to account for asynchrony – are sufficient to prove the correctness property for the two phase commit implementation.

Checking the Synchronous Invariant. We verify the correctness of the syn-

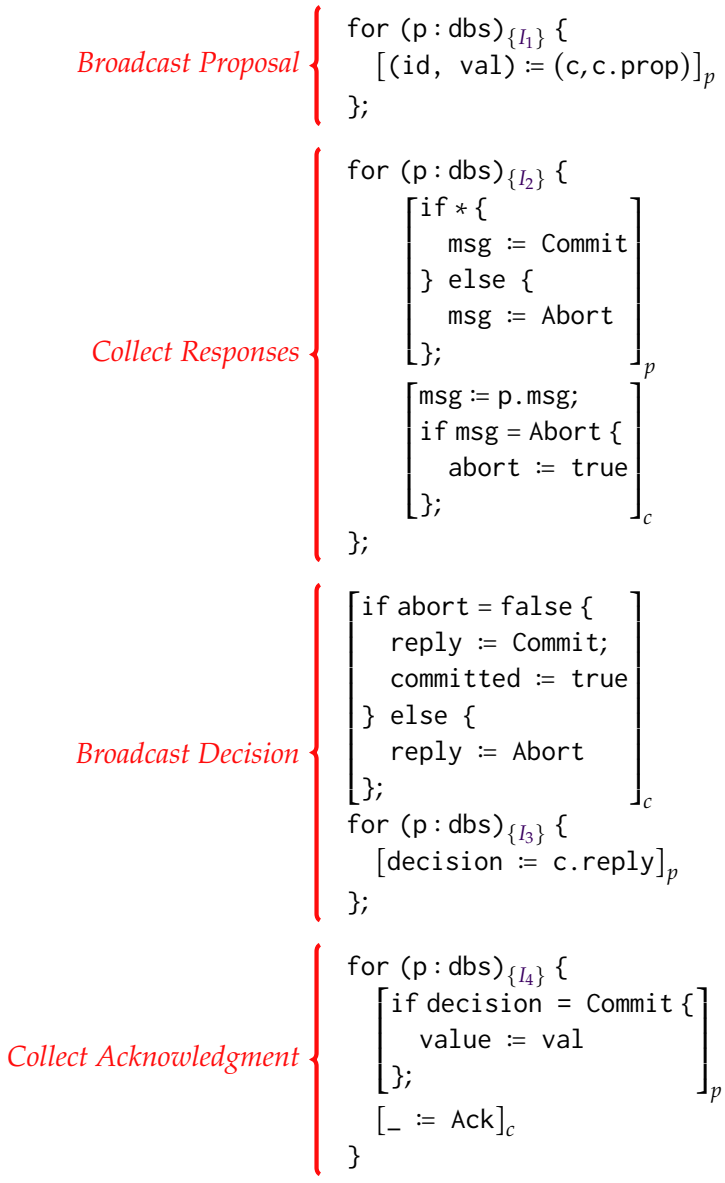


Figure 4.2. Synchronization of two phase commit.

chronous invariant in two steps. First, we compute the *synchronization*; a semantically equivalent synchronous program, by iteratively applying a set of rewriting rules (Section 4.3). Second, we use the synchronization to generate and check verification conditions that ensure that the supplied invariants are inductive and imply the desired correctness property (Section 4.4).

In the first step, our implementation computes this synchronization completely automatically without relying on the user supplied invariants. Each rewriting step produces a new program that comprises a *synchronized prefix* and a suffix that still needs to be rewritten. This process is repeated till the suffix is reduced to `skip`. Figure 4.2 shows the synchronization that CHAI computes for two phase commit. Intuitively, the synchronous version matches up the connected blocks from Figure 4.1. In the synchronized version, all protocol steps are executed one after the other: first, the coordinator assigns its proposal to all database nodes; then, each node decides to either commit or abort; then, the coordinator assigns its decision to each of the database nodes, and, finally, each node assigns the proposed value in case the coordinator decided to commit.

In the second step, CHAI then uses the synchronization and the user supplied invariants to compute verification conditions that ensure that the program satisfies all assertions. CHAI checks the verification conditions by discharging them to an SMT-solver. Importantly, the verification conditions for checking the invariants in the synchronous setting fall into the array property fragment [8] and hence, checking the invariant is decidable.

4.2 Verification with I_S

In order to build a verification framework based on I_S programs, we extend the language as presented in Chapter 2, shown in Figure 4.3. These extensions mostly *add* new forms for specifying (and ultimately verifying) loop invariants.

Invariants. In I_S, each `for`-loop is annotated with two loop invariants $\{I_C, I_S\}$.

Expressions	e	::=	c	*Constants*
		|	$i.x$	*Variables*
		|	X	*Parameters*
		|	$\texttt{f}(\overline{e})$	*Local Computation*
Statements	s	::=	$i.x \coloneqq e$	*Assignment*
		|	$i.x \coloneqq \texttt{recv}(t, \texttt{w})$	*Message Receive*
		|	$\texttt{send}(t, e, e)$	*Message Send*
		|	$\texttt{assume}(e)$	*Assumption*
		|	$\texttt{assert}(e)$	*Assertion*
		|	`break`	*Loop Exit*
Programs	A	::=	`skip`	*No Behavior*
		|	$[s]_x$	*Singleton Program*
		|	$A\,;A$	*Sequential Composition*
		|	$A \,\square\, A$	*Non-deterministic Branch*
		|	$\langle A \triangleright e \rangle$	*Atomic*
		|	$\texttt{for}\ (x : X)_{\{\varphi, \lambda i.\ \varphi\}}\ \{\ A\ \}$	*Iteration*
		|	$\texttt{while true}\ \{\ A\ \}$	*Looping*
		|	$A \parallel A$	*Parallel Composition*
		|	$\bigsqcap_{i:e} A$	*Iterated Parallel Composition*
		|	$\bigsqcup_{i:e} A$	*Nondeterministic Value*

$\texttt{w} \in \{e, *\}$ $\quad i, x \in \mathsf{Identifiers}$ $\quad X \in \mathsf{ParamIdentifiers}$ $\quad t \in \mathsf{TypeIdentifiers}$

Figure 4.3. $\mathrm{I_S}$ Syntax. Changes with respect to Figure 2.1 hilighted in gray.

The first, I_C is a *communication* invariant that is checked during the rewrite to a synchronous program (Section 4.3). We find that, in practice, these invariants are usually simple. For all our examples, CHAI computes communication invariants completely automatically. The invariant I_S is a *synchronous data invariant*, *i.e.*, a loop invariant over the synchronization. Synchronous data invariants are user supplied and used by our verification condition generation procedure (Section 4.4), which checks that the invariants indeed hold on the *synchronization*.

Atomic Actions. The language includes atomic actions $\langle P \triangleright e \rangle$, which are *not* written by the user – instead, they are computed by our rewriting rules (Section 4.3). An action $\langle P \triangleright e \rangle$ comprises a program P that executes *atomically* and an *annotation* e

BUFF	$\in$	$(PID \times PID \times \mathsf{MsgType}) \rightharpoonup \overline{e}$	**Buffers**
ASRT	::=	$\varnothing \mid \text{ASRT} \cup \{\texttt{unfold}(p, x, P)\} \mid \text{ASRT} \cup \{\texttt{split}(X, x, I_C)\}$	**Assertions**
Γ	::=	(BUFF, ASRT)	**Contexts**

Figure 4.4. Syntax for context Γ. Changes with respect to Figure 3.3 highlighted in gray.

that is derived from the supplied synchronous data invariants, and used to generate verification conditions (Section 4.4). We sometimes omit e if it equals true.

4.3 Rewrite Rules

In this section, we formalize our approach by describing our rewriting rules for producing synchronizations. We focus our attention on programs that iterate over sets of processes, only, and briefly discuss how our method extends to while loops later (Section 4.3.5).

Symbolic States. Each rewriting rule defines a relation between a pair of *symbolic states*, whose syntax is summarized in Figure 4.4. A symbolic state comprises a *context* (Γ); a *synchronized prefix* (Δ); and the program (P) to be rewritten. Synchronized prefixes Δ are $\mathrm{I_S}$ programs *without* sends or receives. A context Γ consists of a symbolic message buffer BUFF and a set of assertions ASRT. BUFF maps each channel to the sequence of pending messages on that channel, *i.e.*, $\text{BUFF}(p, q, t)$ returns the sequence of pending messages of type t sent from p to q. ASRT contains assertions about process identifiers and is populated during the rewrite. Finally, for a context $\Gamma \triangleq (\text{BUFF}, \text{ASRT})$ we write $\Gamma(p, q, t)$ to mean $\text{BUFF}(p, q, t)$ and $\Gamma \vdash a$ to mean $a \in \text{ASRT}$.

Rewriting Rules. Each rewriting rule defines a judgment of the form $\Gamma, \Delta, P \rightsquigarrow \Gamma', \Delta', P'$. The goal of each step is to move parts of program P into the prefix Δ such that eventually we can rewrite P to `skip`. We now describe the main rules of our method, starting with rules rules for loop-free programs, programs that contain communication between a single process and a group of processes, and finally communication between

sets of processes.

4.3.1 Rewriting loop-free programs

Figure 4.5 shows our main rules for rewriting loop-free programs. These rules are essentially the same as their counterparts from Chapter 3.

Message Passing. Rule R-SEND handles message sends. The rule rewrites a send from process p to some x into `skip` if x evaluates to some PID q. The rule checks that x corresponds to some PID q through the condition $\Delta \vDash x = q$. The rule updates Γ by enqueuing the expression n in the buffer for the respective channel between p and q, where we use $\Gamma[a \leftarrow b]$ to denote the function that returns the same values as Γ on all inputs except for a where it returns b. Rule R-RECV rewrites a receive from some x into `skip`, if x corresponds to a PID p. R-RECV dequeues the first message m from the respective channel and sequences the corresponding assignment after Δ.

Branching and Context. Branching is handled through nondeterministic choice, where $A \,\square\, B$ is a choice between A and B. The rule R-CHOICE rewrites a program with branching by rewriting each branch independently, where both A and B must be rewritten to `skip`. The resulting synchronous prefix consists of a choice between the prefixes of the individual branches. The rule R-FALSE rewrites unreachable program points to `skip`. Rule R-CONTEXT allows rewriting a program A independently of statements that are executed after, or in parallel. Our method contains an additional rule, R-CONGRUENCE (omitted for brevity, but analogous to Chapter 3), which allows rewriting program A to B if $A \equiv B$. For instance, the rule allows rewriting `skip`; P to P, for any program P.

Example 4.1. Consider example EX1, where we have replaced wildcard receives with receives from the respective processes.

$$\text{EX1}_{\text{ASYNC}} \triangleq \begin{bmatrix} \texttt{send(q, ping);} \\ \texttt{_} \coloneqq \texttt{recv(q)} \end{bmatrix}_p \parallel \begin{bmatrix} v \coloneqq \texttt{recv(p);} \\ \texttt{send(p, Ack)} \end{bmatrix}_q$$

The goal is to rewrite $\text{EX1}_{\text{ASYNC}}$ to `skip` producing the synchronization EX1_{SYNC}:

$$\text{EX1}_{\text{SYNC}} \triangleq \begin{array}{l} [v \coloneqq \texttt{ping}]_q\,; \\ [_ \coloneqq \texttt{Ack}]_p \end{array}$$

For this, we define the initial context as $\Gamma_0 \triangleq (\text{BUFF}_\varnothing, \varnothing)$ where the empty buffer $\text{BUFF}_\varnothing$ maps every channel to an empty sequence, and we assume a single messages type, $\top$. We first apply R-CONTEXT to select the send statement in p's program, and then apply rule R-SEND to reduce the send to `skip`, producing the program below and updating the buffer to $\text{BUFF}_\varnothing[(p, q, \top) \leftarrow \texttt{ping}]$.

$$\begin{bmatrix} \texttt{skip;} \\ \texttt{_ := recv(q)} \end{bmatrix}_p \parallel \begin{bmatrix} v \coloneqq \texttt{recv(p);} \\ \texttt{send(p, Ack)} \end{bmatrix}_q$$

Next, we apply R-CONTEXT with R-RECV to yield the following with buffer $\text{BUFF}_\varnothing$ and prefix $[v \coloneqq \texttt{ping}]_q$.

$$\begin{bmatrix} \texttt{skip;} \\ \texttt{_ := recv(q)} \end{bmatrix}_p \parallel \begin{bmatrix} \texttt{skip;} \\ \texttt{send(p, Ack)} \end{bmatrix}_q$$

Applying the congruence $\texttt{skip}; s \equiv s$ and repeating the same sequence of rules yields the target EX1_{SYNC}.

4.3.2 Communication between a Single Process and a Group

Next we present our method for rewriting loops over symmetric sets of process identifiers. At a high-level, in order to reason about an unbounded collection of processes, we (1) *focus* on a single (arbitrarily chosen) process, (2) *synchronize* the interactions with that process, and (3) *generalize* to the entire set. This strategy is inspired by techniques such as temporal case splitting from model checking [50], materialization & blurring from shape analysis [65], or equivalently, unfold & fold or focus & adoption for linear type systems [26]. Next, we formalize this strategy in our rewrite rules.

Iterating over symmetric processes. Figure 4.6 shows rule R-LOOP-UPD, which rewrites the interaction between a set of symmetric processes P and a single process q that iterates over P. In order to rewrite the entire loop, R-LOOP-UPD requires showing that a single interaction between q and an arbitrary process $u \in P$ can be rewritten. For this, the rule picks a fresh process identifier u and a fresh loop variable x. It then adds a permission to communicate with process u (*i.e.*, to "unfold" u from set P) to the context Γ, and strengthens the prefix Δ by assuming a communication invariant I_C. Intuitively, the communication invariant contains information about who to send to and who to receive from, however, we find that this invariant is often very simple. Finally, the rule requires invariant I_C to be inductive. If the rewrite succeeds, the rule extends the synchronous prefix by repeating the synchronization that was computed for a single iteration, once for every process in P. Note that the context Γ after the rewrite is required to be the same as the initial context. This ensures that all messages sent in a given iteration must also be received in that iteration.

Unfolded processes. Figure 4.6 contains the rules for unfolding process u from a set P. Rule R-VAR-UNFOLD binds u to x, allowing the process that iterates over P to send or receive from u via its loop variable. Rule R-SYM-UNFOLD unfolds u through a symmetric receive, *i.e.*, it transforms a receive from an arbitrary process in P to a receive from u. Note that both rules consume the unfold permission, *i.e.*, only a single process can be unfolded in a given iteration.

Example 4.2. We now rewrite EX2 into its synchronization EX2_{SYNC} starting from the initial context $\Gamma_0 \triangleq (\text{BUFF}_{\varnothing}, \varnothing)$. For brevity, we omit annotation.

$$\text{EX2} \triangleq \left[\begin{array}{l}\texttt{for (q : qs) \{}\\ \quad\texttt{send(q, ping);}\\ \quad\texttt{_ := recv(q);}\\ \texttt{\}}\end{array}\right]_p \parallel \bigsqcap_{q:qs} \left[\begin{array}{l}v \coloneqq \texttt{recv}(p);\\ \texttt{send(p, Ack);}\end{array}\right]_q \qquad \text{EX2}_{\text{SYNC}} \triangleq \begin{array}{l}\texttt{for (q : qs) \{}\\ \quad [v \coloneqq \texttt{ping}]_{\texttt{q}};\\ \quad [_ \coloneqq \texttt{Ack}]_p\\ \texttt{\}}\end{array}$$

The rewrite begins with an application of R-LOOP-UPD which picks an arbitrary iteration

R-CONTEXT

$$\frac{\Gamma, \Delta, A \rightsquigarrow \Gamma', \Delta', A'}{\Gamma, \Delta, A \circ B \rightsquigarrow \Gamma', \Delta', A' \circ B}$$

R-CHOICE

$$\frac{\Gamma, \Delta, A \rightsquigarrow \Gamma, \Delta; \Delta_A, \texttt{skip} \qquad \Gamma, \Delta, B \rightsquigarrow \Gamma, \Delta; \Delta_B, \texttt{skip}}{\Gamma, \Delta, A \,\square\, B \rightsquigarrow \Gamma, \Delta; \Delta_A \,\square\, \Delta_B, \texttt{skip}}$$

R-FALSE

$$\frac{\Delta \vDash \texttt{false}}{\Gamma, \Delta, A \rightsquigarrow \Gamma, \Delta, \texttt{skip}}$$

R-SEND

$$\frac{\begin{array}{ll}\Delta \vDash x = q & \Gamma(p,q,t) = m \\ q \text{ is a PID} & \Gamma' = \Gamma[(p,q,t) \leftarrow m{\cdot}n]\end{array}}{\Gamma, \Delta, [\texttt{send}(t,x,n)]_p \rightsquigarrow \Gamma', \Delta, \texttt{skip}}$$

R-RECV

$$\frac{\begin{array}{ll}\Delta \vDash x = p & \\ p \text{ is a PID} & \Gamma' = \Gamma[(p,q,t) \leftarrow n] \\ \Gamma(p,q,t) = m{\cdot}n & \Delta' = \Delta; [y \coloneqq p.m]_q\end{array}}{\Gamma, \Delta, [y \coloneqq \texttt{recv}(x,t)]_q \rightsquigarrow \Gamma', \Delta', \texttt{skip}}$$

Figure 4.5. Rewrite Rules (Basic Statements)

R-LOOP-UPD

$$\frac{\begin{array}{l}u, x \textit{ fresh} \\ \Gamma_0 \triangleq \Gamma \cup \{\texttt{unfold}(u,x,P)\} \text{ and } \Delta_0 \triangleq \texttt{assume}(I_C) \qquad \Delta' \triangleq \Delta\,; \left[\begin{array}{l}\texttt{for } (p:P)_{\{I_S\}} \,\{ \\ \quad \Delta^u[p/u] \\ \}\end{array}\right] \\ \Delta \vDash I_C \text{ and } (\Delta_0; \Delta^u) \vDash I_C \\ \Gamma_0, \Delta_0, [A]_u \parallel B[x/p] \rightsquigarrow \Gamma, (\Delta_0; \Delta^u), \texttt{skip}\end{array}}{\Gamma, \Delta, \bigsqcap_{p:P} \left[A\right]_p \parallel \left[\begin{array}{l}\texttt{for } (p:P)_{\{I_C, I_S\}} \,\{ \\ \quad B \\ \}\end{array}\right]_q \rightsquigarrow \Gamma, \Delta', \texttt{skip}}$$

R-VAR-UNFOLD

$$\frac{\Gamma \vdash \texttt{unfold}(u,x,P) \qquad \Gamma' \triangleq \Gamma \setminus \{\texttt{unfold}(u,x,P)\}}{\Gamma, \Delta, A \rightsquigarrow \Gamma', \Delta \cup \{\texttt{assume}(x = u)\}, A}$$

R-SYM-UNFOLD

$$\frac{\Gamma \vdash \texttt{unfold}(u,x,P) \qquad \Gamma' \triangleq \Gamma \setminus \{\texttt{unfold}(u,x,P)\}}{\Gamma, \Delta, y \coloneqq \texttt{recv}(P,t) \rightsquigarrow \Gamma', \Delta, y \coloneqq \texttt{recv}(u,t)}$$

Figure 4.6. Rewrite Rules (Iteration over sets of processes and unfolding)

of the loop, generating the rewrite obligation $\textsc{Ex2}_{\textsc{U}}$ shown below.

$$\textsc{Ex2}_{\textsc{U}} \triangleq \begin{bmatrix} \mathtt{send}(x, \mathtt{ping}); \\ _ \coloneqq \mathtt{recv}(x) \end{bmatrix}_p \parallel \begin{bmatrix} v \coloneqq \mathtt{recv}(p); \\ \mathtt{send}(p, \mathtt{Ack}) \end{bmatrix}_u$$

As the new obligation occurs in the context $\Gamma \cup \{\mathtt{unfold}(u, x, Q)\}$, the rewrite consumes $\{\mathtt{unfold}(u, x, Q)\}$ to produce $\textsc{Ex2}'_{\textsc{U}}$ through an application of R-VAR-UNFOLD.

$$\textsc{Ex2}'_{\textsc{U}} \triangleq \begin{bmatrix} \mathtt{assume}(x = u); \\ \mathtt{send}(x, \mathtt{ping}); \\ _ \coloneqq \mathtt{recv}(x) \end{bmatrix}_p \parallel \begin{bmatrix} v \coloneqq \mathtt{recv}(p); \\ \mathtt{send}(p, \mathtt{Ack}) \end{bmatrix}_u$$

The rewrite of the body does not depend on any loop-carried state, so the invariant $I_C \triangleq \mathtt{true}$ allows to rewrite the program to `skip` with prefix $\left([v \coloneqq \mathtt{ping}]_u ; [_ \coloneqq \mathtt{Ack}]_p\right)$. This lets us rewrite EX2 to `skip` with the prefix $\textsc{Ex2}_{\textsc{Sync}}$.

Example 4.3. Next we consider a program $\textsc{Ex}_{\textsc{Bad}}$which our rules fail to rewrite. In $\textsc{Ex}_{\textsc{Bad}}$, each iteration of p's for-loop can receive from an arbitrary process in qs. However, p also sends to the process bound by its loop variable causing it to potentially communicate with two different processes in the same iteration. As this breaks the symmetry between the processes in qs, it is impossible to rewrite $\textsc{Ex}^1_{\textsc{Bad}}$ into a synchronous for-loop. Applying R-LOOP-UPD generates the obligation to rewrite $\textsc{Ex}^1_{\textsc{Bad}}$ to `skip`. Applying R-SYM-UNFOLD to $\textsc{Ex}^1_{\textsc{Bad}}$ yields $\textsc{Ex}^2_{\textsc{Bad}}$ and consumes the permission $\mathtt{unfold}(u, x, qs)$. At this point, rewriting is doomed: applying R-SEND and R-RECV yields the program

$$[\mathtt{send}(x, x);]_p \parallel [_ \coloneqq \mathtt{recv};]_u$$

which cannot be further rewritten, as the value of x is unknown.

$$\mathrm{EX}_{\mathrm{BAD}} \triangleq \begin{bmatrix} \texttt{for (q : } qs\texttt{) \{} \\ \quad _ \coloneqq \texttt{recv(}qs\texttt{);} \\ \quad \texttt{send(q, q)} \\ \texttt{\}} \end{bmatrix}_p \parallel \bigsqcap_{q:qs} \begin{bmatrix} \texttt{send(}p\texttt{,ping);} \\ _ \coloneqq \texttt{recv(}p\texttt{)} \end{bmatrix}_q$$

$$\mathrm{EX}^1_{\mathrm{BAD}} \triangleq \begin{bmatrix} _ \coloneqq \texttt{recv(}qs\texttt{);} \\ \texttt{send(}x\texttt{,}x\texttt{)} \end{bmatrix}_p \parallel \begin{bmatrix} \texttt{send(}p\texttt{,ping);} \\ _ \coloneqq \texttt{recv(}p\texttt{)} \end{bmatrix}_u \qquad \mathrm{EX}^2_{\mathrm{BAD}} \triangleq \begin{bmatrix} _ \coloneqq \texttt{recv(}u\texttt{);} \\ \texttt{send(}x\texttt{,}x\texttt{)} \end{bmatrix}_p \parallel \begin{bmatrix} _ \coloneqq \texttt{recv} \end{bmatrix}_u$$

4.3.3 Communication between Groups of Processes

Finally, we turn to the rules for rewriting pairs of symmetric groups of processes. The addition of these rules marks a crucial increase to the expressiveness of the system described in Chapter 3.

Set to set. The rule R-SYM-RMV handles rewriting the parallel composition of two *sets* of PIDs, P and Q. Analogous to R-LOOP-UPD, the rule works by (a) *focusing* on an element $u \in P$, (b) *rewriting* the interaction of u with the members of Q, and (c) *generalizing* the interactions to all of the members of P. As before, the rule picks a fresh pid u and strengthens the prefix Δ through a communication invariant I_C which is required to be inductive. Note that the rule substitutes the singleton set $\{u\}$ for P causing all loops in process q to iterate over u, only. If the rewrite succeeds, the rule extends the synchronous prefix by repeating the synchronization Δ^u in parallel, once for each $u \in P$.

Atomic actions. The rule R-FOR-ATOMIC complements R-SYM-RMV: given a loop over a set that has been focused on a single element, R-FOR-ATOMIC generates the permission to *use* element by unfolding it. For this, the rule rewrites the loop body to produce the synchronization Δ^u. As unfolding ensures that the rewrite is restricted to the interaction between the process pair u and q, we know that the resulting interaction can be executed *atomically* and satisfies loop invariant I_s.

$$\frac{}{\mathcal{X} \vdash \mathsf{send}(t,p,m) :: \mathcal{X}} \qquad \frac{\mathsf{fv}(e) \cap \mathcal{X} = \varnothing}{\mathcal{X} \vdash x \coloneqq e :: \mathcal{X} \setminus x} \qquad \frac{\mathsf{fv}(e) \cap \mathcal{X} \neq \varnothing}{\mathcal{X} \vdash x \coloneqq e :: \mathcal{X} \cup \{x\}}$$

$$\frac{\mathsf{fv}(e) \cap \mathcal{X} = \varnothing}{\mathcal{X} \vdash x \coloneqq \mathsf{recv}(t,e) :: \mathcal{X} \setminus x} \qquad \frac{\mathsf{fv}(e) \cap \mathcal{X} = \varnothing \qquad \mathcal{X} \vdash A :: \mathcal{X}_1 \qquad \mathcal{X} \vdash B :: \mathcal{X}_2}{\mathcal{X} \vdash \mathtt{if}\ e\ \{\ A\ \}\ \mathtt{else}\ \{\ B\ \} :: \mathcal{X}_1 \cup \mathcal{X}_2}$$

$$\frac{\{x\} \vdash A :: \mathcal{X}}{\vdash A :: \mathsf{cf}(x)}$$

Figure 4.7. Selected rules establishing control flow independence.

Note that this is only sound if there are no control-flow dependencies on the loop iteration variable, p. The judgment $\vdash A :: \mathsf{cf}(p)$ ensures that this is the case. The rules for determining $\vdash A :: \mathsf{cf}(p)$ are given in Figure 4.7, and implement a simple taint analysis.

The combination of R-SYM-RMV and R-FOR-ATOMIC thus produces concurrent programs where message passing has been replaced with coarse-grained atomic actions.

Example 4.4. We now examine EX4.

$$\bigsqcap_{p:ps} \left[\begin{array}{l} \mathtt{for\ (q : qs)\ \{} \\ \quad \mathsf{send}(\mathsf{q}, (p, \mathtt{ping})); \\ \quad _ \coloneqq \mathsf{recv}(q) \\ \mathtt{\}} \end{array}\right]_p \parallel \bigsqcap_{q:qs} \left[\begin{array}{l} \mathtt{for\ (p : ps)\ \{} \\ \quad (id, v) \coloneqq \mathsf{recv}(ps); \\ \quad \mathsf{send}(id, \mathtt{Ack}) \\ \mathtt{\}} \end{array}\right]_q$$

First we apply R-SYM-RMV, whose precondition obliges us to rewrite the following to `skip`:

$$\left[\begin{array}{l} \mathtt{for\ (q : } qs\mathtt{)\ \{} \\ \quad \mathsf{send}(\mathsf{q}, (p^u, \mathtt{ping})); \\ \quad _ \coloneqq \mathsf{recv}(\mathsf{q}) \\ \mathtt{\}} \end{array}\right]_{p^u} \parallel \bigsqcap_{q:qs} \left[\begin{array}{l} \mathtt{for\ (p : \{} p^u \mathtt{\})\ \{} \\ \quad (id, v) \coloneqq \mathsf{recv}(ps); \\ \quad \mathsf{send}(\mathsf{p}, \mathtt{Ack}) \\ \mathtt{\}} \end{array}\right]_q$$

Applying R-LOOP-UPD and R-VAR-UNFOLD transforms the obligation to

$$\left[\begin{array}{l}\texttt{assume}(x = q^u);\\ \texttt{send}(x, (p^u, \texttt{ping}));\\ _ \coloneqq \texttt{recv}(x)\end{array}\right]_{p^u} \parallel \left[\begin{array}{l}\texttt{for (p} : \{p^u\}\texttt{) \{}\\ \quad (id, v) \coloneqq \texttt{recv}(\{p^u\});\\ \quad \texttt{send}(id, \texttt{Ack})\\ \texttt{\}}\end{array}\right]_{q^u}$$

The obligation opened by R-SYM-RMV is closed by R-FOR-ATOMIC and R-SYM-UNFOLD, yielding

$$\left[\begin{array}{l}\texttt{assume}(x = q^u);\\ \texttt{send}(x, (p^u, \texttt{ping}));\\ _ \coloneqq \texttt{recv(x)}\end{array}\right]_{p^u} \parallel \left[\begin{array}{l}(id, v) \coloneqq \texttt{recv}(p^u);\\ \texttt{send}(id, \texttt{Ack})\end{array}\right]_{q^u}$$

which is rewritten to `skip`. The prefix computed by R-FOR-ATOMIC is thus the atomic action

$$\Delta_u(p^u, q^u) \triangleq \langle [v \coloneqq \texttt{ping}]_{q^u}; [_ \coloneqq \texttt{Ack}]_{p^u} \rangle.$$

Propagating Δ_u back through the applications of R-LOOP-UPD and R-SYM-RMV finally yields the rewritten program

$$\bigsqcap_{p:ps} \texttt{for (}q : qs\texttt{) \{}\langle [v \coloneqq \texttt{ping}]_q; [_ \coloneqq \texttt{Ack}]_p \rangle\texttt{\}}.$$

4.3.4 Rewrite Rule Correctness

Interpretations. We interpret prefixes and contexts such that $(\sigma, \mu) \in [\![\Delta, \Gamma]\!]$ when σ and μ are a store and buffer consistent with the states reachable by executing Δ and the assumptions in Γ.

The soundness theorem says that each rewriting step preserves the reachability of halted states. Let $C \approx C'$ when $C = C' = \texttt{crash}$ or C and C' have equal stores.

Theorem 4.1 (Soundness)**.** *Let A contain only almost-symmetric races, and assume*

R-SYM-RMV

u fresh

$\Delta_0 \triangleq \mathtt{assume}(I_C)$

$\Delta \vDash I_C$ and $\Delta_0; \Delta^u \vDash I_C$

$\Delta' \triangleq \Delta \,;\, \bigsqcap_{p:P} [\Delta^u[p/u]]_p$

$\Gamma \cup \mathtt{split}(P, u, I_C), \Delta_0, [A]_u \parallel \bigsqcap_{q:Q} [B[\{u\}/P]]_q \rightsquigarrow \Gamma, (\Delta_0; \Delta^u), \mathtt{skip}$

$$\Gamma, \Delta, \bigsqcap_{p:P} \left[A\right]_p \parallel \bigsqcap_{q:Q} \left[B\right]_q \rightsquigarrow \Gamma, \Delta', \mathtt{skip}$$

R-FOR-ATOMIC

x fresh

$\Delta_0 \triangleq \mathtt{havoc}(\mathtt{vars}(q, \Delta)); \mathtt{assume}(I_C)$

$\Gamma \vdash \mathtt{split}(P, u, I_C)$

$\Gamma_0 \triangleq \Gamma \setminus \{\mathtt{split}(P, u, I_C)\}$

$\vdash A :: \mathtt{cf}(p)$

$\Gamma_0 \cup \{\mathtt{unfold}(u, x, \{u\})\}, \Delta_0, [A[x/p]]_q \parallel [B]_u \rightsquigarrow \Gamma_0, (\Delta_0; \Delta^u), \mathtt{skip}$

$$\Gamma, \Delta, \left[\mathtt{for}\ (p : \{u\})_{\{I_C, I_S\}}\ \{A\}\right]_q \parallel [B]_u \rightsquigarrow \Gamma, (\Delta; \langle \Delta^u \rhd I_S\ \{u\}\rangle), \mathtt{skip}$$

Figure 4.8. Rewrite Rules (case splitting on a single process)

1. $\varnothing, \mathtt{skip}, A \rightsquigarrow \Gamma_B, \Delta_B, B$

2. $C_A \in [\![\mathtt{skip}, \varnothing, A]\!]$ *and* $C_A \to_{\mathrm{I_s}} C_H$ *where* $C_H \in \mathsf{Halted}(C_A)$ *(is halted).*

Then there exists $C_B \in [\![\Delta_B, \Gamma_B, B]\!]$ *and* C'_H *such that* $C_B \to_{\mathrm{I_s}} C'_H$ *and* $C_H \approx C'_H$.

Proof Sketch. We provide the intuition for why the rewrite rules are sound, but do not present a detailed proof. Many of the rules reduce to a special case of the rules for computing canonical sequentializations, so we direct our attention to the new features, namely the rules R-SYM-RMV and R-FOR-ATOMIC. The proof is by induction on the rewriting derivation.

- Case R-SYM-RMV:

  To prove R-SYM-RMV, we strengthen the inductive hypothesis to include an environment that can execute behaviors on behalf of processes in P - the constraint

is that these behaviors must preserve I_C. The conclusion follows by induction on the size of P. The rewritten Δ is collected for each $p \in P$ and composed in parallel with the input environment of the inductive hypothesis. Thus, the output rewritten program is the parallel composition over Δ.

- Case R-FOR-ATOMIC:

  As R-SYM-RMV case splits on a processes u, the key property to show is that the communication invariant I_C summarizes the behavior of statements that interact with processes $p \in P$ when $p \neq u$. In this case, we consider the entire for loop rather than just the iteration corresponding to u. From the rewriting assumption, we can prove that, if the environment maintains I_C, then the halting states of $\left[\texttt{for } (p : P)_{\{I_C\}} \texttt{ \{ } A \texttt{ \}}\right]_q \parallel [B]_u$ are contained in $\left[\texttt{for } (p : P \setminus \{u\})_{\{I_C\}} \texttt{ \{ } A \texttt{ \}}\right]_q \parallel \langle \Delta^u \rangle$. This is justified by the fact that the rewriting hypothesis guarantees that each iteration of the loop terminates, and that the environment can only interrupt (*i.e.*, havoc the state) between iterations.

□

4.3.5 Extensions

We now discuss three extensions to the rules presented above that are implemented in CHAI.

Grouping Statements. Sometimes there is a choice of which `for`-loop to merge a *local* action with. For instance, p's conditional statement in Figure 4.1 can be merged either with c's first or second `for`-loop. To make verification more predictable, our system contains a statement $\texttt{group}(A)$ that ensures that all statements in A are rewritten together.

Nonblocking receives. Our implementation of nonblocking receives follows the description in EX5 of Section 4.1. The rule for $t \coloneqq \texttt{newType}$ binds t to a fresh identifier. The rule for `recvT0` rewrites `recvT0` as if it were a blocking receive and then wraps the resulting prefix into a conditional, non-deterministically either assigning the received

value or `Nothing`.

General While Loops. Our system also handles while loops and iteration over sets of indices. We consider while loops that implement a reactive pattern, where processes serve requests in a non-terminating loop. In this setting, our rules rewrite the finite interaction between a set of clients and the reactive servers. For while loops, we can relax our condition on races in the following way: if a receive occurs in a while loop, we can allow races between sends in the *same* symmetric set (for an example, consider the acceptor process in Section 4.5.2). Intuitively, this is possible since while loops in our

R-WHILE-ATOMIC

$$\frac{\begin{array}{l} x \text{ fresh} \\ \Delta_0 \triangleq \texttt{havoc}(\texttt{vars}(q, \Delta)); \texttt{assume}(I_C) \\ \Gamma \vdash \texttt{split}(P, u, I_C) \\ \Gamma_0 \triangleq \Gamma \setminus \{\texttt{split}(P, u, I_C)\} \\ \Gamma_0 \cup \{\texttt{unfold}(u, x, \{u\})\}, \Delta_0, [A[x/p]]_q \parallel [B]_u \rightsquigarrow \Gamma_0, (\Delta_0; \Delta^u), \texttt{skip} \end{array}}{\Gamma, \Delta, \left[\texttt{while true}_{I_C, I_S}\ \{\ A\ \}\right]_q \parallel [B]_u \rightsquigarrow \Gamma, (\Delta; \langle \Delta^u \triangleright I_S\ \{u\}\rangle), \left[\texttt{while true}_{I_C, I_S}\ \{\ A\ \}\right]_q}$$

Figure 4.9. Reactive while loop.

setting do not terminate: we again employ rely-guarantee style reasoning to ensure that, after one client interacts with the server, the same interaction can be repeated by another client. Finally, we note that our system handles *purely local* computations. In particular, this includes *general* while loops that perform only local computations, *i.e.*, do not send or receive messages.

4.4 Generating Verification Conditions

Given a program A and a property φ, we can verify that $A \vDash_H \varphi$ by first rewriting A to produce the synchronized prefix Δ, and then verifying that $\Delta; \texttt{assert}(\varphi) \nrightarrow_{\mathrm{Is}} \texttt{crash}$. Let φ be a formula over a subset of the processes of A and assume that A can be rewritten to Δ, *i.e.*, $-, \texttt{skip}, A \rightsquigarrow -, \Delta, B$. Then, by Theorem 4.1, the original program is safe (*i.e.*,

$$
\begin{array}{lcl}
\mathsf{wp}(x_p \coloneqq e, Q) & \triangleq & Q[x[p \leftarrow e]/x] \\
\mathsf{wp}(\texttt{assume}(P), Q) & \triangleq & P \Rightarrow Q \\
\mathsf{wp}(\texttt{assert}(P), Q) & \triangleq & P \wedge Q \\
\mathsf{wp}(\langle a \triangleright P \rangle, Q) & \triangleq & P \wedge \mathsf{wp}(a, Q) \\
\mathsf{wp}(s_1; s_2, Q) & \triangleq & \mathsf{wp}(s_1, \mathsf{wp}(s_2, Q)) \\
\mathsf{wp}(s_1 \,\square\, s_2, Q) & \triangleq & \mathsf{wp}(s_1, Q) \wedge \mathsf{wp}(s_2, Q) \\
\mathsf{wp}(\texttt{for } (x : X)_{\{\lambda d.I\}} \{\ A\ \}, Q) & \triangleq & I[\varnothing/d] \\
& \wedge & (\forall\, y.\ I[X'/d] \wedge x \in X' \setminus d \Rightarrow \mathsf{wp}(A, I[(d \cup \{x\})/d])) \\
& \wedge & (\forall\, y.\ (I[X/d] \Rightarrow Q) \\
& & \textit{where } y \triangleq \mathsf{V}(A) \\
\mathsf{wp}(\textstyle\bigsqcap_{p:P} A, Q) & \triangleq & ((\forall\, p \in P.\ \mathsf{pc}_P[p] = l_0) \Rightarrow I) \\
& \wedge & \forall\, p_0 \in P.\ \forall\, y. \\
& & \bigwedge_{a_l \in A} (I \wedge CF(l) \Rightarrow \mathsf{wp}(a[p_0/p], I[\mathsf{pc}'_P/\mathsf{pc}_P])) \\
& \wedge & \forall\, y.\ ((\forall\, p \in P.\ \mathsf{pc}_P[p] = l_{\mathsf{exit}}) \wedge I \Rightarrow Q) \\
& & \textit{where } I \quad \triangleq\ \forall\, p \in P. \\
& & \qquad \bigwedge_{a_l \in A} \mathsf{pc}_P[p] = l \Rightarrow \bigvee_{\langle a \triangleright B_{l'} \rangle \in \mathsf{pred}(A,l)} B_{l'} \\
& & \qquad CF(l) \triangleq\ \mathsf{pc}_P[p] = l \\
& & \qquad\quad \wedge \bigvee_{l' \in \mathsf{out}(A,l)} \mathsf{pc}'_P = \mathsf{pc}_P[p \leftarrow l'] \\
& & \qquad y \quad \triangleq\ \{\mathsf{pc}_P, \mathsf{pc}'_P\} \cup \mathsf{V}(A)
\end{array}
$$

Figure 4.10. Weakest precondition computation. $\mathsf{V}(A)$ denotes the variables that may be assigned in A.

$A \vDash_H \varphi$) if the *rewritten* program is safe (*i.e.*, $\Delta; \texttt{assert}(\varphi) \nrightarrow_{\mathrm{I_S}} \mathsf{crash}$) and φ does not mention state in B.

Verification conditions. To check if $\Delta; \texttt{assert}(\varphi) \nrightarrow_{\mathrm{I_S}} \mathsf{crash}$, we compute its verification condition (VC) by computing its *weakest precondition* and checking to see if it is implied by the initial state. The procedure $\mathsf{wp}(s, Q)$ defines the weakest precondition computation: if s is a program and Q is a formula, then $\mathsf{wp}(s, Q)$ is a formula describing the initial states that guarantees that s only terminates in states satisfying Q. The definition of $\mathsf{wp}(s, Q)$ is mostly standard.

Assignment. As $\mathrm{I_S}$ programs contain unbounded sets of processes, variables are modeled as maps from PID to values. Thus, $\mathsf{wp}(p.x \coloneqq e, Q)$ substitutes x with the *map*

$x[p \leftarrow e]$ in Q.

Loops. Determining the weakest precondition of a loop requires a user-supplied loop invariant. The $\texttt{for}\ (x : X)_{\{I\}}\ \{\ldots$ case uses a user-supplied invariant parameterized by d, an auxiliary variable that contains the elements of X that have already been processed by the loop (*i.e.*, are "done").

Parallel Composition. We define the case for parallel processes $\bigsqcap_{p:P} A$ as an analog of the standard case for loops. First, let the function $\mathsf{actions}(A)$ collect the labeled atomic actions of A. Next, we define an auxiliary variable pc_P (the program counter) that maps each process p in P to its current program location (*i.e.*, the current atomic block). The final component is an inductive invariant. The invariant is constructed from the annotations on the actions in $\mathsf{actions}(A)$: if $\langle a_l \triangleright B_l \rangle_l \in \mathsf{actions}(A)$, then the invariant includes the conjunct $\mathsf{pc}_p[P] = l \Rightarrow B$ where B is the *disjunction* of all annotations for control locations that can transition to l, *i.e.*, all B_m such that $\langle a_m \triangleright B_m \rangle_m \in \mathsf{actions}(A)$ and l is one of the immediate successors of m in the control flow of A.

We define $\mathsf{wp}(\bigsqcap_{p:P} A, Q)$ as three main assertions, requiring that (1) the invariant I holds in the initial state; (2) each transition preserves the invariant (*i.e.*, the invariant is inductive); and (3) when every process in P has terminated (*i.e.*, has its program counter set to l_{exit}), I implies Q.

Example 4.5. Consider the program A.

$$A \triangleq \bigsqcap_{p:ps} \left[\begin{array}{l} \texttt{for}\ (q : qs)_{\{\mathsf{true}, S\}}\ \{ \\ \quad \texttt{send}(q, (q, \texttt{ping})) \\ \}; \\ \texttt{for}\ (q : qs)_{\{\mathsf{true},\mathsf{true}\}}\ \{ \\ \quad x \coloneqq \texttt{recv}(qs); \\ \quad \texttt{assert}(x = \texttt{ping}) \\ \} \end{array}\right]_p \parallel \bigsqcap_{q:qs} \left[\begin{array}{l} \texttt{for}\ (p : ps)_{\{\mathsf{true},\mathsf{true}\}}\ \{ \\ \quad (id, y) \coloneqq \texttt{recv}(ps); \\ \quad \texttt{send}(id, y); \\ \} \end{array}\right]_q$$

$$S \triangleq \lambda d.\ q \in d \Rightarrow q.y = \texttt{ping}$$

We want to prove that A is safe, *i.e.*, that $A \not\twoheadrightarrow_{\mathrm{I_s}} \texttt{crash}$. To do so, we rewrite A to A_{SYNC}, and then check the validity of $\mathsf{wp}(A_{\mathrm{SYNC}}; \texttt{assert}(\mathsf{true}), \mathsf{true})$.

$$A_{\mathrm{SYNC}} \triangleq \prod_{q:qs} \left(\begin{array}{l} \texttt{for } (p : ps) \texttt{ \{} \\ \quad \langle [(q.id, q.y) \coloneqq (q, \texttt{ping})]_q \rhd S\ \{q\} \rangle_{l_0}; \\ \quad \left\langle \left[\begin{array}{l} p.x \coloneqq q.y; \\ \texttt{assert}(p.x = \texttt{ping}) \end{array} \right]_p \rhd \mathsf{true} \right\rangle_{l_1} \\ \texttt{\} }_{l_{\mathsf{exit}}} \end{array} \right)$$

The invariant for A_{SYNC} is derived from its two atomic actions. Because l_1 is reachable from l_0, the annotation from l_0 is used as the assertion when some q is at l_1, reflected in the conjunct $\mathsf{pc}_{qs}[q] = l_1 \Rightarrow S$. Since l_0 and l_{exit} are reachable from l_1, whose annotation is true, the assertion at l_0 is simplified to true. The invariant I is thus defined $I \triangleq \forall\, q \in qs.\ \mathsf{pc}_{qs}[q] = l_1 \Rightarrow y[q] = \texttt{ping}$. The weakest precondition comprises three assertions: (1) the initial state satisfies I, (2) the two inductive steps corresponding to whether q runs l_0 or l_1, and (3) the final state satisfies true (which is trivial). Below, T_0 and T_1 check that the actions at l_0 and l_1 each preserve I. CF_{l_0} and CF_{l_1} encode the control flow by constraining the "current" and "next" program counter. The verification condition is defined below. We omit the assignment to id, and variables are implicitly universally

quantified.

$$\begin{aligned}
\mathsf{wp}(A_{\text{SYNC}}, \mathsf{true}) = {} & \left(\forall\, q \in qs.\ \mathsf{pc}_{qs}[q] = l_0 \Rightarrow I\right) \wedge \\
& (\forall\, q \in qs.\ \forall\, p \in ps.\ (T_0 \wedge T_1)) \ \wedge \\
& \left(\forall\, q \in qs.\ \ \wedge I \wedge \mathsf{pc}_{qs}[q] = l_{\mathsf{exit}} \Rightarrow \mathsf{true}\right)
\end{aligned}$$

where

$$\begin{aligned}
T_0 &= (I \wedge CF_{l_0} \wedge y' = y[q \leftarrow \mathtt{ping}] \Rightarrow I[\mathsf{pc}'_{qs}/\mathsf{pc}_{qs}][y'/y]) \\
T_1 &= (I \wedge CF_{l_1} \wedge x' = x[p \leftarrow y[q]] \Rightarrow (x'[p] = \mathtt{ping}) \wedge I[\mathsf{pc}'_{qs}/\mathsf{pc}_{qs}][x'/x]) \\
CF_{l_0} &= \mathsf{pc}_{qs}[q] = l_0 \wedge \mathsf{pc}'_{qs} = \mathsf{pc}_{qs}[q \leftarrow l_1] \\
CF_{l_1} &= \mathsf{pc}_{qs}[q] = l_1 \wedge \left(\mathsf{pc}'_{qs} = \mathsf{pc}_{qs}[q \leftarrow l_0] \vee \mathsf{pc}'_{qs} = \mathsf{pc}_{qs}[q \leftarrow l_{\mathsf{exit}}]\right)
\end{aligned}$$

As this formula is valid, $A_{\text{SYNC}} \not\twoheadrightarrow_{\text{I}_\text{S}} \mathtt{crash}$ and hence $A \not\twoheadrightarrow_{\text{I}_\text{S}} \mathtt{crash}$.

Proposition 4.1 states the correctness of $\mathsf{wp}(A, \mathsf{true})$ and says that if A executes in a state satisfying $\mathsf{wp}(A, \mathsf{true})$, then A does not reach the crash configuration. The definition of $\mathsf{wp}(s, Q)$ is standard and we do not prove it sound here.

Proposition 4.1 (VC generation)**.** *If* $\mathsf{Valid}(\mathsf{wp}(\Delta, \mathsf{true}))$ *then* $\Delta \not\twoheadrightarrow_{\text{I}_\text{S}} \mathtt{crash}$.

In general, even if the formulas asserted and assumed in a program are decidable, the naive model of message buffers as arrays requires nested array reads, yielding undecidable verification conditions. However, when $-, \mathtt{skip}, A \rightsquigarrow -, \Delta, B$, Δ does not contain message buffers. Hence, if the formulas asserted and assumed in A are in the array property fragment [8], then the verification resulting from $\mathsf{wp}(\Delta, \mathsf{true})$ is as well.

Proposition 4.2 (VC Decidability)**.** *Assume* $-, \mathtt{skip}, P \rightsquigarrow -, \Delta, Q$, $\psi = \mathsf{wp}(\Delta, \mathsf{true})$*, and the local states of P do not contain nested reads. Then, checking the validity of* ψ *is decidable.*

4.5 Case Studies: Synchronous Invariants for Distributed Protocols

In this section, we describe three case studies where we use CHAI to verify functional correctness properties of complex asynchronous programs.

Implementation. We have implemented our approach of pretending synchrony in a tool, CHAI. CHAI takes as input a I_S program A, a property φ and (1) computes a *synchronization* Δ, and, should one exist, (2) *verifies* that Δ satisfies φ, thereby proving the safety of A (Section 4.4), *i.e.*, that $A \vDash_H \varphi$. We implemented the rewriting step by interpreting our rewriting rules (Section 4.3) as a PROLOG predicate. The verification step is implemented by computing the weakest precondition of the rewritten program (Section 4.4) and then using Z3 to prove the validity of the verification condition.

Experiments. We used CHAI to verify the following case studies implemented in I_S:

- *Two-Phase Commit* (as discussed in 4.1). We verify that if the transaction is committed then each participant has accepted the coordinator's proposal.
- *Key-Value Store*, a database implementing a write-once key/value store. We verify that if a client sets the value of a key, a subsequent read of that key returns the same value.
- *Raft Leader Election* (from [55]). We verify that no two candidates both think they are leader in the same term.
- *Single-Decree Paxos* (from [41]). We verify agreement, that is, if two proposers decide on a value, they must have decided on the same value.

Results. The results of our experiments are summarized in Table 4.1. The table indicates (1) the number of lines of I_S required for each benchmark; (2) the number of lines of annotations in the form of loop invariants; (3) the amount of time it took for

Table 4.1. CHAI benchmark results. **#LOC** is the number of lines of code of I_S, **#Annot** is the number of lines of annotations (loop invariants), **Rewrite (s)** is the time CHAI took to compute the benchmark's synchronization, and **VC (s)** is the time it took to prove the validity of the verification condition output by CHAI. All experiments were run on a 2.3 GHz Intel Core i7 CPU.

Benchmark	#LOC	#Annot	Rewrite (s)	VC (s)
Key-Value Store	36	4	0.02	0.02
Two-Phase Commit	33	4	0.09	0.02
Raft Leader Election	54	12	0.05	0.03
Single-Decree Paxos	87	44	1.66	0.39
Total	210	64	1.82	0.46

CHAI to compute the synchronization of the benchmark, and (4) the amount of time it took Z3 to prove the verification condition valid.

Comparison. Our benchmarks required about 0.25x proof overhead *i.e.*, one line of annotation per four lines of code, with Paxos requiring about one line of annotation per two lines of source. This is in contrast to recent systems like Ironfleet [31] (9x overhead), Verdi [82] (90x overhead on Raft), and DISEL [80] (3x overhead on two-phase commit). A direct comparison to these tools is difficult, as, for one, they implement significantly more featureful systems. Nevertheless, the reduction in annotations is significant, and we conjecture that it is largely due to the automatic reduction computation, *i.e.*, the rewriting step. The rewritten program is synchronous composed of coarse-grained atomic blocks which further reduce the number of program interleavings, simplifying the invariants needed to verify correctness.

4.5.1 Raft Leader Election

Figure 4.11 shows an implementation of the leader election phase of the Raft consensus algorithm [55]. We have removed components like logs and timeouts that are not relevant to leader election, and have omitted the initialization phase for brevity.

Asynchronous Program. The protocol contains two sets of symmetric processes:

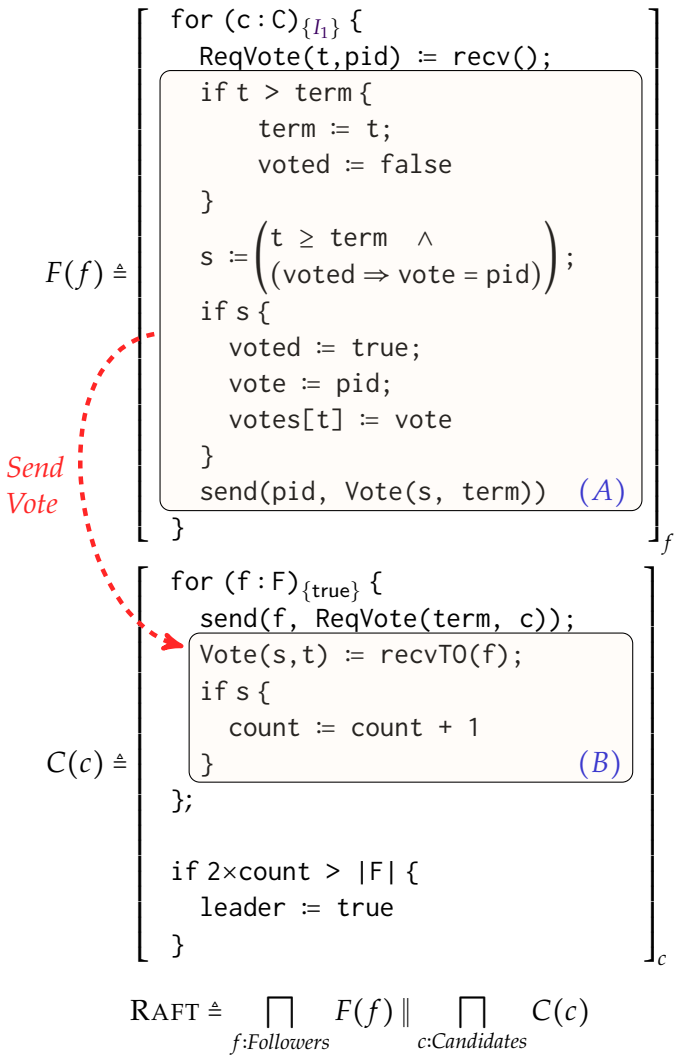


Figure 4.11. Simplified Raft leader election algorithm (asynchronous version)

the set of *candidate* processes C and the set of *follower* processes F. Each follower is initialized with `voted := false`, and each candidate initialized with `leader := false` and `counter := 0`. The algorithm starts with candidates sending a message of type `ReqVote` containing the candidate's current term and its process id to every follower. If the candidate term is larger than the follower's current term, the follower updates its term and sets its `voted` flag to false. The follower then determines whether or not to accept the vote by setting a flag `s`. `s` is set to true if the candidate term is at least as big as the follower's term and either the follower did not vote yet, or it voted for the candidate. The follower then updates its bookkeeping and responds back to the candidate with a message of type `Vote` with a flag set to `true` if the vote is granted, and the current term of the follower, otherwise. After this message exchange, the candidate then claims to be the leader if it receives messages with granted votes from a majority of the followers. In addition to the original protocol, we have added a ghost variable `votes` to keep track of the history of votes that a follower grants.

Property: Election Safety. We want to prove the key *election safety* property [55]: for any given term, *at most one leader is elected* by the followers, *i.e.*, the assertion:

$$\forall c_1, c_2 \in C.\ c_1.\texttt{leader} \land c_2.\texttt{leader} \land c_1.\texttt{term} = c_2.\texttt{term} \Rightarrow c_1 = c_2$$

Upon closely inspecting the follower's code, we can see that each follower votes for *at most* one candidate for a given term due to the condition `voted` $\Rightarrow$ `vote` = `pid` in variable `s`. Unfortunately, this reasoning is not inductive and hence cannot be used to prove election safety. To derive an inductive invariant, we must relate the value of the counter in candidate c with the *number of followers* that have voted for c at some point during the execution. Thus, an asynchronous invariant for the follower's loop must relate the follower's local state to that of the candidates, requiring an extensive case split. However, the invariant becomes easy, once we pretend synchrony.

Invariants. The property is proven by the following invariant I_1. I_1 refers to the

cardinality of two sets: $k[\mathtt{c}]$, the number of followers that *may vote* for $\mathtt{c}$ in the future, and $l[\mathtt{c}]$, the number of followers that *have voted* for $\mathtt{c}$ in the past. Then, the invariant states that the number of followers who voted for a candidate is as least as big as its counter, and the sum of the cardinalities is bounded by the total number of followers.

$$
\begin{array}{rcll}
I_1 & \triangleq & \forall \mathtt{c} \in C.\ |F| \geq k[\mathtt{c}] + l[\mathtt{c}] \wedge \mathtt{c.count} \leq l[c] & \\
k[\mathtt{c}] & \triangleq & \#\{f \mid \mathtt{f.term} < \mathtt{c.term}\} & \text{(\# may vote for c in future)} \\
l[\mathtt{c}] & \triangleq & \#\{f \mid \mathtt{f.term} \geq \mathtt{c.term} \wedge \mathtt{f.votes}[\mathtt{c.term}] = \mathtt{c}\} & \text{(\# have voted for c in past)}
\end{array}
$$

The synchronized version computed by our method contains a single atomic action built from the bodies of the two loops. In particular, synchronization schedules the block (A) before (B), clarifying the relationship between $\mathtt{f.s}$ and $\mathtt{c.count}$: $\mathtt{c.count}$ is incremented only when $\mathtt{f.s}$ is set to true.

Verification. The above invariants let us prove election safety as follows. Suppose that there are two different leaders $\mathtt{c}_1$ and $\mathtt{c}_2$ within the same term. Then, both $\mathtt{c}_1\mathtt{.count}$ and $\mathtt{c}_2\mathtt{.count}$ should be greater than $|F|/2$. But, due to I_1, it must be that $l[\mathtt{c}_1] > |F|/2$ and $l[\mathtt{c}_2] > |F|/2$. By cardinality reasoning, there should exist a follower $\mathtt{f}^*$ in both $l[\mathtt{c}_1]$ and $l[\mathtt{c}_2]$, yielding the contradiction:

$$\mathtt{c}_1 = \mathtt{f}^*\mathtt{.votes}[\mathtt{c}_1\mathtt{.term}] = \mathtt{f}^*\mathtt{.votes}[\mathtt{c}_2\mathtt{.term}] = \mathtt{c}_2$$

While the proof involves cardinality invariants, the key bit is that once we have eliminated asynchrony — specifically, message buffers — via pretend synchrony, we can completely automate the discovery of such invariants using known techniques like #Π [76].

4.5.2 Single-decree Paxos

Figure 4.12 contains the single-decree version of the Paxos consensus protocol [41], where the proposers do not update their proposal number after a failed attempt and try again. For brevity, we omitted the initialization of the local variables. It also does

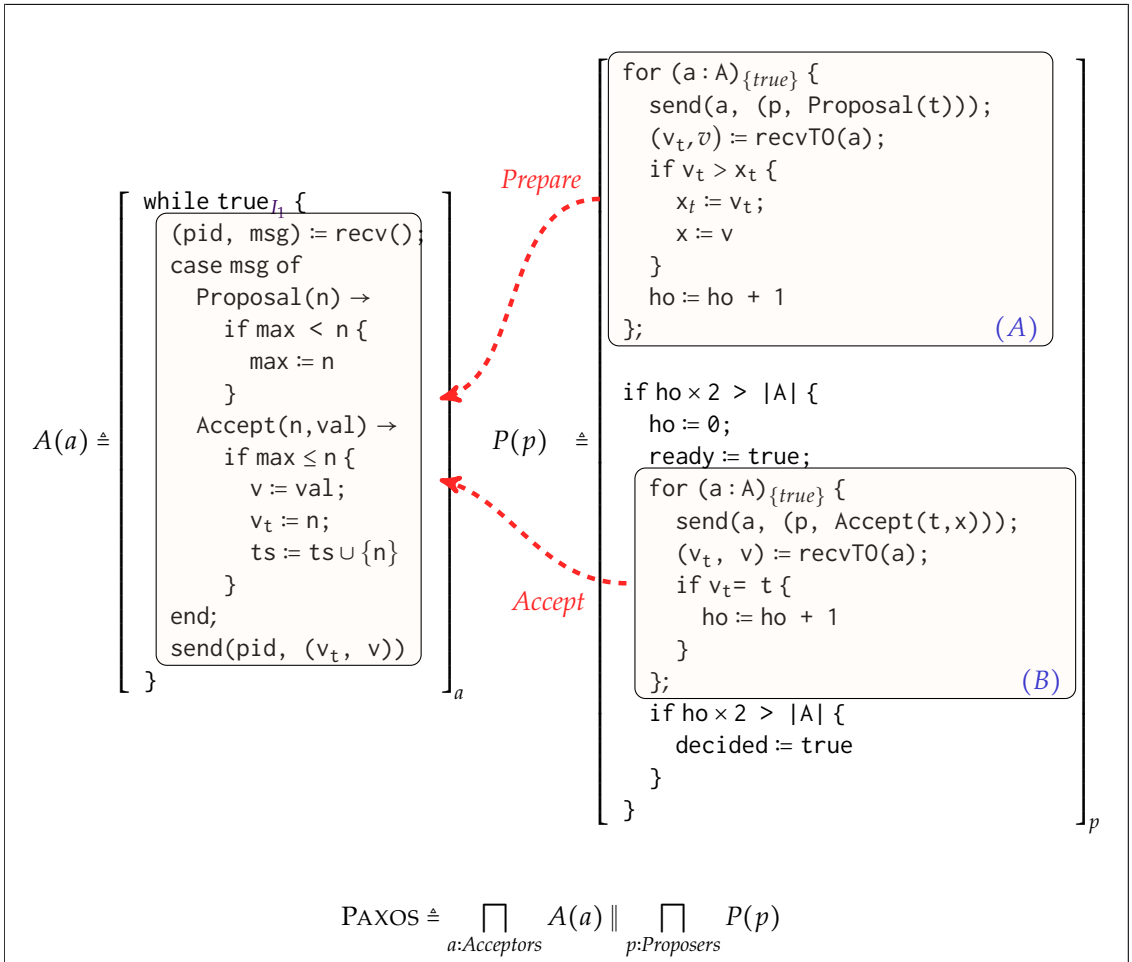


Figure 4.12. Single-decree Paxos (asynchronous version)

not contain the learner processes that do not participate in the value selection.

Asynchronous Program. The protocol consists of a set of proposers P and a set of acceptors A. Some of the local variables are initialized as follows: `max` $\coloneqq -\infty$ contains the smallest proposal number to accept in the future;

$\mathtt{v_t} \coloneqq -\infty$ contains the highest proposal number accepted; `v` $\coloneqq \perp$ that contains the accepted value; `ts` $\coloneqq$ `{}` contains the numbers of all accepted proposals; `t` $\coloneqq$ `*` contains the arbitrary and unique proposal number; `x` $\coloneqq$ `*` is the value to be proposed; and $\mathtt{x_t} \coloneqq -\infty$ keeps track of the highest numbered proposal accepted by the acceptors.

The protocol starts with proposers sending a `Propose` message that contains their proposal number. Acceptors update their `max` to *reject* every proposal with smaller number, and respond back with their current values of $\mathtt{v_t}$ and `v`. After receiving a response, the proposers update their value with the value of the *highest* numbered proposal.

The first phase above deviates from the original Paxos protocol where acceptors respond *only* if the received proposal number is bigger than they have ever seen. However, this can be considered as a performance optimization. In addition, since we model message drops with the `recvTO` primitive, we actually support the same behavior.

If a proposer hears back a response from a majority of acceptors, it starts its second phase by sending an `Accept` message with its proposal number and value to every acceptor. Upon receiving this message, acceptors update their accepted value and proposal number if necessary, and respond back with the same message as before. If the majority of received replies contain the proposer's own number, the proposer declares that number to be the decided value.

Property: Agreement. We want to prove that the *agreement* property is satisfied in this protocol, *i.e.*, that every proposer agrees on the decided value: $\forall p_1, p_2 \in P.\ p_1.\mathtt{decided} \wedge p_2.\mathtt{decided} \Rightarrow p_1.\mathtt{x} = p_2.\mathtt{x}$.

Invariants. The following invariant can be used to satisfy the agreement property: Here, k counts the number of acceptors that *have accepted* `p`'s proposal before, l

counts the ones that *may accept* it in the future, and m counts the ones that *will never* accept it. I_1^m, the main lemma, states that if p is ready and not outdated (*i.e.*, a majority of acceptors have either accepted its proposal or might accept it later), then, whenever an acceptor accepts a proposal that is at least as high as `p.t`, it must in fact accept `p.x`.

$$
\begin{array}{rcll}
I_1 & \triangleq & I_1^4 \wedge I_1^m \wedge \ldots & \\
I_1^4 & \triangleq & \forall\,.\ p.\ k[\mathtt{p}] + l[\mathtt{p}] + m[\mathtt{p}] = |A| & \\
I_1^m & \triangleq & \forall \mathtt{a}, \mathtt{p}.\ \mathtt{p.ready} \wedge k[\mathtt{p}] + l[\mathtt{p}] \geq \frac{|A|}{2} \wedge \mathtt{a.v_t} \geq \mathtt{p.t} \Rightarrow \mathtt{a.v} = \mathtt{p.x} & \\
 & & & \\
k[\mathtt{p}] & \triangleq & \#\{\mathtt{a} \in A \mid \mathtt{p.t} \in \mathtt{a.ts}\} & \text{(\# have accepted)} \\
l[\mathtt{p}] & \triangleq & \#\{\mathtt{a} \in A \mid \mathtt{p.t} \notin \mathtt{a.ts} \wedge \mathtt{a.max} \leq \mathtt{p.t}\} & \text{(\# may accept)} \\
m[\mathtt{p}] & \triangleq & \#\{\mathtt{a} \in A \mid \mathtt{p.t} \notin \mathtt{a.ts} \wedge \mathtt{a.max} > \mathtt{p.t}\} & \text{(\# will never accept)}
\end{array}
$$

Synchronized Program. The synchronized program is the parallel composition of, for each proposer, a *single* `for`-loop that, for each acceptor a, executes two atomic actions: one derived from rewriting the block (A) with the body of a's `while` loop, and one similarly derived from block (B).

As with Raft, once the program is synchronized, the sets and cardinalities we used could be found automatically. In addition, note how simple and intuitive the invariants above are with respect to invariants that one might need to formally verify the original asynchronous protocol.

4.6 Summary

In this chapter we have described *pretend synchrony*, a new approach to verifying asynchronous distributed systems by soundly treating them as if they were synchronous when verifying their correctness. We have implemented pretend synchrony in a tool called CHAI and shown that by automatically computing an asynchronous program's *synchronization*, it is possible to prove challenging protocols such as the Raft leader election algorithm or single decree Paxos correct while reducing annotation burden by an order of magnitude, even compared to previous SMT based methods.

Acknowledgments

Chapter 4, in full, is adapted from the material as it appears in Klaus v. Gleissenthall, Alexander Bakst, Rami Gökhan Kıcı, and Ranjit Jhala. Pretend Synchrony. *In submission to the ACM SIGPLAN International Conference on Principles of Programming Languages, POPL 2018*. The dissertation author was the primary investigator and author of this paper.

Chapter 5

Conclusion

Distributed systems are large, complicated pieces of software, and maintaining an accurate mental model of their behavior is taxing if not impossible. In the article "How Amazon Web Services Uses Formal Methods," [53] the authors note that

> Engineers naturally focus on designing the "happy case" for a system, or the processing path in which no errors occur. This is understandable, as the happy case is by far the most common case. That code path must solve the customer's problem, perform well, make efficient use of resources, and scale with the business – all significant challenges in their own right. When the design for the happy case is done, the engineer then tries to think of "what could go wrong" based on personal experience and that of colleagues and reviewers. The engineer then adds mitigations for these scenarios, prioritized by intuition and perhaps statistics on the probability of occurrence. Almost always, the engineer stops well short of handling "extremely rare" combinations of events, as there are too many such scenarios to imagine.

Biased by motivated reasoning, it is hard for human engineers to exhaustively enumerate the problematic corner cases.

In this dissertation, we have proposed a method of simplifying proofs of correctness that appeal to the "happy case." To do so, we have made three distinct contributions to ease the task of message-passing and distributed system verification.

Symmetric non-determinism. Symmetric non-determinism is intended to capture a "goodness of design" property that allows developers to reason about their systems as if they were synchronous. The essence of the property is that well-designed systems are agnostic of the underlying asynchrony of the programming model. One way

to understand what "well-designed" means is that it is easy to match program locations across processes that are (implicitly) related by the implemented protocol while lacking any formal annotation. This structure becomes even more apparent in programming languages with expressive type systems: each step (and, hence, message) in the protocol is associated with a distinct *type*. Matching complementary parts of a message exchange, when faced with many choices, amounts to matching the types of the send and received messages. While not all systems are symmetrically non-deterministic, our work suggests that well-known algorithms *are* or can be refactored as such.

Canonical sequentialization. Symmetric non-determinism allows us to find proofs of some shallow properties, like deadlock freedom, completely automatically. Moreover, the algorithm we have presented for computing sequentializations is fast enough to incorporate into development tools. We conjecture that relating a distributed program to its sequentialization further aids the developer in understanding her code *during development*.

Pretend synchrony. Finally, we have applied symmetric non-determinism to reason about program that *do not* have sequentializations. Nevertheless, we argue that most verification approaches conflate reasoning about the particulars of the messaging subsystem and the data exchanged *by* the messages. By pretending synchrony, we completely eliminate asynchronous message buffers, which connects a particular implementation to an idealized, synchronized algorithm with coarse grained concurrency. Our case studies proving the correctness of well-known distributed systems suggests this amounts to a significant savings in proof effort.

5.1 Future Directions

Designing distributed systems is likely to remain a challenging problem. However, we believe the techniques described in this dissertation open the door to empowering developers with better programming environments. Necessarily, the automated techniques in this dissertation are incomplete, in the sense that there are correct programs

that can not be verified as such by the presented proof systems. We conjecture that some of these shortcomings may be addressed by somewhat straightforward extensions. For example, we do not discuss programs with dynamic process creation, but it is possible that these can be modeled as programs with a set of *static* (but unbounded) groups of processes that receive a distinguished "initialization" message to model creation.

Parameterized topologies. A class of problems with a murkier set of potential solutions are systems *lacking* symmetric processes in the sense used throughout the dissertation. Notable examples include protocols that overlay a *ring* structure on the participants. In this setting, the topology of the protocol, not merely the set of participants, is parameterized. Reasoning about these protocols might require reasoning about *ordsets* of processes [], for example in the recursive lookup procedure of Chord. A major limitation of the work presented in this dissertation is the minimal treatment of network anomalies.

Failures. While the work applies in a limited capacity to models in which messages can be dropped, we do not consider Byzantine failures. A fruitful research direction would be to use the ideas developed in this dissertation as a foundation for layering additional reasoning targeted at more interesting failure modes.

Effectiveness. We argued that the approach of synchronization as presented Chapter 4 decreases the proof burden. As with any verification framework, a useful question to ask is if this manifests when actual engineers (rather than motivated PhD students) are given the tool in question. However, if it is indeed the case that pretending synchrony eases reasoning about distributed systems, then it should be possible to leverage tools for reasoning about shared memory systems to further automated proofs. It would be interesting to see if, for example, automated verification tools such as #Π [76] could be used to make certain kinds of correctness proofs completely automatic.

IDE integration & synthesis. Finally, sequentializations and synchronizations have the honor of being easy and fast to compute (when they exist). Because they can be used to provide rapid feedback, it would be valuable to study to what extent they

aid developers in designing distributed applications. One could imagine an IDE that presents the developer with the current sequentialization, if it exists, or a set of send and receive statements that are unpaired. Moreover, we showed an implementation of canonical sequentialization that treats the *sequential prefix* as the unification variable in a Prolog predicate and "runs" the rewrite relation to calculate the sequentialization. An alternative idea is to ask the user to supply the *sequentialization* and leave the *input program* unknown, thus yielding a *synthesis* algorithm for distributed systems.

Appendix A

Translation from λ_m to $\mathbf{I_S}$

Lemma 3.1. *Assume $\mathcal{C} \to_{\mathrm{I_S}} \mathcal{C}'$ Then, for all $\langle \mu, \mathcal{P} \rangle \in \lfloor \mathcal{C} \rfloor_\gamma$, there exists $\langle \mu', \mathcal{P}' \rangle \in \lfloor \mathcal{C}' \rfloor_\gamma$ such that $\langle \mu, \mathcal{P} \rangle \twoheadrightarrow_{\lambda_m} \langle \mu', \mathcal{P}' \rangle$.*

Proof. The proof is by induction on the length of the derivation of $\xrightarrow{T}_{\mathrm{I_S}}$, splitting cases on the first step.

- Case Receive:

  By assumption,

  $$\mu[(p,q,t)] = v \cdot v' \tag{A.1}$$

  $$(\sigma, \mu, \left[\mathtt{recv}(t, \mathtt{w})_{\overline{\ell}}\right]_q) \xrightarrow{\left[\mathtt{recv}(t,p)_{\overline{\ell}}\right]_q}_{\mathrm{I_S}} (\sigma, \mu[(p,q,t) \leftarrow v'], [\gamma(q) \coloneqq v]_q) \tag{A.2}$$

  By the definition of $\lfloor \rfloor_\gamma$,

  $$\lfloor (\sigma, \mu, \left[\mathtt{recv}(t, \mathtt{w})_{\overline{\ell}}\right]_q) \rfloor_\gamma = \langle \mu, \mathtt{recv}_q \rangle \tag{A.3}$$

  $$\lfloor (\sigma, \mu[(p,q,t) \leftarrow v'], [\gamma(q) \coloneqq v]_q) \rfloor_\gamma = \langle \mu[(p,q,t) \leftarrow v'], (\mathtt{return}\ @t\ v)_q \rangle \tag{A.4}$$

  By the definition of $\to_{\lambda_m}$ and Equations (A.3) and (A.4),

  $$\lfloor (\sigma, \mu, \left[\mathtt{recv}(t, \mathtt{w})_{\overline{\ell}}\right]_q) \rfloor_\gamma \to_{\lambda_m} \lfloor (\sigma, \mu[(p,q,t) \leftarrow v'], [\gamma(q) \coloneqq v]_q) \rfloor_\gamma \tag{A.5}$$

  and the inductive hypothesis concludes the proof.

- Case Sequence:

  By assumption,

$$(\sigma, \mu, A) \xrightarrow{T}_{\mathrm{I_S}} (\sigma', \mu', \texttt{skip}) \tag{A.6}$$

  There are three cases for each singleton process:

  1. Case $A = \mathcal{L}(f, x, m)$ and $B = \texttt{while true \{ } \mathcal{L}(f, x, e) \texttt{ \}}$:

     Then, by definition

$$\langle \mu, m[\texttt{rec } f.\ (\lambda x.\ e)/f] \rangle \rightarrow_{\lambda_m} \langle \mu', (\texttt{rec } f.\ (\lambda x.\ e))\ \sigma(x) \rangle \tag{A.7}$$

     and unfolding f yields the conclusion, since $\vdash_f m$.

  2. Case Otherwise:

     There are again two cases to consider for $\langle \mu, \mathsf{Close}(\sigma, \gamma, \mathcal{P}) \rangle \in \lfloor (\sigma, \mu, A; B) \rfloor_\gamma$

     (a) Case $\mathcal{P} = m \texttt{ >>= } \lambda x.\ n$:

     By the inductive hypothesis,

$$\langle \mu, \mathsf{Close}(\sigma, \gamma[p \leftarrow x], m) \rangle \rightarrow_{\lambda_m} \langle \mu', \mathsf{Close}(\sigma', \gamma[p \leftarrow x], \sigma'(p, x)) \rangle \tag{A.8}$$

     And hence by contextual evaluation,

$$\langle \mu, \mathsf{Close}(\sigma, \gamma, (m \texttt{ >>= } \lambda x.\ n)_p) \rangle \rightarrow_{\lambda_m} \langle \mu, (\mathsf{Close}(\sigma', \gamma, n)[\sigma'(x)/x])_p \rangle \tag{A.9}$$

     (b) Case $\mathcal{P} = \texttt{let } y^t = m \texttt{ in } n$:

     Then (since we assume no variable shadowing)

$$\langle \mu, \mathsf{Close}(\sigma, \gamma, \texttt{let } y^t = m \texttt{ in } n) \rangle$$
$$= \langle \mu, \texttt{let } y^t = \mathsf{Close}(\sigma, \gamma, m) \texttt{ in } \mathsf{Close}(\sigma, \gamma, n) \rangle \tag{A.10}$$

and

$$\langle \mu, \texttt{let } y^t = \mathsf{Close}(\sigma, \gamma, m) \texttt{ in } \mathsf{Close}(\sigma, \gamma, n) \rangle \rightarrow_{\lambda_m} \langle \mu, \mathsf{Close}(\sigma, \gamma, n)[\mathsf{Close}(\sigma, \gamma, m)/y] \rangle \quad \text{(A.11)}$$

$$\langle \mu, \mathsf{Close}(\sigma, \gamma, \texttt{let } y^t = m \texttt{ in } n) \rangle \rightarrow_{\lambda_m} \langle \mu, \mathsf{Close}(\sigma, \gamma, \texttt{let } y^t = m \texttt{ in } n) \rangle \quad \text{(A.12)}$$

i. Case t is pure:

Therefore,

$$A = [y \coloneqq \mathsf{Apply}(\sigma, p, m)]_p \quad \text{(A.13)}$$

$$B = \mathcal{T}(n, \gamma(p), p) \quad \text{(A.14)}$$

And thus

$$(\sigma, \mu, A; B) \rightarrow_{\mathrm{I_S}} (\sigma[(p, y) \leftarrow \mathsf{Apply}(\sigma, p, m)], \mu, B) \quad \text{(A.15)}$$

Finally, by definition,

$$\langle \mu, \mathsf{Close}(\sigma, \gamma, \texttt{let } y^t = m \texttt{ in } n) \rangle \in \lfloor (\sigma[(p, y) \leftarrow \mathsf{Apply}(\sigma, p, m)], \mu, B) \rfloor_\gamma \quad \text{(A.16)}$$

ii. Otherwise:

The case is identical to the case for $m >\!\!>\!\!= n$ after the substitution.

- Case While-Expand:

By assumption,

$$(\sigma, \mu, \texttt{while true \{ } A \texttt{ \}}) \xrightarrow{\varepsilon}_{\mathrm{I_s}} (\sigma', \mu', A; \texttt{while true \{ } A \texttt{ \}}) \tag{A.17}$$

By definition,

$$\mathcal{L}(f, x, f\ x) = \texttt{skip} \tag{A.18}$$

$$\tag{A.19}$$

By the definition of $\lfloor\rfloor_\gamma$ and Equation (A.18), for some e, f, x, and t,

$$\lfloor(\sigma, \mu, \texttt{while true \{ } A \texttt{ \}})\rfloor_\gamma = \langle \mu, ((\texttt{rec } f.\ (\lambda x.\ \mathsf{Close}(\sigma \setminus x, \gamma, e)))\ \sigma(x))_p \rangle \tag{A.20}$$

By the definition of $\lfloor\rfloor_\gamma$,

$$\lfloor(\sigma, \mu, A; \texttt{while true \{ } A \texttt{ \}})\rfloor_\gamma = \langle \mu, \mathsf{Close}(\sigma, \gamma, e[\texttt{rec } f.\ (\lambda x.\ e)/f]) \rangle \tag{A.21}$$

By the definition of $\rightarrow_\beta$ (to unfold the recursive application) and hence $\rightarrow_{\lambda_m}$, and Equations (A.20) and (A.21),

$$\lfloor(\sigma, \mu, \texttt{while true \{ } A \texttt{ \}})\rfloor_\gamma \rightarrow_{\lambda_m} \lfloor(\sigma, \mu, A; \texttt{while true \{ } A \texttt{ \}})\rfloor_\gamma \tag{A.22}$$

The remaining cases are proved by similar expansions of the definition. □

Lemma 3.2. *For all well-formed E and σ, if $\mathcal{T}(E, \gamma)$ is defined, then there exists E'' such that $\mathcal{T}(E'', \gamma)$ is defined,* $\mathsf{Close}(\sigma', \gamma, E'') = E'$ *and*

$$\langle \mu, \mathsf{Close}(\sigma, \gamma, E) \rangle \twoheadrightarrow_{\lambda_m} \langle \mu', E' \rangle \Rightarrow (\sigma, \mu, \mathcal{T}(E, \gamma)) \rightarrow_{\mathrm{I_s}} (\sigma', \mu', \mathcal{T}(E'', \gamma))$$

Proof. By induction on the length of the transition sequence, splitting cases on the first

step.

- Case $E = \mathcal{H}[\texttt{recv @}t]_p$:

  By definition,

$$\langle \mu[(q,p,t) \leftarrow v \cdot vs], \mathsf{Close}(\sigma, \gamma, \mathcal{H}[\texttt{recv @}t]_p) \rangle \rightarrow_{\lambda_m} \langle \mu, \mathcal{H}[\texttt{return @}t\ v]_p \rangle \quad \text{(A.23)}$$

  There are two cases based on the shape of $\mathcal{H}$:

  1. Case $[]_p \parallel \mathcal{P}$:

     Then by definition,

$$\mathcal{T}(E, \gamma) = [\gamma(p) \coloneqq \texttt{recv}(t, *)]_p \parallel A \quad \text{(A.24)}$$

     and thus

$$(\sigma, \mu[(q,p,t) \leftarrow v \cdot vs], [\gamma(p) \coloneqq \texttt{recv}(t, *)]_p \parallel A) \rightarrow_{\mathrm{I_s}} (\sigma, \mu, [\gamma(p) \coloneqq v]_p \parallel A) \quad \text{(A.25)}$$

     Let $E'' = (\texttt{return @}t\ v)_p \parallel \mathcal{P}$ concludes the proof, as by definition,

$$\begin{aligned} [\gamma(p) \coloneqq v]_p &= \mathcal{T}(\texttt{return @}t\ v, \gamma(p), p) \\ &= \mathsf{Close}(\sigma, p, \texttt{return @}t\ v) \\ &= \texttt{return @}t\ v \end{aligned}$$

     which proves the case.

  2. Case $[]_p \texttt{>>=} \lambda x.\ m$:

     Then by definition,

$$\mathcal{T}(E, \gamma) = [x \coloneqq \texttt{recv}(t, *)]_p ; \mathcal{T}(m, \gamma(p), p) \parallel A \quad \text{(A.26)}$$

and thus

$$\begin{aligned}&(\sigma, \mu[(q,p,t) \leftarrow v \cdot vs], [\gamma(p) \coloneqq \texttt{recv}(t,*)]_p ; \mathcal{T}(m, \gamma(p), p) \parallel A) \\ \rightarrow_{\mathrm{I_s}} &(\sigma, \mu, [\gamma(p) \coloneqq v]_p ; \mathcal{T}(m, \gamma(p), p) \parallel A)\end{aligned} \tag{A.27}$$

Letting $E'' = \texttt{return}\ @t\ v \texttt{>>=} \lambda x.\ m \parallel \mathcal{P}$ concludes the proof as above.

- Case $E = \mathcal{H}[\texttt{return}\ @t\ n \texttt{>>=} \lambda x.\ m]_p$:

  Then we have by assumption and the form of $\rightarrow_{\lambda_m}$,

$$\begin{aligned}&\langle \mu, \mathsf{Close}(\sigma, \mu, \mathcal{H}[\texttt{return}\ @t\ n \texttt{>>=} \lambda x.\ m]_p) \rangle \\ \rightarrow_{\lambda_m} &\langle \mu, \mathsf{Close}(\sigma, \mu, \mathcal{H}[m[n/x]]_p) \rangle\end{aligned} \tag{A.28}$$

  And by definition,

$$\mathcal{T}(\mathcal{H}[\texttt{return}\ @t\ n \texttt{>>=} \lambda x.\ m]_p, \gamma) = [x \coloneqq n]_p ; \mathcal{T}(m, \gamma(p), p); A \parallel B \tag{A.29}$$

  By the rule for sequencing,

$$(\sigma, \mu, [x \coloneqq n]_p ; \mathcal{T}(m, \gamma(p), p); A \parallel B) \rightarrow_{\mathrm{I_s}} (\sigma[x \leftarrow \mathsf{Apply}(\sigma, p, n)], \mu, A \parallel B) \tag{A.30}$$

  It is easy to see that $m[n/x] = \mathsf{Close}(\sigma[x \leftarrow \mathsf{Apply}(\sigma, p, n)], \gamma, m)$, so letting $E'' = \mathcal{H}[m]_p$ finishes the case.

- Case $E = \mathcal{H}[(\texttt{rec}\ f.\ \lambda x.\ m)\ n]_p$:

  Then let $e' = m[n/x][(\texttt{rec}\ f.\ \lambda x.\ m)/f]$ and so by definition,

$$\langle \mu, \mathsf{Close}(\sigma, \gamma, \mathcal{H}[(\texttt{rec}\ f.\ \lambda x.\ m)]_p) \rangle \rightarrow_{\lambda_m} \langle \mu, \mathsf{Close}(\sigma, \gamma, \mathcal{H}[e']_p) \rangle \tag{A.31}$$

Next, by the definition of $\mathcal{T}$,

$$\mathcal{T}(\mathcal{H}[(\texttt{rec}\ f.\ \lambda x.\ m)\ n]_,\gamma) = [x \coloneqq n]_p\,;\texttt{while true \{}\ \mathcal{L}(f, x, \mathcal{T}_1(e, x, p))\ \texttt{\}}; A \parallel B \tag{A.32}$$

Since E is well formed, we know that m must have the form, for some c, a, b, m_0 where f does not appear in m_0.

$$m_0 \mathbin{>\!\!>\!=} \lambda r.\ (\lambda f.\ (\textsc{Loop}\ f\ c\ a\ b))\ f$$

and hence e' has form

$$m_0[n/x] \mathbin{>\!\!>\!=} \lambda r.\ (\lambda f.\ (\textsc{Loop}\ f\ c\ a\ b))\ (\texttt{rec}\ f.\ \lambda x.\ m)$$

so by definition

$$\mathcal{T}(\mathcal{H}[e']_p, \gamma(p), p) = \mathcal{L}(f, x, (\textsc{Loop}\ f\ c\ a\ b))\texttt{;while true \{}\ \mathcal{L}(f, x, \mathcal{T}(m, x, p))\ \texttt{\}} \tag{A.33}$$

and hence by the definition of $(\textsc{Loop}\ f\ c\ a\ b)$ and $\textsc{While-Expand}$,

$$\mathcal{T}(\mathcal{H}[(\texttt{rec}\ f.\ \lambda x.\ m)\ n]_p, \gamma(p), p) \rightarrow_{\mathrm{I_s}} \mathcal{T}(\mathcal{H}[e']_p, \gamma(p), p)$$

The remaining cases are similar. □

Appendix B

Sequentialization

Definition B.1 (Composition of buffers)**.** As an abuse of notation, define $\mu \circ \mu'$ as follows:

$$(\mu \cdot \mu')(p, q, t) = \mu(p, q, t) \cdot \mu'(p, q, t)$$

Definition B.2 (Interpretation of stores)**.** $\sigma \in [\![\Delta]\!]_{\sigma_0}$ if and only if $(\sigma_0, \varnothing, \Delta) \to^* (\sigma, \varnothing, \texttt{skip})$

Definition B.3 (Interpretation of contexts)**.** $\mu \in [\![\Gamma]\!]_\sigma$ if and only if $\forall (p, q, t) \in \mathsf{dom}(\Gamma)$, $\mu(p, q, t) = [\![\Gamma(p, q, t)]\!]_\sigma$

Definition B.4 (Interpretation of stores and contexts)**.** Let $(\sigma, \mu) \in [\![\Delta, \Gamma]\!]_{\sigma_0}$ if and only if

1. $\sigma \in [\![\Delta]\!]_{\sigma_0}$
2. $\mu \in [\![\Gamma]\!]_\sigma$
3. $\forall \{x \in X\} \in \Gamma.\ \sigma(x) \in \sigma(X)$
4. $\forall \{\varnothing \subset X \subseteq Y\} \in \Gamma.\ \varnothing \subset \sigma(X) \subseteq \sigma(Y)$

Definition B.5 (Preorder on stores and buffers)**.**

$$\sigma \preceq \sigma' \leftrightarrow \mathsf{dom}(\sigma) \subseteq \mathsf{dom}(\sigma') \wedge \forall\, x \in \mathsf{dom}(\sigma).\ \sigma'(x) = \sigma(x)$$

$$\mu \preceq \mu \leftrightarrow \mathsf{dom}(\mu) \subseteq \mathsf{dom}(\mu') \wedge \forall\, x \in \mathsf{dom}(\mu).\ \exists\, m.\ \mu'(x) = \mu(x) \cdot m$$

Definition B.6 (Halted processes)**.** $p \in \mathsf{hprocs}(\sigma, \mu, P)$ if and only if $P \equiv \left[s_p\right]_p \parallel P_0$ and any of the following hold:

1. $s_p \equiv \texttt{skip}$ or $s_p \equiv \texttt{error}$
2. The next action in s_p is $x \coloneqq \texttt{recv}(t, y)$, $\sigma(y) = q$, and for all (σ', μ', P') such that $(\sigma, \mu, P) \to^* (\sigma', \mu', P')$, $\mu'(q, p, t) = \varnothing$.
3. The next action in s_p is $x \coloneqq \texttt{recv}(t, S)$ and for all (σ', μ', P') such that $(\sigma, \mu, P) \to^* (\sigma', \mu', P')$, $\mu'(q, p, t) = \varnothing$ for all $q \in S$.

Definition B.7 (Restriction of program stores and buffers)**.**

$$\sigma|_P(x_p) = \begin{cases} \sigma(x_p) & p \in P \\ \bot & \textit{otherwise} \end{cases}$$

$$\mu|_Q(p, q, t) = \begin{cases} \mu(p, q, t) & q \in Q \\ \bot & \textit{otherwise} \end{cases}$$

Definition B.8 (Preorder on halted states)**.**

$$(\sigma, \mu, P) \preceq (\sigma', \mu', P') \leftrightarrow \sigma|_H \preceq \sigma'|_H \wedge \mu|_H \preceq \mu'|_H \qquad \textit{where } H = \mathsf{hprocs}(\sigma, \mu, P)$$

Definition B.9 (Simulation on states)**.** Define the relation $\sqsubseteq$ on program states such that $(\sigma, \mu, P) \sqsubseteq (\sigma', \mu', P')$ if and only if whenever

$$(\sigma, \mu, P) \to^* (\sigma_f, \mu_f, P_f)$$

then there exists $(\sigma'_f, \mu'_f, P'_f)$ such that

$$(\sigma', \mu', P') \to^* (\sigma_f, \mu_f, P_f) \quad \text{and} \quad (\sigma_f, \mu_f, P_f) \preceq (\sigma_f, \mu_f, P_f)$$

Definition B.10. We overload $\sqsubseteq$ to rewriting states as follows.

Let $\Gamma, \Delta, P, \Psi \sqsubseteq \Gamma', \Delta; \Delta', P', \Psi'$ if and only if, for all P_x such that $\mathsf{rf}(P \ltimes P_x)$,

$$\forall\, (\sigma, \mu) \in [\![\Delta, \Gamma]\!]_{\varnothing}.\ \exists\, (\sigma', \mu') \in [\![\Delta', \Gamma']\!]_{\sigma}.\ (\sigma, \mu, \Psi_x \ltimes \Psi; P \ltimes P_x) \sqsubseteq (\sigma', \mu', \Psi_x \ltimes \Psi'; P' \ltimes P_x)$$

Definition B.11 (Left movers)**.** Let λ be a *left-mover in* $(\sigma, \mu, P \parallel [\lambda; s]_p)$ if and only if

1. If $(\sigma, \mu, P \parallel [\lambda; s]_p) \to^* (\sigma', \mu', P' \parallel [\lambda; s]_p)$ then λ is enabled in $(\sigma', \mu', P' \parallel [\lambda; s]_p)$. ($\lambda$ does not become disabled.)
2. If $(\sigma, \mu, P \parallel [\lambda; s]_p) \xrightarrow{[t]_q} (\sigma_0, \mu_0, P_0 \parallel [\lambda; s]_p) \xrightarrow{[\lambda]_p} (\sigma', \mu', P' \parallel [s]_p)$ then there exists (σ_1, μ_1, P_1) such that $(\sigma, \mu, P \parallel [\lambda; s]_p) \xrightarrow{[\lambda]_p} (\sigma_1, \mu_1, P_1 \parallel [s]_p) \xrightarrow{[t]_q} (\sigma', \mu', P' \parallel [s]_p)$. ($\lambda$ commutes to the left.)

Lemma B.1. *If* $\sigma \in [\![\Delta]\!]_{\sigma'}$ *and* $\sigma' \in [\![\Delta_0]\!]_{\sigma_0}$ *then* $\sigma \in [\![\Delta_0; \Delta]\!]_{\sigma_0}$.

Proof. By the definition of $[\![\cdot]\!]$. □

Lemma B.2 (Send is a left mover)**.** *For all* $\sigma, \mu, P, p, s,$ $\mathsf{send}(t, x, m)$ *is a left-mover in* $(\sigma, \mu, P \parallel [\mathsf{send}(t, x, m); s]_p)$.

Proof. The proof is by the definition of left movers.

1. By the definition of $\to^*$ it is clear that $\mathsf{send}(t, x, m)$ in $[\mathsf{send}(t, x, m); s]_p$ is always enabled.
2. Let $(\sigma, \mu, P \parallel [\mathsf{send}(t, x, m); s]_p) \xrightarrow{[t]_q} (\sigma_0, \mu_0, P_0 \parallel [\mathsf{send}(t, x, m); s]_p)$. Then

$$(\sigma_0, \mu_0, P_0 \parallel [\mathsf{send}(t, x, m); s]_p) \xrightarrow{[\mathsf{send}(t,x,m)]_p} (\sigma', \mu', P_0 \parallel [s]_p)$$

with $\sigma' = \sigma_0$ and $\mu'(p, \sigma(x), t) = \mu_0[(p, \sigma(x), t) \leftarrow \mu_0(p, \sigma(x), t) \cdot \sigma(m)]$.

□

Lemma B.3. *If λ is a left mover in $(\sigma, \mu, P \parallel [\lambda; s]_p)$ then*

$$(\sigma, \mu, P \parallel [\lambda; s]_p) \sqsubseteq (\sigma, \mu, \lambda; (P \parallel [s]_p))$$

Proof. Suppose $(\sigma, \mu, P \parallel [\lambda; s]_p) \to^* (\sigma', \mu', P')$. Then there is a sequence of transitions $T \in \mathsf{Traces}(\sigma, \mu, P \parallel [\lambda; s]_p)$ that reaches (σ', μ', P') We need to show that there exists a configuration (σ'', μ'', P'') reachable from $(\sigma, \mu, [\lambda]_p; (P \parallel [s]_p))$ such that $(\sigma', \mu', P') \preceq (\sigma'', \mu'', P'')$.

- Case $T_i = \lambda$ for some i.

  Then by the definition of left movers, T_i can be moved to the left in T until it appears first. The resulting trace is also a trace of $(\sigma, \mu, [\lambda]_p; (P \parallel [s]_p))$, so $\sigma'' = \sigma'$, $\mu'' = \mu'$, and $P'' = P'$.

- Case $T_i \neq \lambda$ for all i.

  Then, by definition, $P' \equiv P'_\lambda \parallel [\lambda; s]_p$ and $(\sigma', \mu', P'_\lambda \parallel [\lambda; s]_p) \to^* (\sigma'_\lambda, \mu'_\lambda, P'_\lambda \parallel [s]_p)$ by executing $[\lambda]_p$.

  It must be the case that $p \notin \mathsf{hprocs}(\sigma', \mu', P')$ by definition, since λ is enabled. The proof that $(\sigma', \mu', P') \preceq (\sigma'_\lambda, \mu'_\lambda, P'_\lambda \parallel [s]_p)$ is by contradiction, so we assume $(\sigma', \mu', P') \not\preceq (\sigma'_\lambda, \mu'_\lambda, P'_\lambda \parallel [s]_p)$. Letting $H = \mathsf{hprocs}(\sigma', \mu', P')$ there are three cases for some $q \in H$:

  - Case $\sigma'(x_q) \neq \sigma'_\lambda(x_q)$.

    Then x_q must be assigned in λ and hence $q = p$ since processes only assign to local variables. But $p \in H$ is a contradiction, as $[\lambda]_p$ is enabled in P'.

  - Case $\forall\, m.\ \mu'_\lambda(q', q, t) \neq \mu'(q', q, t) \cdot m$. Since p executes λ, $p = q$ or $p = q'$. In the former case, then $p \in H$ which is a contradiction. In the latter λ could have only *appended* messages to $\mu'(p, r', t)$, which contradicts the assumption that $\mu'_\lambda(p, q, t)$ does not contain $\mu'(p, q, t)$ as a prefix.

□

Lemma B.4. *If* $\Gamma, \Delta, P, \Psi \rightsquigarrow \Gamma', \Delta; \Delta', P', \Psi'$ *then* $\Gamma, \Delta, P, \Psi \sqsubseteq \Gamma', \Delta; \Delta', P', \Psi'$

Proof. The proof is by induction on the derivation of $\Gamma, \Delta, P, \Psi \rightsquigarrow \Gamma', \Delta; \Delta', P', \Psi'$, and by case analysis on the last step.

- Case R-SEND.

  Let $(\sigma, \mu) \in [\![\Gamma, \Delta]\!]$ and assume

  $$(\sigma, \mu, \Psi_x \ltimes \Psi; [\mathsf{send}(t, x, n)]_p \ltimes P_x) \to^* (\sigma', \mu', H)$$

  and thus by definition of sequencing,

  $$(\sigma, \mu \cdot \mu_\Psi, [\mathsf{send}(t, x, n)]_p \ltimes P_x) \to^* (\sigma', \mu', H)$$

  then by lemma B.3:

  $$(\sigma, \mu \cdot \mu_\Psi, [\mathsf{send}(t, x, n)]_p ; P_x) \to^* (\sigma', \mu', H)$$

  and hence

  $$(\sigma, \mu \cdot \mu_\Psi \cdot [(p, q, t) \leftarrow (\mu \circ \mu_\Psi) \cdot n], P_x) \to^* (\sigma', \mu', H).$$

  By assumption, $\sigma(x) = q$ and $\Gamma \not\vdash \mathsf{E}(q)$, so $(p, q, t) \notin \mathsf{dom}(\mu_\Psi)$ and thus

  $$\mu \cdot \mu_\Psi \cdot [(p, q, t) \leftarrow (\mu \circ \mu_\Psi)(p, q, t) \cdot n] = \mu \cdot [(p, q, t) \leftarrow \mu(p, q, t) \cdot n] \cdot \mu_\Psi,$$

  which gives us

  $$(\sigma, \mu[(p, q, t) \leftarrow \mu(p, q, t) \cdot n] \circ \mu_\Psi, P_x) \to^* (\sigma', \mu', H).$$

  and hence

  $$(\sigma, \mu[(p, q, t) \leftarrow \mu(p, q, t) \cdot n], \Psi_x \ltimes \Psi; P_x) \to^* (\sigma', \mu', H).$$

- Case R-RECV.

  Let $(\sigma, \mu) \in [\![\Delta, \Gamma]\!]$ and assume

  $$(\sigma, \mu, \Psi_x \ltimes \Psi; [x \coloneqq \mathtt{recv}(t, p)]_q \ltimes P_x) \to^* (\sigma', \mu', H).$$

  Then,

  $$(\sigma, \mu \circ \mu_\Psi, [x \coloneqq \mathtt{recv}(t, p)]_q \ltimes P_x) \to^* (\sigma', \mu', H),$$

  and by Lemma B.3,

  $$(\sigma, \mu \circ \mu_\Psi, [x \coloneqq \mathtt{recv}(t, p)]_q ; P_x) \to^* (\sigma', \mu', H),$$

  and hence

  $$(\sigma, \mu \cdot \mu_\Psi \cdot [(p, q, t) \leftarrow \mathtt{pop}(\mu_\Psi(p, q, t))], P_x) \to^* (\sigma', \mu', H).$$

  As in the case of R-SEND, we exchange buffers to produce

  $$(\sigma, \mu \cdot [(p, q, t) \leftarrow \mathtt{pop}(\mu(p, q, t))] \cdot \mu_\Psi, P_x) \to^* (\sigma', \mu', H).$$

- Case R-CONTEXT.

  By straightforward application of the inductive hypothesis.

- Case R-CONGRUENCE.

  By definition, as $A \equiv B$ if and only if the set of program traces in A is equivalent to the program traces in B.

- Case R-RECV-UNFOLD.

  Let $(\sigma, \mu) \in [\![\Delta, \Gamma]\!]$ and assume

  $$(\sigma, \mu, \Psi_x \ltimes \Psi; [x \coloneqq \mathtt{recv}(Q, t)]_p \parallel \prod_{q:Q'} [A]_q \ltimes P_x) \to^* (\sigma', \mu', H).$$

via some sequence of actions T. If $[x \coloneqq \texttt{recv}(Q,t)]_q$ does not appear in T then T must also be a trace of $(\sigma, \mu, \Psi_x \ltimes \Psi; [x \coloneqq \texttt{recv}(q^*,t)]_p \parallel \bigsqcap_{q:Q'} [A]_q)$ and hence of $(\sigma, \mu, \Psi_x \ltimes \Psi; [x \coloneqq \texttt{recv}(q^*,t)]_p \parallel \bigsqcap_{q:Q' \setminus q^*} [A]_{q^*})$.

Otherwise, there are configurations such that

$$\begin{aligned}(\sigma, \mu, \Psi_x \ltimes \Psi; [x \coloneqq \texttt{recv}(Q,t)]_p \parallel \prod_{q:Q'} [A]_q \ltimes P_x) &\xrightarrow{T_0} (\sigma_0, \mu_0, P_0) \\ &\xrightarrow{[x \coloneqq \texttt{recv}(Q,t)]_p} (\sigma_1, \mu_1, P_1) \\ &\xrightarrow{T_1} (\sigma', \mu', H)\end{aligned}$$

and there is some $q \in Q$ such that $\mu_0(q,p,t) \neq \mu_1(q,p,t)$ and hence was the matching sender for p's receive. First, we show by contradiction we that $q \in Q'$, for if $q \in Q \setminus Q'$ and the unique syntactic matching send must appear in P_x. But by the IH the syntactic matching send must be in A. Since $\mathsf{rf}(\bigsqcap_{q:Q'} [A]_q \ltimes P_x)$ by assumption, this is a contradiction. Finally, since $q \in Q'$, the trace $T_0 \cdot [x \coloneqq \texttt{recv}(Q,t)]_p \cdot T_1$ is a trace of the output program and we are done.

- Case R-SEND-UNFOLD.

  The case follows by definition since the output is congruent with the input program.

- Case R-SEND-RESID.

  Let $(\sigma, \mu) \in [\![\Delta, \Gamma]\!]$ and assume

  $$(\sigma, \mu, \Psi_x \ltimes \Psi; [\texttt{send}(t,x,n)]_p \ltimes P_x) \to^* (\sigma', \mu', H).$$

  Then because $\Psi_x \ltimes \Psi$ are sequenced, we can conclude for some μ_Ψ,

  $$(\sigma, \mu \cdot \mu_\Psi; [\texttt{send}(t,x,n)]_p \ltimes P_x) \to (\sigma, \mu \cdot \mu_\Psi \cdot [(p,q,t) \leftarrow \sigma(n)]; P_x) \to^* (\sigma', \mu', H).$$

  Now, there exists $(\sigma_V, \mu) \in [\![\Delta; V \coloneqq V \cup n, \Gamma]\!]$ such that $\sigma(n) \in \sigma_V(V)$. And further-

more,

$$(\sigma_V, \mu, \Psi_x \ltimes \Psi; [\textstyle\sum v{:}V.\mathsf{send}(t, x, v)]_p \ltimes P_x) \to^* (\sigma', \mu', H).$$

And hence, as before,

$$\begin{aligned}
&(\sigma_V, \mu, \Psi_x \ltimes \Psi; [\textstyle\sum v{:}V.\mathsf{send}(t, x, v)]_p \ltimes P_x) \\
\to^* &(\sigma_V, \mu \cdot \mu_\Psi; [\textstyle\sum v{:}V.\mathsf{send}(t, x, v)]_p \ltimes P_x) \\
\to^* &(\sigma', \mu', H).
\end{aligned}$$

Since $\sigma(n) \in \sigma_V(V)$, there is a choice of v such that $\sigma(n) = v$ resulting in

$$\begin{aligned}
&(\sigma_V, \mu \cdot \mu_\Psi; [\textstyle\sum v{:}V.\mathsf{send}(t, x, v)]_p \ltimes P_x) \\
\to^* &(\sigma_V, \mu \cdot \mu_{\Psi \cdot}; [\mathsf{send}(t, x, n)]_p \ltimes P_x) \\
\to^* &(\sigma', \mu', H).
\end{aligned}$$

which concludes the case.

- Case R-COMPOSE-RESID.

  This case follows by straightforward application of the inductive hypothesis, and from the fact that the rule explicitly checks that the residual term satisfies symmetric non-determinism.

- Case R-WHILE-REPEAT and R-WHILE-REMOVE.

  These cases follow by definition, as they simply "unfold" a single iteration of the while loop.

- Case R-IF-THEN and R-IF-ELSE.

  These cases follow by definition, as (`if true { `s_1` } else { `s_2` }`) $\equiv s_1$ (resp. `false` and s_2).

- Case R-LOOP-UPD.

The proof of this case is by induction on the size of Q.

– $|Q| = 1$.

  Then for some q_0, $Q = \{q_0\}$ and by definition,

$$\texttt{for } (q : Q) \texttt{ \{ } A \texttt{ \}} \equiv A[q_0/q]$$

$$\prod_{q:Q} B; C \equiv B; C[q_0/q].$$

  By hypothesis,

$$(\sigma, \mu) \in [\![\Gamma, \Delta]\!]$$

$$(\sigma, \mu, \Psi_x \ltimes \Psi; (A[q_0/q] \parallel B; C[q_0/q]) \ltimes P_x) \to^* (\sigma', \mu', H).$$

  By the definition of Γ_0 and Δ_0,

$$(\sigma, \mu) \in [\![\Gamma_0, \Delta_0]\!].$$

  By the rewriting hypothesis,

$$(\sigma, \mu, \Delta^{q_0}; \Psi_x \ltimes \Psi \ltimes \Psi^{q_0}; X) \to^* (\sigma'', \mu'', H'')$$

  such that $(\sigma', \mu', H) \preceq (\sigma'', \mu'', H')$. By definition again,

$$\Delta' = \Delta; \Delta^{q_0} = \Delta; \texttt{for } (q : \{q_0\}) \texttt{ \{ } \Delta^q \texttt{ \}}$$

$$\Psi' = \Psi \parallel \Psi^{q_0}$$

  so

$$(\sigma, \mu, \texttt{for } (q : Q) \texttt{ \{ } \Delta^{q_0}[q/q_0] \texttt{ \}}; \Psi_x \ltimes \Psi \parallel \Psi^{q_0}; P_x) \to^* (\sigma'', \mu'', H'')$$

– $|Q| = n + 1$ and $IH(n)$.

Then $Q = Q' \cup \{q_n\}$ for some $q_n \notin Q'$, so

$$\texttt{for}\ (q : Q)\ \{\ A\ \} \equiv A(q_n); \texttt{for}\ (q : Q')\ \{\ A[q_n/q]\ \}.$$

and hence the obligation is solved by the inductive hypothesis, since

$$\texttt{for}\ (q : Q)\ \{\ A\ \} \equiv A(q_n); \texttt{for}\ (q : Q')\ \{\ A[q_n/q]\ \} \parallel \prod_{q:Q} B; C \ltimes P_x$$

is symmetrically nondeterministic as p is a concrete process. By the rewriting assumption, if for $(\sigma, \mu) \in [\![\Gamma, \Delta]\!]$,

$$(\sigma, \mu, \Psi_x \ltimes \Psi; A[q_n/q] \parallel \prod_{q:Q} B \ltimes (\texttt{for}\ (q : Q)\ \{\ A\ \} \parallel \prod_{q:Q} C)) \to (\sigma', \mu', H),$$

then

$$(\sigma, \mu, \Delta'; \Psi_x \ltimes \Psi'; \prod_{q:Q \setminus u} B \ltimes (\texttt{for}\ (q : Q)\ \{\ A\ \} \parallel \prod_{q:Q} C)) \to (\sigma', \mu', H),$$

By "running" Δ' we obtain $\sigma_{\Delta'}$, and hence

$$(\sigma_{\Delta'}, \mu, \Psi_x \ltimes \Psi'; \prod_{q:Q \setminus u} B \ltimes (\texttt{for}\ (q : Q)\ \{\ A\ \} \parallel \prod_{q:Q} C)) \to (\sigma', \mu', H),$$

and apply the inductive hypothesis to finish the obligation.

□

Bibliography

[1] Cloud Haskell Semantics (draft). https://haskell-distributed.github.io/static/semantics.pdf. Accessed: 2017-07-31.

[2] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991.

[3] Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Optimal dynamic partial order reduction. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 373–384, 2014.

[4] Alexander Bakst and Ranjit Jhala. Predicate abstraction for linked data structures. In *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings*, pages 65–84, 2016.

[5] Alexander Bakst, Klaus v. Gleissenthall, Rami Gökhan Kıcı, and Ranjit Jhala. Verifying Distributed Programs via Canonical Sequentialization. *Accepted for publication in the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2017*.

[6] Samik Basu, Tevfik Bultan, and Meriem Ouederni. Synchronizability for verification of asynchronously communicating systems. In *Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings*, pages 56–71, 2012.

[7] Nikolaj Bjørner, Kenneth L. McMillan, and Andrey Rybalchenko. On solving universally quantified horn clauses. In *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*, pages 105–125, 2013.

[8] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays? In *Verification, Model Checking, and Abstract Interpretation, 7th International Conference, VMCAI 2006, Charleston, SC, USA, January 8-10, 2006, Proceedings*, pages 427–442, 2006.

[9] Saksham Chand, Yanhong A. Liu, and Scott D. Stoller. Formal verification of multi-paxos for distributed consensus. In *FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings*, pages 119–136, 2016.

[10] Minas Charalambides, Peter Dinges, and Gul A. Agha. Parameterized, concurrent session types for asynchronous multi-actor interactions. *Sci. Comput. Program.*, 115-116:100–126, 2016.

[11] Bernadette Charron-Bost and André Schiper. The heard-of model: computing in distributed systems with benign faults. *Distributed Computing*, 22(1):49–71, 2009.

[12] Alessandro Cimatti, Iman Narasamdya, and Marco Roveri. Boosting lazy abstraction for systemc with partial order reduction. In *Tools and Algorithms for the Construction and Analysis of Systems - 17th International Conference, TACAS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, pages 341–356, 2011.

[13] Robert L. Constable, Stuart F. Allen, Mark Bromley, Rance Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, Todd B. Knoblock, N. P. Mendler, Prakash Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing mathematics with the Nuprl proof development system*. Prentice Hall, 1986.

[14] Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts, and Hernán Vanzetto. TLA + proofs. In *FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*, pages 147–154, 2012.

[15] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252, 1977.

[16] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 137–150, 2004.

[17] Pierre-Malo Deniélou and Nobuko Yoshida. Dynamic multirole session types. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 435–446, 2011.

[18] Pierre-Malo Deniélou, Nobuko Yoshida, Andi Bejleri, and Raymond Hu. Parameterised multiparty session types. *Logical Methods in Computer Science*, 8(4), 2012.

[19] Ankush Desai, Shaz Qadeer, and Sanjit A. Seshia. Systematic testing of asynchronous reactive systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 73–83, 2015.

[20] Disco-Project. Disco MapReduce. http://discoproject.org, April 2017.

[21] Emanuele D'Osualdo, Jonathan Kochems, and Luke Ong. Soter: an automatic safety verifier for erlang. In *Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions, AGERE! 2012, October 21-22, 2012, Tucson, Arizona, USA*, pages 137–140, 2012.

[22] Cezara Dragoi, Thomas A. Henzinger, Helmut Veith, Josef Widder, and Damien Zufferey. A logic-based framework for verifying consensus algorithms. In *Verification, Model Checking, and Abstract Interpretation - 15th International Conference, VMCAI 2014, San Diego, CA, USA, January 19-21, 2014, Proceedings*, pages 161–181, 2014.

[23] Cezara Dragoi, Thomas A. Henzinger, and Damien Zufferey. Psync: a partially synchronous language for fault-tolerant distributed algorithms. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 400–415, 2016.

[24] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. A calculus of atomic actions. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 2–15, 2009.

[25] Jeff Epstein, Andrew P. Black, and Simon L. Peyton Jones. Towards haskell in the cloud. In *Proceedings of the 4th ACM SIGPLAN Symposium on Haskell, Haskell 2011, Tokyo, Japan, 22 September 2011*, pages 118–129, 2011.

[26] Manuel Fähndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, pages 13–24, 2002.

[27] Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. Proofs that count. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 151–164, 2014.

[28] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 110–121, 2005.

[29] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996.

[30] Tim Harris, Simon Marlow, Simon L. Peyton Jones, and Maurice Herlihy. Composable memory transactions. *Commun. ACM*, 51(8):91–100, 2008.

[31] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. Ironfleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 1–17, 2015.

[32] Kohei Honda. Types for dyadic interaction. In *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*, pages 509–523, 1993.

[33] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 273–284, 2008.

[34] C. Norris Ip and David L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1/2):41–75, 1996.

[35] Simon L. Peyton Jones. Compiling haskell by program transformation: A report from the trenches. In *Programming Languages and Systems - ESOP'96, 6th European Symposium on Programming, Linköping, Sweden, April 22-24, 1996, Proceedings*, pages 18–44, 1996.

[36] Ming Kawaguchi, Patrick Maxim Rondon, Alexander Bakst, and Ranjit Jhala. Deterministic parallelism via liquid effects. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 45–54, 2012.

[37] Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin Vahdat. Mace: language support for building distributed systems. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 179–188, 2007.

[38] Igor Konnov, Helmut Veith, and Josef Widder. SMT and POR beat counter abstraction: Parameterized model checking of threshold-based distributed algorithms. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pages 85–102, 2015.

[39] Igor V. Konnov, Marijana Lazic, Helmut Veith, and Josef Widder. A short counterexample property for safety and liveness verification of fault-tolerant distributed algorithms. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 719–734, 2017.

[40] Igor V. Konnov, Helmut Veith, and Josef Widder. What you always wanted to know about model checking of fault-tolerant distributed algorithms. In *Perspectives of System Informatics - 10th International Andrei Ershov Informatics Conference, PSI 2015, in Memory of Helmut Veith, Kazan and Innopolis, Russia, August 24-27, 2015, Revised Selected Papers*, pages 6–21, 2015.

[41] Leslie Lamport. Paxos made simple. pages 51–58, December 2001.

[42] Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[43] Butler Lampson and Howard E. Sturgis. Crash recovery in a distributed data storage system. In *Technical report XEROX Palo Alto Research Center*, 1976.

[44] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, pages 348–370, 2010.

[45] Richard J. Lipton. Reduction: A new method of proving properties of systems of processes. In *Conference Record of the Second ACM Symposium on Principles of Programming Languages, Palo Alto, California, USA, January 1975*, pages 78–86, 1975.

[46] Ognjen Maric, Christoph Sprenger, and David A. Basin. Cutoff bounds for consensus algorithms. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, pages 217–237, 2017.

[47] Simon Marlow. *Parallel and concurrent programming in Haskell: Techniques for multicore and multithreaded programming*. " O'Reilly Media, Inc.", 2013.

[48] Eduardo R. B. Marques, Francisco Martins, Vasco T. Vasconcelos, Nicholas Ng, and Nuno Martins. Towards deductive verification of MPI programs against session types. In *Proceedings 6th Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, PLACES 2013, Rome, Italy, 23rd March 2013.*, pages 103–113, 2013.

[49] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.

[50] Kenneth L. McMillan. Verification of infinite state systems by compositional model checking. In *Correct Hardware Design and Verification Methods, 10th IFIP WG 10.5 Advanced Research Working Conference, CHARME '99, Bad Herrenalb, Germany, September 27-29, 1999, Proceedings*, pages 219–234, 1999.

[51] Neil Mitchell and Colin Runciman. Losing functions without gaining data: another look at defunctionalisation. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell, Haskell 2009, Edinburgh, Scotland, UK, 3 September 2009*, pages 13–24, 2009.

[52] David Monniaux and Francesco Alberti. A simple abstraction of arrays and maps by program translation. In *Static Analysis - 22nd International Symposium, SAS 2015, Saint-Malo, France, September 9-11, 2015, Proceedings*, pages 217–234, 2015.

[53] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How amazon web services uses formal methods. *Commun. ACM*, 58(4):66–73, 2015.

[54] Nicholas Ng and Nobuko Yoshida. Static deadlock detection for concurrent go by global session graph synthesis. In *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, pages 174–184, 2016.

[55] Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014.*, pages 305–319, 2014.

[56] Dominic A. Orchard and Nobuko Yoshida. Effects as sessions, sessions as effects. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 568–581, 2016.

[57] Sam Owre, John M. Rushby, Natarajan Shankar, and Friedrich W. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Trans. Software Eng.*, 21(2):107–125, 1995.

[58] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: safety verification by interactive generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 614–630, 2016.

[59] Doron A. Peled. All from one, one for all: on model checking using representatives. In *Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings*, pages 409–423, 1993.

[60] Ruzica Piskac, Leonardo Mendonça de Moura, and Nikolaj Bjørner. Deciding effectively propositional logic using DPLL and substitution sets. *J. Autom. Reasoning*, 44(4):401–424, 2010.

[61] Amir Pnueli, Jessie Xu, and Lenore D. Zuck. Liveness with (0, 1, infty)-counter abstraction. In *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, pages 107–122, 2002.

[62] Vincent Rahli. Interfacing with proof assistants for domain specific programming using eventml. 2012.

[63] Patrick Maxim Rondon, Alexander Bakst, Ming Kawaguchi, and Ranjit Jhala. Csolve: Verifying C with liquid types. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, pages 744–750, 2012.

[64] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001, IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, Germany, November 12-16, 2001, Proceedings*, pages 329–350, 2001.

[65] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, pages 105–118, 1999.

[66] Alejandro Sánchez, Sriram Sankaranarayanan, César Sánchez, and Bor-Yuh Evan Chang. Invariant generation for parametrized systems using self-reflection - (extended version). In *Static Analysis - 19th International Symposium, SAS 2012, Deauville, France, September 11-13, 2012. Proceedings*, pages 146–163, 2012.

[67] Nicolas Schiper, Vincent Rahli, Robbert van Renesse, Mark Bickford, and Robert L. Constable. Developing correctly replicated databases using formal tools. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*, pages 395–406, 2014.

[68] Stephen F. Siegel. Efficient verification of halting properties for MPI programs with wildcard receives. In *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Paris, France, January 17-19, 2005, Proceedings*, pages 413–429, 2005.

[69] Stephen F. Siegel and George S. Avrunin. Modeling wildcard-free MPI programs for verification. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2005, June 15-17, 2005, Chicago, IL, USA*, pages 95–106, 2005.

[70] Stephen F. Siegel and Ganesh Gopalakrishnan. Formal analysis of message passing - (invited talk). In *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, pages 2–18, 2011.

[71] Stephen F. Siegel and Timothy K. Zirkel. Automatic formal verification of mpi-based parallel programs. In *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2011, San Antonio, TX, USA, February 12-16, 2011*, pages 309–310, 2011.

[72] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, 2003.

[73] Samira Tasharofi, Rajesh K. Karmani, Steven Lauterburg, Axel Legay, Darko Marinov, and Gul Agha. Transdpor: A novel dynamic partial-order reduction technique for testing actor programs. In *Formal Techniques for Distributed Systems - Joint 14th IFIP WG 6.1 International Conference, FMOODS 2012 and 32nd IFIP WG 6.1 International Conference, FORTE 2012, Stockholm, Sweden, June 13-16, 2012. Proceedings*, pages 219–234, 2012.

[74] Klaus v. Gleissenthall, Alexander Bakst, Rami Gökhan Kıcı, and Ranjit Jhala. Pretend Synchrony. *In submission to the ACM SIGPLAN International Conference on Principles of Programming Languages, POPL 2018*.

[75] Niki Vazou, Alexander Bakst, and Ranjit Jhala. Bounded refinement types. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, pages 48–61, 2015.

[76] Klaus von Gleissenthall, Nikolaj Bjørner, and Andrey Rybalchenko. Cardinalities and universal quantifiers for verifying parameterized systems. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 599–613, 2016.

[77] Björn Wachter, Daniel Kroening, and Joël Ouaknine. Verifying multi-threaded software with impact. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 210–217, 2013.

[78] Philip Wadler. Propositions as sessions. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, pages 273–286, 2012.

[79] David Walker and J. Gregory Morrisett. Alias types for recursive data structures. In *Types in Compilation, Third International Workshop, TIC 2000, Montreal, Canada, September 21, 2000, Revised Selected Papers*, pages 177–206, 2000.

[80] James R. Wilcox, Ilya Sergey, and Zachary Tatlock. Programming language abstractions for modularly verified distributed systems. In *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA*, pages 19:1–19:12, 2017.

[81] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 357–368, 2015.

[82] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas E. Anderson. Planning for change in a formal verification of the raft consensus protocol. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, Saint Petersburg, FL, USA, January 20-22, 2016*, pages 154–165, 2016.

[83] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: transparent model checking of unmodified distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009, April 22-24, 2009, Boston, MA, USA*, pages 213–228, 2009.

[84] Pamela Zave. Using lightweight modeling to understand chord. *Computer Communication Review*, 42(2):49–57, 2012.