UC Berkeley
UC Berkeley Electronic Theses and Dissertations

Title
Specification Mining: New Formalisms, Algorithms and Applications

Permalink
https://escholarship.org/uc/item/4027r49r

Author
Li, Wenchao

Publication Date
2013

Peer reviewed|Thesis/dissertation

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

Specification Mining: New Formalisms, Algorithms and Applications

by

Wenchao Li

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

in

Engineering – Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Sanjit A. Seshia, Chair
Professor Andreas Kuehlmann
Professor Francesco Borrelli

Fall 2013

Specification Mining: New Formalisms, Algorithms and Applications

Copyright 2013
by
Wenchao Li

Abstract

Specification Mining: New Formalisms,
Algorithms and Applications

by

Wenchao Li

Doctor of Philosophy in Engineering – Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Sanjit A. Seshia, Chair

Specification is the first and arguably the most important step for formal verification and correct-by-construction synthesis. These tasks require understanding precisely a design's intended behavior, and thus are only effective if the specification is created right. For example, much of the challenge in bug finding lies in finding the specification that mechanized tools can use to find bugs. It is extremely difficult to manually create a complete suite of good-quality formal specifications, especially given the enormous scale and complexity of designs today. Many real-world experiences indicate that poor or the lack of sufficient specifications can easily lead to misses of critical bugs, and in turn design re-spins and time-to-market slips.

This dissertation presents research that mitigates this manual and error-prone process through automation. The overarching theme is specification mining – the process of inferring likely specifications by observing a design's behaviors. We explore formalisms and algorithms to mine specifications from different sources, and demonstrate that the mined specifications are useful if not essential for a variety of applications such as verification, diagnosis and synthesis. The first part of the dissertation presents two approaches to mine specifications dynamically from simulation or execution traces. The first approach offers a simple but effective template-based remedy to the aforementioned problem. The second approach presents a novel formalism of specification mining based on the notion of sparse coding, which can learn latent structures in an unsupervised setting, and thus are not restricted by predefined templates. Additionally, we show that the mined specifications from both approaches can be used to localize bugs effectively.

In the second part of the dissertation, we study the problem of synthesis from temporal logic specifications. This synthesis approach offers an attractive proposition – one can automatically construct a functionally correct system from its behavioral description. The downside, however, is that it completely relies on the user to not only specify the intended behaviors of the system but also the assumptions on the environment. The latter is especially tricky in practice as environment assumptions are often implicit knowledge and seldom

documented. We propose a framework that learns assumptions from the counterstrategies of an unrealizable specification to systematically guide it towards realizability. We further show that, the proposed counterstrategy-guided assumption mining approach enables the automatic synthesis of a new class of semi-autonomous controllers, called human-in-the-loop (HuIL) controllers. A crucial component of such a controller is an advisory that determines when to switch control from the autonomous controller to the human operator. We formalize the criteria that characterize a HuIL controller, by taking into account of human factors such as response time, and describe how to construct the advisory using assumption mining.

Human inputs are still critical in specification. In the last part of this dissertation, we describe two efforts on broadening the scope of specification mining with creative use of human inputs. The first is the design of a crowdsourced specification mining game called CrowdMine. The main idea of CrowdMine is to transform a design's traces into images and leverage the human ability to recognize patterns in images to assist the process of mining specifications. The second effort examines the feasibility of converting natural language specifications to formal specifications, with a focus on how specification mining encapsulated in a natural language processing (NLP) layer may assist non-expert users of formal methods at the requirement stage of a design.

To my family.

Contents

Contents ii

List of Figures vi

List of Tables viii

1 Introduction 1
1.1 Thesis Statement . . . 3
1.2 Thesis Contributions . . . 4
1.2.1 New Formalisms . . . 4
1.2.2 New Algorithms . . . 5
1.2.3 New Applications . . . 6
1.2.4 Broadening the Scope of Specification Mining . . . 7
1.3 Related Work . . . 8
1.3.1 System . . . 8
1.3.2 Type of Specification . . . 9
1.3.3 Method . . . 11
1.3.4 Application . . . 12
1.4 Thesis Organization . . . 13

2 Preliminaries 15
2.1 Notation . . . 15
2.2 Automata . . . 16
2.2.1 Büchi Automaton . . . 16
2.2.2 Finite-State Transducers . . . 16
2.3 Linear Temporal Logic . . . 17
2.3.1 Syntax and Semantics . . . 17
2.3.2 Satisfiability and Realizability . . . 18
2.4 Specification Mining with Templates . . . 19

I Requirement Generation and Error Localization 21

3 Background 22
3.1 Formalism and Notations . . . 22
3.1.1 Transition System . . . 22
3.1.2 Traces and Subtraces . . . 22
3.2 Running Example . . . 23

4 Specification Mining for Digital Circuits 25
4.1 Overview . . . 25
4.2 Preliminaries . . . 26
4.2.1 Notations . . . 27
4.2.2 Specification Templates . . . 27
4.3 Mining Algorithm . . . 28
4.3.1 Mining from Delta Traces . . . 28
4.3.2 Merging Simple Specifications . . . 32
4.3.3 Specification Ranking . . . 34
4.4 Error Localization . . . 34
4.5 Experiments . . . 37
4.5.1 Benchmarks . . . 37
4.5.2 Results . . . 38
4.6 Summary and Discussion . . . 41

5 A Sparse Coding Framework for Specification Mining 43
5.1 Introduction . . . 43
5.2 Background . . . 45
5.2.1 Traces and Matrices . . . 45
5.2.2 Bipartite Graphs . . . 46
5.3 Specification Formalism – Basis Subtraces . . . 48
5.4 Algorithm: Sparsity-Constrained Biclique Cover . . . 50
5.4.1 Formulation as a Sparse Coding Problem . . . 50
5.4.2 Solving the Sparse Coding Problem . . . 51
5.4.3 Example Illustration . . . 53
5.5 Application to Error Localization . . . 54
5.5.1 Problem Definition . . . 54
5.5.2 Localization by Construction . . . 55
5.5.3 Example Illustration . . . 57
5.6 Results and Experiments . . . 58
5.6.1 Theoretical Guarantees . . . 58
5.6.2 Case Study . . . 60
5.7 Additional Related Work . . . 61
5.7.1 Boolean Matrix Factorization . . . 61

5.7.2 Bug Localization . . . 61
5.8 Summary . . . 62

6 Crowdsourced Specification Mining **63**
6.1 Introduction . . . 63
6.2 CrowdMine – Game Design . . . 65
6.2.1 CrowdMine1: An Open-Loop Design . . . 65
6.2.2 CrowdMine2: A Closed-Loop Design . . . 67
6.3 Discussion . . . 69

II Assumption Mining for LTL Synthesis **72**

7 Background **73**
7.1 Synthesis from GR(1) Specifications . . . 75
7.1.1 Generalized Reactivity (1) Specifications . . . 75
7.1.2 Games and Strategies . . . 75
7.1.3 Counterstrategy Graph . . . 77
7.2 Related Work . . . 78

8 Mining Environment Assumptions **80**
8.1 Solution Overview . . . 82
8.2 Version-Space Learning with Templates . . . 85
8.3 Experimental Results . . . 87
8.3.1 AMBA AHB Bus Protocol . . . 87
8.3.2 Generalized Buffer . . . 88
8.3.3 Robotic Vehicle Controller . . . 89
8.4 Summary . . . 90

9 Human-in-the-Loop Controller Synthesis **91**
9.1 Introduction . . . 91
9.2 Motivating Example . . . 94
9.3 Human-in-the-Loop Controller . . . 95
9.3.1 Agents as Automata . . . 95
9.3.2 Criteria for Human-in-the-loop Controllers . . . 96
9.4 Controller Synthesis . . . 97
9.4.1 Weighted Counterstrategy Graph . . . 98
9.4.2 Counterstrategy-Guided Synthesis of HuIL Controllers . . . 100
9.4.3 Switching from Human Operator to Auto-Controller . . . 102
9.5 Experimental Results . . . 103
9.5.1 Car-Following . . . 103
9.5.2 Gridworld Hallway . . . 105

9.6 Additional Related Work . . . 107
9.7 Summary . . . 108

10 Mining Assumptions from Natural Language Specifications 109
10.1 Related Work in NLP . . . 110
10.2 Natural Language to LTL Formula . . . 111
10.2.1 Preprocessor . . . 111
10.2.2 Stanford Type Dependency Parser (STDP) . . . 112
10.2.3 Semantic Processor . . . 113
10.2.4 Formula Generation . . . 115
10.3 Case Study . . . 117
10.4 Summary . . . 121

11 Conclusion and Future Work 122
11.1 Closing Remarks . . . 122
11.2 Future Work . . . 123
11.2.1 Combining Sparse Coding and Automata-Based Specification Mining 123
11.2.2 Compositional Analysis . . . 123
Assumption Mining for Verification . . . 124
Contract-Based Synthesis . . . 124
11.2.3 Improving Sparse Coding . . . 125
11.2.4 Other Application Domains . . . 125

Bibliography 126

List of Figures

1.1 Thesis overview . . . 4
1.2 Dimensions of specification mining. . . . 8

2.1 Monitor for $\mathbf{G}\ (a \rightarrow \mathbf{X}\ b)$. . . . 19

3.1 A two-port arbiter design. . . . 23
3.2 Sample trace of the arbiter design. . . . 23

4.1 High-level architecture of the proposed specification mining tool SAM. . . . 26
4.2 Equivalent delta trace of the trace shown in Figure 3.2. . . . 29
4.3 Monitor for the formula in Equation 3.1 over delta traces (propositions not shown on certain transitions are understood to be complementary such that the automaton is deterministic). . . . 30
4.4 Illustration of the procedure **createTable**. . . . 31
4.5 Iterative row and column updates of Tab over $\tau_i^\delta = \{e_j^\delta, e_{j+1}^\delta\}$. . . . 32
4.6 Specification mining-based bug localization. . . . 35
4.7 CMP router comprising four high-level components. . . . 38
4.8 Example mined specification from the CMP router. . . . 39
4.9 State machine in the "vcstate" module. . . . 39

5.1 Bipartite graph G_D. . . . 46
5.2 Biclique edge cover Cov. . . . 47
5.3 A subtrace as superimposition of two basis subtraces. . . . 50
5.4 A normal trace of a 2-port round-robin arbiter . . . 53
5.5 Three basis subtraces computed via sparse coding. . . . 54
5.6 Subtrace correctness characterization using basis subtraces. . . . 55
5.7 Bit flip on g_1 at cycle 97. . . . 57
5.8 Error subtrace as identified. . . . 58
5.9 Two error explanations. . . . 58

6.1 Overview of CrowdMine1. . . . 66
6.2 Top ranked patterns obtained in CrowdMine1. . . . 67
6.3 Overview of CrowdMine2 . . . 68

6.4 Finding contradicting subtraces in a counterexample. 69

8.1 Counterstrategy-guided synthesis enabled by assumption mining (the highlighted portions are our contributions). 82
8.2 Counterstrategy graph G^c for unrealizable specification ψ. 84
8.3 Diagrammatic representation of a version space from the most general hypothesis to the most specific hypothesis. 86

9.1 Human-in-the-Loop Controller: Component Overview and Synthesis from Specification 94
9.2 Controller Synthesis – Car A Following Car B 94
9.3 Condensed graph $\hat{G}^c$ for G^c (Figure 8.2) after contracting all SCCs. 99
9.4 Gridworld hallway example. 105
9.5 Illustration of ϕ: arrows indicate all possible movements of car B from the current position to the next position. 107

10.1 Non-expert user uses formal methods to analyze problems in a design document, facilitated by a NLP layer. 110
10.2 Workflow: NL $\rightarrow$ LTL $\rightarrow$ formal analysis. 112
10.3 Dependencies generated using STDP. 113
10.4 Predicate graph after the application of type rules. 115
10.5 Transition of Regulator_Mode based on Requirement 13 and 15. 121

List of Tables

2.1 Mathematical Notations 15
2.2 Abbreviations 16
2.3 Semantics of LTL 18

4.1 Performance results on generation of likely specifications. 38
4.2 Bug localization results on the CMP router. 41

10.1 Formula Translation Rules 116
10.2 Isolette requirements in English 118

Acknowledgments

This dissertation would not have been possible without the help of many people over the years. First and foremost, I would like to thank my advisor, Sanjit A. Seshia, for his guidance and support. His contagious enthusiasm, intense motivation and determination, and seemingly limitless patience, have set an example that I can only hope to imitate. I am truly grateful to have had the opportunity to work and learn under his mentorship.

I would also like to thank Prof. Andreas Kuehlmann and Prof. Francesco Borrelli for reviewing my dissertation and giving me invaluable feedback. I would also like thank Prof. Rastislav Bodík for chairing my qualifying exam committee. They have given me much critical and constructive feedback that helped shape this thesis.

I am also thankful to my internship mentors at Microsoft Research, Redmond and SRI International, Menlo Park: Dr. Alessandro Forin, Dr. Natarajan Shankar and Dr. Shalini Ghosh. Some of the ideas on specification mining were first developed while I was working as an intern with Dr. Alessandro Forin during the summer of 2008, in Microsoft Research, Redmond. He is always very resourceful and has given me much help and advice. I am thankful to Dr. Natarajan Shankar, Dr. Shalini Ghosh and other amazing researchers at SRI International for the intriguing conversations, insightful feedback, and overall the fun and productive times at Menlo Park. Dr. Shalini Ghosh really spearheaded the work on converting natural language requirements to their formal representations. Her inputs were critical to the development of the work on connecting natural language processing to specification mining. I would also like to thank Dr. Natarajan Shankar for his continual encouragement and guidance. His expertise deeply impresses me, and if anyone asks about my role models, Shankar would definitely be one of them. I am also grateful to my other collaborators: Lili Dworkin, Dorsa Sadigh, Daniel Elenius, Prof. Somesh Jha and Prof. S. Shankar Sastry. Some of the work done in this dissertation could not have been possible without their assistance and guidance. I would also like to thank Prof. Orna Kupferman who visits Berkeley regularly in the summers, during which we have had many fruitful discussions. Her depth of knowledge and rigor in approach exemplify the meaning of a world-class researcher. She is, without a doubt, a constant source of my inspiration.

During my tenure as a graduate student at Berkeley, I also had the opportunity to work alongside some of the most talented and creative people colleagues and friends. The list include, but not limited to: Daniel Holcomb, Bryan Brady, Susmit Jha, Jia Zou, Chenjie Gu, Hao Zhang, Qi Zhu, Yang Yang, Haibo Zeng, Wei Zheng, Qiliang Xu, Yenhao Chen, Liangpeng Guo, Chung-Wei Lin, Alberto Puggelli, Pierluigi Nuzzo, Mehdi Maasoumy, Xuening Sun, Tobias Welp, Baruch Sterin, Sayak Ray, Aadithya Karthik, Prateek Bhanshali, Luigi Di Guglielmo, Alexandre Donze, Rüdiger Ehlers, Indranil Saha, Rohit Sinha, Dorsa Sadigh, Jonathan Kotker, Zach Wasson, Wei Yang Tan, Garvit Juniwal, Nishant Totla, Ankush Desai and Daniel Fremont.

Last but not least, I would like to thank my family, for their unadulterated and unconditional love and support. My parents are always supportive of my choices, and many times at their own sacrifices. Without them, I would not have been able to achieve my goals. My

wife, Nuo, has worked alongside with me also as a Ph.D. student in Electrical Engineering and Computer Sciences at UC Berkeley. She is the most loving and caring person I know. Her wonderful smile brightens my every day at Berkeley.

Chapter 1

Introduction

We live in a world of ubiquitous computing. From home appliances, to smart phones, to cars, we constantly engage and interact with computing devices. It has been projected that there will be over 1,000 embedded devices per person by 2015 [Joh08]. However, a problem that arises from ubiquitous computing is the potential proliferation of bugs, where even tiny bugs can result in large-scale failures, e.g., a single bit error brought down all Amazon S3 servers in July 20, 2008 [Tea08]. The stakes are even higher for safety-critical systems, such as pacemakers, auto-pilot systems and automotive control systems, where a failure may result not only in financial losses but also loss of human lives.

The proliferation of computing devices, shrinking of semiconductor fabrication process nodes, as well as increase in design complexity are making the already challenging problem of providing assurance for these devices an even more daunting task. In fact, the dominant problem faced by engineers today lies not in creating *new* designs, but in creating them *correctly*. *Verification* and *correct-by-construction synthesis*[1] are two main approaches for providing such assurances. In [Ses12], the author has identified *the lack of good-quality specifications* as one of the major problems that still plagues these two fields. We elaborate on this below.

Verification:

Design and verification engineers now have an arsenal of tools that they can apply to verify computing systems. However, despite decades of efforts, especially in automation, the cost of verification is steadily rising. It has been estimated that verification and validation makes up about 60-70 percent of the total development cost of a processor or a system-on-a-chip (SoC) design [BAM03]. According to International Business Strategies, the cost of verification and validation will soon completely dwarf the rest of the overall design cost as technology scales

[1] The term "synthesis" here refers to the synthesis of formal artifacts from high-level specifications, such as those given in temporal logic, as opposed to "logic synthesis" (turning a register transfer level (RTL) description into its gate-level implementation), or "high-level synthesis" (e.g., create a RTL description from a C or SystemC description), which are more commonly used in the electronic design automation (EDA) community.

beyond 45 nm [San].

The first and a fundamental step of every verification process is *specification*. Specification at large is the process of determining the requirements of the target design and *formalizing* them mathematically. It is important to note that the effectiveness of many mechanized procedures, such as assertion-based verification [Sto02] and model checking [CGP00], relies heavily on an engineer being able to create correct and complete specifications of a design.

Consider the classic example of an elevator controller design. The first and foremost step is to determine and define the intended behaviors of the elevator. These behaviors are supposed to capture *precisely* how the elevator should operate. For instance, the door should remain closed *until* the elevator reaches the target floor. Temporal logic, such as Linear Temporal Logic (LTL) [Pnu77] which has been assimilated into industry standards (IEEE 1850 Standard for Property Specification Language) [EF06], can be used to *formalize* this requirement, as shown below.

$$\psi = \mathbf{G}\ \big(\text{elevator_moves} \rightarrow (\text{door_closed}\ \mathbf{U}\ \text{reaches_target_floor})\big) \tag{1.1}$$

We review the syntax and semantics of LTL in Section 2.3. In general, this kind of property, which describes temporal behaviors of a system, is commonly known as *temporal property*. These properties, as opposed to *single-state assertions*, which describe constraints at a single state of a circuit or program, are intended to capture the evolution and relationship among behaviors of multiple states in time. Therefore, they are typically more tricky to write and formalize correctly. According to a report from IBM Haifa, roughly 30% of the formulas were created incorrectly during the first formal verification runs of a new hardware design [BBDER97]. In this dissertation, we propose *mining* likely formal specifications from traces of a design as a way to mitigate this problem. The mined specifications allow an engineer to better understand the design, verify its correctness, and manage possible evolutional changes. We explore novel formalisms of specification and present a suite of algorithms for mining meaningful specifications from traces.

The practical effectiveness of verification, especially formal verification, lies much in its ability to find bugs. Hence, much of the challenge in finding bugs in fact lies in *finding the specification* that mechanized tools can use to find bugs. Mined specifications, which are only behaviors exhibited in traces (of a possibly buggy design), cannot be used directly as targets for further (formal) verification until they are validated by a human engineer. In this dissertation, we present a novel bug localization technique that leverages mined specifications but is not limited by this restriction.

Synthesis:

The majority of a design process is in fact expensive iterations of verification and debug. Wouldn't it be nice if we can start with the requirements and then have a way to automatically generate a design that satisfies these requirements? Temporal logic synthesis [PR89], the proposition of automatically synthesizing an implementation from its temporal logic specification, offers an attractive solution to this problem. However, it faces the same problem as in verification – its complete reliance on having good-quality specifications.

In this dissertation, we demonstrate that specification mining can also be valuable in addressing the specification challenge in temporal logic synthesis. Specifically, we focus on the problem of missing environment assumptions, which is a common cause of a specification being unrealizable (see Section 2.3.2 for a precise definition of unrealizability) due to the difficulty of capturing the environmental model (e.g., as a result of implicit knowledge, third-party IP and simply poor documentation), and propose a novel algorithm based on analyzing how the environment should not behave in order for the system to satisfy its specification. Our work is motivated, in part, by the recent advance in temporal logic synthesis, which does not only result in more efficient algorithms [PPS06b], but also demonstrates its applicability to a spectrum of domains, ranging from digital circuits [BGJ+07a, BGJ+07b], to aircraft electric power system [XTM12], to mission planning for autonomous robots [KGFP07, Won10], to industrial automation [CGR+12].

1.1 Thesis Statement

In this dissertation, we explore the following thesis:

Temporal specifications can be mined systematically from example behaviors of a design and its environment, and can be used to automate tedious tasks in verification and synthesis such as bug localization and finding missing assumptions.

This thesis takes the position that the manual and error-prone process of writing formal specifications can be mitigated by specification mining – an automatic procedure that infers likely specifications by analyzing a design's behaviors. We show that specification mining is not only useful for requirement generation (i.e. finding candidate formal specifications that a designer may miss), but also applicable to a variety of other applications. Figure 1.1 illustrates the composition of this thesis.

The central theme of the thesis is specification mining. We present a collection of novel formalisms and techniques that are tailored for a variety of different applications. This thesis is the first to propose the use of mined specifications for bug localization in reactive systems, such as digital circuits. We study two topics at large – verification and synthesis. In verification, we show that relevant and meaningful specifications can be mined dynamically by observing traces of a digital design, and are useful also for bug localization. In synthesis, we study the topic of synthesis from temporal logic specifications and focus on the problem of finding missing environment assumptions. We show that, by analyzing the counterstrategy of an unrealizable specification, one can systematically generate candidate environment assumptions that guides the specification towards realizability. In addition, this *counterstrategy-guided synthesis* approach enables the automatic construction of a new class of semi-autonomous controller, called human-in-the-loop (HuIL) controllers. Lastly, this thesis makes an effort to broaden the scope of specification mining, by proposing creative uses of human computer interaction (HCI). Specifically, we present a crowdsourced specification

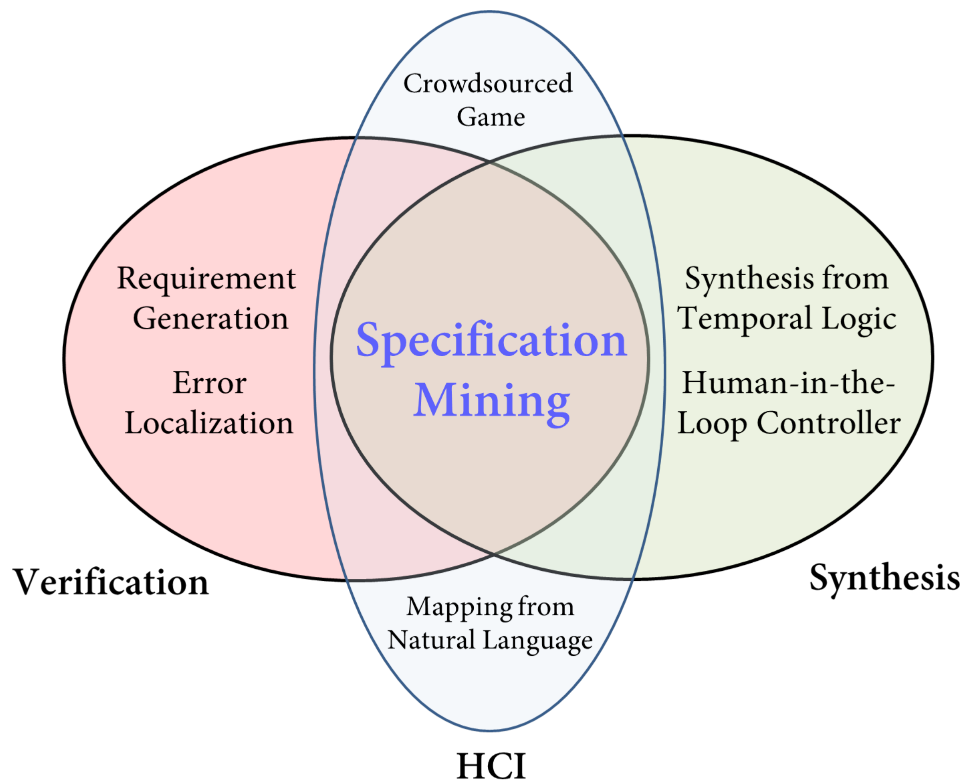


Figure 1.1: Thesis overview

mining game called CrowdMine that can be played even by young children but effectively infers likely properties behind the scene. Our second effort in this direction is motivated by the fact that much documentation is still written in natural language (NL) today. We examine the feasibility of automatically deriving formal specifications from NL sentences, and demonstrate that this process can be tightly integrated with our assumption mining framework to assist non-expert users of formal methods to debug design requirements.

1.2 Thesis Contributions

The contributions of this thesis are four-fold.

1.2.1 New Formalisms

There has been extensive work on specification formalisms. These formalisms can be generally put into two categories: (1) automata-based [Muk96, Rus12] or (2) logic based (e.g., CTL [CES86], LTL [Pnu77], PSL [EF06]), and often with a tight connection between members of the two categories (e.g., two formalisms describe the same formal language).

In this thesis, we explore a different direction, inspired by the notion of a sparse code [OF97]. In comparison, in the automata-theoretic view of specification, an automaton (or an equivalent logic formula) characterizes a set of traces with the condition that it accepts each trace in this set (or each trace satisfies the logic formula). In our sparse-coding formalism, we offer a "linear algebra" view of specification. Instead of using an automaton, we propose to use a basis to characterize a set of traces. Similar to the notion of a vector space, we consider a finite-length subtrace (see the definition of a subtrace in Section 3.1.2) as a k dimensional Boolean vector, where k is the number of Boolean signals present in the trace. Hence, any finite trace can be viewed as a collection of such subtraces of the same dimension, and we can characterize a design in terms of the set of subtraces it can produce. With this observation, we propose the use of "basis subtraces" as a specification formalism. These basis subtraces, just like basis vectors in linear algebra, *span* a k-dimensional subspace ($\subseteq \mathbb{B}^k$) including all possible k-dimensional behaviors that can be exhibited from the design under analysis. Particularly, we use a linear combination analog in the Boolean domain, where multiplication is interpreted as the Boolean conjunction and addition is interpreted as the Boolean disjunction. In Chapter 5, we elaborate on this formalism in detail, and show the specification mining problem is equivalent to computing a basis that spans the observed traces.

1.2.2 New Algorithms

In a dynamic specification mining framework, where specifications are mined from observations of the design under analysis, the problem of specification mining is closely related to the problem of runtime verification. In runtime verification, traces of a design are tested against one or more patterns usually in the form of monitors. In specification mining, traces are examined to deriving a set of patterns that the traces satisfy[2], which presumably also capture some aspects of the underlying design.

By prescribing to a set of templates, which determine the form these patterns take, one can construct *instantiations* of these templates so that they can be checked against the traces. Many existing algorithms follow this strategy (see [LKLH11] for a compilation of techniques). In Chapter 4, we also use this strategy but adapt it in a way that it is efficient for digital circuits. Specifically, two main challenges for porting this strategy to circuits are addressed by the new algorithm – (1) many concurrent signals in a circuit; (2) traces can easily be millions or even billions cycles long.

Using predefined templates can be restrictive. In Chapter 5, we leverage the new sparse coding-based specification formalism, which does not use hard-coded templates and only parameterize a specification by the length of a subtrace, and describe a graph-theoretic algorithm for computing a basis from traces. The key novelty of this algorithm is a connection from computing a basis from Boolean traces to finding a biclique edge cover for a bipartite graph. We describe this algorithm in detail in Chapter 5.

[2] The term "satisfy" is used a in loose sense here.

In Part II of the thesis, we present the first counterstrategy-guided assumption mining algorithm that systematically guides an unrealizable specification towards realizability. Similar to counterexample-guided abstraction refinement in verification, our algorithm iteratively refines the environment model by adding an assumption that eliminates certain moves by the environment that contribute to the specification being unrealizable. This technique also extends the existing methods for finding specifications, which often focus on desired behaviors exhibited in traces of a design in traces – (1) it derives assumptions from *counteracting* behaviors that prevents the target design from fulfilling certain objectives; (2) it examines a new form of evidence – counterstrategies (represented as a graph) as opposed to traces. In Chapter 9 of the thesis, we further show that, by adapting this algorithm suitably, we can synthesize a new class of semi-autonomous controllers that incorporates constraints representing the interaction with a human operator.

1.2.3 New Applications

Specification mining has been used to address a number of challenges in verification, such as assisting in program understanding [CJK07], bug finding [ECH+01, WN05], bug classification [LCH+09] and meeting coverage goals [LSTV12]. In this thesis, we expand this envelope further and demonstrate novel applications of specification mining both in verification and synthesis. We list these applications below.

Bug Localization in Reactive Systems, such as Digital Circuits:

A mined specification, if validated that it matches design intent, can be used as a target for verification. However, not all mined specifications are real specifications, since their validity is only supported by the given evidence (e.g., traces). Thus, we cannot use them directly in verification or to do find bugs until they are validated by a third party (e.g, a human designer).

In this thesis, we make the observation that, by finding likely temporal properties that certain signals hold in different locations and at different time points of traces of a circuit, we can effectively *localize* bugs *both in space and in time* given correct and erroneous traces of the circuit[3]. An additional advantage of this approach is that we need not even care about the problem that the correct traces may also be produced by the same buggy design. In Chapter 4, we demonstrate that this approach enables effective bug localization for a variety of bugs in hardware, including transient bugs.

Temporal Logic Synthesis:

Temporal logic synthesis is a correct-by-construction proposition that an implementation can be generated automatically from its temporal logic specifications. It relies not only on the system guarantees to be specified correctly but also on the environment assumptions, which are the trickier parts of the specifications, to be captured correctly and completely. A missing

[3] We assume correctness can be determined by some end-to-end mechanism, such as whether an application running on a processor crashes.

assumption often leads to the specification being *unrealizable* – no implementation can exist to satisfy the specification. As a result, the usefulness of this approach is undermined.

In Chapter 8, we re-purpose specification mining to address this problem of missing assumptions. Using the counterstrategy-guided assumption mining approach mentioned above, we show that meaningful candidate assumptions can be produced in a systematic way to guide the specification towards realizability.

Synthesis of Human-in-the-Loop Controllers:
Our contribution in this part of the thesis is two-fold. First, we formalize a new class of semi-autonomous controllers, called human-in-the-loop (HuIL) controllers, as a composition of an autonomous controller, a human operator, and an advisory controller that acts as a switching mechanism between these two components. Instances of these controllers, although usually recognized and subsequently designed as separate entities, are pervasive: a pilot constantly needs to interact with the auto-pilot system; a driver still needs to be behind the wheel when cruise control is turned on. The correctness of these systems depends not only on the correctness of the autonomous controller and that of the actions of the human operator, but also on the proper interaction between them.

Motivated, in part, by the recent work on synthesizing robotic controllers from temporal logic specifications [KGFP07, Won10], we apply this approach to the automatic synthesis of HuIL controllers. In Chapter 9, we demonstrate the effectiveness of this approach with an application on examples motivated by driver-assistance systems.

1.2.4 Broadening the Scope of Specification Mining

Most existing literature today considers specification mining as a mechanized procedure. In this thesis, we expand this view by first identifying parts of the process that can benefit from human inputs and then building mechanisms that can use these inputs in creative ways. Two novel approaches are listed below.

Crowdsourcing and Gamification:
As indicated in Section 1.3, many existing specification mining techniques rely on the use of templates. The process of construction of these templates usually requires expert insight and can still easily miss some aspects of the design. In Chapter 6, we explore the use of human computation to assist the process of specification mining. We present a game called CrowdMine, which "democratize" specification mining by transforming specific instances of it to a game. In particular, the gamification leverages human's ability to recognize patterns in images by turning traces into images. We have designed and deployed prototypes of the game. We discuss our findings in detail in Chapter 6.

Analyzing Specifications in Natural Language:
Formalizing specifications requires thorough understanding the underlying logic, which is no easy feat even for experienced engineers. In Chapter 10, we propose to bridge the gap between the common practice of writing specifications in natural language and the formalization of these specifications which is necessary for verification, with judicious use of natural language processing techniques. By encapsulating the proposed assumption mining technique inside

a natural language processing (NLP) layer, this requirement analysis is made applicable in a much larger context. Experimental results on analyzing a portion of the "Requirement Engineering Management Handbook" released by the Federal Aviation Administration [LM09] demonstrate the utility of the proposed technique.

1.3 Related Work

Specification mining is an active area of research and much work has been done related to this topic. The term was coined by Ammons et al. in [ABL02] but the study of automatically generating specifications goes back as early as 1974 [Weg74, Cap75]. In this section, we survey related approaches in this area. Specifically, we aim to distill the commonality and differences among the rich body of work on specification mining by categorizing them along four dimensions – system, type of specification, method and application, as shown in Figure 1.2.

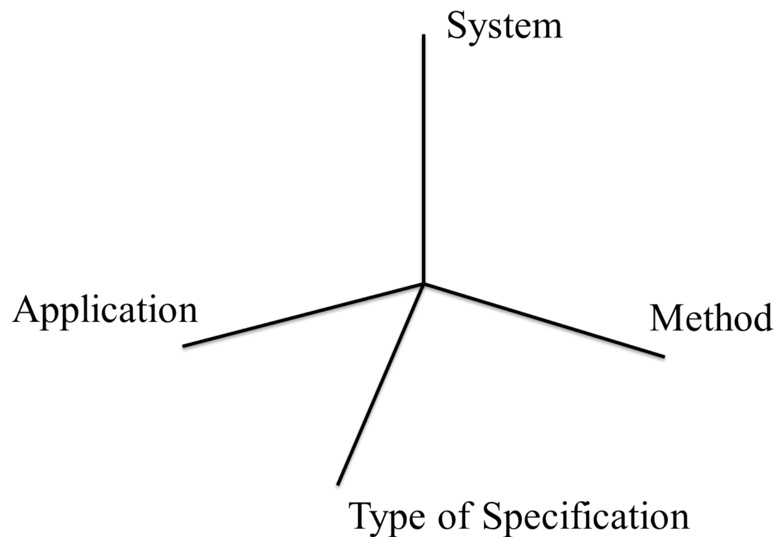


Figure 1.2: Dimensions of specification mining.

These dimensions are not necessarily orthogonal to one another. For example, the type of specification we are looking for can significantly influences the method for finding these specifications. We elaborate on each of the four dimensions below and highlight the novelties of our approaches compared to related work.

1.3.1 System

System concerns with the question of *what the specification is about*. Related work can be largely put into two domains – circuits and programs.

Circuit:

Various circuit-specific mining techniques have been proposed for hardware-specific properties. The IODINE tool [HNCC05] mines simple likely invariants such as one-hot encodings or fixed-delay pairs. Fey and Drechsler [FD04] present an approach to mine repeated patterns where patterns are valuations of signals at various time steps (e.g. $s_t = 1 \wedge s_{t+1} = 0$). While their approach is general, the timing requirement can be too strict for complex interactions and it deals with only a small set of signals over a predefined interval each round. The Dianosis [RKF+08] tool mines properties based on checkers defined in the Open Verilog Library (OVL) [Org07]. Isaksen and Bertacco [IB06] propose to generate transaction diagrams from a trace for analyzing protocols. Vasudevan et al. [VSP+10] propose to mine simple implications over the signals in a small window, supported by statistical measures. In comparison, our approach mines a class of general temporal properties, possibly spanning a large number of cycles (e.g., thousands) for digital circuits in a scalable manner and, to our knowledge, is the first to effectively use these mined properties for bug localization.

Program:

There is a rich body of work on mining specifications in programs. We highlight some of the work below and refer interested readers to [LKLH11] for a more thorough survey and categorization of specification mining along this dimension.

Started about a decade ago, there has also been a surge in software engineering to adopt machine learning and data mining techniques to reverse engineer or mine formal specifications [LKLH11]. Daikon [EPG+07] is one of the earliest tools that mine single-state invariants or pre-/post-conditions in programs. Alur et al. [AvMN05] consider the problem of synthesizing interface specifications for Java classes. Ammons et al. [ABL02] propose to mine state machines that encode legal partial orders of API calls with data-flow constraints on their arguments. Lorenzoli et al. [LMP08] propose to extract component interaction models in the form of extended finite state machines by using a technique called GK-tail. In this dissertation, we focus instead on mining specifications in circuits, which is a domain less studied, and propose a portfolio of techniques tailored to this domain.

1.3.2 Type of Specification

Formal specification is a way of describing the behavior of a design precisely, typically supported by a formal language. While many formalisms of specifications are general (e.g, LTL), the choice of the types of specifications to mine is heavily influenced by the domain insight and the target application. We highlight some of these approaches below.

Automata:

This class of techniques aims to learn a single complex specification, usually as a finite automaton, over a specific alphabet. The finite automaton is either treated as the specification, or is used as the substrate for extracting simpler properties afterwards. For instance, Ammons et al. [ABL02] first produce a probabilistic automaton that accepts the trace and then extract from it likely properties. Similarly, the work by Isaksen and Bertacco [IB06] on generating transaction diagrams described previously also falls into this category. However,

this kind of approaches can often be computationally expensive. In fact, learning a single finite state machine from traces is NP-hard [Gol78]. Additionally, the specification can be as large as the implementation, and thus can be difficult for a human to understand. In this dissertation, we focus on mining multiple simple properties, by making structural assumptions on the form of these properties appropriate for the target domains. We describe related work of this approach below.

Temporal Properties:

To achieve better scalability, an alternative is to learn multiple small specifications, each describing a specific aspect of the design. *Temporal properties*, which are the type of specification we focus in this thesis, are examples are these small specifications Engler et al. [ECH+01] first introduce the idea of mining simple alternating patterns. Several subsequent efforts [GS08b, WN05, YEB+06, GS08a] built upon this work. For example, Perracotta [YEB+06] and Javert [GS08a] locate instances of the alternating pattern $(a\,b)^*$ and a resource usage pattern $(a\,b^*\,c)^*$. In addition, Javert composes these patterns into larger ones by using a set of inference rules. Part of the work in this thesis, which we will describe in detail in Chapter 4, is inspired by ideas in Perracotta and Javert. Additionally, we focus on patterns that are meaningful for digital circuits and also provide a merging procedure to compose patterns in time. We also note that there is extensive related work in the data mining community dating back to the work by Agrawal and Srikant on mining sequential patterns [AS95]. The tool PR-Miner, developed by Li and Zhou [LZ05], builds on a data mining technique called frequent itemset mining to extract implicit programming rules from executions of software code. The closest to our approach presented in Chapter 4 is perhaps the work by Lo et al. [LKL08] on mining past-time temporal rules from execution traces. The main difference is we employ an automaton-monitoring approach with optimizations on hardware traces and they use statistical measures such as support and confidence. We again point interested readers to [LKLH11] for a more detailed survey in this area.

Single-State Invariants:

Value-based invariants are constraints over some variables in a program or a circuit. For example, Daikon [EPG+07] mines invariants such as "y = 2x + 3" where "x" and "y" are variables of a program. Another example is the IODINE tool described above, which simple invariants such as one-hot encodings. In this dissertation, we focus on mining temporal properties, which are often the more tricky properties to write and formalize. However, we note that some of the ideas that leverage single-state invariants are applicable to our problem. An example of this is the work done by Liblit et al. [LAZJ03], which uses predicates for bug isolation in programs but does not infer the predicates. Our work on bug localization in hardware is inspired by this idea. We additionally make the observation and demonstrate with experimental results that temporal properties are particularly useful addressing this problem in hardware.

Sequence Diagrams:

Sequence charts, such as message sequence charts (MSCs) [AEY00] and their extensions such as Harel's live sequence charts (LSCs) [DH01], are especially useful for describing behavioral scenarios involving interworking of processes and objects. Uchitel et al. [UKM01]

propose to an algorithm that builds a labeled transition system behavior model based on MSCs. Prior work by Lo et al. [LMK07, LM08] uses dynamic analysis on traces to infer LSCs for software programs. We further note that, the work of Merlin by Livshits et al. [LNRB09], aims at inferring explicit information flow specifications from program code by using probabilistic inference techniques.

1.3.3 Method

This section does not attempt to be comprehensive but covers a representative sample of work that is relevant to this thesis.

Dynamic:

The underlying assumption of this approach is that the design under analysis is mostly correct. Thus, one can *mine* likely specifications by observing simulation or execution traces of the design. A majority of the specification mining techniques mentioned above can be categorized as dynamic. DAIKON [EPG+07], for instance, essentially reports properties that are true over sample runs of a program. A common philosophy shared amongst these techniques is that frequently occurring patterns are likely specifications. Thus, the idea is to generalize observations from a set of traces to all possible traces of a design. Hence, the representativeness of the observed traces (e.g., coverage of test vectors) inherently affect the performance of these techniques. We show in Chapter 5 that the theoretical guarantee offered by our sparse-coding approach for bug localization is also closely related to this criterion.

In hardware, Goldmine, a system developed by Vasudevan et al.[VSP+10], uses decision-tree based learning and considers input in the form of sequence of signals over a small bounded window. Our work on specification mining for digital circuits, using temporal property templates and sparse coding, also fall under this category.

Static:

Specifications can also be generated by reasoning about the program statically. For example, Engler et al. [ECH+01] use static analyses on source code of a program to directly infer a set of rules that the program should obey. Alur et al. [AvMN05] propose the use of predicate abstraction together with automata learning to automatically synthesize interface specifications for Java classes. Ramanathan et al. [RGJ07] mine precedence rules from program source code of the form: "Whenever an event occurs, previously, another series of events has happened." One drawback of static approaches is that it has to deal with infeasible paths. In programs, another issue of such an approach is that it needs to handle pointers and references. We note that, additionally, static analysis is particularly challenging for hardware designs, because it is difficult to infer causal dependencies between events across multiple cycles from the structure of a Register Transfer Level (RTL) description.

Static and dynamic analyses complement each other. We refer the readers to [Ern03] for a detailed comparison of the two techniques. Static and dynamic information can also be combined to generate better quality specifications. GoldMine, for example, uses cone-of-influence information from a RTL description of the design to improve variable selection [HSV13].

1.3.4 Application

Design Understanding:

Specification mining, sometimes also known as specification discovery, often aims at discovering behavioral patterns that are either unknown to the user or missed during a design and verification cycle. The lack of (formal) specifications (and poor documentation), a prominent problem in both hardware and software engineering, can be very costly due to the difficulty ensued on maintaining these systems [Erl00]. Hence, specification mining is often used as a tool to aid program understanding. Our work on assumption mining addresses a similar challenge, albeit for a different application.

From another perspective, it is also useful to reverse engineer the formal description of designs and protocols in a number of security-related contexts. For example, Prospex [CWKK09] is a tool that combines message clustering with state machine inference to reverse engineer network protocols. Other related work include the use of specification mining for understanding malwares [CJK07], and model inference for vulnerability discovery [CBP+11].

Verification:

A mined specification, if validated that it matches design intent, can then be used as a target for further verification. It can either be converted to a monitor and used in assertion-based verification or used directly in model checking. Much of the results of specification mining can be naturally applied in this setting. In this dissertation, we choose to focus on small temporal properties, motivated also by the presence of many tools that support the verification of these properties. Additionally, a novelty of this dissertation is the use of natural language processing techniques to further expand the scope of specification mining to verify natural language requirements (see Chapter 10).

Synthesis:

Our work on assumption mining, to our knowledge, is the first application of specification mining to the problem of temporal logic synthesis. Chatterjee et al. [CHJ08] consider also the problem of unrealizable LTL specification. Their approach aims at constructing an environment model that is as weak as possible, by analyzing the game graph constructed during the synthesis process. However, the environment model generated is a single Büchi automaton, which can have a large state space and thus can be difficult for a human user to inspect. Our work, on the other hand, provides a simpler but practical approach by using a template-based mining approach. The novelty of our approach lies in the use of counterstrategies, such that we can produce candidate assumptions iteratively for easy human validation. Subsequent work by Alur et al. [AMT13] uses a similar approach but consider direct counterstrategy graph exploration.

Classification:

An interesting application of specification mining is to consider the mined specifications as features in a classification framework. Lo et al. [LCH+09], for instance, propose the idea of mining discriminative patterns to help classify software failures. Iterative patterns are first mined from different failure traces in a supervised setting to produce a selected set of patterns that maximizes the Fisher's linear discriminant. New failure traces can then be

classified accordingly using this set of patterns. Our work on bug localization is similar to theirs in spirit. Instead of using mined patterns in a classification setting, we use the time and location information of distinguishing patterns between correct and error traces to localize bugs in hardware.

1.4 Thesis Organization

We first introduce preliminary material in Chapter 2. The rest of the dissertation is then divided into two parts. In Part I of the thesis, we explore the use of specification mining for generating requirements for verification as well as localizing tricky bugs in digital designs. Chapter 3 gives additional background materials on mining temporal properties from traces along with a simple illustrative example. In Chapter 4, we present the techniques for efficiently mining temporal properties from hardware traces and using the mined properties for bug localization, supported by experimental results on several benchmarks. This is based on joint work with Alessandro Forin and Sanjit A. Seshia [LFS10]. In Chapter 5, we present the sparse-coding based formalism of specification, and describe formulations and algorithms for extracting a sparse basis from traces as well as using it to perform bug localization. We also present experimental results to demonstrate the effectiveness of this approach. This is based on joint work with Sanjit A. Seshia [LS12]; in particular, the idea of viewing a trace as a sequence of images is due to my co-author Seshia. In Chapter 6, we present designs of the crowdsourced specification game, CrowdMine, and discuss directions in creatively leveraging human computation in verification. This is based on joint work with Somesh Jha and Sanjit A. Seshia [LSJ12].

In Part II of the thesis, we study the topic of synthesis from temporal logic specifications and focus on the problem of finding missing environment assumptions. Chapter 7 reviews background on LTL synthesis with a focus on GR(1) specifications and contrasts our work with existing techniques for debugging LTL specifications in the context of synthesis. In Chapter 8, we present the counterstrategy-guided assumption mining approach and validate its usefulness on case studies drawn from existing literature. This is based on joint work with Lili Dworkin and Sanjit A. Seshia [LDS11]. In Chapter 9, we present the formalization and characterization of a human-in-the-loop controller and describe an adaption of the assumption mining technique for automatically synthesizing such controllers. This is based on joint work with Dorsa Sadigh, S. Shankar Sastry and Sanjit A. Seshia [LSSS14]. In Chapter 10, we present our efforts on using natural language processing techniques to expand the applicability of assumption mining to natural language requirements. We further support the proposed approach with a case study on a portion of a publicly available document released by the Federal Aviation Administration. This is based on joint work with Shalini Ghosh, Daniel Elenius, Natarajan Shankar, Patrick Lincoln and Wilfried Steiner [GEL+13]. In particular, this work was done as part of the ARSENAL project, lead by Ghosh, with Elenius as the main software architect, and the idea of using type rules to formalize a natural language sentence is due to Ghosh. Finally, we summarize the contributions of this dissertation and

identify future directions in Chapter 11.

This research was supported in part by the Gigascale Systems Research Center and the Multiscale Systems Center, two of six research centers funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation entity, the TerraSwarm Research Center, one of six centers supported by the STARnet phase of the Focus Center Research Program (FCRP), a Semiconductor Research Corporation program sponsored by MARCO and DARPA, a Hellman Family Faculty Fund Award, an Alfred P. Sloan Research Fellowship, NSF grant CNS-0644436, NSF grant CCF-1116993, and NSF grant CCF-1139138.

Chapter 2

Preliminaries

In this chapter, we describe the necessary theoretical background for most of the work in this dissertation. We begin by introducing the notations and abbreviations in Section 2.1. In Section 2.2, we review materials on Büchi automaton and finite state transducers. Afterwards, we describe the syntax and semantics of Linear Temporal Logic (LTL) in Section 2.3. Finally, in Section 2.4, we introduce the problem of mining specifications from templates.

2.1 Notation

We use the following notations and abbreviations throughout the dissertation.

Table 2.1: Mathematical Notations

Symbols	**Meaning**
$\mathbb{B}$	the Boolean field represented by the set $\{\texttt{true}, \texttt{false}\}$
$\mathbb{B}^N$	a Boolean vector of size N
$\mathbb{B}^{M \times N}$	a Boolean matrix of size M by N
X	a set of Boolean input variables
Y	a set of Boolean output variables
$\mathcal{X}$	the set of all possible assignments to X, i.e. 2^X (input alphabet)
$\mathcal{Y}$	the set of all possible assignments to Y, i.e. 2^Y (output alphabet)
$\bar{b}$	a literal which is the negation of the Boolean variable b
M^e	a Mealy transducer
M^o	a Moore transducer
$\mathcal{C}$	a sequential circuit
M^t	a discrete transition system
$\mathcal{L}$	language as a set of traces

Table 2.2: Abbreviations

Symbols	Meaning
DFA	Deterministic Finite Automaton
FST	Finite State Transducer
BMF	Boolean Matrix Factorization
LTL	Linear Temporal Logic
GR(1)	Generalized Reactivity (1)
HuIL	Human-In-the-Loop
SAM	Scalable Assertion Miner
IP	Intellectual Property
HCI	Human Computer Interaction
RTL	Register Transfer Level
AI	Artificial Intelligence
EDA	Electronic Design Automation
VCD	Value Change Dump

2.2 Automata

2.2.1 Büchi Automaton

Definition 2.1. *A deterministic Büchi word automaton (DBW) is a tuple* $\mathcal{A} = (Q^b, \Sigma, \rho^b, q_0^b, F^b)$, *where*

- Q^b *is a finite set of states,*
- Σ *is a finite alphabet,*
- $\rho^b : Q^b \times \Sigma \to Q^b$ *is a deterministic and complete transition function,*
- $q_0^b \in Q^b$ *is the initial state, and*
- $F^b \subseteq Q^b$ *is a set of accepting states.*

An infinite word $\pi \in \Sigma^\omega$ is an infinite sequence of symbols from Σ. Denote π_i as the ith symbol in π. Given a word π, a run of $\mathcal{A}$ is an infinite sequence of states $\tau = q_0, q_1, \ldots \in Q^{b^\omega}$ such that $q_{i+1} = T(q_i, \pi_i)$ for all $i \geq 0$. We use $\inf(\tau)$ to be the set of states that appear infinitely often in τ. τ is accepting if and only iff $\inf(\tau) \cap F^b \neq \emptyset$.

A *nondeterministic* Büchi automaton (NBW) is similar, except that ρ^b is a relation instead of a function, i.e. $\rho^b \subseteq Q^b \times \Sigma \times Q^b$.

2.2.2 Finite-State Transducers

Definition 2.2. *A finite-state Mealy transducer (machine) with input alphabet* $\mathcal{X}$ *and output alphabet* $\mathcal{Y}$ *is a tuple* $M^e = (Q^m, q_0^m, \mathcal{X}, \mathcal{Y}, \delta^m, \theta^m)$, *where*

- Q^m *is a finite set of states,*
- $q_0^m \in Q^m$ *is the initial state,*
- $\delta^m : Q^m \times \mathcal{X} \rightarrow Q^m$ *is a transition function, and*
- $\theta^m : Q^m \times \mathcal{X} \rightarrow \mathcal{Y}$ *is an output function.*

Given an input trace $\vec{x} = x_0, x_1, \ldots$, a run of M^e is the sequence $\vec{q} = q_0, q_1, \ldots$ of states such that $\vec{q}_0 = q_0$, and $q_{k+1} = \delta^m(q_k, x_k)$ for all $k \geq 0$. The run $\vec{q}$ on $\vec{x}$ produces the output trace $M^e(\vec{x}) = \theta^m(q_0, x_0), \theta^m(q_1, x_1), \ldots$. The language of M^e is then denoted by the set $\mathcal{L}(M^e) = \{(x, y)^\omega \mid M^e(\vec{x}) = \vec{y}\}$.

A *Moore* transducer M^o is similar, except that $\theta^m : Q^m \rightarrow \mathcal{Y}$ is the output function. Similarly, the run $\vec{q}$ on an input trace $\vec{x}$ produces the output trace $M^o(\vec{x}) = \theta^m(q_0), \theta^m(q_1), \ldots$. The language of M^o is thus $\mathcal{L}(M^o) = \{(x, y)^\omega \mid M^o(\vec{x}) = \vec{y}\}$. With a slight abuse of notations, we use M to denote a transducer when it is clear from the context whether it refers to a Moore transducer or a Mealy transducer.

A sequential circuit $\mathcal{C}$ with Boolean input variables X, Boolean output vairables Y and a set of state-holding elements (flip-flops) FF can also be modeled as a Mealy transducer M with input alphabet 2^X and output alphabet 2^Y. The state space of the transducer is defined by the possible valuations of FF, i.e. $Q^m \subseteq 2^{FF}$, and the initial state is determined by the initial assignment ff_0 to FF[1]. For convenience, we use sequential circuits and Mealy transducers interchangably in this thesis.

2.3 Linear Temporal Logic

Linear Temporal Logic was first introduce by Pnueli in [Pnu77]. Since its introduction, it has been widely used to specify properties and reason about the behaviors of sequential circuits [MP92]. In this section, we review background on the syntax and semantics of LTL. In addition, we describe a few decision problems related to LTL.

2.3.1 Syntax and Semantics

An LTL formula is built from atomic propositions AP, Boolean connectives (i.e. negations, conjunctions and disjunctions), and temporal operators $\mathbf{X}$ (*next*) and $\mathbf{U}$ (*until*). Given an atomic proposition $p \in AP$, a formula in linear temporal logic (LTL) can be constructed as follows.

$$\psi ::= p \mid \neg\psi \mid \psi \vee \psi \mid \mathbf{X}\ \psi \mid \psi\ \mathbf{U}\ \psi$$

[1] This definition can be generalized to include multiple initial states.

Other temporal operators $\mathbf{F}$ (*eventually*) and $\mathbf{G}$ (*globally*) can be derived using the temporal operators $\mathbf{X}$ and $\mathbf{U}$, and Boolean connectives: $\mathbf{F}\,\psi = \texttt{true}\;\mathbf{U}\,\psi$ and $\mathbf{G}\,\psi = \neg\mathbf{F}\,\neg\psi$.

LTL formulas are usually interpreted over infinite words (paths) over $\Sigma = 2^{AP}$. Let $\pi = \pi_0, \pi_1, \pi_2, \ldots \in \Sigma^\omega$ be an infinite path over Σ and $\pi^i = \pi_i, \pi_{i+1}, \pi_{i+2}, \ldots$ be the suffix of π starting at π_i. The semantics of an LTL formula is then defined inductively, as shown in Table 2.3.

Table 2.3: Semantics of LTL

$\pi \models \texttt{true}$	
$\pi \not\models \texttt{false}$	
$\pi \models \psi$	iff $\psi \in \pi_0$ for $\psi \in AP$
$\pi \not\models \psi$	iff $\psi \notin \pi_0$ for $\psi \in AP$
$\pi \models \psi \vee \phi$	iff $\pi \models \psi$ or $\pi \models \phi$
$\pi \models \psi \wedge \phi$	iff $\pi \models \psi$ and $\pi \models \phi$
$\pi \models \mathbf{X}\,\psi$	iff $\pi^1 \models \psi$
$\pi \models \psi\,\mathbf{U}\,\phi$	iff $\exists\, i \geq 0$ such that $\pi^i \models \phi$ and $\forall\, 0 \leq j < i$, $\pi^j \models \psi$.

The language of an LTL formula ψ is then the set of infinite words that satisfy ψ, given by $\mathcal{L}(\psi) = \{w \in \Sigma^\omega \mid w \models \psi\}$.

We introduce two more abbreviations, similar to those used in PSL [EF06], that correspond to rising or falling edge of a signal (in a digital circuit). Given an atomic proposition p, we define

- $rise(p) = \neg p \wedge \mathbf{X}\,p$, and
- $fall(p) = p \wedge \mathbf{X}\,\neg p$.

Thus, $rise(p)$ and $fall(p)$ denote the cases when a propositional variable p changes its value from `false` to `true`, and from `true` to `false`, respectively.

One classic example is the LTL formula $\mathbf{G}\,(p \rightarrow \mathbf{F}\,q)$, which means every occurrence of p in a trace must be followed by some q in the future.

Properties are usually classified as being *safety* or *liveness* properties. Informally, safety properties are those that specify that “nothing bad happens in the trace” and their violation can be demonstrated on a finite-length trace. Liveness properties on the other hand specify that “something good happens in the future” and their violation can only be shown with an infinite-length (lasso-shaped) trace.

2.3.2 Satisfiability and Realizability

An LTL formula ψ is *satisfiable* if there exists an infinite word that satisfies ψ, i.e. $\exists\,\pi \in \Sigma^\omega$ such that $\pi \models \psi$. A transducer M *satisfies* an LTL formula ψ if $\mathcal{L}(M) \subseteq \mathcal{L}(\psi)$ over $AP = X \cup Y$. We write this as $M \models \psi$. *Realizability* is then the decision problem of

determining whether there exists a transducer M with input alphabet $\mathcal{X} = 2^X$ and output alphabet $\mathcal{Y} = 2^Y$ such that $M \models \psi$.

An LTL formula ψ can also be "interpreted" over finite words (traces), such as in a dynamic verification setting. The idea is to construct a (deterministic) monitor M^d (also more generally known as a synchronous observer [HLR93, Rus12]) from ψ such that M^d outputs a signal `err` if and only if a finite prefix of trace π violates ψ. For example, Figure 2.1 shows a monitor for the LTL formula $\mathbf{G}\ (a \to \mathbf{X}\ b)$. The monitor outputs `err` if and only if it is in the "err" state.

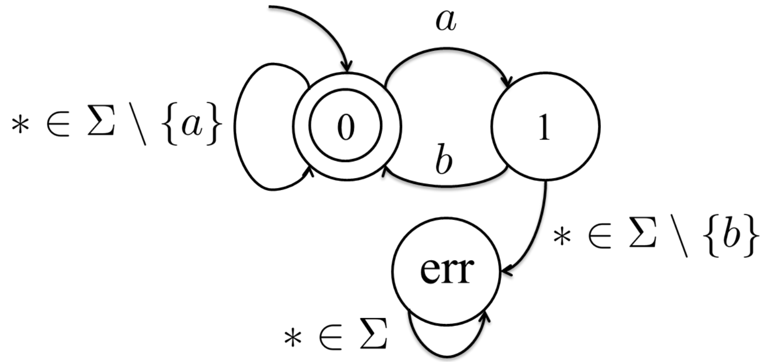


Figure 2.1: Monitor for $\mathbf{G}\ (a \to \mathbf{X}\ b)$.

Multiple ways for the construction of such a monitor exist. We point interested readers to [KYV01, BGHS04, AKT+06] for more detailed discussions of this topic.

2.4 Specification Mining with Templates

A *specification template* is a syntactically constrained formula or automaton. We start with a semi-formal definition of *template-based specification mining*, with terms to be concretized later.

Definition 2.3. *Given a* specification template ξ^p *over a pattern alphabet* Σ^p *and some* evidence *ev over a disjoint alphabet* Σ*, the* specification mining problem *is to find all total and one-to-one mappings* $\kappa : \Sigma^p \to \Sigma$ *such that the evidence* satisfies ξ^p *with symbols in* Σ^p *replaced by symbols in* Σ *under the mapping* κ*.*

For example, a specification template can be an LTL formula $\xi^p = \mathbf{G}\ (a \to \mathbf{X}\ b)$ for an alphabet $\Sigma^p = \{a, b\}$. Consider the following evidence which is a finite length trace π over the alphabet $\Sigma = \{\text{req}, \text{reset}, \text{grant}\}$: $\text{reset}, \text{req}, \text{reset}, \text{req}, \text{grant}$. An LTL formula that is satisfied by π conforming to the template is $\xi = \mathbf{G}\ (\text{reset} \to \mathbf{X}\ \text{req})$, which is associated with the mapping κ where $\kappa(a) = \text{reset}$ and $\kappa(b) = \text{req}$. We call such a formula over the projected alphabet Σ under some mapping κ an *instantiation* ξ of the specification template ξ^p. For simplicity, we use Ξ to denote the set of all possible instantiations over Σ given a set

of templates Ξ^p over Σ^p. Note that, the specific instance of the template-based specification mining problem where the specification ψ is given in LTL, the evidence is given as a trace π and the satisfaction criterion is determined by the output of the monitor for ψ, is equivalent to having monitors for all possible instantiations of the specification template and then finding a subset of those which do not produce any `err` when evaluated on π.

The definition is intentionally given liberally, so that the notions of *specification template*, *evidence* and *satisfaction* can be tailored to specific problems and applications. In this example, our specification template is an LTL formula, the evidence is a finite trace, and satisfaction amounts to an interpretation of the LTL formula over a finite trace. Similar definitions have been given in prior work, but often restricted to a particular model or domain. For instance, Gabel and Su [GS08b] use a finite-state automaton (FSA) as ξ^p and a finite trace as evidence and satisfaction as the trace accepted by the FSA. In this thesis, we show that this template-based specification mining approach can be generalized to different sources and representations, such as ξ^p given as an LTL formula, and evidence given as a discrete transition system. Moreover, in Chapter 5, we present a technique for mining specifications without the need to assume some pre-determined templates, thereby further automating the process of specification mining.

Part I

Requirement Generation and Error Localization

Chapter 3

Background

In this chapter, we present additional background materials on mining temporal properties from traces. These materials will be useful in explaining our monitor-based technique in Chapter 4 and the sparse-coding technique in Chapter 5.

3.1 Formalism and Notations

3.1.1 Transition System

It is convenient to view a sequential circuit as a transition system $M^t = (V^t, Q_0^t, \rho^t)$ where $V^t = X \cup Y \cup FF$ is a finite set of Boolean variables. We denote the state space of M^t as $Q^t \subseteq 2^{V^t}$. $Q_0^t \subseteq Q^t$ is a set of initial states of the system, and $\rho^t \subseteq Q^t \times Q^t$ is the transition relation. A state of the system $q \in Q^t$ is a Boolean vector comprising valuations to each variable in V^t. For clarity, we restrict ourselves in this thesis to synchronous systems in which transitions occur at the tick of a clock, such as digital circuits, although the ideas can be applied in other settings as well.

3.1.2 Traces and Subtraces

Let the state of the system at the ith cycle (step) be denoted by q_i. A *complete trace* $\tilde{\tau}$ of the system of length l is a sequence of states $q_0, q_1, q_2, \ldots, q_{l-1}$ where $q_0 \in Q_0^t$, and $(q_i, q_{i+1}) \in \rho^t$ for $0 \leq i < l-1$. We denote the ith state of $\tilde{\tau}$ as $\tilde{\tau}_i$. Note, however, that the full system state and/or inputs might not be observed or recorded during execution. We therefore define a *trace* τ as a sequence of valuations to an observable subset $V^{t,o}$ of the variables V^t; i.e., $\tau = q_0^o, q_1^o, q_2^o, \ldots, q_{l-1}^o$ where each q_i^o is the corresponding q_i restricted to $V^{t,o}$. Similarly, We denote the ith state of τ as τ_i. For simplicity, in the rest of this dissertation, we use V^t in place of $V^{t,o}$ to refer to the set of observable variables when it is clear from the context.

A *subtrace* $\tau_i^w = \tau_i, \tau_{i+1}, \ldots, \tau_{i+w-1}$ of length w ($w > 1$) in τ is defined as the segment of τ starting at cycle i and ending at cycle $i+w-1$, such that $i \geq 0$ and $i+w \leq l$. For example, a subtrace of length 2 contains the evolution of the observed variables when the underlying

system makes one transition. We use $\mathcal{L}^w_{V^t}(M^t)$ to denote the set of all distinct w-length subtraces observable from M^t over the variable set V^t. We say a τ^w can be *exhibited* by M^t if $\tau^w \in \mathcal{L}^w_{V^t}(M^t)$. It is easy to see that $\mathcal{L}^w_{V^t}(M^t)$ is finite for any finite-state transition system.

3.2 Running Example

In this section, we describe a running example which we will use to illustrate the concepts and techniques in Chapter 4 and Chapter 5.

Figure 3.1 shows a simple two-port arbiter design. The two-port arbiter is a digital circuit that takes requests from its two input channels, represented by Boolean signals r_0 and r_1, and produces grants respectively corresponding to these requests, represented by Boolean signals g_0 and g_1. If $r_i = \texttt{true}$, it indicates that there is a request on input channel i. Similarly, if $g_i = \texttt{true}$, it indicates that a grant is produced at output channel i.

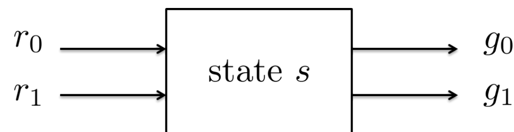


Figure 3.1: A two-port arbiter design.

In this design, when there is only a single request at a cycle, a corresponding grant is produced for that request at the same cycle. To handle competing requests, the arbiter uses a Boolean variable s to indicate which channel currently has priority. When $s = 0$, r_0 has priority over r_1. Otherwise, r_1 has priority over r_0. When $r_0 = \texttt{true} \wedge r_1 = \texttt{true}$, $g_i = \texttt{true}$ for the channel i that currently has a higher priority (and $g_j = \texttt{false}$ for the channel j that has a lower priority). Additionally, the arbiter implements a round-robin scheme of arbitration. This means that if it produces a grant for channel i at the current cycle, then it will give priority to the other channel in the next cycle. Figure 3.2 shows a trace of length 5 over the input and output signals r_0, r_1, g_0, g_1.

cycle	1	2	3	4	5
r_0	0	1	0	1	1
r_1	0	0	0	1	0
g_0	0	1	0	0	1
g_1	0	0	0	1	0

Figure 3.2: Sample trace of the arbiter design.

Observe that at cycle 2, a grant is produced at output channel 0. Thus, when there are competing requests at cycle 3, a grant is produced at output channel 1 due to this channel having a higher priority.

We further note that the arbiter design, if operating correctly, should satisfy the following LTL properties.

- Liveness: $\mathbf{G}\ (r_i \rightarrow \mathbf{F}\ g_i)$[1]
- Round-robin: $\mathbf{G}\ \big((g_i \wedge \mathbf{X}\ r_j) \rightarrow (\mathbf{X}\ g_j)\big)$ for $i \neq j$.
- No simultaneous grant: $\mathbf{G}\ \neg(g_i \wedge g_j)$ for $i \neq j$.

It is important to note that the behaviors of a trace is also influenced by the environment that interacts the arbiter, i.e. how r_0 and r_1 are produced. In this case, we assume an environment that keeps track of whether a request has been granted, and keeps issuing it until it is granted. For example, in Figure 3.2, when a request is initiated on channel 0 at cycle 3 but not granted at that cycle, r_0 holds the value `true` *until* the next cycle at which this request is then granted. We can also use the following LTL formula to describe this behavior.

$$\mathbf{G}\big(rise(r_0) \wedge \neg rise(g_0) \rightarrow \mathbf{X}\ (r_0\ \mathbf{U}\ rise(g_0))\big) \tag{3.1}$$

Given traces of a design (and with the arbiter as an illustrative example), we describe two approaches in Chapter 4 and Chapter 5, for mining temporal properties that reflect the behaviors of the arbiter described above, dynamically from these traces. In addition, we show that the mined properties can be used to effectively localize bugs (especially in time) in case when the design fails to operate correctly.

[1] In fact, **F** can be replaced by **X** in this formula.

Chapter 4

Specification Mining for Digital Circuits

To understand is to perceive patterns.

– Isaiah Berlin

Formal specifications can precisely capture a system's desired behavior. In fact, much of the challenge in developing a system lies in specifying it – understanding the properties that characterize precisely the behaviors of the system. Once these properties have been established, assertion-driven verification such as model checking [CGP00] and runtime-monitoring [Sto02] can then be used to ensure the system is developed correctly according to specification. However, the difficulty of manually creating a complete set of formal properties (assertions) and of maintaining those properties through design changes and evolution has significantly hindered the wide-spread adoption of formal specifications. There is therefore an urgent need for scalable techniques for automatically generating formal specifications.

In this chapter, we present our specification mining approach to address this problem, with a particular focus on digital circuits. In Section 4.1, we give an overview of the proposed technique. Then, in Section 4.2, we describe background materials and introduce the specification templates we use in this work. After that, we present our algorithm in detail in Section 4.3. One key contribution of this chapter is a novel way of using mined specification to localize errors *in time and in space* for digital circuit designs. We elaborate on this in Section 4.4. Finally, we present experimental results in Section 4.5 and discuss our findings in Section 4.6.

4.1 Overview

In this chapter, we present a novel approach to scalably mine temporal properties in the form of recurring patterns from simulation or execution traces of a digital design. These specifications can then be examined by the engineer to see whether they match the designer's

intent or can be checked with further verification. Figure 4.1 illustrates the high-level tool flow.

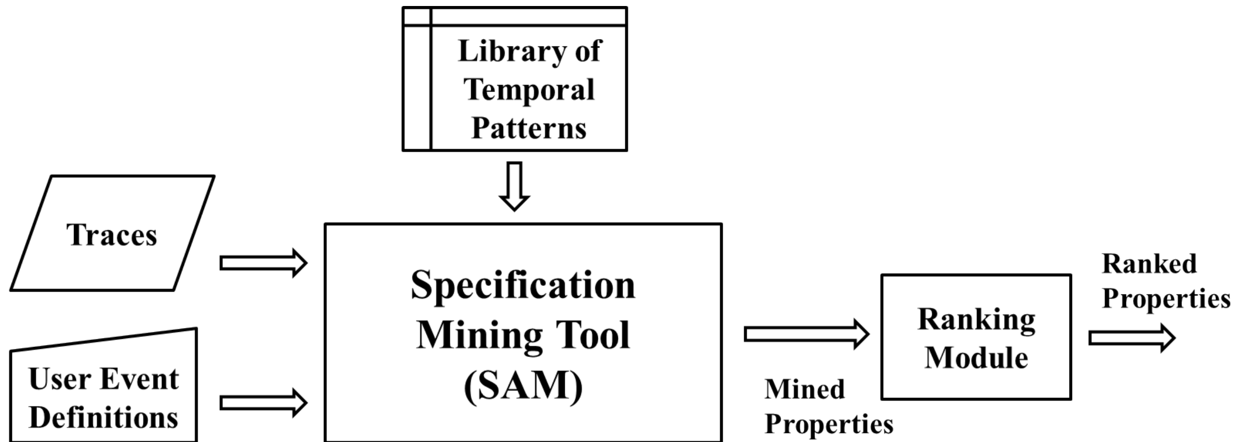


Figure 4.1: High-level architecture of the proposed specification mining tool SAM.

Our tool takes a set of traces and optionally a user-defined event definition as input, and generates a set of behavioral patterns which conform to the trace as output. We use a specification template library that is specially designed for the digital circuit domain. We describe these templates in Section 4.2.2 and the associated mining algorithm in detail in Section 4.3. Lastly, a post-processing ranking module is used to produce a heuristically-ranked list of properties.

Specification mining not only helps to automate coverage-driven simulation or formal verification, it can also provide useful information for diagnosis. In Section 4.5, we present a technique that is able to leverage the mined properties to effectively localize bugs in both RTL (e.g. programming mistakes such as erroneous state machine transitions) as well as faults that may arise from physical defects (e.g., stuck-at or transient faults).

To summarize, we make the following key contributions:

- A new dynamic specification mining technique especially designed for general digital circuits. Our tool SAM (Scalable Assertion Miner) efficiently mines non-trivial specifications and is highly scalable: for a design with over 20,000 signals, over 1000 properties were mined in under a minute;
- A novel trace-diagnosis technique based on specification mining that achieves good localization accuracy for large circuits.

4.2 Preliminaries

We first define formal notations and then describe the specification templates that the tool uses.

4.2.1 Notations

Definition 4.1. *An event e is a tuple $(v, \dot{v})$, where v is a bit-vector variable (an array of Boolean variables) and $\dot{v}$ is a value assignment to each Boolean variable in the array. Additionally, we use t_e to denote a cycle at which e occurs.*

Similar to $rise(p)$ and $fall(p)$ for a Boolean variable p, we define a change of a bit-vector variable as a *delta event*.

Definition 4.2. *A delta event $\delta(v)$ can be characterized by the following LTL formula,*

$$\delta(v) = \big((v = \dot{v}_1) \wedge (\boldsymbol{X}\,(v = \dot{v}_2)) \wedge (\dot{v}_1 \neq \dot{v}_2)\big)$$

which indicates some change of value of v. We use $\dot{v}_\delta^+$ to denote the value that v changes to ($\dot{v}_2$) after a delta event $\delta(v)$ occurs. Similar to Definition 4.1, we use $t_{\delta(v)}$ to denote a cycle at which $\delta(v)$ is satisfied.

4.2.2 Specification Templates

We consider specification templates with a small alphabet. Gabel and Su [GS08b] show that the problem of mining properties represented as finite automata from (finite-length) traces is NP-hard through reduction from the Hamiltonian Path problem. Hence, it makes sense to mine patterns with a small alphabet to avoid the potential worst-case exponential blow-up. Their approach builds on that of Perracotta [YEB+06] which requires $O(n^k)$ space and $O(n^{k-1}l)$ time for an input alphabet size of n, a pattern alphabet size of k, and a trace of length l. Gabel and Su further propose the use of Binary Decision Diagrams (BDDs) [Bry86] to improve the tractability of the problem. However, while they show some speed-up using the symbolic technique, the alphabet size is still limited to 3 in practice. In addition, the performance of BDD-based techniques depends heavily on having a good variable ordering, and finding the optimum variable ordering is again NP-hard [BW96].

We extend the simple alternating patterns $(a\ b)^*$ described in [YEB+06] and [GS08b] to include properties in LTL. As noted in Section 2.3.2, an LTL property can be "interpreted" over *finite* traces by using a monitor. Hence, the acceptance of a FSA, in the case of an alternating pattern, can be replaced by whether the corresponding monitor outputs `err` for an LTL template instantiation.

Additionally, in order to handle the large number of signals and long (easily millions to billions cycles long) simulation/execution traces in modern digital designs, we focus on properties that can be efficiently monitored. Below, we list the three basic types of templates we consider in this work. In the next section, we elaborate on how these temporal properties can be monitored (mined) efficiently from hardware traces.

- **Alternating:** the regular expression template $(\delta(a)\ \delta(b))^*$ specifies that a change of value of signal a must always alternate with a change of value of signal b.

- **Until:** the LTL template $\mathbf{G}\ \big(\delta(a) \wedge \neg\delta(b) \rightarrow \mathbf{X}\ (\dot{a}^{+}_{\delta}\ \mathbf{U}\ \delta(b))\big)$ for specifying the property that whenever a signal a changes value, it should retain that value until another signal b changes value. The property described in Equation 3.1, for example, describes one such behavior (in Equation 3.1) that the arbiter example satisfies.
- **Eventually:** the LTL template $\mathbf{G}\ \big(\delta(a) \rightarrow \mathbf{X}\ (\mathbf{F}\ \delta(b))\big)$ describes that a change of value of a must be eventually followed by a change of value of b. Notice that this behavior almost always holds for any pair of signals. In Section 4.3, we describe further refinement of this property, based on the "responsiveness" of $\delta(b)$. For example, the **Next** pattern $\mathbf{G}\ (\delta(a) \rightarrow \mathbf{X}\ \delta(b))$ restricts that the change must always happen in the next cycle.

4.3 Mining Algorithm

Our algorithm is based on constructing a monitor[1] (see Section 2.4) for each instantiation of the specification templates defined in the previous section. To cope with the large number of signals and long traces, we employ the following two trace reduction techniques.

- In case module information is available, e.g., from the RTL description of a hardware design, we partition a trace into disjoint traces, one for each module.
- Convert a trace to a *delta trace*, which contains only *delta events*. We show that for our set of predefined templates, it is sufficient to consider only delta traces.

We elaborate on these techniques and describe the basic mining algorithm in Section 4.3.1. The predefined specification templates only describe relationships between two signals, thus cannot directly capture more complex interactions among multiple signals. In Section 4.3.2, we present two merging techniques for synthesizing more complex specifications from these simple properties. Finally, also due to the simplicity of the templates, a large number of properties are likely to hold on the given traces. In Section 4.3.3, we discuss some metrics that we employ to rank the mined properties.

4.3.1 Mining from Delta Traces

We assume we are given a partition of the observable variables V^t into disjoint bit-vector variables. This is motivated by the fact that, in a RTL description, a signal a may be declared as "a[0:3]" indicating that it is a bit-vector of length 4. It is therefore more meaningful to consider the bits of a as a whole than to consider them as separate Boolean signals. Formally, let $V^t = \biguplus_{1 \leq i \leq k} V_i^t$, where each V_i^t is a bit-vector variable of length greater than 0. Hence, each state in a trace consists of a set of *events* E, where each event $e_i = (V_i^t, \dot{V}_i^t)$.

[1] In the case of the **Alternating** template, which is essentially a regular expression, the monitor is a DFA.

Our first trace reduction technique is based on the observation that not all valuations of a variable V_i^t is always present in a trace (similar to the notion of reachable states). Hence, by considering an alphabet comprising only events seen in a given trace, we can reduce the size of the alphabet. In a Perracotta [YEB+06]-style algorithm, this in turn reduces the memory requirement ($O(n^k)$ space where n is the alphabet size and the pattern alphabet size $k = 2$ in our case). However, the running time ($O(n^{k-1}l)$) of such an algorithm linearly depends on l. In the digital circuit domain, l can easily be millions or even billions cycles long. Thus, we use the second trace reduction technique, which turns a trace into a delta trace as mentioned earlier. While the idea of compressing a trace into a delta trace is not new (e.g., the value change dump (VCD) format commonly used in logic simulation), the key novelty of our approach is that we can design specification templates, as shown in Section 4.2.2, that are specific to hardware-relevant behaviors and at the same time can be interpreted and efficiently monitored solely over delta traces.

Take the two-port arbiter described in Section 3.2 as an example. We can turn the trace in Figure 3.2 into the equivalent *delta trace* shown in Figure 4.2.

cycle	1	2	3	4	5
r_0	$rise(r_0)$		$rise(r_0)$		
r_1			$rise(r_1)$	$fall(r_1)$	
g_0	$rise(g_0)$	$fall(g_0)$		$rise(g_0)$	
g_1			$rise(g_1)$	$fall(g_1)$	

Figure 4.2: Equivalent delta trace of the trace shown in Figure 3.2.

A *delta trace* τ^δ is a trace where each state contains a set of *delta events* E^δ that occur (are true) in that state. Observe that this simple transformation already reduces the number of events we need to consider for the arbiter example in two dimensions. First, the effective length of the delta trace is now 4 (as opposed to 5), since there are no delta events in the last cycle (such states are subsequently removed from the delta trace). Second, the maximum number of events per cycle is now 3 (as opposed to 4), which effectively reduces the "n" in the $O(nl)$ time algorithm, for reasons that will become obvious later. For a bit-vector signal v, a delta event $\delta(v)$ specifies a change from $\dot{v}_1$ at the current cycle to $\dot{v}_2$ at the next cycle. We further simplify this (at the cost of forgetting $\dot{v}_1$ but gains a reduction in the number of delta events) by considering only $\dot{v}_\delta^+$, i.e. all delta events with the same $\dot{v}_\delta^+$ are considered to be the same event.

Consider the LTL formula described in Equation 3.1. We can determine whether this formula is violated given the trace in Figure 3.2 by monitoring it over the delta trace in Figure 4.2. Figure 4.3 shows the corresponding monitor, which operates on delta traces. Specifically, it takes as input a word (possibly infinite) over 2^{AP} where the set of delta events

forms the set of atomic propositions AP. The absence of a delta event $\delta(v)$ at a cycle is interpreted as $\neg\delta(v)$.

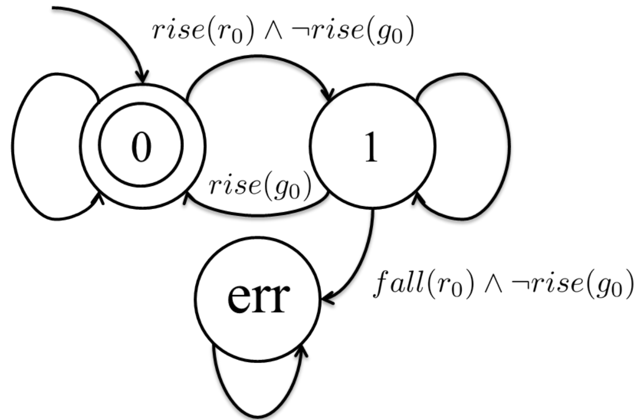


Figure 4.3: Monitor for the formula in Equation 3.1 over delta traces (propositions not shown on certain transitions are understood to be complementary such that the automaton is deterministic).

As an illustration, a run of this monitor on the example delta trace produces the sequence of states "0", "0", "1", "0" (recall that the last state of the delta trace is removed). Since this run does not produce the output `err`, we claim that the corresponding LTL formula is a *likely* specification of the arbiter (which turns out to be a real one in this case). In addition, observe that the run of the monitor revisits state "0" once after transiting to state "1", which corresponds exactly to the pattern $\big(rise(r_0) \land \neg rise(g_0) \rightarrow \mathbf{X}\ (r_0\ \mathbf{U}\ rise(g_0)\big)$ occurring once when the antecedent of the implication is true[2]. We use this information to record the number of *occurrences* of the pattern associated with a particular template instantiation. Additionally, we call a (delta) event that cause a transition from state "0" to state "1" or state "1" to state "0" a *constituent event* of the pattern. In this example, the delta events $rise(r_0)$ at cycle 3 and $rise(g_0)$ at cycle 4 are constituent events of the **Until** pattern described by the LTL formula in Equation 3.1. The cycles at which a constituent event occurs (cycle 3 and 4 in the above example) are recorded as timestamps for the respective event if the corresponding specification is not violated by the trace.

The overall algorithm is outlined below, using a single specification template ξ^p given a trace τ. This algorithm is repeated in a similar fashion for each of the predefined templates for all partitioned (by module) traces.

Our algorithm is adapted from that of Perracotta [YEB+06]. The main idea is to first allocate a 2D table Tab with dimensions $|E^\delta| \times |E^\delta|$ such that each entry (i, j) in the table records the current state of the corresponding monitor for the instantiation of ξ^p with a mapping κ that maps $a \in \Sigma^p$ to $e_i^\delta \in E^\delta = \Sigma$ and $b \in \Sigma^p$ to $e_j^\delta \in \Sigma$. Figure 4.4 illustrates this idea.

[2]State "0" thus behaves like an accepting state.

Algorithm 1 Mine all likely specifications given a template and a single trace.

Input: A finite trace τ.
Input: The set of events E in τ.
Input: A specification template ξ^p.
Output: A set of likely properties Ξ as instantiations of ξ^p.
1: $(\tau^\delta, E^\delta) = \textbf{deltaTrace}(\tau, E)$
2: $Tab = \textbf{createTable}(E^\delta, \xi^p)$
3: **for** each state $\tau_i^\delta \in \tau^\delta$ **do**
4: $\quad \textbf{updateTable}(Tab, \tau_i^\delta)$
5: **end for**
6: $\Xi = \textbf{outputPatterns}(Tab)$

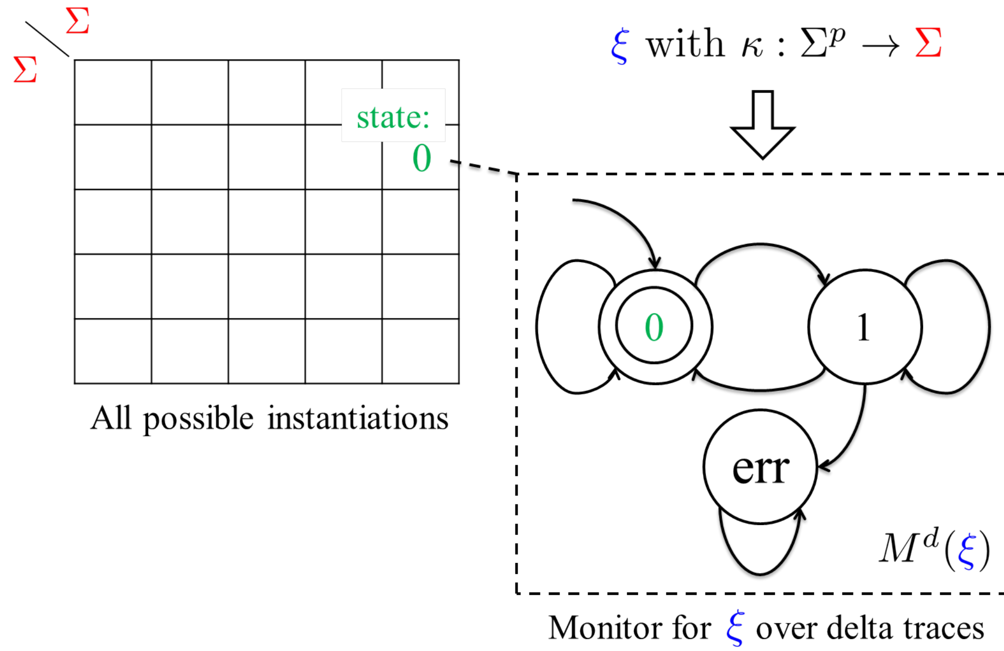


Figure 4.4: Illustration of the procedure **createTable**.

Then, the delta trace τ^δ is processed sequentially starting from τ_0^δ. For each state τ_i^δ, the corresponding entries in Tab are updated, i.e. monitors make state transitions according to τ_i^δ. Specifically, given a delta event $e^\delta \in \tau_i^\delta$, every entry with a mapping ρ which maps a to e^δ is first updated. This corresponds to a row in Tab. Then, every entry with a mapping κ which maps b to e^δ is updated. This corresponds to a column in Tab. When performing the alternating row and column updates, we need to avoid updating the same entry multiple times at the same cycle. Figure 4.5 illustrates one such scenario, when the procedure **updateTable** operates on $\tau_i^\delta = \{e_j^\delta, e_{j+1}^\delta\}$. Notice that when updating Tab for e_{j+1}^δ, the entry $(j, j+1)$ remains unchanged because the corresponding monitor has already

made a transition when e_j^δ was processed.

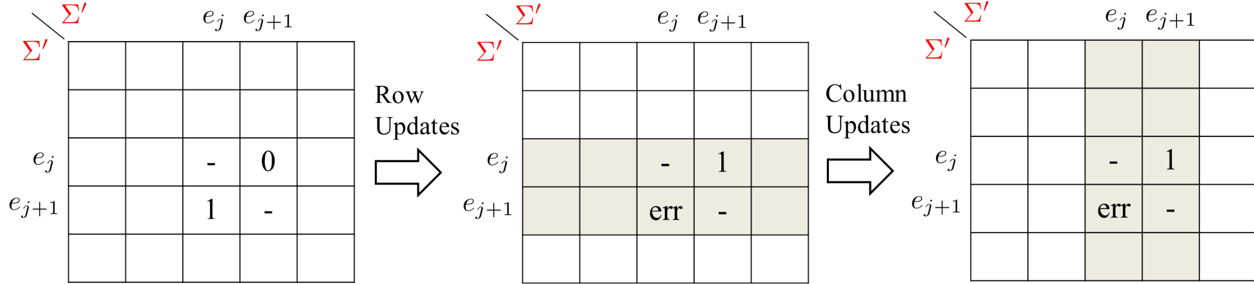


Figure 4.5: Iterative row and column updates of Tab over $\tau_i^\delta = \{e_j^\delta, e_{j+1}^\delta\}$.

Finally, each entry in Tab not ending at an "err" state corresponds to a candidate specification[3]. The procedure **outputPatterns** produces the set of candidate specifications Ξ.
Complexity: Given an input trace τ of length l over a set of events E, the procedure **deltaTrace** converts the trace to a delta trace τ^δ of length $l^\delta < l$ with delta events E^δ. In our experience, typically, $l^\delta \ll l$ and $|E^\delta| \ll 2^{|V^t|}$. This thus greatly enhances the scalability of the proposed technique. Hence, Algorithm 1 requires $O(|E^\delta|^2)$ space and runs in $O(|E^\delta| l^\delta \max_i |\tau_i^\delta|)$ time.

4.3.2 Merging Simple Specifications

The predefined templates can only express relationships between two (delta) events. In this section, we describe two merging procedures for deriving more complex properties, e.g., properties over multiple events, from the simple mined properties.

Property merging: We first merge properties by matching the time of occurrences of the constituent events. If two instantiations of the same template contain events that always occur at the same cycles, then there is potential for merging the two properties into a single one. For example, if both $(e_1^\delta\ e_3^\delta)^*$ and $(e_2^\delta\ e_3^\delta)^*$ are mined properties from running Algorithm 1 with the **Alternating** template on a trace τ, and the constituent events e_1^δ and e_2^δ always occur at the same cycles, then we can merge the two properties into $(e_1^\delta \wedge e_2^\delta\ e_3^\delta)^*$.

For each mined property, a list of timestamps $(t_{\delta(v)})$ is maintained for each of its constituent events $\delta(v)$ during the property mining phase. These lists are processed afterwards in a property merging phase by first partitioning the set of properties into mergeable subsets, and then merging the properties within each partition by taking the conjunction of events that always appear together. For example, if the constituent events $\delta(a)$ and $\delta(b)$ always occur at the same time in the mined properties $\mathbf{G}\ \big(\delta(a) \rightarrow \mathbf{X}\ (\mathbf{F}\ \delta(c))\big)$ and $\mathbf{G}\ \big(\delta(b) \rightarrow \mathbf{X}\ (\mathbf{F}\ \delta(c))\big)$, then we can merge them as a single property $\mathbf{G}\ \big(\delta(a) \wedge \delta(b) \rightarrow \mathbf{X}\ (\mathbf{F}\ \delta(c))\big)$. The algorithm is outlined in Algorithm 2.

[3]Our tool allows a user to further require that each of these entries end at a state that signifies *complete* occurrence of a pattern.

Algorithm 2 Merge Properties

1: Ξ : a list of mined properties.
2: L_ξ^t : a list of timestamps for each $\xi \in \Xi$, with length $|L_\xi^t|$.
3: ind_ξ : index of the current timestamp in L_ξ^t, starting from 1.
4: min_t: $\min_{\xi in \Xi} L_\xi^t[ind_\xi]$.
5: **function Partition**(Ξ)
6: **if** $(|\Xi| = 1) \vee (ind_\xi = |L_\xi^t|, \forall \xi \in \Xi)$ **then**
7: **return** $\{\Xi\}$
8: **else if** $\exists \xi, ind_\xi = |L_\xi^t|$ **then**
9: **return append**($\{\xi \mid ind_\xi = |L_\xi^t|\}$, **Partition**($\{\xi \mid ind_\xi \neq |L_\xi^t|\}$)
10: **else**
11: $\Xi' = \{\xi \mid L_\xi^t[ind_\xi] = min_t\})$
12: $\forall \xi' \in \Xi'$, $ind_{\xi'} := ind_{\xi'} + 1$
13: **return append**(**Partition**(Ξ'), **Partition**($\{\xi \mid L_\xi^t[ind_\xi] \neq min_t\}$))
14: **end if**
15: **end function**
16: **function Merge**(Ξ)
17: $\{\Xi_1, \ldots, \Xi_k\}$ = **Partition**(Ξ)
18: Merge events to form a single property in each Ξ_i by taking the *conjunction* of the constituent events occurring at the same time.
19: **end function**

The algorithm essentially first partitions the mined properties into sets of properties such that the occurrences of events all match in time for each set, and then merge these properties into a single one. This merging procedure is particularly useful when no event definition is provided *a priori*. A hardware module in a typical microprocessor core can have hundreds of signals running in parallel and many of them are highly correlated. In Section 4.5, we demonstrate that this simple recursive procedure significantly reduces the number of mined specifications and in practice generates better quality specifications for the user.

Property chaining: After we merge the mined properties, we further *chain* them by repeatedly applying a set of inference rules to the results to obtain even more complex properties. Two simple inference rules are given below.

Rule for **Alternating** (adapted from [GS08a]):

$$\frac{(\delta(a)\ \delta(b))^* \quad (\delta(b)\ \delta(c))^* \quad (\delta(a)\ \delta(c))^*}{(\delta(a)\ \delta(b)\ \delta(c))^*}$$

Rule for **Eventually**:

$$\frac{\mathbf{G}\ (\delta(a) \rightarrow (\mathbf{X}\ \mathbf{F}\ \delta(b))) \quad \mathbf{G}\ (\delta(b) \rightarrow (\mathbf{X}\ \mathbf{F}\ \delta(c)))}{\mathbf{G}\ (\delta(a) \rightarrow \mathbf{X}\ \mathbf{F}\ (\delta(b) \rightarrow \mathbf{X}\ \mathbf{F}\ \delta(c)))}$$

4.3.3 Specification Ranking

The process of merging and chaining also allows us to further sieve through the set of specifications for the most interesting ones. For example, if one is interested in complex interactions, we can output only properties with an alphabet size greater than a user-specified threshold. We also allow the user to put a restriction on the “responsiveness” of an event in an **Eventually**-type specification, e.g., a request must be granted within a certain number of cycles. We parameterize this by a positive integer r, such that the **Eventually** template with responsiveness r is given by the following LTL formula.

$$\mathbf{G}\ \big(\delta(a) \rightarrow \mathbf{X}^{\leq r}\ \delta(b))\big)$$

This formula requires that a $\delta(a)$ must be followed by a $\delta(b)$ within r cycles. The set of r-responsive specifications can be obtained from the set of **Eventually**-type specifications produced by Algorithm 1 by using a post-processing step. For a **Eventually**-type specification, we check if $t_{\delta(b)} - t_{\delta(a)} \leq r$ for all consecutive pairs of constituent events $\delta(a)$ and $\delta(b)$. The advantage of using this post-processing step, as opposed to treating a r-responsive specification as a new template, is that we do not need to create a new monitor and rerun Algorithm 1 for a different value of r, thereby improving the efficiency of the overall approach.

In our tool, we can also rank patterns (properties) according to either *frequency of occurrences* or *time of first occurrence* of a pattern. The *time of first occurrence* of a pattern is marked by the timestamp of the first constituent event $\delta(a)$ in the pattern.

4.4 Error Localization

Our specification mining-based bug localization framework is inspired by prior work in diagnosis, such as cooperative bug isolation by Liblit [Lib04], which proposes the use of predicates at different program locations to help isolating bugs. The mined properties considered in our approach can also be viewed as predicates over specific signals in a circuit. Thus, they provide location information with respect to these signals. Moreover, the occurrences of a property also convey information about when certain behaviors appear. Combining these two ideas, we propose to use our specification mining algorithm as a subroutine to determine properties that distinguish between a correct trace (or a set of correct traces) and an error trace. The location and time information associated with a distinguishing pattern (which we will describe later), can then be used to help localize the bug *both in space and in time.*

Figure 4.6 illustrates the proposed bug localization tool flow, where the specification-mining algorithm is performed on both the correct and the error trace to find distinguishing patterns.

Distinguishing patterns (properties) are essentially those that exist in one trace but not in the other. After finding these patterns, we rank them according to time of first occurrence to output the most likely bug location. As also noted in [Lib04], the qualities of

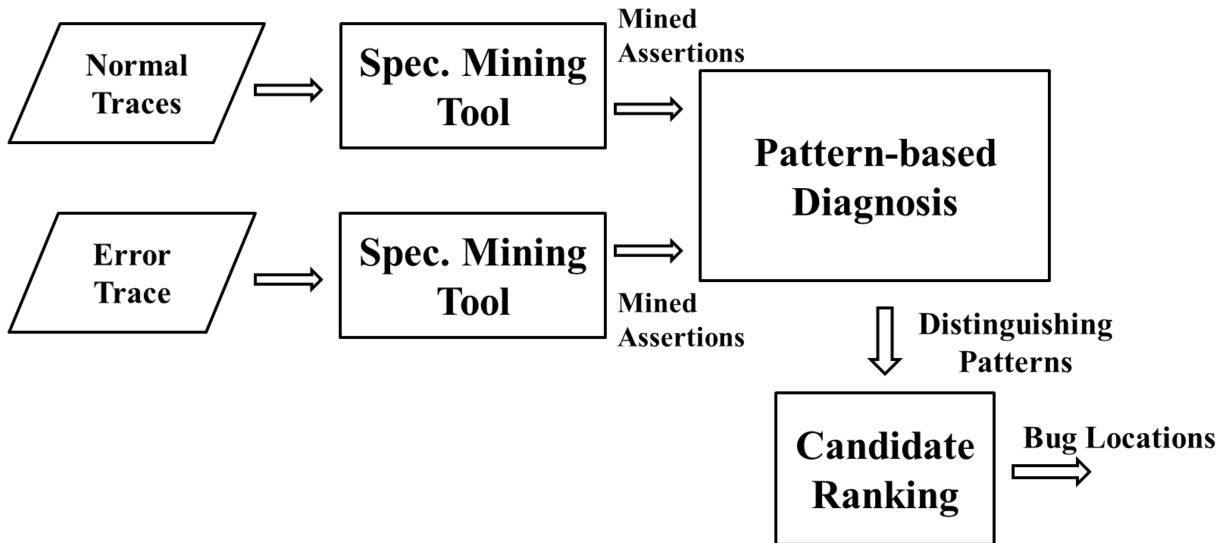


Figure 4.6: Specification mining-based bug localization.

predicates fundamentally influence the performance of a pattern-based diagnosis tool. Thus, the research question we explore in this chapter is *whether we can use many simple mined properties to localize complex bugs in hardware designs.*

Consider the problem of diagnosing an error given a set of correct traces and a single error trace. Our goal is to localize the root cause of the error to the part of the circuit where the bug occurred (*in space*). For transient faults (e.g., as a result of electrical defects), another important goal is to localize them *in time*, i.e. to find the approximate time of occurrence of a transient fault. One potential application is post-silicon debugging [MSN10] where bugs are difficult to diagnose due to limited observability, limited reproducibility, and possible dependence on physical parameters.

A number of diagnosis approaches have been proposed in the classic AI literature. As observed by Console et al. [CT91], these approaches either require models that describe the correct behaviors of the system or they need models for the abnormal (faulty) behaviors. Our approach is similar to the consistency-based methods [dKMR92]. In the traditional consistency-based reasoning approach, if a system can be described using a set of constraints, then diagnosis can be accomplished by identifying the set (often minimal) of constraints that must be excluded in order for the remaining constraints to be consistent with the observations. While this approach does not require knowledge of how a component fails (a fault model), it requires a reasonably complete specification of the correct system. In the EDA literature, while there has been substantial work on fault diagnosis and debugging in a post-silicon environment [PM08, PBWM10, dPGH+08, ZWM11, ZWSM11, WB11, AKB12], to our knowledge our work is the first to make use of automatically mined specifications. Related work in bug localization will be discussed in more detail in Section 5.7.

Our approach is similar to the consistency-based method but we do not need to start with

a set of specifications. Instead, we mine specifications from traces and use them to localize the errors. Our approach does not directly make use of the RTL description for diagnosis (other than the module hierarchy), which makes it scalable and appealing for post-silicon debug. In addition, we do not need to time-align the correct traces with the incorrect trace. The *trace-based bug localization problem* can be defined as follows.

Definition 4.3. *Given a correct trace τ jointly produced by a set of modules M, and an incorrect trace τ' over the same alphabet Σ produced by M' such that some $m \in M'$ is erroneous, the* bug localization problem *is to localize the bug to m, and if the bug occurred at cycle t, then localize its occurrence to $[t-k, t+k]$ for some small k.*

Remark 4.1. *We assume that the error is detectable at the system level. This means that there exists a mechanism to label a trace (erroneous or otherwise) with respect to some correctness criteria. Typically, such a mechanism relies on checking or monitoring some end-to-end behaviors, such as whether a program running on the processor under analysis crashes. Additionally, while this problem definition seems to suggest that we need to assume the presence of a single bug, we remark that it simply targets one bug at a time. Thus, an approach that solves this problem can be used iteratively in case multiple bugs need to be localized.*

Consistency is defined with respect to the specifications mined from the correct trace. Specifically, consistency is violated if

- A pattern is observed in the error trace but it fails at some point in the correct trace; or
- A pattern is observed in the correct trace but it fails at some point in the error trace.
- A pattern is observed in both traces but it has a different responsiveness (e.g., for an **Eventually**-type pattern) in the error trace than in the correct trace[4].

A pattern that violates consistency is termed a *distinguishing pattern*. An error can propagate to other modules and in turn cause more erroneous behaviors later. In light of this, we rank the mined distinguishing patterns by the first time of violation – the point where a pattern is expected to hold but does not. For example, the first occurrence of $\delta(a)$ that is not followed by an occurrence of $\delta(b)$ in the **Eventually** pattern $\mathbf{G}\ \big(\delta(a) \rightarrow \mathbf{X}\ (\mathbf{F}\ \delta(b))\big)$. The module which the top ranked pattern belongs to is returned as the space-localization of the bug. The time of the pattern's first violation is returned as the time-localization of, for example, transient fault. Since the pattern itself describes a specific erroneous behavior, our approach not only localizes the bug, but can also produces useful insights about the error (e.g., a local failure that may start a chain of reactions that eventually lead to the end-to-end error).

[4]We did not use this criterion in our experiments.

4.5 Experiments

We have implemented the proposed approach in a tool called SAM (Scalable Assertion Miner). The tool is available online at `http://verifun.eecs.berkeley.edu/sam/`. In this section, we present case studies illustrating that our approach has the following desirable characteristics:

- Scalable: We can mine specifications from traces that are millions of cycles long, with thousands of signals, all within two minutes.
- Effective for bug localization: In fault injection experiments, our mined specifications accurately localized the fault to within module boundaries, and in the case of transient faults, localized them to a small time window within when they occurred.
- Relevant: The mined specifications, after the merging procedure, are sufficiently high-level so as to be useful to designers.

We used ModelSim and iVerilog to simulate the designs and to record the traces in the VCD format. The experiments are run on a netbook with an Intel Atom 1.60 GHz processor and 1.0 GB of RAM.

4.5.1 Benchmarks

We used the following four benchmarks in our experiments: a MIPS-based microprocessor design [PLF06], a chip-multiprocessor (CMP) router [Peh01], a I^2C interface core and a CAN interface design.

eMIPS processor: The eMIPs processor is a MIPS-based processor developed at Microsoft Research, Redmond [PLF06]. eMIPS is a dynamically extensible processor architecture based on the MIPS R4000 instruction set. The version that was used in the experiments had 278 modules and contained more than 20000 signals.

CMP router: The CMP router was designed by Peh [Peh01]. We used a simplified two-port version with 14 modules in the experiments. The overview of the CMP router design is illustrated in Figure 4.7, as a composition of four high-level components..

The input controller comprises a set of FIFOs buffering incoming flits and interacting with the arbiter. When the arbiter grants access to a particular output port, a signal is sent to the input controller to release the flits from the buffers, and at the same time, an allocation signal is sent to the encoder which in turn configures the crossbar to route the flits to the appropriate output port.

Lastly, the I^2C and CAN are the Inter-Integrated Circuit (I^2C) interface and Control Area Network (CAN) interface designs publicly available on Opencores [ope].

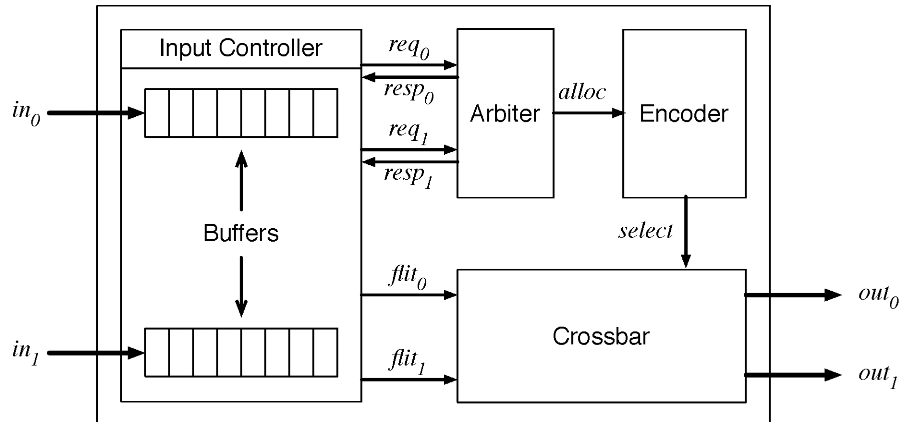


Figure 4.7: CMP router comprising four high-level components.

4.5.2 Results

In our experiments, we only recorded values of latches with width less than 5 in any simulation trace. This is a simple heuristic to prune away the data paths, whose behaviors are less interesting from a specification perspective, and due to the fact that we did not start with any manual event definition.

Scalability: The first experiment is meant to evaluate both the efficiency of our specification mining algorithm and the usefulness of the specification merging procedure. We simulated each of the benchmarks on the default test-bench supplied with it to generate one very long trace for that benchmark. Then we applied our mining tool to the resulting trace. (Applying our tool to multiple traces will yield similar results.)

Table 4.1 summarizes the performance of our tool on each benchmark circuit.

	$\lvert\tau\rvert$	$\lvert\tau^\delta\rvert$	$n_m^{\max}$	$\lvert S\rvert$	$\lvert S_m\rvert$	R_t (sec)
eMIPS	5.0×10^6	5408	108	2079	1028	51
Router	2.3×10^5	12420	28	120	74	13
I^{2}C	1.6×10^6	20904	33	389	308	9
CAN	2.6×10^7	36100	175	3272	1356	71

Table 4.1: Performance results on generation of likely specifications.

$\lvert\tau\rvert$ is the size of the original trace. $\lvert\tau^\delta\rvert$ and $n_m^{\max}$ are the average length of delta traces and the maximum number of delta events per module respectively. $\lvert S\rvert$ is the total number of specifications mined before any merging is performed. $\lvert S_{merge}\rvert$ is the total number of specifications resulting from applying the parallel merging and chaining procedures. R_t is the overall runtime of the tool for each benchmark in seconds.

Observe that the compression of a trace to a delta trace significantly reduces its length – by about 1000X. Moreover, we enforce that the number of delta events per module to be

less than 200 (modularization was then done automatically along module boundaries given in the RTL descriptions). As a result, the number of delta events to be processed by our algorithm was at most 175. The number of simple specifications we generated is in the few thousands. After the application of property merging, this number was reduced by 2X. The runtime was very small – less than 2 minutes for all benchmarks.

Relevance: Figure 4.8 shows part of the specifications mined in the "vcstate" module for the CMP router. This module contains a state machine that controls the handshake with the arbiter and the decision to move the flits in the buffer to the corresponding output channels. Figure 4.9 shows the actual behavior of the state machine.

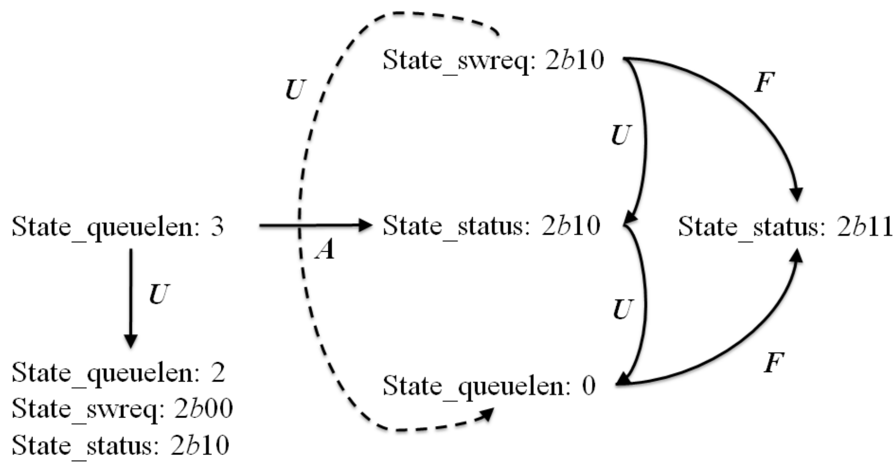


Figure 4.8: Example mined specification from the CMP router.

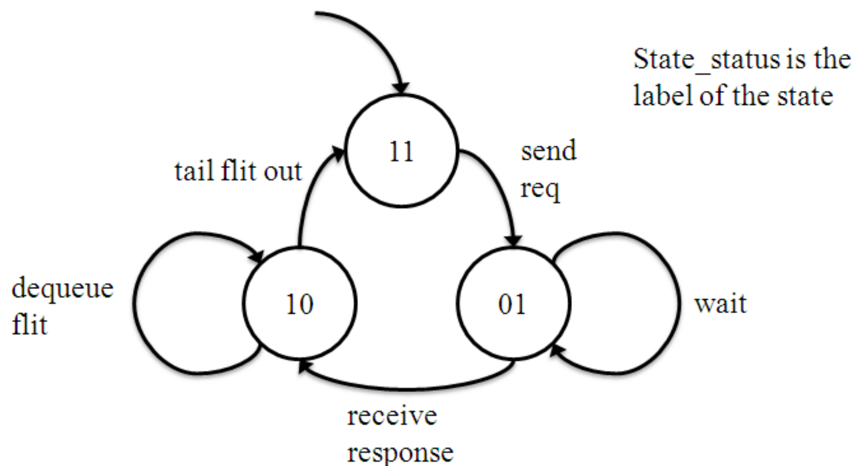


Figure 4.9: State machine in the "vcstate" module.

The mined specifications are illustrated in a directed graph. A node is labeled by a conjunction of delta events. The number after the colon corresponds to the value ($\dot{v}_\delta^+$) of the

signal in front of the colon. Together they constitute an event (in this case a delta-event $\delta(v)$ by considering only $\dot{v}_\delta^+$). "State_status" corresponds to the state in the input controller state machine. An edge from a node a to a node b corresponds to a specification template with $\Sigma = \{a, b\}$ with a mapped to the conjunct labeled at a and b mapped to the conjunct labeled at b. The label of an edge indicates the type of the specification – ***A*** for **Alternating**, ***U*** for **Until**, and ***F*** for **Eventually**.

In the mined properties, we can see that, for example, "State_swreq" always stays at $2b10$ (requesting output channel 1) until "State_status" moves to $2b10$. In fact, it stays high until "State_queuelen" (which registers the number of flits in the buffer) goes to 0. If "State_queuelen" goes to 0, then "State_status" eventually goes back to its initial state $2b11$. In this particular configuration of the router, it has a buffer size of 4 on both input channels. A data packet consists of three flits – a head, a body and a tail flit. The test-bench was set up in such a way that it imitates an upstream router that maintains credits on whether an outgoing packet from that router has been successfully routed by the router under analysis. Packets were randomly issued by the test-bench, and the frequency of packet injection was measured at about 23%.

The mined specifications match well with the behaviors of the router – a data packet (3 flits) comes in and fills up the buffer; the head flit triggers a request and before another packet comes to the same input channel, it manages to secure an output channel and the entire packet gets dequeued through that channel. Although the quality of the specification depends heavily on the quality of the simulation trace, the mined specifications are still useful since it can indicate the parts of behaviors that the current test-bench has covered, allowing future simulation efforts to be directed to the uncovered behaviors quickly, e.g., test with different router configurations and traffic patterns.

Bug localization: In the second set of experiments, we syntactically injected faults into the designs and then used our mining-based diagnosis approach to localize the fault. We focus on the eMIPS processor design and the CMP router, due to the flexibility of using different and meaningful (with the environment set up suitably) test-benches during simulation.

eMIPS Processor. The faults we injected included single-bit errors, multiple-bit errors, and erroneous state transitions. In the first case, we inverted the "dne_r" signal in the BlockRAM Controller from $1b0$ to $1b1$. In the second, we changed the "we_r" signal in the BlockRAM Interface module from $4b0$ to $4b6$. In the last case, we changed the "rdstate" signal conditionally in the "Register_File_Read" module from $2b00$ to $2b10$ when it is in state $2b01$. This represents an erroneous transition in a state machine. We produced from each case an error trace of one million cycles, which is also approximately one million cycles before the error failed some system-level end-to-end specification. *In all three fault injection experiments, our diagnosis technique ranked the faulty module as a top candidate among the* 278 *modules.* However, on average 6 other modules are also ranked as top candidates. This is due to the fact that some signals in these modules are combinational output of the error, and these signals in turn violated some local properties mined in their modules. While it is possible to overcome this by tracking only the registers, the trade-off is that since we track less signals, we will lose some behavioral coverage.

Table 4.2: Bug localization results on the CMP router.

Type of Fault	N_f	Cov_A	$Local_t$	$Local_m$
stuck-at	5	100%	–	100%
erroneous transition	3	100%	–	100%
erroneous assignment	7	100%	–	57%
transient	16	100%	81%	56%

CMP Router. With the same configuration of the router used in the specification generation experiments, we used a test-bench with random packet generation at the input but with a fixed traffic pattern. The same test-bench was used to generate both the correct trace and the faulty trace. This time, our mined specifications only covered signals that are registers. We injected different types of faults into the router, including stuck-at faults on wires and registers, erroneous transition in state machines, erroneous assignment (syntactical) for wires and registers, and single transient errors in registers. Table 4.2 shows the localization results. N_f is the number of fault injection experiments that were performed for each type of fault. Cov_A is the percentage of times that an assertion mined from the correct trace is falsified by the error trace – existence of a distinguishing pattern and this is also a form of assertion coverage. $Local_t$ (for transient errors) is the percentage of times that error was localized by some distinguishing pattern within before or after 15 cycles of the transient fault – this measures localization in time. Finally, $Local_m$ is the percentage of times that our tool returned the correct module where the error occurred – this measures localization in space.

As we observe, the technique was able to achieve perfect assertion coverage and localize transient faults in space 100% of the time for simple faults (stuck-at and erroneous transition), and more than 50% of the time for trickier faults (erroneous assignment and transient). For transient faults, the top-ranked distinguishing pattern was able to localize them in more than 80% of the time. For example, the assertion $\mathbf{G}$ ((state_queuelen = 2) $\rightarrow$ $\mathbf{X}$ $\mathbf{F}$ (state_swreq = 0))[5] is the distinguishing pattern that successfully localized a transient error in the "state_status[0]" signal and returned the correct module localization, even though the distinguishing pattern itself does not contain the error signal.

4.6 Summary and Discussion

In this chapter, we have proposed a scalable specification-mining tool that is suitable for general digital circuits. In addition, we have shown that our mined specifications are effective for bug localization. Experimental evaluation shows that (a) the mining algorithm is practical, requiring only minutes of computation even for a very large-scale example; (b) for human benefit, the mined specifications can be automatically compacted by a significant factor; (c)

[5]The assignments represent $\dot{v}_\delta^+$ of delta events.

the diagnostic use is effective in pinpointing the bug location to the correct module when programming mistakes and hardware faults are applied to the benchmarks.

An inherent limitation of dynamic specification mining is that the quality of the specification mined is only as good as the set of traces. In the case of digital circuits, we can leverage techniques in coverage-directed testing to simulate many behaviors as fast as possible. An alternative to using coverage tools is to improve the mined specifications online, as the circuit runs in a testing or production mode on real workloads. For example, in a testing mode one can prototype the circuit on a piece of reconfigurable logic and iteratively generate online assertion checkers for specifications mined from each trace.

We use templates in this work, and thus can only mine properties that conform with the predefined templates (even though they match well with common behavioral patterns in hardware). In the next chapter, we present a novel specification mining technique that does not have this restriction.

Chapter 5

A Sparse Coding Framework for Specification Mining

All fixed set patterns are incapable of adaptability or pliability. The truth is outside of all fixed patterns.

– Bruce Lee

In this chapter, we present a new specification mining formalism inspired by the notion of *sparse coding* in the machine learning literature.

5.1 Introduction

Different kinds of formal specifications provide different trade-offs in terms of ease of generation from traces, generality, and usefulness for error localization. Finite automata provide a very general formalism, and are typically inferred from finite-length traces. However, such automata tend to overfit the traces they are mined from, and do not generalize well to unseen traces – i.e., they are very sensitive to the choice of traces $\mathcal{T}$ they are mined from and can easily exclude valid executions outside of the set $\mathcal{T}$. Linear temporal logic (LTL) formulas [MP92] are an alternative. One typically starts with templates for common temporal logic formulas and learns LTL formulas that are consistent with a set of traces. If the templates are chosen carefully, such formulas can generalize well to unseen traces. However, the biggest challenge is in coming up with a suitable set of templates that capture all relevant behaviors.

In this chapter, we introduce a third kind of formal specification, which we term as *basis subtraces*. To understand the idea of a subtrace, consider the view of a trace as a two-dimensional matrix, where one dimension is the space of system variables and the other dimension is time. A *subtrace* is a finite window, or a snapshot, of a trace. Thus, just as a movie is a sequence of overlapping images, a trace is a sequence of overlapping subtraces. Restricting ourselves to Boolean variables, each subtrace can be viewed as a binary matrix.

Given a set of finite-length traces, and an integer w, the traces can be divided into subtraces of time-length w. The set of all such subtraces constitutes a set of binary matrices. The basis subtraces are simply a set of subtraces that form a basis of the set of subtraces, in that every subtrace can be expressed as a superimposition of the basis subtraces.

The form of superimposition depends on the type of system being analyzed. In this chapter, we consider digital systems such as sequential circuits. In this context, one can define superimposition as a "linear" combination over the semi-ring with Boolean OR as the additive operator and Boolean AND as the multiplicative operator. The coefficients in the resulting linear combination are either 0 or 1. The problem of computing a basis of a set of subtraces is equivalent to a Boolean matrix factorization problem, in which a Boolean matrix must be decomposed into the product of two other Boolean matrices.

Given a set of subtraces, several bases are possible. Following Occam's Razor principle, we seek to compute a "simple" basis that generalizes well to unseen traces. More concretely, we seek to find a basis that is "minimal" in the sense that each subtrace is a linear combination of *only a small number* of basis subtraces. This yields the *sparse basis problem*. In this chapter, we formally define this problem in the context of Boolean matrix factorization and propose a graph-theoretic algorithm for solving it. Such a problem is often referred to as a *sparse coding* problem in the machine learning literature, since it involves encoding a data set with a "code" in a sparse manner using only a few non-zero coefficients.

Similar to Chapter 4, we apply the generated basis subtraces to the problem of bug localization. In digital circuits, an especially vexing problem today is that of post-silicon debugging, where, given an error trace with potentially only a subset of signals observable and no way to reproduce the trace, one must localize the problem in space (to a small collection of error modules) and time (to a small window within the trace). Similar problems arise in debugging distributed systems. In addition, bug localization is very relevant to "pre-silicon" verification as well. Our approach attempts to *construct* windows of an error trace using a basis computed from subtraces of length w sliced from a set of good traces. The hypothesis is that the earliest window that cannot be constructed is likely to indicate the time when the bug occurred, and the portions that cannot be constructed are likely to indicate the signals (variables) that correspond to the location of the problem. The technique can thus be applied for *simultaneous* error localization and explanation.

To summarize, the main contributions of this chapter are:

- We introduce the idea of *basis subtraces* as a formal way of capturing behavior of a design as exhibited by a set of traces;
- We formally define the *sparsity-constrained Boolean matrix factorization problem* and propose a graph-theoretic algorithm for solving it;
- We demonstrate with experimental results that we can mine useful specifications using our sparse coding method, and

- We show that the computed bases can be effective for simultaneous error localization and error explanation, even for transient errors, such as bit flips, that arise not just due to logical errors but also from electrical effects.

The rest of the chapter is organized as follows. We first describe additional notations and formalisms in Section 5.2. Then, we formally introduce our novel specification formalism, called basis subtraces, in Section 5.3. Afterwards, we describe an optimization formulation and a graph-theoretic algorithm for mining basis subtraces in Section 5.4. In Section 5.5, we describe an approach that leverages the mined basis to localize errors in traces. In Section 5.6, we present a case study on the CMP router design described in Section 4.5. We survey related work in Section 5.7 and finally summarize the chapter in Section 5.8.

5.2 Background

In this section, we make the connection from traces to their graph representations. Section 5.2.1 introduces notations representing traces of a reactive system as matrices, and Section 5.2.2 connects the matrix representation with a graph representation.

5.2.1 Traces and Matrices

Equation 5.1 shows a trace τ of length 4 where each state comprises a valuation to two Boolean variables. We depict the trace in matrix form, where the rows correspond to variables and the columns to cycles.

$$\begin{bmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{bmatrix} \tag{5.1}$$

The subtrace τ_0^2 of τ thus can be similarly represented by the following matrix.

$$\begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix}$$

Let $\mathcal{T}^w$ be the set of all subtraces of length w in a trace τ of length l, i.e. $\mathcal{T}^w = \{\tau_i^w \mid 0 \leq i \leq l - w\}$. For any $\tau^w \in \mathcal{T}^w$ over the set of observable variables V^t, we can view it as a Boolean matrix of dimensions $|V^t| \times w$. We can also conveniently represent it using a vector $v^w \in \mathbb{B}^{|V^t| \times w}$ by stacking the columns in τ^w (i.e., using a column-major representation). For example, v_0^2 as shown below (the apostrophe represents matrix transpose) represents the subtrace τ_0^2.

$$v_0^2 = \begin{bmatrix} 1 & 1 & 0 & 0 \end{bmatrix}'$$

We use $\|v_i^w\|_1$ to denote the number of 1s in v_i^w (the L_1-norm for the Boolean vector v_i^w). For convenience, we use v_i^w and τ_i^w interchangeably in the rest of this chapter. Also, for brevity, we use v_i for v_i^w when the length of each subtrace w is obvious from the context.

Now we can represent $\mathcal{T}^w$ as a Boolean matrix with $|V^t| \times w$ rows and $l - w + 1$ columns. For example, we can represent all the subtraces of length 2 for the trace in Equation 5.1 as the matrix D shown in Equation 5.2.

$$\begin{bmatrix} 1 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix} \tag{5.2}$$

5.2.2 Bipartite Graphs

A Boolean matrix can be viewed as an adjacency matrix for a *bipartite graph* (*bigraph*, for short). A bipartite graph $G = \langle U, V, E \rangle$ is a graph with two disjoint non-empty sets of vertices U and V and such that every edge in $E \subseteq U \times V$ connects one vertex in U and one in V. For a Boolean matrix $D \in \mathbb{B}^{k_1 \times k_2}$, denote $D_{i,j}$ as the entry in the ith row and j^{th} column of D. Then, D can be represented by a bigraph G_D with $U = \{u_1, u_2, \ldots, u_{k_1}\}$ and $V = \{v_1, v_2, \ldots, v_{k_2}\}$, such that there is an edge connecting $u_i \in U$ and $v_j \in V$ if and only if $D_{i,j} = 1$. For example, the matrix D in Equation 5.2 can be represented by the bigraph G_D shown in Figure 5.1.

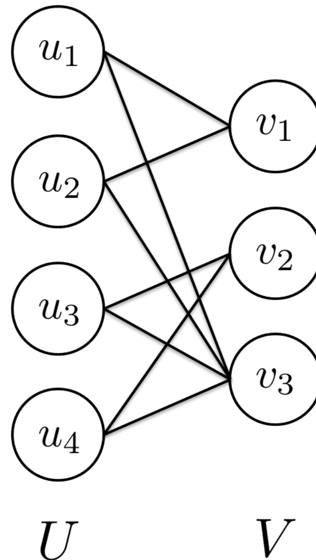


Figure 5.1: Bipartite graph G_D.

A *biclique* is a complete bipartite graph; i.e., a bipartite graph $G' = \langle U', V', E' \rangle$ where $E' = U' \times V'$. Given a bigraph G, a *maximal edge biclique* of G is a biclique $B_1 = \langle U_1 \subseteq U, V_1 \subseteq V, E_1 = U_1 \times V_1 \rangle$ if it is not contained in another biclique of G, that is, there does not exist another biclique $B_2 = \langle U_2 \subseteq U, V_2 \subseteq V, E_2 = U_2 \times V_2 \rangle$ and either $U_1 \subset U_2$ or $V_1 \subset V_2$.

In the rest of this chapter, we use the pair of vertices (U_1, V_1) to denote the maximal edge biclique B_1. For a set of bicliques Cov and a bigraph G, denote E_{Cov} as the set of edges in G covered by Cov, i.e. $\forall e \in E_{Cov}, \exists G' = \langle U' \subseteq U, V' \subseteq V, E' \rangle \in Cov$, s.t. $e \in E'$. Cov is a *biclique edge cover* of G if and only if all the edges E in G are covered by the set, i.e. $E_{Cov} = E$. Abusing notation a little, we use E_v to denote the set of edges connected to vertex v. The smallest number of bicliques needed to cover all the edges in G is called the *bipartite dimension* of G. For example, a biclique edge cover Cov for the bigraph G_D in Figure 5.1 is shown in Figure 5.2.

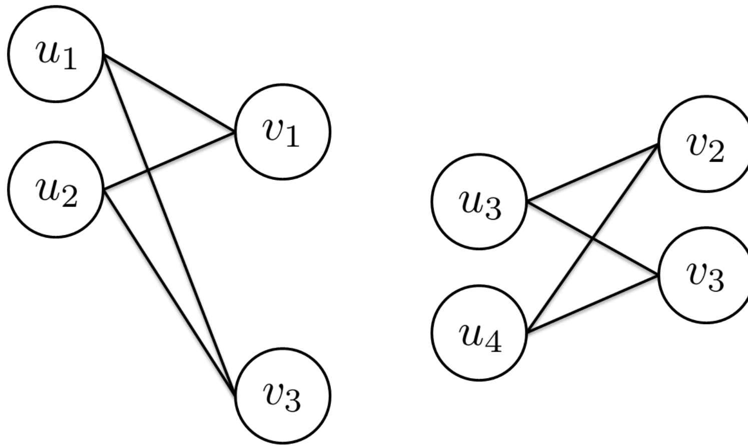


Figure 5.2: Biclique edge cover Cov.

The view of Boolean matrices as bigraphs is relevant for decomposing a set of traces into a set of basis subtraces. The following problem is important in this context.

Definition 5.1. *Consider a Boolean matrix $D \in \mathbb{B}^{m \times n}$, the* Boolean matrix factorization problem *is to find k and Boolean matrices $B \in \mathbb{B}^{m \times k}$ and $S \in \mathbb{B}^{k \times n}$ such that*

$$D = B \circ S \tag{5.3}$$

That is, D is decomposed into a Boolean combination (denoted by the operator $\circ$) of two other Boolean matrices, in which scalar multiplication is the Boolean `AND` operator $\wedge$, and scalar addition is the Boolean `OR` operator $\vee$. In other words, we perform matrix/vector operations over the Boolean semi-ring with $\wedge$ as the multiplicative operator and $\vee$ as the additive operator. For example, the matrix D in Equation 5.2 can be factorized in the following way.

$$\begin{bmatrix} 1 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix} \circ \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}$$

We use $D_{\cdot,i}$ to denote the ith column vector of a matrix D, and $D_{i,\cdot}$ to denote the ith row vector of D. Thus, the columns of matrix D are $D_{\cdot,1}, D_{\cdot,2}, \ldots, D_{\cdot,n}$. We will refer to D as

the *data matrix* since it represents traces which are the input data. We call the matrix B the *basis matrix* because each $B_{\cdot,i}$ can be viewed as some basis vector in $\mathbb{B}^m$. We call the matrix S the *coefficient matrix*. Each $S_{\cdot,i}$ is a Boolean vector in which a 1 in the jth entry indicates that the jth basis vector is used in the decomposition and 0 otherwise.

We can also rewrite the factorization in the following way as a Boolean sum of the matrices formed by taking the tensor (outer) product of the ith column in B and the ith row in S.

$$\begin{bmatrix} 1 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix}$$

Notice that the two matrices on the right hand side correspond to the bicliques in Figure 5.2.

Remark 5.1. *Clearly, a solution always exists for the problem in Definition 5.1. This is because one can always pick $k = n$ such that $B = I$ $(B = D)$ and $S = D$ $(S = I)$ (where I is the identity matrix). However, this is not particularly revealing in terms of the behaviors which each $D_{\cdot,i}$ is composed of. One alternative is to minimize k. The smallest k for which such a decomposition exists is called the ambiguous rank [Fro95] of the Boolean matrix D. It is also equal to the bipartite dimension of the bigraph G_D corresponding to matrix D. The problem of finding a Boolean factorization of D with the smallest k is equivalent to finding a biclique edge cover of G_D with the minimum number of bicliques. Both problems have been shown to be NP-hard [Sie00]. On the other hand, one can choose to find an overcomplete basis $(k > n)$ such that each $D_{\cdot,i}$ can be expressed as a Boolean sum of* only a few *basis vectors. We discuss this formulation in detail in the following section.*

5.3 Specification Formalism – Basis Subtraces

In this section, we present a novel specification formalism, based on the idea of sparse coding, that characterize the behaviors of a circuit (i.e. traces) in terms of "basis subtraces". This formalism departs from traditional specification formalisms, which often prescribe to certain logic descriptions, e.g., CTL or LTL. CTL, for example, is commonly interpreted over Kripke structures [Kri63]. In contrast, our formalism is trace-based – the behavior of a system is characterized solely based on its *observed* behaviors. Hence, it is especially useful for reasoning about black-box systems.

The notion of sparsity is borrowed from the wealth of literature in machine learning such as sparse coding [OF97, LBRN07] and sparse principal component analysis (PCA) [ZHT04]. The main idea of sparse coding is to use an overcomplete code (basis) but enforce certain sparsity constraint. Because the basis vectors are non-orthogonal and not linearly independent of each other, sparsifying the code will bias towards recruiting only those basis vectors necessary for representing an input. As a result, it often helps to generate a good interpretation of the data in terms of the underlying concepts or patterns. In the setting of mining

specifications from a trace, we argue that each subtrace of a trace can be viewed as a superimposition of patterns, and a potential specification is a pattern that is commonly shared by multiple subtraces. We call these patterns *basis subtraces*.

Without loss of generality, consider V^t to be the set of observable variables. Let $m = |V^t| \times w$ for some positive integer w representing the length of a subtrace. We first define the notion of *constructing* a subtrace from a set of basis subtraces.

Definition 5.2. *A set of k subtraces, represented by a Boolean matrix $B \in \mathbb{B}^{m \times k}$, where each column $B_{\cdot,i}$ corresponds to some subtrace of length w over V^t,* constructs *a subtrace $v^w \in \mathbb{B}^m$ if there exists a coefficient vector $s \in \mathbb{B}^k$ such that $v^w = B \circ s$.*

In particular, if $\|s\|_1 \leq C$, then we say the set of subtraces represented as B *constructs* the subtrace v^w with *sparsity* C. Now we are ready to define the meaning of a *basis*, with respect to a set of subtraces.

Definition 5.3. *A set of k subtraces $B \in \mathbb{B}^{m \times k}$* forms a basis *of a set of subtraces $\mathcal{T}^w$ if for every subtrace $v^w \in \mathcal{T}^w$ ($v^w \in \mathbb{B}^m$), B* constructs *v^w.*

Similarly, if B constructs every subtrace $v^w \in \mathcal{T}^w$ with sparsity C, then B forms a *sparse basis* of $\mathcal{T}^w$ with sparsity C. In addition, we say B *specifies* M^t if B forms a (sparse) basis of $\mathcal{L}^w_{V^t}(M^t)$ (recall the definition of $\mathcal{L}^w_{V^t}(M^t)$ in Section 3.1.2) given some w and V^t (and for some C).

Observe that a B that specifies M^t may also construct a subtrace $v^w \notin \mathcal{L}^w_{V^t}(M^t)$. This is, however, aligned with the notion of a specification. In general, we requires only the specification to be superset of the behaviors of the system that it specifies. For example, in the case of model checking a finite-state transducer M with an LTL property ψ, we check if $\mathcal{L}(M) \subseteq \mathcal{L}(\psi)$ but not the other direction.

The basis subtraces that B represents can be viewed as temporal patterns of V^t over a finite time window w. Hence, the notion of B constructing a subtrace v^w can be thought of as having some of the patterns in B *superimpose* together to form v^w. Figure 5.3 illustrates this idea.

In this picture, the occurrence of an event is represented by the appearance of a shaded bubble, and correspondingly a 1 in the matrix. Similarly, the absence of an event corresponds to a 0 in the matrix. We are given two basis subtraces of length 2 consisting of three events. For instance, $basis_1$ may describe a behavioral pattern where the presence of events A and B triggers event C in the next cycle. The superimposition of $basis_1$ and $basis_2$ forms the subtrace v. A trace thus can be viewed as a sequence of subtraces where each subtrace contains a specific subset of patterns in the basis. Sparsity C then constrains the maximum number of (concurrent) patterns that a subtrace can contain.

A basis can also be thought of as a *behavioral encoding* of the underlying transition system. This encoding allows one to separate potentially erroneous subtraces from correct subtraces, by checking if a subtrace can be encoded by the basis. As we will show in Section 5.5, this allows one to effectively localize bugs (in time) by solely analyzing the traces of a design.

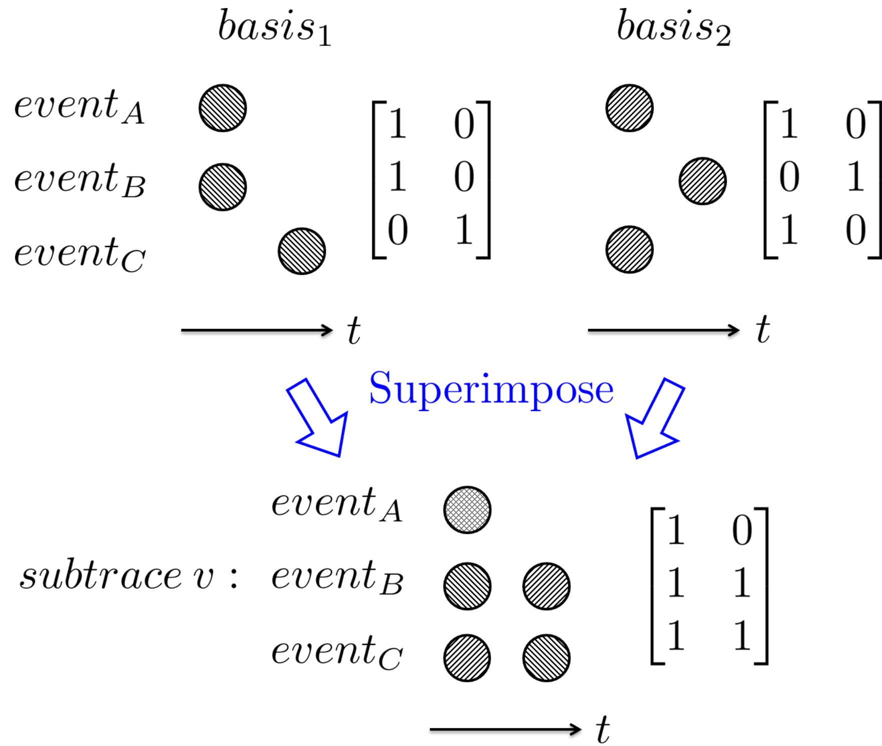


Figure 5.3: A subtrace as superimposition of two basis subtraces.

In the next section, we describe formulations and algorithms for learning B based on observed traces of a system M^t. In particular, motivated by the idea of sparse coding, we are interested in whether enforcing some sparsity C will help identifying a basis B that contains meaningful behaviors of M^t.

5.4 Algorithm: Sparsity-Constrained Biclique Cover

In this section, we formulate the problem of mining a basis B with sparsity C from observed traces of a transition system M^t, and describe an algorithm for solving it.

5.4.1 Formulation as a Sparse Coding Problem

We first extend the Boolean matrix factorization problem described in Section 5.2.2 as follows.

Definition 5.4. *Given $D \in \mathbb{B}^{m \times n}$ and a positive integer C, the* sparsity-constrained Boolean matrix factorization problem *is to find k, $B \in \mathbb{B}^{m \times k}$, and $S \in \mathbb{B}^{k \times n}$ such that*

$$\begin{aligned} D &= B \circ S \\ \textit{and} \quad \|S_{\cdot,i}\|_1 &\leq C, \ \forall_i \end{aligned} \tag{5.4}$$

It is easy to see that this formulation extends Definition 5.1 by enforcing the additional constraint that the number of 1s in each column of S cannot exceed C. That is, B needs to form a *sparse basis* of D with sparsity C.

However, similar to the Boolean matrix factorization problem, this formulation admits a trivial solution – $B = D$ and $S = I$, and $k = n$. This solution is not particularly revealing in terms of potential patterns embedded in the traces, since each subtrace is constructed by itself. The following improved formulation addresses this weakness by using an optimization objective that encourages *sharing of basis subtraces*, i.e. each subtrace is constructed by multiple basis subtraces.

Definition 5.5. *Given* $D \in \mathbb{B}^{m \times n}$ *and a positive integer* C*, the* sparsity-constrained sharing-maximized Boolean matrix factorization problem *is to find* k*,* $B \in \mathbb{B}^{m \times k}$*, and* $S \in \mathbb{B}^{k \times n}$ *to*

$$\begin{aligned} \textit{maximize} \quad & \sum_i \sum_j S_{i,j}/k \\ \textit{subject to} \quad & D = B \circ S \\ & \|S_{\cdot,i}\|_1 \leq C, \ \forall_i \end{aligned} \tag{5.5}$$

A basis subtrace $B_{\cdot,i}$ is shared in the construction of more than one subtraces if $\sum_j S_{i,j} \geq 1$. Hence, by normalizing this quantity with the number of basis subtraces k, maximizing the objective function $\sum_i \sum_j S_{i,j}/k$ is equivalent to maximizing the total number of sharing across all basis subtraces. In addition, since the quantity $\sum_j S_{i,j}$ represents the *frequency* at which basis subtrace $B_{\cdot,i}$ is used in the construction of the subtraces, it can be used as a metric to *rank* the basis subtraces. In Section 5.4.2, we describe in detail an algorithm that heuristically solve this problem.

Remark 5.2. *The choice of the sparsity constraint* C *can influence the value of* B *and* S*, as a decomposition of* D*. When* C *is not available as an input, a meta-level optimization can be performed to find the best* C *(e.g., by sampling or searching over a specific range of values) such that the value of the objective function in Equation 5.5 is maximized.*

5.4.2 Solving the Sparse Coding Problem

In this section, we describe an algorithm that solves the *sparsity-constrained sharing-maximized Boolean matrix factorization problem*, as formalized in Equations 5.5. Our solution is guaranteed to satisfy the sparsity constraint and heuristically maximizes the objective function. The key idea of the algorithm is to exploit the connection between matrix factorization and biclique edge covering for bigraphs, as described in Section 5.2. Specifically, it is based on growing a biclique edge cover Cov for the bigraph $G_D = \langle U, V, E \rangle$ corresponding to the data matrix D. At each step, a maximal edge biclique that covers some number of previously uncovered edges is added to Cov until Cov covers all the edges. Each biclique $\langle U', V', E' \rangle$ added this way represents some basis subtrace $B_{\cdot,j}$ where $B_{i,j} = 1$ if $u_i \in U'$. Additionally, $S_{j,k} = 1$ if $v_k \in V'$, i.e. subtrace $D_{\cdot,k}$ uses $B_{\cdot,j}$ in its construction. Thus, the cardinality of

V' indicates the total number of times that this basis subtrace is used in the construction of observed subtraces. By choosing maximal bicliques to form the cover, we are essentially maximizing the sharing of basis. On the other hand, the sparsity constraint is a constraint on the maximum number of maximal bicliques that can be used to cover the edges that connect each vertex in V.

Observe that this algorithm relies on a way to generate maximal edge bicliques of a bigraph. Computing these bicliques is not easy: for instance, the closely-related problem of finding a maximum (not maximal) edge biclique in a bigraph is NP-complete [Pee03]. Additionally, the number of maximal bicliques in a bigraph can be exponential in the number of vertices [GKL08].

However, there exist enumeration algorithms that are polynomial in the combined input and output size, such as the Consensus algorithm in [AAC+04].

Algorithm 3 solves the sparsity-constrained Boolean matrix factorization problem by building upon some key concepts in the Consensus algorithm and adapting them for our problem context. These concepts are described below.

- **Consensus:** For two bicliques $B_1 = (U_1, V_1)$ and $B_2 = (U_2, V_2)$, the consensus of B_1 and B_2 is $B_3 = (U_3, V_3)$ where $U_3 = U_1 \cap U_2$ and $V_3 = V_1 \cup V_2$.
- **Extend to a maximal biclique:** For a consensus biclique $B_1 = (U_1, V_1)$, we can extend it to a maximal biclique $B_2 = (U_2, V_2)$ where $U_2 = U_1$ and $V_2 = \{v \mid \forall u \in U_1,\ (u, v) \in E\}$ (V_2 is the set of vertices in V that are connected to every vertex in U_1).
- **v-rooted star biclique:** A v-rooted star biclique B_v^{star} is the biclique formed by the node $v \in V$ and all the nodes connected to v (and the edges), i.e. $(\{u \mid (u, v) \in E\}, \{v\})$

The main idea of Algorithm 3 is the following. We try to cover the edges in the bigraph with as many maximal bicliques as possible, until we are about to violate the sparsity constraint at some vertex $v \in V$. In that case, we cover the remaining edges of v with the v-rooted star biclique. If there is still some $v \in V$ with uncovered edges at the end of the iteration, then we just cover it with the v-rooted star biclique as well. The final cover will be the union of the set of maximal bicliques added in the consensus steps $Cov_1 \setminus Cov_0$ with the set of star bicliques Cov_2.

5.4.3 Example Illustration

Consider the two-port arbiter example presented in Section 3.2. The router is simulated under an environment which randomly generates requests at the two input channels and holds a request until it has been granted. Figure 5.4 shows part of a trace of 100 cycles long over the request and grant signals of the arbiter. We used a sliding window of length 3 to collect a set of subtraces. In addition, a sparsity of 4, which is one third of the total number of entries in a subtrace, was used.

We ask the following two research questions.

Algorithm 3 Sparsity-constrained cover

1: **Input:** a set Cov_0 containing all B_v^{star} and a sparsity constraint C.
2: **Initialize:** $Cov_1 := Cov_0$, $Cov_2 := \emptyset$, $\alpha_v := C$, $\forall\, v \in V$, and $V_{cov} := \emptyset$.
3: **repeat**
4: Pick a new pair of bicliques $B_1 = (U_1, V_1)$ from Cov_1 and $B_2 = (U_2, V_2)$ from Cov_0, form the consensus B_3.
5: Extend B_3 to a maximal biclique $B_4 = (U_4, V_4)$.
6: **if** $(B_4 \notin Cov_1) \wedge (V_4 \cap V_{cov} = \emptyset)$ **then**
7: Add B_4 to Cov_1.
8: **for** $v \in V_4 \setminus V_{cov}$ **do**
9: $\alpha_v := \alpha_v - 1$
10: **if** $\alpha_v = 1$ **then** Add B_v^{star} to Cov_2 and add v to V_{cov} **end if**
11: **end for**
12: **end if**
13: **until** $E_{(Cov_1 \setminus Cov_0) \cup Cov_2} = E$ or cannot find a new pair of bicliques B_1 and B_2
14: **for** $v \in V \setminus V_{cov}$ **do**
15: Add B_v^{star} to Cov_2.
16: **end for**
17: **Output:** $(Cov_1 \setminus Cov_0) \cup Cov_2$ as the sparsity-constrained cover.

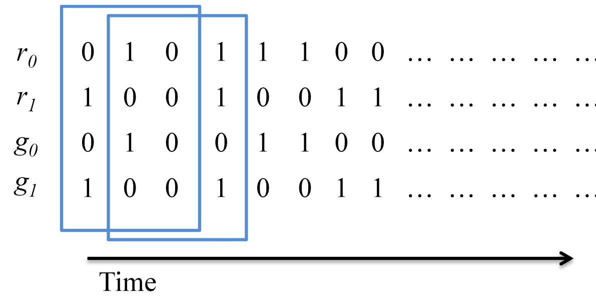


Figure 5.4: A normal trace of a 2-port round-robin arbiter

- Are the computed "basis subtraces" meaningful? That is, do they correspond to some relevant specifications of the underlying system?
- Do the "basis subtraces" capture sufficient underlying structure of a trace? That is, can they be used to construct traces that are generated from unseen input sequences?

Figure 5.5 shows the three top ranked (based on how frequent a basis is used in the construction of subtraces) basis subtraces, computed by using Algorithm 3.

Observe that basis subtraces (a) and (b) correspond to the behavior of the arbiter granting a request at the same cycle when there is no competing request. Basis subtrace (c) shows that when there are two competing requests at the same cycle, the arbiter first grants one

0	0	0
1	0	0
0	0	0
1	0	0

(a)

0	1	0
0	0	0
0	1	0
0	0	0

(b)

0	1	1
0	1	0
0	0	1
0	1	0

(c)

Figure 5.5: Three basis subtraces computed via sparse coding.

of the requests and the ungranted request will stay asserted the next cycle and then gets granted. Moreover, each basis subtrace exhibits a pattern *in isolation*; viz. the sparse-coding approach produces a meaningful *behavioral basis*.

In order to evaluate the *coverage* of the computed basis, i.e. whether the basis can construct unseen subtraces, we further simulated the arbiter with random inputs another 100 times each for 100 cycles. For each of these traces, we also used a sliding window of length 3 to partition them into subtraces. Using the basis computed from the previous trace, we tried to construct these subtraces and succeeded in every attempt. This means that many (if not all) of the sub-behaviors (of length 3) of the arbiter were fully covered by the computed bases, even though unseen subtraces were present in the new traces.

5.5 Application to Error Localization

The notion of basis subtraces as specifications naturally gives rise to a notion of correctness. Specifically, a subtrace is deemed correct if it can be constructed using the basis (with some sparsity). On the other hand, a subtrace can be viewed as erroneous if it cannot be constructed by the basis. Moreover, given known correct traces, a basis can be *mined* from these traces. This basis characterizes a space containing all the known correct subtraces and potentially more. The idea of using the mined basis to localize errors in a trace is thus to check if any segment of this trace (subtrace) cannot be constructed using the basis. Figure 5.6 illustrates this idea.

5.5.1 Problem Definition

Without loss of generality, consider the problem of localizing an error given a single correct trace and a single error trace, similar to the setup described in Chapter 4. Our goal is localize the error *in time* – identify a small interval of the timeline at which the error occurred. Particularly, we assume a setting where the input sequence that generated the error trace is either unknown (or only partially known) or it is extremely slow to re-simulate the input sequence (if known) on the correct design (also sometimes referred to as a "golden model").

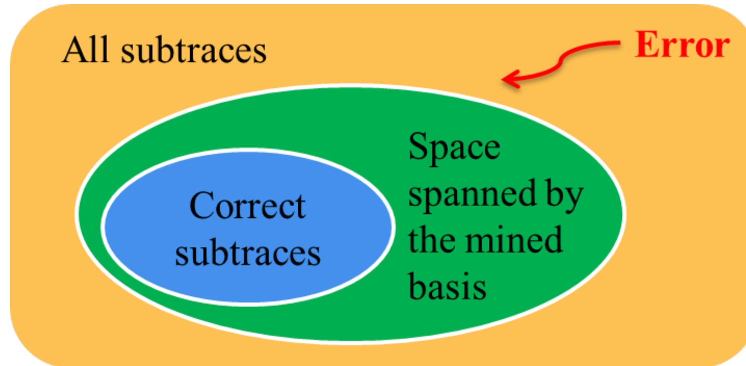


Figure 5.6: Subtrace correctness characterization using basis subtraces.

This means that a simple anomaly detection technique which checks the first divergence of the error trace and the correct trace obtained by simulating the golden model on the same input sequence does not work. One has to use the correct trace to help localize the bug in the error trace. This setting is especially applicable to post-silicon debugging where the bugs are often difficult to diagnose due to limited observability, limited reproducibility and susceptibility to environmental variations.

More formally, the error localization problem we address in this section can be defined as follows.

Definition 5.6. *Given an error trace of length l and an integer w, partition the trace into non-overlapping subtraces each of length w (without loss of generality, assume l is an integer multiple of w; otherwise, the last subtrace can be treated specially).*

Then, the error localization problem *is to identify the subtrace containing the first point of deviation of the error trace from the correct trace.*

One might note that the problem we define is not the only form of error localization that is desirable. For instance, one might also want to narrow down the fault to the signals/variables that were incorrectly updated. Also, there might be more than one source of an error, in which case one might want to identify all of the sources.

While these goals are important, we contend that our algorithm to address the problem defined above can also be used to achieve these additional objectives. For example, the error explanation technique we present below can be used to identify which variables were incorrectly updated and how. Similarly, one can apply our construction-based localization algorithm iteratively to identify multiple subtraces that cannot be constructed from the basis subtraces, and could potentially be used to identify multiple causes of an error.

5.5.2 Localization by Construction

As described above, the key hypothesis underlying our approach is that the earliest section (subtrace) of the error trace that cannot be constructed contains the likely cause of the error.

Our error localization algorithm operates in the following steps:

1. Given a correct trace τ, first obtain the set $\mathcal{T}^w$ of all *unique* subtraces of length w in τ. Using the approach described in Section 5.2, convert the set $\mathcal{T}^w$ to a data matrix D.

2. Solve the *sparsity-constrained sharing-maximized Boolean matrix factorization problem* for D given a sparsity constraint C. Obtain B as the basis matrix.

3. Given an error trace τ', partition it into an ordered set of q subtraces of length w. Denote this set by $\mathcal{T}^{w\prime}$. The elements in $\mathcal{T}^{w\prime}$ are ordered by their positions in τ'. Convert $\mathcal{T}^{w\prime}$ to a data matrix D'.

4. Starting from $D'_{\cdot,0}$, try to construct $D'_{\cdot,i}$ using the basis B computed above with the same sparsity constraint C. Return i as the location of the bug if the construction fails. In case all constructions succeed, return $\perp$ indicating inability to localize the error.

Algorithm 4 describes the above approach in more detail using pseudo-code. It uses the following subroutines:

- **dataMatrix** is the procedure that converts a set of subtraces to the corresponding data matrix described in Section 5.2.

- **sparseBasis** solves the *sparsity-constrained sharing-maximized Boolean matrix factorization problem* using the graph-theoretic algorithm presented in Section 5.4 for D with a given sparsity C, and returns the computed basis B.

- **constructTrace** solves the following minimization problem.

$$\begin{aligned} \underset{S_{\cdot,i}}{\text{minimize}} \quad & \|D'_{\cdot,i} \oplus (B \circ S_{\cdot,i})\|_1 \\ \text{subject to} \quad & \|S_{\cdot,i}\|_1 \leq C \end{aligned} \tag{5.6}$$

  where $\oplus$ is the bit-wise Boolean `XOR` operator, and is interpreted to apply entry-wise on matrices.

  Notice that for fixed C, this problem is *fixed-parameter tractable* because we can use a brute-force algorithm that enumerates all the $\sum_{1\leq i\leq C} \binom{k}{i}$ possible $S_{\cdot,i}$. It can also be solved using a pseudo-Boolean optimization formulation, where the Boolean variables in the optimization problem are the entries in $S_{\cdot,i}$.

Error Explanation. Denote $S^*_{\cdot,i}$ as the optimal solution to the minimization problem in Equation 5.6. If the minimum value is non-zero, then $E = D'_{\cdot,i} \oplus (B \circ S^*_{\cdot,i})$ is the minimum difference between the error subtrace $D'_{\cdot,i}$ and the constructed subtrace $B \circ S^*_{\cdot,i}$. Notice that E is also a subtrace, and can be interpreted as a finite sequence of assignments to the observed variables. We illustrate these ideas below.

Algorithm 4 Error localization in time

Input: Set of subtraces $\mathcal{T}^w$ from a correct trace τ, $\mathcal{T}^{w\prime}$ from an error trace τ'
Input: Integer $C > 0$
$D = \textbf{dataMatrix}(\mathcal{T}^w)$; $D' = \textbf{dataMatrix}(\mathcal{T}^{w\prime})$; $B = \textbf{sparseBasis}(D, C)$
for $i := 0 \rightarrow q-1$ **do**
  $E = \textbf{constructTrace}(D'_{\cdot,i},\, B,\, C)$
  if $E \neq \mathbf{0}$ **then** **return** τ'^{w}_{i} **end if**
end for
return $\perp$

5.5.3 Example Illustration

Consider again the setup of the two-port arbiter in Section 5.4.3. For each of the extra 100 traces, we randomly injected a single bit error (flipping its value) at a random cycle to one of the four signals in the trace. Our task was to test if we could localize the error to a subtrace of length 3 that contained it.

The following example illustrates one of the experiments. Figure 5.7 shows a snapshot of the trace.

	95	96	97	98	99
r_0	0	0	0	1	0
r_1	0	0	1	0	0
g_0	0	0	0	1	0
g_1	0	0	**0**	0	0

Figure 5.7: Bit flip on g_1 at cycle 97.

Using the approach described in Algorithm 4, the subtrace containing the error was correctly identified. Among the 100 traces, we successfully identified the window at which the error was injected for 84 of them. Figure 5.8 shows one the error subtrace identified for the trace used in Figure 5.7.

Following Equation 5.6, Figure 5.9a shows the (differential) subtrace $D'_i \oplus (B \circ S_i)$ that minimizes $|D'_i \oplus (B \circ S_i)|_1$. This subtrace is returned as an *error explanation*.

We argue that this subtrace reveals the injected error (without needing to assume a particular fault model) since the erroneous behavior is singled out – a grant was not produced at g_1 at cycle 97 even when the corresponding request was made at r_1.

	96	97	98
r_0	0	0	1
r_1	0	1	0
g_0	0	0	1
g_1	0	0	0

Figure 5.8: Error subtrace as identified.

	96	97	98
r_0	0	0	0
r_1	0	0	0
g_0	0	0	0
g_1	0	1	0

(a) Error explanation subtrace.

	96	97	98
r_0	0	0	0
r_1	0	1	0
g_0	0	0	0
g_1	0	0	0

(b) Alternative error explanation subtrace.

Figure 5.9: Two error explanations.

Remark 5.3. *Note that multiple error explanations (solutions to the minimization problem in Equation 5.6) can exist. Figure 5.9b shows an alternative error explanation subtrace for this example where g_1 was asserted but r_1 was not asserted at cycle 97. This is due to the symmetry of the `XOR` operator. We plan to investigate this further in future work.*

5.6 Results and Experiments

5.6.1 Theoretical Guarantees

We now give conditions under which our error localization approach is *sound*. By sound, we mean that when our algorithm reports a subtrace as the cause of an error, it is really an erroneous subtrace that deviates from correct behavior.

Since our approach mines specifications from traces, its effectiveness fundamentally depends on the quality of those traces. Specifically, our soundness guarantee relies on the set

of complete traces $\tilde{\mathcal{T}}$ (which induces traces $\mathcal{T}$ over observable variables $V^{t,o}$) satisfying the following *coverage metrics* defined over the transition system $M^t = (V^t, Q_0^t, \rho^t)$ of the golden model (the model itself may not be given):

- *Initial state coverage:* For every initial state $q_0 \in Q_0^t$, there exists some trace $\tilde{\tau} \in \tilde{\mathcal{T}}$ in which q_0 is the initial state, i.e. $q_0 = \tilde{\tau}_0$.
- *Transition coverage:* For every transition $(q, q') \in \rho^t$, there exists some trace $\tilde{\tau} \in \tilde{\mathcal{T}}$ in which the transition (q, q') occurs, i.e. there exists a i such that $\tilde{\tau}_i = q$ and $\tilde{\tau}_{i+1} = q'$.

While full transition coverage can be difficult to achieve for large designs, there is significant work in the simulation-driven hardware verification community on achieving a high degree of transition coverage [TK01]. If achieving transition coverage is challenging for a design, one could consider slicing the traces based on smaller module boundaries and computing tests that ensure full transition coverage within modules, at the potential cost of missing cross-module patterns.

Our soundness theorem relates test coverage with effectiveness of error localization.

Proposition 5.1. *Given a transition system M^t and a set of finite-length and complete traces $\tilde{\mathcal{T}}$ of M^t satisfying initial state and transition coverage, let $\mathcal{T}$ be the projection of $\tilde{\mathcal{T}}$ over the observable variables $V^{t,o}$ and let $\mathcal{T}^2$ be the set of all distinct subtraces of length 2 in $\mathcal{T}$, we have $\mathcal{T}^2 = \mathcal{L}^2_{V^{t,o}}(M^t)$.*

It is easy to see that Proposition 5.1 is true by the definitions of *initial state coverage*, *transition coverage* and $\mathcal{L}^2_{V^{t,o}}(M^t)$.

Theorem 5.1. *Given a transition system M^t for the golden model and a set of finite-length traces $\mathcal{T}$ over observable variables $V^{t,o}$ induced by complete traces $\tilde{\mathcal{T}}$ of M^t satisfying initial state and transition coverage, if Algorithm 4 is invoked on $\mathcal{T}$ and an arbitrary error trace τ' with $p = 2$, then Algorithm 4 is sound; viz., if it reports a subtrace of τ' as an error location, that subtrace cannot be exhibited by M^t.*

Proof. The proof proceeds by contradiction. Suppose Algorithm 4 reports a subtrace τ^2 of τ' as the location of the error. If τ^2 were to be exhibited by M^t, then $\tau^2 \in \mathcal{L}^2_{V^{t,o}}(M^t)$.

The procedure **sparseBasis** produces a sparse basis B for $\mathcal{T}^2$ with sparsity C. By Proposition 5.1, $\mathcal{T}^2 = \mathcal{L}^2_{V^{t,o}}(M^t)$. Hence, B forms a sparse basis of $\mathcal{L}^2_{V^{t,o}}(M^t)$ with sparsity C. By Definition 5.3, this means that B can construct τ^2. However, by the definition of the procedure **constructTrace**, we know that B cannot construct τ^2. Therefore, by proof of contradiction, τ^2 cannot be exhibited by M^t. □

We also note that, in theory, it is possible for Algorithm 4 to miss reporting a subtrace that is an error location, due to the fact that B may construct subtraces not in $\mathcal{L}^2_{V^{t,o}}(M^t)$. However, experiments, as presented in the next section, indicate that it is usually accurate in pinpointing the location of an error.

5.6.2 Case Study

In this section, we present a case study on the CMP router described in Section 4.5.1 that demonstrates the usefulness of the sparse-coding approach when applied to error localization. Particularly, we assume a black-box setting where one needs to localize the error(s) solely based on observed traces of the design under analysis. The main goal of this case study is to explore how the technique scales to a larger design, and how effective it is for error localization.

The router was simulated with two flit-generating modules that each issued random data packets (each consists of a head, some body and a tail flit) to the respective input ports of the router. We observed 14 Boolean control signals in the router and a trace was generated for these 14 signals with a simulation length of 1000 cycles. We used a subtrace width of 2 cycles and obtained 93 distinct subtraces each with 14 signals over 2 cycles. A basis was computed from these 93 distinct subtraces subject to a sparsity constraint of 52 (see explanation for the choice of this number at the end of this section). It took 0.243 seconds on a laptop machine with an Intel i7 2.70 GHz processor and 8.0 GB of RAM to obtain this basis which contained 189 basis subtraces.

The router was simulated 100 times with different inputs. We used the first simulation trace to obtain the basis as described in the previous paragraph and the rest 99 traces for error localization. For each of these 99 traces, a single bit flip was injected to a random signal at a random cycle. The goal of this experiment is to localize this bit error to a subtrace of 2 cycles (among the 999 subtraces for each trace) in which the error was introduced.

Following the localization approach described in Section 5.5.2 of the paper, 55 out of 99 of the errors were correctly localized. The remaining 44 errors were not localized (all the subtraces including error subtrace were constructed using the computed basis). The overall accuracy of the error localization procedure in this experiment was 55.6%.

Why is this error localization approach useful? Imagine you are given a good trace (or a collection of good traces) and then an error trace (that cannot be reproduced), and you are asked to localize the error without knowing very much about the underlying system that generates these traces (this situation arises when dealing with legacy systems, for example). Here are two plausible alternative options to our sparse coding approach and the corresponding results:

(a) Hash all the distinct subtraces of 2 cycles in length in the good trace. For each of the subtraces of the same dimension in the bad trace, check if it is contained in the hash, and report an error if it is not contained. For the same traces used above, an error was reported for each of the 99 traces *even before any bit flip was injected.*

(b) Use a basis that spans the entire space of subtraces of 2 cycles, e.g. 14×2 subtraces where each contains only a single 1 in its entries and is orthogonal to the others. However, it is obvious that we cannot localize any error using this basis since it spans all possible subtraces.

Our method can be viewed as something in between (a) and (b). It finds a subspace that not only contains all the good subtraces but also generalizes well to unseen good subtraces from the basis. The generalization is a sparse composition of some key patterns in the good subtraces. An error is reported if a subtrace lies outside this subspace, as illustrated in Figure 5.6. The number 52 for the sparsity constraint was chosen as the value that maximizes the objective function of the optimization problem in Equation 5.5, amongst multiple runs of Algorithm 3 with different Cs.

5.7 Additional Related Work

In this section, we review additional work related to Boolean matrix factorization and bug localization.

5.7.1 Boolean Matrix Factorization

Matrix factorization or factor analysis methods are prevalent in the data mining community, with a common goal to discover the latent structures in the input data. While most of these methods are focusing on real-valued matrices, there have been several works recently that target Boolean matrices, for applications such as role mining [VAG07]. Miettinen et al. [MMG+06] introduced the discrete basis problem (DBP). DBP is similar to our definition of the Boolean matrix factorization problem in which k is fixed and the objective is to minimize the reconstruction error. They showed that DBP is NP-hard and gave a simple greedy algorithm for solving it. In terms of sparse decomposition, Miettinen [Mie10] showed the existence of sparse factor matrices for a sparse data matrix. Our paper describes a different notion of sparsity – we seek to express each data vector as a combination of only a few basis vectors, which can be dense themselves.

5.7.2 Bug Localization

The problem of bug localization and explanation has been much studied in literature in several communities: software testing, model checking, and electronic design automation. In model checking, Groce et al. [GCKS06] present an approach based on distance metrics which, given a counterexample (error trace), finds a correct trace as "close" as possible to the error trace according to the distance metrics. Ball et al. [BNR03] present an approach to localizing errors in sequential programs. They use a model checker as a subroutine, with the core idea to identify transitions of an error trace that are not in any correct trace of the program, and use this for error localization. Both of these approaches operate on error traces generated by model checking, and thus have full observability of the inputs and state variables. In contrast, in our context, the trace includes only-partially observed state and is not reproducible.

In the software testing community, researchers have attempted to use predicates and mined specifications to localize errors [LAZJ03, DLE03]; however, these rely on human insight in choosing a good set of predicates/templates. In contrast, our approach automatically derives specifications in the form of basis subtraces, which can be seen as temporal properties over a finite window. Program spectra [HRS+00], which include computing profiles of program behavior such as summaries of the branches or paths traversed, have also been proposed as ways to separate good traces from error traces; however, these techniques are of limited use for digital circuits since they rely on the path structure of sequential programs and give no guarantees on soundness.

In the area of post-silicon debugging (see [MSN10] for a recent survey), the problem of bug localization has received wide attention. The IFRA approach [PBWM10], which is largely specialized for processor cores, is based on adding on-chip recorders to a design to collect "instruction footprints" which are analyzed offline with some input from human experts. Zhu et al. [ZWM11] propose a SAT-based technique for post-silicon fault localization, where *backbones* are used to propagate information across sliding windows of an error trace. This additional information helps make the approach more scalable and addresses the problem of limited observability. Backspace [dPGH+08] addresses the problem of reproducibility by attempting to reconstruct one or more "likely" error traces by performing backward reachability guided by recorded signatures of system state; such a system is complementary to the techniques proposed herein for bug localization.

5.8 Summary

In this chapter, we have presented a novel specification formalism based on the notion of *basis subtraces*, to capture system behavior from simulation or execution traces. We showed how to compute a *sparse basis* from a set of traces using a graph-theoretic algorithm. We further demonstrated that the generated basis subtraces can be effectively used for error localization and explanation.

Chapter 6

Crowdsourced Specification Mining

The human brain is an incredible pattern-matching machine.

– Jeff Bezos

In this chapter, we propose the use of crowdsourcing and gamification to help solve the specification mining problem. Many existing behavioral or specification mining techniques rely on the use of templates [EPG+07, GS08a, LFS10]. Hence, it is the user's responsibility to come up with a good set of templates. This process requires expert insight and is often incomplete. We introduce a web-based game called *CrowdMine*, that leverages human's natural ability to recognize patterns in images which encode simulation traces of a design, to assist the process of mining specifications. We show that the patterns collectively identified by non-expert humans, who were oblivious to the underlying task, match well with the desired behaviors of the design under analysis.

In Section 6.1, we give background on human computation and identify problems in verification that can benefit from human insights. In Section 6.2, we describe in detail the design of two versions of *CrowdMine* and present experimental results. Lastly, in Section 6.3, we discuss lessons learned and related issues such as privacy and incentive mechanisms.

6.1 Introduction

The field of electronic design automation (EDA), in general, and formal verification, in particular, has relentlessly pushed for automation. For several problems, this is indeed the right strategy. But for many problems, human insight and involvement remain invaluable. Consider, for example, the process of verifying a design. First of all, one needs to write a specification, typically in the form of properties (assertions) or a reference model. Second, one must create an environment model, typically in the form of constraints on the inputs or a state machine description. Next, one runs the verifier, such as a model checker, which is usually thought of as a "push-button" technique. While this is largely true, human insight is not entirely absent; e.g., one might need to supply hints to the verifier in the form of

suitable abstraction techniques or (templates for) inductive invariants. If the verifier returns with a counterexample trace, one must debug the design by localizing the cause of error in time (relevant part of the trace) and space (relevant part of the design). Finally, the process of repairing the design to eliminate the bug is also one that needs human input. To summarize, even after decades of work on automating the verification process, we continue to need human insight in a variety of tasks, including writing specifications, creating models, guiding the verification engine, and localizing and debugging errors.

This work takes the position that while we cannot completely remove human insight from the verification process, we can change the way humans provide insight to the verifier. Today, such input typically comes from expert verification engineers, trained in the tools of their field. But such experts are few and expensive. And even experts have a hard time answering questions such as: When are we done verifying? Have we written enough properties? Where is the bug? And so on. We contend that the experts and automated tools can be assisted in the verification process by a large crowd of non-expert humans *performing simple and repetitive tasks*. Each task involves a pattern recognition or other cognitive operations that humans are typically good at. The main technical challenges are to identify steps in the verification process where human insight is critical, find ways to transform these steps into tasks that non-expert humans can perform, and combine the results to resolve those steps in the verification process. As preliminary evidence to show that these challenges can be met, we present a game called CrowdMine for finding specifications from traces based on pattern recognition by humans.

The idea of tapping into a crowd of humans to assist in a computational task is not new. *Crowdsourcing* is the act of taking a job traditionally performed by a designated agent (usually an employee) and outsourcing it to an undefined, generally large group of people in the form of an open call [How]. *Human computation* is a paradigm for utilizing human processing power to solve problems that computers cannot yet solve [vA05]. (See Quinn and Bederson [QB11] for a more detailed description of these and related terms.) Our proposal is to use a combination of crowdsourcing and human computation to improve the state-of-the-art in verification. The availability of tools like Amazon's Mechanical Turk [tur] and TurKit [LCGM10] make such a combination easier to deploy today.

In recent years, others have also advocated the use of crowdsourcing and human computation in design and verification, both for hardware and software. DeOrio and Bertacco [DB09] propose having humans assist in solving NP-complete problems arising in EDA, such as Boolean satisfiability (SAT) solving. Schiller and Ernst [SE10] propose the use of crowdsourcing and human computation for solving problems in software engineering, including software verification. The important difference between our proposal and these works is that we target steps in the verification process that *already* require human input, and which we think are unlikely to be automated entirely (similar to hard AI problems in the class of passing the Turing test, but unlike many NP-hard problems). We seek to leverage crowdsourcing and human computation to scale up the productivity in these steps manyfold.

To summarize, we make the following contributions:

- Advocate the use of crowdsourcing and human computation for sub-tasks in verification that require human insight;
- Demonstrate the idea through CrowdMine, a novel game devised for finding patterns from system traces that can suggest likely specifications, and
- Sketch out the landscape of similar applications.

6.2 CrowdMine – Game Design

The main idea of CrowdMine is to tap into the human ability to recognize patterns in images to assist the process of mining specifications. For example, a trace can be visualized as a 2D image, where the rows are signals and the columns are cycles. CrowdMine first transforms segments of a trace into images and then queries a non-expert crowd to identify common patterns in those images. We have designed a game, described below, that incorporates these ideas.

6.2.1 CrowdMine1: An Open-Loop Design

The first prototype of CrowdMine is an open-loop design. Figure 6.1 shows an overview of CrowdMine1.

First, we decompose traces of a design into a collection of subtraces, each of length w, as described in Section 3.1. In this example, the design is the arbiter example we described in Section 3.2. In Figure 6.1, $w = 4$ and thus each image represents the values of the two input signals r_0, r_1 and two output signals g_0, g_1 in 4 consecutive cycles. The signals r_0, r_1, g_0, g_1 are ordered from bottom to top, with r_0 at the bottom. There are many ways to color-code the signals. In this example, we choose the following scheme, to enhance the contrast between inputs and outputs as well as the contrast between taking the values 1 and 0.

- If $g_i = 0$, the corresponding square is colored red.
- If $g_i = 1$, the corresponding square is colored yellow.
- If $r_i = 0$, the corresponding square is colored blue.
- If $r_i = 1$, the corresponding square is colored teal.

Now we are ready to explain a session of the game.

Game objective: The game instructs the players to find a common pattern amongst the three displayed images. A *pattern* is simply a collection of squares. Two patterns are considered equal if one pattern can be obtained from the other by a horizontal translation (but not by or combined with a vertical translation, since the patterns would then map to different

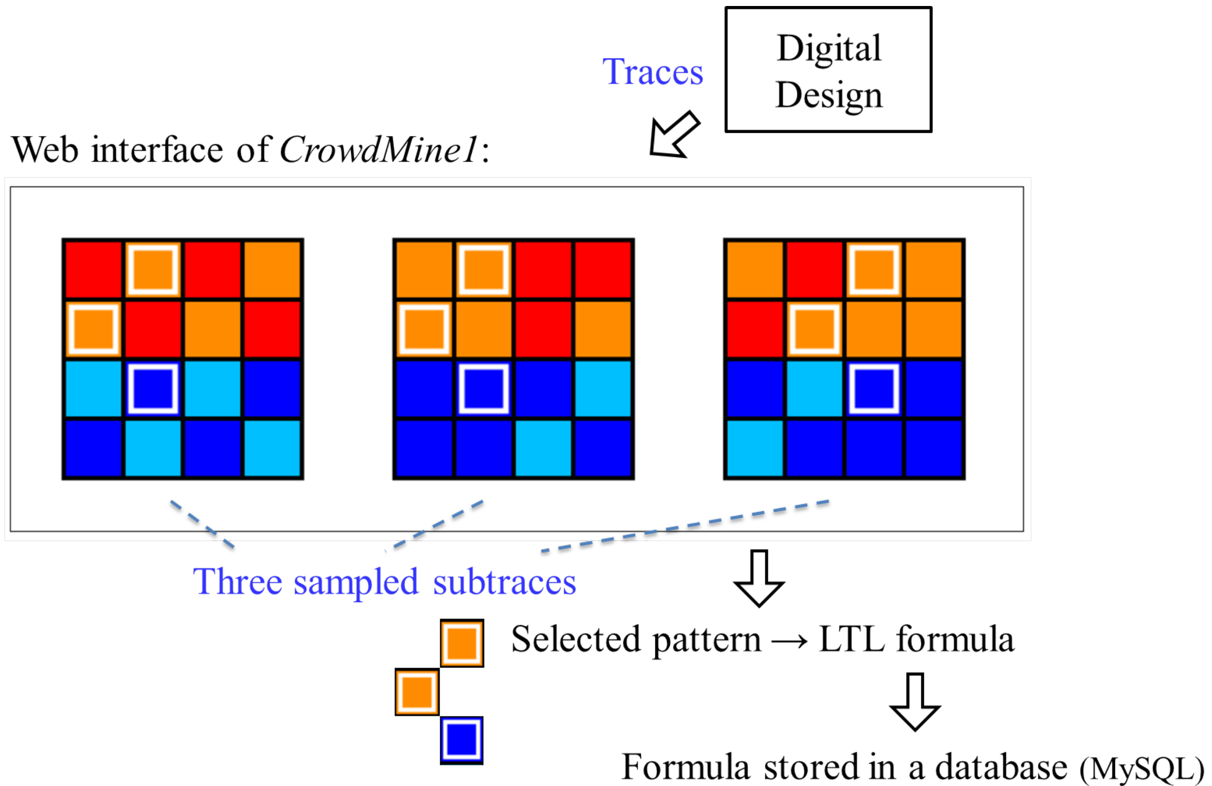


Figure 6.1: Overview of CrowdMine1.

signals). To obtain more interesting results, we further enforce that a pattern must consist of at least two squares in one image.
GUI Design: The web interface is done is HTML5 and Javascript. A player can select a square by clicking the corresponding area. The selected square will be highlighted, as shown in Figure 6.1, by a white box. The player can de-select the same square by clicking it again. When the player is done selecting the patterns in the three images, she can click the "Done" button. However, if the patterns are not the same, a message will appear instructing the player to modify the current selection.
Back-end: If the pattern selected is a common pattern amongst the three images, it is sent to the back-end. The back-end contains a MySQL database that stores the patterns and how many times each of them is found by the players. The back-end also has a simple script that converts a pattern into its corresponding LTL formula. We describe this translation in more detail in Section 6.2.2.
Experimental evaluation:

The game can be played online at http://verifun.eecs.berkeley.edu/crowdmine/. We collected data from anonymous players on the web over a period of approximately one week. Instructions of how the game should be played are available on the URL. The partic-

ipants were not informed about the underlying objective and the mapping from subtraces to images. A total of 165 common patterns were identified, out of 283 successful attempts. The top three ranked patterns, in terms of the number of times that they were found, are shown in Figure 6.2.

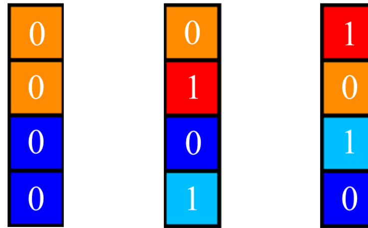


Figure 6.2: Top ranked patterns obtained in CrowdMine1.

For each pattern, its interpretation (given as LTL formulas) and the number of times it was identified, are given below.

- Left: $\mathbf{G}\ ((\neg r_0 \wedge \neg r_1) \rightarrow (\neg g_0 \wedge \neg g_1))$. This pattern was found 31 times.
- Middle: $\mathbf{G}\ ((r_0 \wedge \neg r_1) \rightarrow (g_0 \wedge \neg g_1))$. This pattern was found 16 times.
- Right: $\mathbf{G}\ ((\neg r_0 \wedge r_1) \rightarrow (\neg g_0 \wedge g_1))$. This pattern was found 7 times.

For this example, observe that all three formulas, which correspond to the most frequently identified patterns, match desired behaviors of the arbiter. However, we also note that a majority of the other identified patterns do not correspond to any meaningful behavior – not a property satisfied by the arbiter. This means a lot of human computation cycles were wasted on finding irrelevant patterns. In the next section, we describe an improved version of CrowdMine which uses a closed-loop design to ensure that any stored formula is satisfied by the underlying circuit.

6.2.2 CrowdMine2: A Closed-Loop Design

CrowdMine2 is the second prototype of CrowdMine, which can be played at `http://verifun.eecs.berkeley.edu/crowdmine2/`. Figure 6.3 shows an overview of CrowdMine2.

The main difference of CrowdMine2 from CrowdMine1 is the addition of a model checker in the loop. Instead of three images, the player is first asked to find a common pattern between only two images (as shown in the left window in Figure 6.3). After a selected pattern is converted to its corresponding formula, a model checker is used (the game currently uses Cadence SMV [smv]) to determine if the underlying design satisfies the formula. If the formula is satisfied, it is then stored in a MySQL database. Otherwise, the model checker produces a counterexample, which is then displayed as a third image in the window on the right. Now the player has to find a common pattern amongst the three images. The play

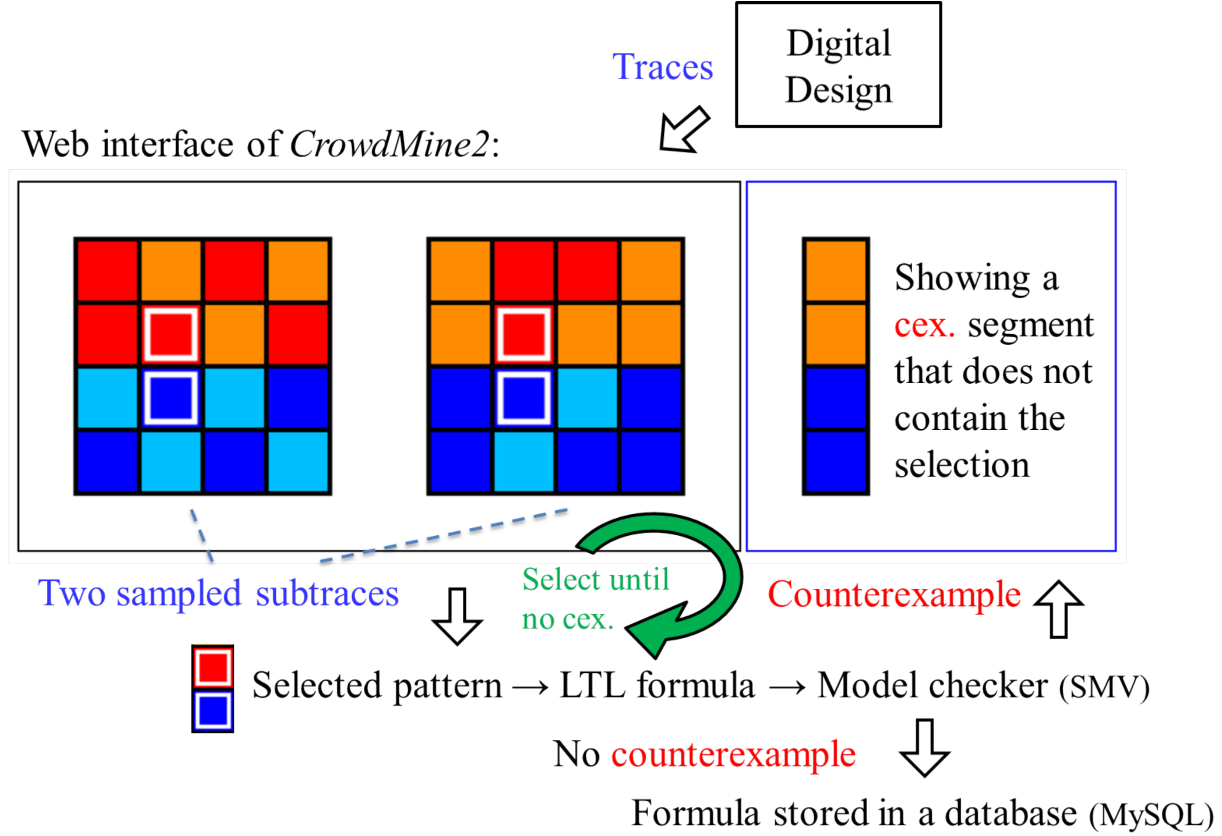


Figure 6.3: Overview of CrowdMine2

continues until an identified pattern is verified by the model checker. At any point in the game, the player can also click the "Reset" button to play with a fresh set of images. We detail two key processes in CrowdMine2 below.

Translation from patterns to LTL formulas: A square either represents an input literal or an output literal. We use l_i to represent the literal corresponding to a square i in the selected pattern. If $i \in \bar{X}$, where $\bar{X}$ is the set of squares corresponding to input values, then l_i is an input literal. Similarly, we use $\bar{Y}$ to denote the set of squares in the selected pattern that corresponds to output values. Since every pattern has at least one left-most square, we additionally use superscript t in l_i^t to denote the position of the literal starting at 0 from the left-most literal. An LTL formula is then generated from a selected pattern consisting of a set of squares as follows.

$$\mathbf{G}\Big(\big((\bigwedge_{i\in\bar{X}} l_i^0) \wedge (\mathbf{X}\bigwedge_{i\in\bar{X}} l_i^1) \wedge (\mathbf{XX}\bigwedge_{i\in\bar{X}} l_i^2) \wedge (\mathbf{XXX}\bigwedge_{i\in\bar{X}} l_i^3)\big) \rightarrow \\ \big((\bigwedge_{i\in\bar{Y}} l_i^0) \wedge (\mathbf{X}\bigwedge_{i\in\bar{Y}} l_i^1) \wedge (\mathbf{XX}\bigwedge_{i\in\bar{Y}} l_i^2) \wedge (\mathbf{XXX}\bigwedge_{i\in\bar{Y}} l_i^3)\big)\Big)$$

In this case, a selected pattern is interpreted as a conjunction of input literals over a bounded window implying a conjunction of output literals over the same window.
Displaying a counterexample: The counterexample produced by the model checker can sometimes be much longer than the length w of a subtrace ($w = 4$ in both CrowdMine1 and CrowdMine2), or even has infinite length. In order to be consistent with the sampled subtraces, we search in the counterexample for a segment as large as a subtrace that contradicts the selected pattern. In general, a counterexample τ_c can be conveniently viewed as a lasso-shaped trace consisting of a prefix of length k_1 and a loop suffix of length k_2. In the case of finite traces, the length of the loop suffix is zero, i.e. $k_2 = 0$.

We first construct a finite trace τ_f of length $k_1 + k_2 + w - 1$ such that this finite trace is guaranteed to contain all subtraces up to length w that are possible in the counterexample τ_c. Figure 6.4 illustrates this construction.

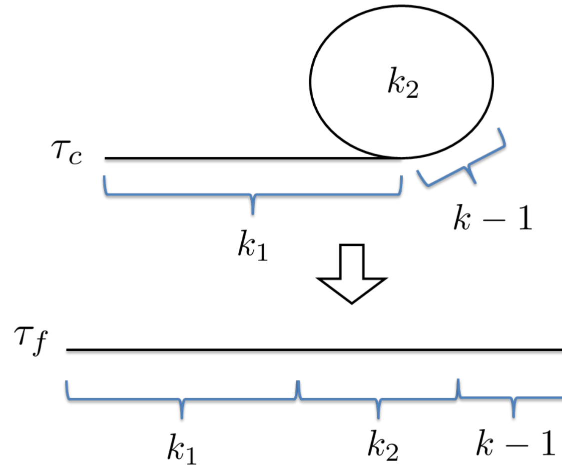


Figure 6.4: Finding contradicting subtraces in a counterexample.

We then search for a subtrace in τ_f with width as large as w such that it contradicts with the pattern. Using the same color-coding scheme for converting a sampled subtrace to an image, we convert the counterexample segment to the third image and display it in the window on the right.

6.3 Discussion

In this section, we summarize our experiences and findings with both versions of CrowdMine, and discuss related issues on crowdsourcing verification in general.
Lessons learned: Gamification can be generally thought of as a process of engaging non-expert users in solving problems through playing games. CrowdMine is a specific instance of this where the problem of specification mining is turned into a simple pattern-finding puzzle. Naturally, CrowdMine also inherits some of the difficulties in gamification. Notably,

not every task admits a natural game transformation. For example, in CrowdMine, the information that the rows correspond to signals and columns correspond to cycles (and additionally cycle count increases from left to right) are not made available to the players. When mapping a pattern to an LTL formula, the pattern is interpreted as having its input part implies the output part. This only makes sense if there is always an input signal in the pattern that either precedes or occurs at the same cycle as an output signal in the pattern. This constraint is however not enforced in CrowdMine. On the other hand, a signal is color-coded only based on its value but independent of the cycle number. Hence, the property that the same temporal pattern still holds if it is shifted in time is preserved.

In a practical setting, a design can have hundreds and thousands of signals and a trace of the design can easily be millions cycles long. Similar to the sparse coding approach presented in Chapter 5, CrowdMine focuses on mining temporal patterns over a short time window. These patterns, in turn, can be viewed as events and the approach presented in Chapter 4 can then to be used to mine temporal properties spanning a much larger timespan over these events. We plan to investigate this particular synergy in future work. To further cope with the large number of signals, a trace can be partitioned based on modules (in the same way that was described in Chapter 4). If the number of input and output signals is still large, a subset of them (e.g., 4) can be randomly sampled so that the simplicity and accessibility of the CrowdMine game is maintained.

Human inputs are often noisy. Without a way to filter out or correct the noisy input, the yield of the game output can be low – a majority of the patterns identified in CrowdMine1 were not useful patterns. To address this problem, CrowdMine2 employs a closed-loop design such that a model checker is used to ensure that any pattern stored at the end is a specification of (satisfied by) the underlying design.

Lastly, since subtraces are selected randomly, frequent patterns are also likely to appear more frequently than other patterns. Hence, it may take a long time for the players to discover a less frequently occurring pattern. To mitigate this problem, we have added an extra window containing three "invalid patterns". These are patterns which a player must not choose during the game. We use the three most frequently identified patterns over a period of time as these "invalid patterns". This feature thus forces the players to look for other patterns. We have incorporated it into the current version of CrowdMine1.

Future landscape: CrowdMine is only our first step in leveraging human computation to assist verification tasks. We specifically focus on the problem of mining specifications (from traces of digital circuits), which is the subject of this thesis. However, crowdsourcing and gamification can be applicable to other problems in verification, such as the problem of finding loop invariants that help speed up program verification. For example, DARPA's Crowd Sourced Formal Verification program [csf] is an initiative that aims to make formal verification more cost effective by enabling non-experts to assist in the formal verification process. Below, we discuss some of the emergent issues related to crowdsourced verification and sketch out the landscape for future work.

- Privacy. Our design is particularly attractive for companies that value confidentiality

because the internals of the circuit are not revealed. For IP protection, the mapping of subtraces to images should be kept confidential. This mapping include the correspondence of signals in the circuit, the color code, and any additional transformation on the subtraces. Randomization and sampling can be used in selecting subtraces and the mapping to images. Finally, *secret sharing* methods such as the *threshold schemes* developed by Shamir [Sha79] are particularly relevant in this context.

- Incentives. Three mechanisms are possible. (1) *Necessity:* Authentication systems such as reCAPTCHA [vA05] embed queries into a human challenge with partially known answers. Our game design can be augmented for this purpose. For example, two plays are presented to the user in series in which the answer is known for one of the plays. (2) *Enjoyment:* Our game design can be viewed as a puzzle game and the player derives enjoyment by solving it. In addition, the scoring-based system invites human competition and can attract a larger crowd. (3) *Profit:* Platforms such as Amazon's Mechanical Turk [tur] provide a *for-profit* medium for crowdsourcing any human intelligence task. We plan to deploy our game on the Mechanical Turk to evaluate the proposed approach in the future.

- Human-Computer Collaboration. It is possible to combine algorithmic techniques with inputs from humans to achieve something better than what can be accomplished by either solely humans or a completely automated approach. In our setting, the human-identified patterns can be further refined (e.g. ranked) to produce the most relevant ones based on feedback from the back-end verification and debugging processes. They can also be used in automated tasks such as the problem error localization discussed in Chapter 4 and 5.

- Looking beyond specification mining. We believe several games similar to CrowdMine can be created and applied to a range of applications in verification, debugging, and related areas. For example, one can improve coverage of a design by properties (or tests) by highlighting parts of a trace corresponding to variables not covered by (enough) properties, and users can be provided incentives to find patterns involving those parts. Properties generated by a system like CrowdMine can be hypothesized as auxiliary inductive invariants to speed up verification. Finally, human-observed patterns in spurious counterexamples could potentially enable better abstraction-refinement in model checking. Finally, the process of debugging has similarities to investigating a crime scene (!) – the "crime" is the manifestation of the error (the failure), and one seeks to find a cause-and-effect chain that explains how the failure happened; this analogy suggests a natural game that could be formulated for non-expert humans to assist in debugging.

Part II

Assumption Mining for LTL Synthesis

Chapter 7

Background

Part II of this thesis focuses on the problem of synthesis from LTL specifications as well as its applications. In this chapter, we first give a brief history of the synthesis problem, and motivate the need for assumption mining to address the problem of specification incompleteness. Then, we present the relevant formalisms and notations in Section 7.1. In Section 7.2, we survey related work on debugging specifications in the context of synthesis from temporal logic.

Synthesis, and specifically synthesis from logic specifications, refers to the proposition that one can automatically synthesize an implementation from its specification. First formalized by Church [Chu57] in 1957 and now commonly known as Church's Problem, the *synthesis problem* for sequential circuits can be stated as follows.

Given input signals X and output signals Y, and a formula ψ over $X \cup Y$, construct, if possible, a sequential circuit $\mathcal{C}$ with inputs X and output Y such that $\mathcal{L}(\mathcal{C}) \subseteq \mathcal{L}(\psi)$.

Church [Chu62] considered using different fragments of *restricted recursive arithmetic* (S1S) to specify ψ. Church's problem was solved in the following years independently by Büchi and Landweber [BL69] in 1969 for specifications in monadic second-order logic (MSO)[1] over $(\mathbb{N}, <)$ and by Rabin in 1972 [Rab72] using tree automata.

However, specifying the behavior of a sequential circuit, and reactive systems in general, in MSO can be cumbersome. Temporal logic[2] was proposed by Pnueli in 1977 [Pnu77] as an alternative for specification language. Since its introduction, temporal logic has been widely adopted in the formal verification community. Particularly, *linear* temporal logic (LTL) later forms the basis of Accellera's Property Specification Language (PSL) [EF06]. The introduction of temporal logic also spurred further development in the synthesis problem. Notably, Pnueli and Rosner [PR89] provided a solution based on infinite trees and proved that synthesis from LTL specifications is 2EXPTIME-complete [Ros92]. While this theoretical solution has existed for a period of time, its computational intractability has essentially made

[1] S1S is the logic of MSO over ω-words.

[2] The semantics of LTL is subsumed by that of MSO.

this approach practically irrelevant. To overcome the complexity issue of LTL synthesis, attempts have been made to identify subsets of the language that allows a more efficient synthesis algorithm [WHT03, ALT04].

More recently, Piterman et al. [PPS06a] proposed an efficient symbolic algorithm for synthesizing sequential circuits from a subclass of LTL known as Generalized Reactivity (1) [GR(1)] specifications. In particular, the algorithm runs in $O(N^3)$ time where N is the size of the state space of the design. Subsequent research shows that GR(1) specifications are expressible enough to cover many properties used in practice over a wide spectrum of applications, ranging from digital circuits [BGJ+07b, BGJ+07a], to robot mission planners [KGFP07, WTM09], to reactive protocols in an aircraft electric power distribution system [XTM12], to controllers in an industrial automation setting [CGR+12]. Hence, GR(1) specifications present a good trade-off between algorithmic efficiency and expressiveness.

A lesser studied but arguably more important problem is the problem of specification. While LTL synthesis offers the attractive proposition that one can automatically construct a functionally correct system from its behavioral description, it is important to note that the correctness of the system now solely relies on the correctness of its specification. The challenge of ensuring the correctness of specification is two-fold. First, specification must be formalized properly in such as way that it reflects the original design intent. Second, every aspect of the intended behavior of the design must be specified. As noted in [BGJ+07b, BGJ+07a], non-trivial efforts were involved in formalizing the informal (and not quite bug-free) specification to GR(1), and it was even more difficult to write a complete specification. According to the authors, "Many aspects of the arbiter are not defined in ARM's standard." The problem of buggy or incomplete specification in synthesis also resonates with the experience of using formal specification in verification. According to a report from IBM Haifa [BBDER97], during the first formal verification runs of a new hardware design, typically 20% of the formulas were found to be trivially valid.

Hence, there is an urgent need for developing techniques and tools to detect and fix bugs in specifications. One common phenomenon of buggy specification is that the specification is *unrealizable*[3] (no implementation exists that can satisfy the specification). In general, unrealizability can come from over-constrained system assertions or insufficient environment assumptions. Our thesis focuses on the latter problem because environment assumptions are especially tricky to get right in practice as these are often implicit knowledge and seldom documented. In Section 7.2, we present a more detailed survey on related work that addresses buggy specifications.

This part of the thesis also expands the envelope on the potential of temporal logic synthesis. In Chapter 9, we present a novel application of GR(1) synthesis, in conjunction with our assumption mining technique, that enables the automatic synthesis of a new class of semi-autonomous controllers. These controllers, called human-in-the-loop (HuIL) controllers, are crucial components in the emerging revolution of car automation.

Finally, not every designer or verification engineer is a logician. In Chapter 10, we present

[3]Definition of unrealizability can be found in Section 2.3.2.

our effort on liberalizing the use of formal specification via natural language processing. In particular, we demonstrate that assumption mining can be useful in debugging requirements written in English, supported by a case study on a portion of a publicly available document released by the Federal Aviation Administration.

7.1 Synthesis from GR(1) Specifications

In this section, we formally present the problem of synthesis from GR(1) specifications. We begin by introducing the syntax of GR(1). Then, we review materials on the game-based algorithms for synthesizing and analyzing GR(1) specification.

7.1.1 Generalized Reactivity (1) Specifications

A Generalized Reactivity (1) [GR(1)] formula is a subclass of LTL with restricted syntax [PPS06b]. A GR(1) formula has the form $\psi = \psi^{env} \rightarrow \psi^{sys}$, where ψ^{env} is the *environment assumption* and ψ^{sys} is the *system guarantee*. The syntax of GR(1) formulas is given as follows. We require ψ^l for $l \in \{env, sys\}$ to be a conjunction of sub-formulas in the following forms:

- ψ^l_i: a Boolean formula that characterizes the initial states.
- ψ^l_t: an LTL formula that characterizes the transition, in the form $\mathbf{G}\ f$, where f is a Boolean combination of variables in $I \cup O$ and expression $\mathbf{X}\ u$ where $u \in I$ if $l = env$ and $u \in I \cup O$ if $l = sys$.
- ψ^l_f: an LTL formula that characterizes fairness, in the form $\mathbf{G}\ \mathbf{F}\ f$, where f is a Boolean formula over variables in $I \cup O$.

Note that given a DBW $\mathcal{A}$, one can produce ψ_i, ψ_t and ψ_f that are essentially symbolic encoding of the initial state q^b_0, the transition function ρ^b and the acceptance condition on F^b, of $\mathcal{A}$. Hence, any formula $\psi = \psi^{env} \rightarrow \psi^{sys}$ where ψ^{env} and ψ^{sys} are specified by DBWs is also considered a GR(1) specification here.

7.1.2 Games and Strategies

Following [PPS06b], we define the following game structure, and then make connections to GR(1) specifications. A game structure $\mathcal{G}$ is given by the tuple $(X, Y, Q^g, \theta, \rho^{env}, \rho^{sys}, Win)$, where

- X is a set of *input* variables, which are controlled by the environment.
- Y (disjoint from X) is a set of *output* variables, which are controlled by the system. Without loss of generality, we assume all input and output variables are Boolean.

- $Q^g \subseteq 2^{X \cup Y}$ is the state space of the game defined over the input and output variables. A state in Q^g is thus an interpretation of $X \cup Y$, assigning each variable to either `true` or `false`.

- θ is a Boolean formula over $X \cup Y$ that specifies the initial states of the game structure. We use $Q^g_0 \subseteq Q^g$ to denote the set of initial states.

- $\rho^{env} \subseteq Q^g \times 2^X$ or $\rho^{env}(X, Y, X')$ is the environment transition relation relating a present state in Q^g to the possible next inputs the environment can pick in 2^X. We use the primed copies X' of X to denote the next input variables.

- $\rho^{sys} \subseteq Q^g \times 2^X \times 2^Y$ or $\rho^{sys}(X, Y, X', Y')$ is the system transition relation relating a present state in Q^g and a next input in 2^X picked by the environment to the possible next outputs the system can pick in 2^Y. Similarly, we use the primed copies Y' of Y to denote the next output variables.

- Finally, Win is the winning condition of the game.

Given a set of GR(1) specifications over $X \cup Y$, i.e. $\psi_i^{env}, \psi_i^{sys}, \psi_t^{env}, \psi_t^{sys}, \psi_f^{env}, \psi_f^{sys}$, we can define a game structure $\mathcal{G}$ in the following way.

- We set $\theta = \psi_i^{env} \wedge \psi_i^{sys}$.

- We set $\rho^{env} = \psi_t^{env}$ by replacing all occurrences of $\mathbf{X}\, u$ by u'.

- We set $\rho^{sys} = \psi_t^{sys}$ by replacing all occurrences of $\mathbf{X}\, u$ by u'.

- Finally, Win is given by $Win = \psi^{env} \rightarrow \psi^{sys}$.

In the context of synthesis of reactive systems, the game is played between the environment *env* and the system *sys*. At each step of the game, *env* first chooses a value for the next input variables X' and then *sys* chooses a value for the next output variables Y'. A play π of $\mathcal{G}$ is a maximal sequence of states $\pi = q_0, q_1, \ldots$ of states such that $q_0 \models \theta$ and $(q_i, q_{i+1}) \in \rho^{env} \wedge \rho^{sys}$ for all $i \geq 0$. A play π is winning for the system iff it is infinite and $\pi \models Win$. Otherwise, π is winning for the environment.

A finite memory strategy for *sys* in $\mathcal{G}$ is a tuple $\mathcal{S}^{sys} = (\Gamma^{sys}, \gamma^{sys_0}, \eta^{sys})$, where,

- Γ^{sys} is a finite set representing the memory,

- $\gamma^{sys_0} \in \Gamma^{sys}$ is the initial memory content, and

- $\eta^{sys} \subseteq Q^g \times \Gamma^{sys} \times \mathcal{X} \times \mathcal{Y} \times \Gamma^{sys}$ or $\eta^{sys}(X, Y, \gamma^{sys}, X', Y', \gamma^{sys\prime})$ is a relation mapping a state in $\mathcal{G}$, some memory content $\gamma^{sys} \in \Gamma^{sys}$ and a next input value chosen by the environment to the possible next outputs the system can pick and an updated memory content.

Similarly, a (finite memory) strategy for *env* is a tuple $\mathcal{S}^{env} = (\Gamma^{env}, \gamma^{env_0}, \eta^{env})$, where

- Γ^{env} is a finite set representing the memory,
- $\gamma^{env_0} \in \Gamma^{env}$ is the initial memory content, and
- $\eta^{env} \subseteq Q^g \times \Gamma^{env} \times \mathcal{X} \times \Gamma^{env}$ or $\eta^{env}(X, Y, \gamma^{env}, X', \gamma^{env\prime})$ is a relation mapping a state in $\mathcal{G}$ and some memory content $\gamma^{env} \in \Gamma^{env}$ to the possible next inputs the environment can pick and an updated memory content.

A play $\pi = q_0, q_1, \ldots = (x_0, y_0), (x_1, y_1), \ldots$ is said to conform to a strategy $\mathcal{S}^{sys}$ (or respectively, $\mathcal{S}^{env}$) iff there exists a sequence $\gamma_0^{sys}, \gamma_1^{sys}, \ldots$ $(\gamma_0^{env}, \gamma_1^{env}, \ldots)$ such that, for all $i \geq 0$, $(q_i, \gamma_i^{sys}, x_{i+1}, y_{i+1}, \gamma_{i+1}^{sys}) \in \eta^{sys}$ $((q_i, \gamma_i^{env}, x_{i+1}, \gamma_{i+1}^{env}) \in \eta^{env})$. A strategy $\mathcal{S}^{sys}$ (or respectively, $\mathcal{S}^{env}$) is winning for *sys* (*env*) from a state q if all plays starting in q and conforming to $\mathcal{S}^{sys}$ (*env*) are won by *sys* (*env*). We use W^{sys} (or respectively, W^{env}) to denote the winning region, or the set of states from which such a winning strategy exists, for *sys* (*env*). Following [KHB09], we call a winning strategy for the environment a *counterstrategy*. Similar to [PPS06b], Könighofer et al. [KHB09] show that counterstrategies can be extracted from the intermediate results of the fixpoint computation for the winning regions, consisting of a disjunction of four sub-strategies.

Hence, given a GR(1) specification $\psi = \psi^{env} \rightarrow \psi^{sys}$, it is realizable iff all the initial states are winning for the system in the corresponding game structure. In this dissertation, we focus on scenarios when the specification is *unrealizable* (but satisfiable), and explore ways to generate environment assumptions as potential fixes by *mining them from the counterstrategy*.

A winning strategy can be turned to a correct implementation by finding a (deterministic) circuit with $|X'|$ inputs, $|X| + |Y|$ flip-flops and $|Y|$ outputs. At each clock cycle, the values of the next outputs Y' are determined by the current (stored) values of $X \cup Y$ and the values of the next inputs X'. Hence, any combinational logic with inputs $X \cup Y \cup X'$ and outputs Y' that is consistent with the winning strategy, which is a relation over $X \cup Y \cup X' \cup Y'$. essentially yields the desired circuit. There are multiple ways to find functions compatible with a Boolean relation. We refer the readers to [WB91, BGJ+07b, BGJ+07a, BCK09, JLH09] for more details. A more recent research also shows that a circuit can be directly constructed from iterates of the fixpoint computation that computes the winning region [SHB12]. In this thesis, we use M_ψ to denote the sequential circuit that is synthesized from specification ψ.

7.1.3 Counterstrategy Graph

It is often convenient to view the counterstrategy as transition system or a directed graph. A *counterstrategy graph* G^c is a discrete transition system $G^c = (Q^c, Q_0^c \subseteq Q^c, \rho^c \subseteq Q^c \times Q^c)$, where

- $Q^c \subseteq Q^g \times \Gamma^{env}$ is the state space,
- $Q_0^c = Q_0^g \times \gamma^{env_0}$ is the set of initial states, and

- $\rho^c = \eta^{env} \wedge \rho^{sys}$ is the transition relation.

In a nutshell, G^c describes evolutions of the game state where *env* adheres to η^{env} and *sys* adheres to ρ^{sys}. Clearly, each state in G^c is associated with an assignment over the input variables X and output variables Y (a Boolean *cube*). With a slight abuse of notation, we use the function $\theta^c : Q^c \to 2^{X \cup Y}$ to denote the Boolean cube xy for $x \in 2^X$ and $y \in 2^Y$ associated with a state $q^c \in Q^c$. A run π^c of G^c is a maximal sequence of states $\pi^c = q_0^c, q_1^c, \ldots$ of states such that $q_0^c \in Q_0^c$ and $(q_i^c, q_{i+1}^c) \in \rho^c$ for all $i \geq 0$.

We can also view G^c as a directed graph, where each state in Q^c is given its own node, and there is an edge from node q_i^c to node q_j^c if given the current state at q_i^c, there exists a *next* input picked from the counterstrategy for which the system can produce a legal next output so that the game proceeds to a new state at q_j^c.

7.2 Related Work

In this section, we focus our discussion on debugging LTL specifications in the context of synthesis. Unrealizability can come from over-constrained system assertions or insufficient environment assumptions. Our work assumes that unrealizability is due to insufficient environment assumptions, and tackles this by generating additional assumptions.

Cimatti et al. [CRST08] formally define the notion of (minimal) explanation for unrealizability using an unrealizability core, which is the set of specifications responsible for unrealizability. In particular, they suggest using the removal of guarantees as a way to explain and fix unrealizability. Our approach can be viewed as orthogonal to theirs. We aim to add environment assumptions to make the specification realizable. We argue that this is also a natural way to fix unrealizability. In fact, formal and even informal descriptions of the environment are often not available or far less accessible than those of the system. The challenge here is finding the right assumptions to add.

Counterstrategies have been used to explain failures in synthesizing a system that respects the specification, such as in the context of Live Sequence Charts [BSL04]. Könighofer et al. [KHB09] provide an explanation for unrealizability of GR(1) specifications by computing a finite-state counterstrategy for the environment. The counterstrategy is further simplified by removing specifications and variables that are not responsible for unrealizability. A heuristic is also provided for computing from the counterstrategy a *countertrace* — a fixed infinite input sequence that, regardless of what the system outputs, will still ensure that the system specification will be violated. Our work builds upon this work – we mine specifications from these counterstrategies. Similar to their work, we also focus our attention on GR(1) specifications for which efficient (relative to LTL) algorithms exist for dealing with the synthesis problem.

In a separate paper, Könighofer et al. [KHB10] present a model-based diagnosis technique that identifies components (specification or variables) in the system guarantees which are over-constrained. For GR(1) specifications, the authors also show how to compute the

realizable and unrealizable core quickly using approximations. In our work, we focus on analyzing the weakness in the environment assumptions instead of the constraints in the system guarantees. Moreover, we produce additional assumptions, which is a step beyond localizing errors.

The problem of correcting the assumption of an unrealizable LTL specification has also been studied in depth in [CHJ08]. The authors construct an additional assumption that constrains only the environment as weakly as possible, and makes the resulting specification realizable. The approach proceeds by first computing a safety assumption that removes a minimal set of environment edges from the game graph, and then computing a liveness assumption that puts fairness on the remaining environment edges. Finding a minimal set of fair edges is shown to be NP-hard. To address this potential intractability, the authors use probabilistic games to compute a locally minimal fairness assumption. The approach was implemented as a tool called GIST [CHJR10]. An advantage of their work is that they can synthesize general environment assumptions (as an intersection of safety and liveness assumptions [AS87]) for any LTL synthesis problem. Our work provides a simpler yet practical approach by restricting the form of missing assumptions and uses specification mining to identify a set of assumptions to restrict the environment in a reasonable way to make the specification realizable.

Previous work by Hagihara et al. [HKSY09] has also attempted to extract environment constraints of simple forms to make specifications strong satisfiable, where strong satisfiability means that for all input sequences which are given in advance, there exists an output sequence such that the specification is satisfied. Their method is based on deriving these constraints from Büchi automata representing the specifications. Our method is different because we tackle realizability directly instead of strong satisfiability. In addition, we use a counterstrategy-guided approach instead of the constructive derivation used in this previous work.

Vasumathi and Kress-Gazit [RKG11] study unsynthesizable specifications in the setting of automatically constructing robotic controllers from LTL specifications. They perform a series of simple checks to identify parts of the LTL specifications that might be flawed, and relate them back to the structured English input. Livingston et al. [LMB12, LPJM13] have also studied the synthesis problem and used a notion of locality that allows "patching" a nominal solution. They update the local parts of the strategy as new data accumulates. This approach allows incremental synthesis and prevents global re-synthesis if the nominal plan fails. The patching algorithm considers changes (addition and deletion of edges) in the game graph, and identifies the affected nodes for each system goal and modifies the game graph locally. In our approach, we do not start with a synthesized game graph, and we mine assumptions from counterstrategies. Also, the uncertainties we consider are not limited to the topological changes in the system's environment.

Chapter 8

Mining Environment Assumptions

> There are known knowns; there are things we know that we know.
> There are known unknowns; that is to say, there are things that we now know we don't know.
> But there are also unknown unknowns – there are things we do not know we don't know.
>
> – Donald Rumsfeld

In this chapter, we present a specification mining technique, called *counterstrategy-guided assumption mining*, which utilizes the counterstrategy produced for an unrealizable GR(1) specification to suggest candidate assumptions to make the specification realizable. In particular, we use a *template-based specification mining* approach to find LTL properties that are satisfied by the counterstrategy. Our choice of templates is designed to match the kinds of specifications allowed in GR(1), thus enabling an iterative synthesis framework that exploits the efficiency of GR(1) synthesis. Further, we argue that these templates, albeit simple and have limited expressiveness, are more useful in practice than a monolithic but complex automaton that tries to capture the missing environment assumption.

The key idea of our approach is that by asserting the negation of the mined properties as additional assumption of the original specification, we effectively rule out the moves described by these properties in the environment. Hence, we iterate this process, which increasingly constrains the environment, until either we cannot find a specification in our library of templates that is satisfied by the counter-strategy or the resulting specification becomes realizable. In this chapter, we present this approach and make a connection to version-space learning, and describe the algorithms and optimizations involved in detail.

Our work can be viewed as a debugging approach to unrealizability. Unrealizability typically arises either from an under-constrained environment or from an over-constrained system. We focus on debugging environmental assumptions rather than system guarantees; note however, that these two are complementary. Könighofer et al. [KHB10] propose to identify guarantees that can be weakened or signals that can be less restricted in order to obtain

a realizable specification. Our focus is on generating *additional environment assumptions*, since this is the more tricky problem in practice — it is often easier to miss an environment behavior than to miss a system specification because the former is not inherently part of the design and thus seldom formally defined. Particularly, we want additional assumptions that are tailored to the type of synthesis (GR(1) in this case here) and the existing specification. As also noted in [KHB10], the Boolean formula `false` is also a valid assumption to resolve unrealizability but a trivial and uninteresting one.

We propose a template-based specification mining approach to address this problem. By imposing a particular structure on the form of specifications (using templates), we reduce the possibility of generating uninteresting assumptions. Also, by biasing the search towards simple templates (that mention few variables), we bias our approach towards assumptions that generalize well.

One may argue in favor of an alternate approach where the additional environment assumption is constructed directly during the synthesis process so that the resulting specification is realizable. Chatterjee et al. [CHJ08] show that in fact one can construct such an assumption by analyzing the game graph that is used to answer the realizability question. In fact, the assumption synthesized (as a Büchi automaton) is minimal in terms of the number of safety and fair edges manipulated in the game graph during synthesis. However, such a monolithic environment assumption (see, e.g., Figure 3 in [CHJ08]) can be difficult for a human user to understand even for correcting a simple specification. Moreover, the Büchi automaton synthesized does not directly translate to an LTL description, which is the original motivation for using LTL specifications due to their affinity to design descriptions. In addition, the behavior of the weakest assumption does not necessarily coincide with the designer's intent. We offer an alternative approach that is simpler from the theoretical viewpoint but very useful in practice.

Specifically, the work discussed in this chapter contributes to the state-of-the-art in the following ways.

- When LTL specifications are satisfiable but unrealizable, we present a novel *counterstrategy-guided synthesis* approach that can produce additional environmental assumptions to make the specification realizable.
- We formalize this specification mining approach as a version-space learning problem, and present optimization techniques for speeding up the learning process.
- We demonstrate the effectiveness of our approach with examples in digital circuits and robotic controllers.

The rest of the chapter is organized as follows. First, we describe our assumption mining algorithm in Section 8.1. We then further formalize our counterstrategy-guided synthesis approach as a version-space learning problem in Section 8.2. Afterwards, we present experimental results in Section 8.3 and we finally conclude in Section 8.4.

8.1 Solution Overview

Figure 8.1 shows the main flow of our method. Our specification mining algorithm takes in as input a library of specification templates, and a counter-strategy state machine generated from unrealizability analysis, and produces as output a candidate assumption that can be added to the existing specification to make it realizable.

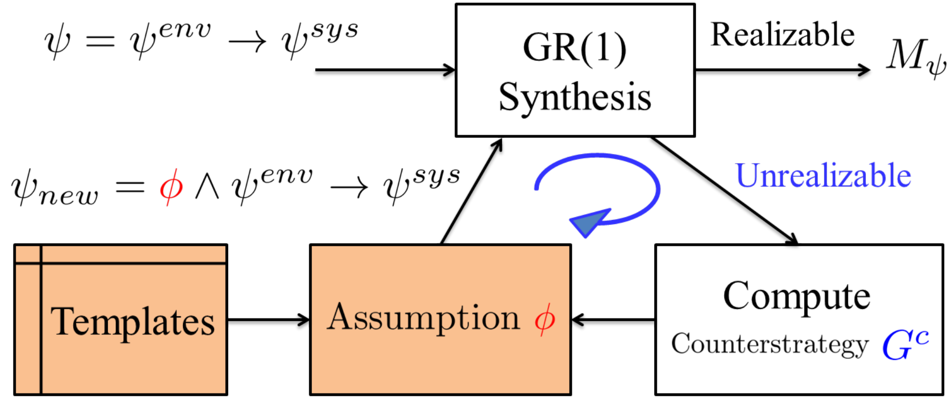


Figure 8.1: Counterstrategy-guided synthesis enabled by assumption mining (the highlighted portions are our contributions).

The counterstrategy G^c is computed using the approach described in [KHB09] for an unrealizable GR(1) specification. The counterstrategy summarizes the next moves of the environment in response to the current output of the system, which will force a violation of the specification. We then use a template-based mining approach to find a specification that is satisfied by G^c. By asserting the negation of such a specification as an assumption ϕ to the original specification, we effectively rule out such moves by the environment. Afterwards, the synthesis procedure is executed on the new specification ψ_{new}. We iterate this process until either we cannot find a specification in our library of templates that is satisfied by the counterstrategy or the resulting specification becomes realizable. We give an overview of the mining algorithm below.

There are four main procedures used by our approach.

- **GenerateCandidates**(Ξ^p, Σ) generates a set of template instantiations Ξ in a particular order as candidates of the additional environment assumptions.
- **Realizable**(ψ) checks if the specification ψ is realizable.
- **CounterStrategy**(ψ) returns a counterstrategy graph G^c for the environment to force a violation of the specification if ψ is not realizable.
- **Mine**(G^c, Ξ, ψ^{env}), returns a formula ϕ as an additional candidate assumption.

Algorithm 5 Mine Assumptions for $\psi = \psi^{env} \rightarrow \psi^{sys}$

Input: $\Sigma = \mathcal{X} \cup \mathcal{Y}$: the alphabet
Input: $\psi = \psi^{env} \rightarrow \psi^{sys}$: initial specification
Input: Ξ^p: a set of specification templates
Output: ϕ: additional assumption required for realizability
1: Ξ := **GenerateCandidates**(Ξ^p, Σ)
2: **while** $\neg$**Realizable**(ψ) **do**
3: G^c := **CounterStrategy**(ψ)
4: ϕ := **Mine**(G^c, Ξ, ψ^{env})
5: **if** $\phi =$ `false` **then**
6: return *Insufficient Template*
7: **Quit**
8: **end if**
9: $\psi^{env} := \psi^{env} \wedge \phi$
10: $\psi := \psi^{env} \rightarrow \psi^{sys}$
11: **end while**

We illustrate the algorithm with an example below.

Example 1. *Consider $X = \{x\}$, $Y = \{y\}$ and the following GR(1) sub-formulas which together form $\psi = \psi^{env} \rightarrow \psi^{sys}$.*

1. $\psi_f^{env} = \boldsymbol{G}\,(\boldsymbol{F}\neg x)$
2. $\psi_t^{sys} = \boldsymbol{G}\,(\neg x \rightarrow \neg y)$
3. $\psi_f^{sys} = \boldsymbol{G}\,(\boldsymbol{F} y)$

Specification ψ is not realizable. Figure 8.2 shows the computed counterstrategy graph G^c. The literal $\bar{x}$ ($\bar{y}$) denotes the negation of the propositional variable x (y). The memory content is denoted by γ_i with γ_0 being the initial memory content. The three shaded states on the left are the initial states. The literals on the edges indicate that the environment first chooses $\bar{x}$ and then the system chooses $\bar{y}$. (the system is forced to pick $\bar{y}$ due to ψ_t^{sys}). Observe that, according the counterstrategy, the system will be forced to pick $\bar{y}$ perpetually. Hence, the other system guarantee ψ_f^{sys} cannot be satisfied.

The key observation we make here is that $G^c \models \neg\phi$, where $\neg\phi = \boldsymbol{F}\,(\boldsymbol{G}\,\neg x)$. Thus, if we assert ϕ as an additional assumption $\phi = \boldsymbol{G}\,(\boldsymbol{F}\,x)$ to ψ, then we effectively rule out this counterstrategy. In fact, the new specification $\psi_{new} = (\phi \wedge \psi^{env}) \rightarrow \psi^{sys}$ is realizable.

In this work, we consider the following GR(1) specification templates Ξ^p[1] over the pattern alphabet $\Sigma = \{a, b\}$.

[1]We omit assumptions about the initial states of the environment since they are less interesting.

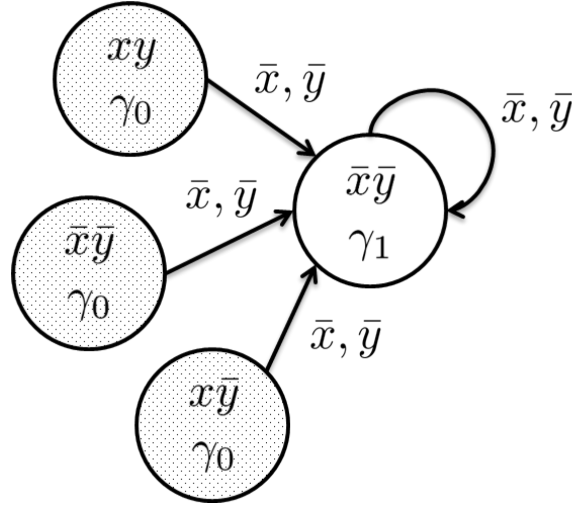


Figure 8.2: Counterstrategy graph G^c for unrealizable specification ψ.

- $\phi_{a1} = \mathbf{G}\ (\mathbf{F}\ a)$
- $\phi_{a2} = \mathbf{G}\ (\mathbf{F}\ (a \vee b))$
- $\phi_{a3} = \mathbf{G}\ a$
- $\phi_{a4} = \mathbf{G}\ (a \vee b)$
- $\phi_{a5} = \mathbf{G}\ (a \rightarrow \mathbf{X}\ b)$

Both ϕ_{a1} and ϕ_{a2} are instances of ψ_f^{env} (fairness assumptions on the environment). ϕ_{a3}, ϕ_{a4} and ϕ_{a5} correspond to ψ_t^{env} (invariants and transitions). Definition 2.3 thus can be adapted to the setting of mining assumptions for temporal logic synthesis. Specifically, the *specification templates* are the GR(1) templates Ξ^p, an *evidence* is a counterstrategy (graph) G^c, and *satisfaction* can be determined by whether $G^c \models \neg\phi$, $\phi \in \Xi$.

Hence, the procedure **GenerateCandidates**(Ξ^p, Σ) produces a set of specifications based on the templates Ξ^p and conform to the syntax of ψ^{env} (so that the specifications are assumptions on the environment). Given a counterstrategy graph G^c, the set of candidate assumptions Ξ, and the current environment assumption ψ^{env}, the procedure **Mine**(G^c, Ξ, ψ^{env}) selects an assumption $\phi \in \Xi$ such that $G^c \models \neg\phi$ and $\phi \wedge \psi^{env} \neq$ `false`[2] if possible. The latter guarantees that we do not add any assumption that is inconsistent with the current assumptions. In Section 8.2, we present a selection strategy based on version-space learning.

Remark 8.1. *G^c is a transition system that can potentially have deadlocks. This means when model checking G^c against the specification $\neg\phi$ may result in the specification being*

[2]This amounts to checking the satisfiability of $\phi \wedge \psi^{env}$.

vacuously true. In the ***Mine*** *procedure, we first check if there exist any deadlock state. For every deadlock state, we check only that state against instantiations from templates ϕ_{a3} and ϕ_{a4}.*

In general, our approach can be viewed as a recommendation system for the user, which can incorporate the user's design intent in an interactive way. The user can engineer the templates based on his knowledge of the environment (possibly another system) or inspect and rule out any candidate assumption as the mining algorithm proceeds. To come up with multiple recommendations, one can simply restart the algorithm with a reduced set of template instantiations (in which the first additional assumption found in constructing the previous ϕ is discarded) and try to find another set that achieves realizability.

8.2 Version-Space Learning with Templates

We first formalize the assumption mining problem in the version space learning framework.

Originally due to Tom Mitchell [Mit79], given *hypothesis space* H and *examples* E, the version space is a subset of H that is *consistent* with the examples.

Definition 8.1. *A hypothesis $h_1 \in H$ is a more general hypothesis than $h_2 \in H$ if h_2 implies h_1. In this case, we also say h_2 is a more specific hypothesis than h_1.*

Definition 8.2. *The general boundary of a version space, G, is the set of maximally general members of the version space. Similarly, the specific boundary of a version space, S, is the set of maximally specific members of the version space.*

Hence, H can be represented by the boundaries G and S. In our setting, given the set of all possible instantiations Ξ of the assumption templates, a *hypothesis* h is a formula $\phi^h = \bigwedge_{\Xi'} \xi$, $\xi \in \Xi'$ for some nonempty subset $\Xi' \subseteq \Xi$. Additionally, let $\texttt{true} \in H$ be the most general hypothesis and $\texttt{false} \in H$ be the most specific hypothesis. An *example* is a counterstrategy graph G^c. Such an example is a *negative* example since it represents behaviors that the target environment should not have. In this case, we only have negative examples[3]. *Consistency* is then defined as $G^c \models \neg\phi^h$.

Language inclusion (or the reverse of a logical implication) thus induces a partial order on the set of hypotheses. According to Definition 8.1, h_1 is more general than h_2 (conversely, h_2 is more specific than h_1) if $\mathcal{L}(\phi^{h_1}) \supseteq \mathcal{L}(\phi^{h_2})$. Figure 8.3 illustrates this concept, where hypotheses are ordered (direction of the arrows) from most general to most specific.

In this case, we seek the most general and consistent hypothesis. The rationale behind this choice is that we want the additional assumption to constrain the environment as little as possible so that it is just sufficient to ensure the realizability of the resulting specification. Hence, starting from true, which is the most general hypothesis, we select a candidate

[3] User may choose to present desired behaviors that the target design should satisfy in the form of traces. Such a trace τ can then be used as a positive example and thus we seek $\tau \models \phi^h$.

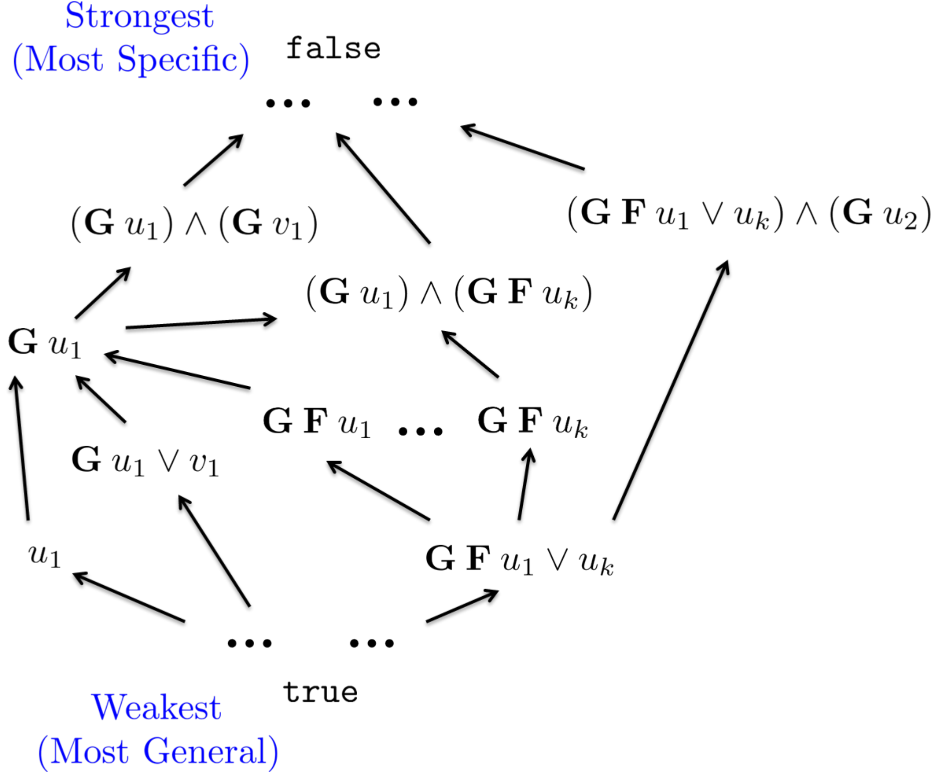


Figure 8.3: Diagrammatic representation of a version space from the most general hypothesis to the most specific hypothesis.

assumption ϕ^h from the set of hypotheses in the next partial order. Observe that, however, there is still a large number of possible candidates to choose from, and not all of them are consistent with the example. To reduce the amount of redundant checks, we use the following modification to the algorithm.

- For a counterstrategy $\mathcal{S}^{env} = (\Gamma^{env}, \gamma^{env_0}, \eta^{env})$, compute $det(\eta^{env}) : Q^g \times \Gamma^{env} \rightarrow \mathcal{X} \times \Gamma^{env}$ as the deterministic version of η^{env} by randomly selecting a next input and a memory update if more than one choice is available. Compute the counterstrategy graph G^c using $det(\eta^{env})$ instead of η^{env}.

- In each iteration of the **while** loop in Algorithm 5, consider a hypothesis space where each hypothesis is a single template instantiation. This is motivated by the observation that, after η^{env} is simplified to $det(\eta^{env})$, it is more likely to satisfy a simple formula without the search resorting to a more complicated disjunction (obtained by negating the conjunction of multiple instantiations). Now among the smaller set of hypotheses during each iteration, we randomly pick one from the set as the candidate assumption ϕ^h. If $G^c \models \neg\phi^h$, then ϕ^h is added to (conjuncted with) ψ^{env} and will not be picked again.

8.3 Experimental Results

In this section, we present case studies from the domains of digital circuit design and robotic controller synthesis. We use RATSY [BCG+10] to generate the counterstrategy graph G^c in case of unrealizability. We use the model checking capabilities in SAL [BGL+00] check if the current set of assumptions is satisfiable, and to model check if $G^c \models \neg\phi$. Our experiment proceeds as follows. Starting with a known realizable specification ψ,

1. Remove an arbitrary assumption ϕ^r from ψ;
2. Proceed with Algorithm 5 to generate a replacement assumption ϕ;
3. Evaluate the relationship between ϕ and ϕ^r;
4. Restart from Step 1 by removing a different assumption.

8.3.1 AMBA AHB Bus Protocol

ARM's Advanced Microcontroller Bus Architecture (AMBA) Advanced High Performance Bus (AHB) is an on-chip communication protocol. The specification allows for up to 16 masters and 16 slaves. The masters initiate communication (reading or writing) with the slaves, and the slaves respond to a master's request. There is an address bus and a data bus, each of which can only be accessed by one master at a time. An arbiter controls access to the address bus. A bus access can either be a single transfer, or a burst, which is a transaction consisting of multiple transfers. A bus access can also either be locked (incapable of being interrupted) or unlocked. Our specifications for this protocol are taken from the example files provided with RATSY [BCG+10]. For details of this protocol, we refer the readers to [BGJ+07a]. There are four environment signals. The first three are driven by the masters and the last one is driven by the slaves.

- `HBUSREQ[i]` - master `i` requests access to the bus
- `HLOCK[i]` - master `i` requests locked access to the bus (used in combination with `HUBUSREQ[i]`)
- `HBURST[1:0]` - one of following: single transfer (SINGLE), 4-transfer burst (BURST4), or unspecified size burst (INCR) Signals driven by the slaves:
- `HREADY` - high if the slave has finished working with the current data; needs to be high before bus ownership can change or transfers can begin

Our experiment was on the configuration of the protocol with 1 master and 2 slaves. We ignored the assumptions that characterize the initial states. In fact, the removal of any of them does not lead to unrealizability. We illustrate our approach with the following example below. First, we remove the following environment fairness.

$$\phi^r = \mathbf{G}\ (\mathbf{F}\ \texttt{HREADY} = 1)$$

We applied our algorithm on the remaining specification and we found ϕ^r in exactly one iteration. In fact, we can choose a different candidate assumption when multiple ones are possible, as indicated in Section 8.2. The following is another possible valid replacement.

$$\mathbf{G}\ (\texttt{HREADY} = 0 \rightarrow \mathbf{X}\ \texttt{HBUSREQ[0]} = 0) \wedge\ \mathbf{G}\ (\mathbf{F}\ \texttt{HREADY} = 1)$$

Next, we removed the following environment transition.

$$\phi^r = \mathbf{G}\ (\texttt{HLOCK[0]} = 1 \rightarrow \texttt{HBUSREQ[0]} = 1)$$

Our algorithm produced $\mathbf{G}\ (\mathbf{F}\ \texttt{HLOCK[0]} = 0)$ as a replacement. The original assumption says whenever master `0` sets `HLOCK[0]` to high, it should be sending a request at the same time by setting `HBUSREQ[0]` to high. Our replacement says master `0` should de-assert `HLOCK[0]` infinitely often. This assumption may not be completely desirable because it does not pinpoint the condition on which the signal should be de-asserted. However, it clearly indicates the need to prevent a master from having a locked access to the bus permanently, and the fact that adding this requirement will make the specification realizable.

In general, we may get a number of possible replacements. It is debatable which of these replacements best represents the designer's intent. We are simply offering a recommendation system in which the user can choose from a pool of possible assumptions. The quality of this pool can be improved if the user can provide more information to the synthesis process in the form of desired traces of the target design. Our experiments show that even without this additional user input, our method is able to generate good quality assumptions that achieve realizability.

8.3.2 Generalized Buffer

IBM's generalized buffer (GenBuf) transmits data from n senders to two receivers. The senders provide data in any order, but the receivers must receive the data in turns. The handshake protocol between sender, buffer, and receiver is as follows. The senders request permission from the buffer to send their data. The buffer must acknowledge each sender's request. The buffer then sends a request to a receiver for permission to transmit the data. The receiver must also acknowledge the buffer's request. We used the specifications provided with the tool ANZU [JGWB07]. For details of this example, we refer the readers to [BGJ+07b]. It has the following environment signals.

- `StoB_REQi` - sender `i` requests a send from the buffer
- `RtoB_ACKj` - receiver `j` acknowledges the buffer's request
- `FULL` - the FIFO is ready to send data
- `EMPTY` - FIFO is ready to receive data

We performed a similar experiment as the one in Section 8.3.1. In the first experiment, we removed the following fairness assumption.

$$\phi^r = \mathbf{G}\ (\mathbf{F}\ (\texttt{BtoR_REQ0} = 1 \leftrightarrow \texttt{RtoB_ACK0} = 1))$$

Our algorithm produced the following replacement.

$$\mathbf{G}\ (\mathbf{F}\ \texttt{RtoB_ACK0} = 1)$$

In this example, we were not able to recover the missing assumption. Our replacement represents an environment where the receiver acknowledges infinitely often. The original assumption states that only true request (when BtoR_REQ0 is high) is acknowledged infinitely often. The replacement assumption is stronger than the original, but still provides a hint to the desired behavior of the environment.

8.3.3 Robotic Vehicle Controller

Our example of a robotic vehicle controller is motivated by the work done by Wongpiromsarn et al. [Won10]. Their work aims to synthesize a discrete planner for an autonomous vehicles to navigate in an urban environment while following traffic rules and avoiding obstacles. We use the following simplified variant in our experiment. Given a rectangular grid of length X (length of the road) and width Y (number of lanes), define the coordinates of where the vehicle is located as $l_{x,y}$. We use $o_{x,y}$ to denote if there is an obstacle at position (x, y) at every time step ($o_{x,y} = 1$ if there is one). This is used to simulate moving obstacles such as other cars in the urban environment. The requirements are the following.

- **A**: All squares are clear of obstacles infinitely often: $\mathbf{G}\ \mathbf{F}\ o_{x,y} = 0$.
- **G1**: The car is at initial position $l^i_{x,y}$ and there is no obstacle at the initial position.
- **G2**: The vehicle can move to an adjacent square or stay in the current square at each time step.
- **G3**: The vehicle cannot move into a square occupied by an obstacle.
- **G4**: The vehicle eventually reaches its final destination.

We expressed these requirements in GR(1) formulas. One way to make the specification unrealizable is to have the destination square permanently blocked (by removing the corresponding fairness assumption). We followed Algorithm 5 and were able to recover the assumption in one iteration.

8.4 Summary

In this chapter, we propose a systematic technique for mining candidate assumptions to guide an unrealizable specification towards realizability. Our technique mines assumptions from *counteracting behaviors*, which are summarized by the counterstrategy of an unrealizable specification. Further, we formalize the problem as an instance of version-space learning to elucidate the proposed counterstrategy-guided synthesis framework. We also propose optimizations to speed up the assumption mining process. Experimental results are encouraging – missing assumptions are found in a majority of the cases and reasonable replacements are found for the rest.

Chapter 9

Human-in-the-Loop Controller Synthesis

> When a machine begins to run without human aid, it is time to scrap it – whether it be a factory or a government.
>
> – Alexander Chase

In this chapter, we show that the proposed counterstrategy-guided assumption mining approach described in Chapter 8 enables the automatic synthesis of a new class of semi-autonomous controllers, called human-in-the-loop (HuIL) controllers. A crucial component of such a controller is an advisory that determines when to switch control from the autonomous controller to the human operator. We formalize the criteria that characterize a HuIL controller, by taking into account of human factors such as response time, and describe how to construct the advisory using assumption mining.

9.1 Introduction

Many safety-critical systems are *interactive*, i.e., they interact with a human being, and the human operator's role is central to the correct working of the system. Examples of such systems include fly-by-wire aircraft control systems (interacting with a pilot), automobiles with driver assistance systems (interacting with a driver), and medical devices (interacting with a doctor, nurse, or patient). We refer to such interactive control systems as *human-in-the-loop control systems*. The costs of incorrect operation in the application domains served by these systems can be very severe. Human factors are often the reason for failures or "near failures", as noted by several studies (e.g., [Fed95, KCD00]).

One alternative to human-in-the-loop systems is to synthesize a fully autonomous controller from a high-level mathematical specification. The specification typically captures both assumptions about the environment and correctness guarantees that the controller must provide, and can be specified in a formal language such as Linear Temporal Logic. While this

correct-by-construction approach looks very attractive, the existence of a fully autonomous controller that can satisfy the specification is not always guaranteed. For example, in the absence of adequate assumptions constraining its behavior, the environment can be modeled as being overly adversarial, causing the synthesis algorithm to conclude that no controller exists. Additionally, the high-level specification might abstract away from inherent physical limitations of the system, such as insufficient range of sensors, which must be taken into account in any real implementation. Thus, while full manual control puts too high a burden on the human operator, some element of human control is desirable. However, at present, there is no systematic methodology to synthesize a combination of human and autonomous control from high-level specifications. In this chapter, we address this limitation of the state of the art. Specifically, we consider the following question:

Can we devise a controller that is mostly automatic and requires only occasional human interaction for correct operation?

A particularly interesting domain is that of automobiles with "self-driving" features, otherwise also termed as "driver assistance systems". Such systems, already capable of automating tasks such as lane keeping, navigating in stop-and-go traffic, and parallel parking, are being integrated into high-end automobiles. However, these emerging technologies also give rise to concerns over the safety of an ultimately driverless car. Recognizing the safety issues and the potential benefits of vehicle automation, the National Highway Traffic Safety Administration (NHTSA) recently published a statement that provides descriptions and guidelines for the continual development of these technologies [Nat13]. Particularly, the statement defines five levels of automation ranging from vehicles without any control systems automated (Level 0) to vehicles with full automation (Level 4). In this paper, we focus on Level 3 which describes a mode of automation that requires only limited driver control:

> *"Level 3 - Limited Self-Driving Automation: Vehicles at this level of automation enable the driver to cede full control of all safety-critical functions under certain traffic or environmental conditions and in those conditions to rely heavily on the vehicle to monitor for changes in those conditions requiring transition back to driver control. The driver is expected to be available for occasional control, but with sufficiently comfortable transition time. The vehicle is designed to ensure safe operation during the automated driving mode."* [Nat13]

Essentially, this mode of automation stipulates that the human driver can act as a fail-safe mechanism and requires the driver to take over control should something go wrong. The challenge, however, lies in identifying the complete set of conditions under which the human driver has to be notified ahead of time. Based on the NHTSA statement, we identify four important criteria required for a human-in-the-loop controller to achieve this level of automation.

- *Monitoring*. The controller should be able to determine if human intervention is needed based on monitoring past and current information about the system and its environment.
- *Minimally Intervening*. The controller should only invoke the human operator when it is necessary, and does so in a minimally intervening manner.
- *Prescient*. The controller can determine if a specification may be violated ahead of time, and issues an advisory to the human operator in such a way that she has sufficient time to respond.
- *Conditionally Correct*. The controller should operate correctly until the point when human intervention is deemed necessary.

We further elaborate and formally define these concepts later in Section 9.3. In general, a human-in-the-loop controller, as shown in Figure 9.1 is a controller consists of three components: an automatic controller, a human operator, and an advisory control mechanism that orchestrates the switching between the auto-controller and the human operator.[1] In this setting, the auto-controller and the human operator can be viewed as two separate controllers, each capable of producing outputs based on inputs from the environment, while the key responsibility of the advisory controller is to determine precisely when the human operator should assume control, while giving her enough time to respond.

Similar to the previous chapter, we study the construction of such controller in the context of temporal logic synthesis.

In summary, the main contributions of this chapter are:

- A formalization of human-in-the-loop control systems and the problem of synthesizing such controllers from high-level specifications, including four key criteria these controllers must satisfy.
- An algorithm for synthesizing human-in-the-loop controllers that satisfy the aforementioned criteria.
- An application of the proposed technique to examples motivated by driver-assistance systems for automobiles.

The rest of the chapter is organized as follows. Section 9.2 describes an motivating example discussing a *car following* example. Section 9.3 provides a formalism and characterization of the human-in-the-loop controller synthesis problem. Section 9.4 describes our algorithm for the problem. We then present case studies of safety critical driving scenarios in Section 9.5. Finally, we discuss related work in Section 9.6 and conclude in Section 9.7.

[1] In this paper, we do not consider explicit dynamics of the plant. Therefore it can be considered as part of the environment also.

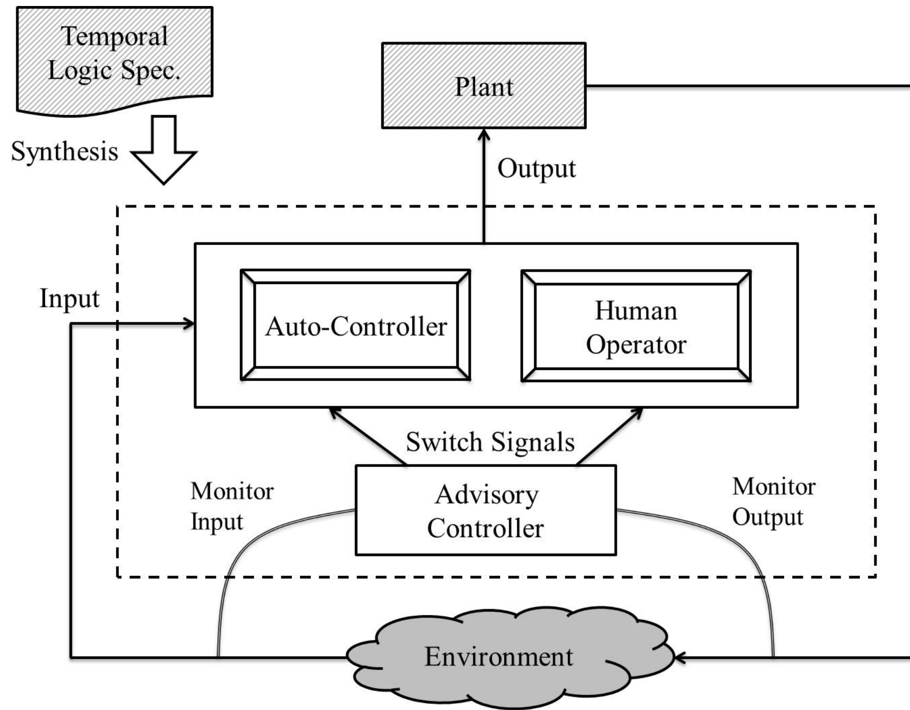
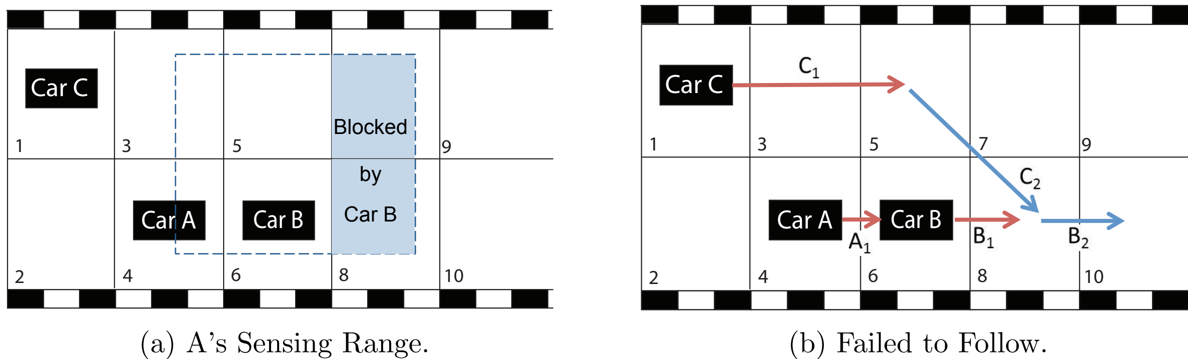


Figure 9.1: Human-in-the-Loop Controller: Component Overview and Synthesis from Specification

9.2 Motivating Example



(a) A's Sensing Range.

(b) Failed to Follow.

Figure 9.2: Controller Synthesis – Car A Following Car B

Consider the example in Figure 9.2. In this example, car A is the autonomous vehicle, car B and C are two other cars on the road. We assume that the road has been divided into discretized regions that encode all the legal transitions for the vehicles on the map, similar to the discretization setup used in receding horizon temporal logic planning [WTM12]. The objective of car A is to *follow* car B. Note that car B and C are part of the *environment*

and cannot be controlled. The notion of following can be stated as follows. We assume that car A is equipped with sensors that allows it to see two squares ahead of itself if its view is not obstructed, as indicated by the enclosed region by blue dashed lines in Figure 9.2a. In this case, car B is blocking the view of car A, and thus car A can only see regions 3, 4, 5 and 6. Car A is said to be able to *follow* car B if it can always move to a position where it can see car B. Furthermore, we assume that car A and C can move at most 2 squares forward, but car B can move at most 1 square ahead, since otherwise car B can out-run or out-maneuver car A.

Given this objective, and additional safety rules such as cars not crashing into one another, our goal is to automatically synthesize a controller for car A such that

- car A follows car B whenever possible;
- and in situations where the objective may not be achievable, *switches control* to the human driver but also allowing *sufficient time* for the driver to respond and take control.

In general, it is not always possible to come up with a fully automatic controller that satisfies all requirements. Figure 9.2b illustrates such a scenario where car C blocks the view as well as the movement path of car A after two time steps. The brown arrows indicate the movements of the three cars in the first time step, and the purple arrows indicate the movements of car B and C in the second time step. Positions of a car X at time t is indicated by X_t. In this failure scenario, the autonomous vehicle needs to notify the human driver since it has lost track of car B.

Hence, a human-in-the-loop synthesis approach is tasked with producing an autonomous controller along with advisories for the human driver in situations where her attention is required. Our challenge, however, is to *identify the conditions that we need to monitor and notify the driver when they may fail.* In the next section, we discuss how human constraints such as response time can be simultaneously considered in the solution, and mechanisms for switching control between the auto-controller and the human driver.

9.3 Human-in-the-Loop Controller

9.3.1 Agents as Automata

We model a discrete controller as a finite-state transducer (FST), as described in Section 2.2. To characterize correctness of M, we assume that we can label if a state is *unsafe* or not, with a function $\mathcal{F} : Q^m \to \{\texttt{true}, \texttt{false}\}$, i.e. a state q is *unsafe* if $\mathcal{F}(q) = \texttt{true}$. With a slight abuse of terminologies, we say a run (or a finite prefix of it) of a controller M is *safe* if it does not contain any unsafe state.

We model two of the three agents in a human-in-the-loop controller, the automatic controller $\mathcal{AC}$ and the advisory controller $\mathcal{VC}$, as finite-state transducers (FSTs). The human

operator can be viewed as another FST $\mathcal{HC}$ that uses the same input and output interface as the auto-controller. The overall controller $\mathcal{HIL}$ is then a composition of the models of $\mathcal{HC}$, $\mathcal{AC}$ and $\mathcal{VC}$.

We use the binary variable *auto* to denote the internal advisory signal that $\mathcal{VC}$ sends to both $\mathcal{AC}$ and $\mathcal{HC}$. Hence, $\mathcal{X}^{\mathcal{HC}} = \mathcal{X}^{\mathcal{AC}} = \mathcal{X} \cup \{auto\}$, and $\mathcal{Y}^{\mathcal{VC}} = \{auto\}$. When $auto = \texttt{false}$, it means the advisory controller is requiring the human operator to take over control, and the auto-controller can have control otherwise.

We assume that the human operator (e.g., driver behind the wheel) can take control at any time by transitioning from a "non-active" state to an "active" state, e.g. by hitting a button on the dashboard or simply pressing down the gas pedal or the brake. When $\mathcal{HC}$ is in the "active" state, the human operator essentially acts as the automaton that produces outputs to the plant (e.g. a car) based on environment inputs. We use a binary variable *active* to denote if $\mathcal{HC}$ is in the "active" state. When $active = \texttt{true}$, the output of $\mathcal{HC}$ can overwrite[2] the output of $\mathcal{AC}$. Similarly, when $active = \texttt{false}$, the output of $\mathcal{HIL}$ is the output of $\mathcal{AC}$. Note that even though the human operator is modeled as a FST here, since we do not have direct control of the human operator, it can in fact be any arbitrary relation mapping $\mathcal{X}$ to $\mathcal{Y}$.

9.3.2 Criteria for Human-in-the-loop Controllers

One key distinguishing factor of a human-in-the-loop controller from traditional controller is the involvement of a human operator. Hence, human factors such as response time cannot be disregarded. In addition, we would like to minimize the need to engage the human operator. Based on the NHTSA statement, we derive four criteria for any effective human-in-the-loop controller, as described below.

- *Monitoring.* An advisory *auto* is issued to the human operator under specific conditions. These conditions in turn need to be determined unambiguously *at runtime*, potentially based on history information but not predictions. In a reactive setting, this means we can use trace information only up to the point when the environment provides a next input from the current state.

- *Minimally intervening.* Our mode of interaction requires only selective human intervention. An intervention occurs when $\mathcal{HC}$ transitions from the "non-active" state to the "active" state (we discuss mechanisms for suggesting a transition from "active" to "non-active" in Section 9.3.1, after prompted by the advisory signal *auto* being $\texttt{false}$). However, frequent transfer of control would mean constant attention is required from the human operator, thus nullifying the benefits of having the auto-controller. In order to reduce the overhead of human participation, we want to minimize a joint cost function f_C that combines two elements: (i) the *probability* that when *auto* is set to

[2] The "overwrite" action happens when a sensor senses the human operator is in control, e.g., putting her hands on the wheel.

`false`, the environment will eventually force $\mathcal{AC}$ into a failure scenario, and (ii) the *cost* of having the human operator taking control. We formalize this objective function in Section 9.4.

- *Prescient.* It may be too late to seek the human operator's attention when failure is imminent. We also need to allow extra time for the human to respond and study the situation. Hence, an advisory should be issued ahead of any failure scenario. In the discrete setting, we are given a positive integer T representing human response time, and require that *auto* is set to `false` at least T number of transitions ahead of a state (in $\mathcal{AC}$) that is *unsafe*.

- *Conditionally-Correct.* The auto-controller is responsible for correct operation as long as *auto* is set to `true`. Formally, if $auto = \texttt{true}$ when $\mathcal{AC}$ is at a state q, then $\mathcal{F}(q) = \texttt{false}$. Additionally, when *auto* is set to `false`, the auto-controller should still maintain correct operation in the next $T-1$ time steps, during or after which we assume the human operator take over control. Formally, if *auto* changes from `true` to `false` when $\mathcal{AC}$ is at a state q, let $R_T(q)$ be the set of states reachable from q within $T-1$ transitions, then $\mathcal{F}(q') = \texttt{false}$, $\forall q' \in R_T(q)$.

While other criteria may be desirable, we believe at least the above four are necessary. Now we are ready to state the *HuIL controller Synthesis Problem:*

Given a model of the system and its specification expressed in a formal language, synthesize a HuIL controller $\mathcal{HIL}$ that is, by construction, monitoring, minimally intervening, prescient, and conditionally correct.

Similar to Chapter 8, we study the synthesis of a HuIL controller in a setting where the controller is synthesized from its temporal logic specifications. In Section 9.4, we propose an algorithm based on the assumption mining technique presented in Chapter 8 for solving the HuIL controller synthesis problem.

9.4 Controller Synthesis

Given an unrealizable specification, a counterstrategy $\mathcal{S}^{env}$ exists for *env* which describes moves by *env* such that it can force a violation of the system guarantees. The key insight of our approach for synthesizing a HuIL controller is that we can synthesize an advisory controller that monitors these moves and prompts the human operator with sufficient time ahead of any danger. These moves are essentially assumptions on the environment under which the system guarantees can be ensured. When these assumptions are not violated (the environment may behave in a benign way in reality), the auto-controller fulfills the objective of the controller. On the other hand, if any of the assumptions is violated, as flagged by the advisory controller, then the control is safely switched to the human operator in a way that

she can have sufficient time to respond (both to the notification and to the situation). The challenge, however, is to decide when an advisory should be sent to the human operator, in a way that it is also *minimally intervening* to the human operator.

9.4.1 Weighted Counterstrategy Graph

Recall that a counterstrategy can be viewed as a discrete transition system or a directed graph G^c. We consider two types of *imminent* failures (violation of some system guarantee specification) described by G^c.

- **Safety violation.** For a node (state) $q_1^c \in Q^c$, if there does not exist a node q_2^c such that $(q_1^c, q_2^c) \in \rho^c$, then we say q_1^c is *failure-imminent*. In this scenario, after *env* picks a next input according to the counterstrategy, *sys* cannot find a next output such that all of the (safety) guarantees are satisfied (some ψ_i^{sys} or ψ_t^{sys} is violated).
- **Fairness violation.** If a node q is part of a strongly connected component (SCC) *scc* in Q^c, then we say q is also *failure-doomed*. For example, the node $(\bar{x}, \bar{y}, \gamma_1)$ in Figure 8.2 is a failure-doomed node. Starting from q, *env* can always pick inputs in such a way that the play is forced to get stuck in *scc*. Clearly, all other states in *scc* are also *failure-doomed*.

Now we make the connection of the labeling function $\mathcal{F}$ for a controller M to the counterstrategy graph G^c which describes behaviors that M should not exhibit. Consider an auto-controller M and a run $\vec{q} = q_0, q_1, \ldots, q_k$ of M up to state q_k which induces the word $(x_0, y_0), (x_1, y_1), \ldots, (x_k, y_k)$. $\mathcal{F}(q_k) = \texttt{true}$ if and only if there exist some $q^c \in Q^c$ such that $\theta^c(q^c) = x_k y_k$ and q^c is either *failure-imminent* or *failure-doomed*.

Proposition 9.1. *Any run of G^c must either end at a* failure-imminent *state or end with a lasso which consists entirely of* failure-doomed *states.*

To see this, recall that a counterstrategy graph for a specification ψ describes how ψ^{sys} is violated. Hence, a run of G^c either violates the safety part of ψ^{sys} (ψ_i^{sys} or ψ_t^{sys}) or violates the fairness part of ψ^{sys} (ψ_f^{sys}).

In practice, it is not always the case that the environment will behave in the most adversarial way. For example, a car in front may yield if it is blocking our path. Hence, even though the specification is not realizable, it is still important to assess, at any given state, whether it will actually lead to a violation. For simplicity, we assume that the environment will adhere to the counterstrategy once it enters a *failure-doomed* state.

We can convert G^c to its directed acyclic graph (DAG) embedding $\hat{G}^c = (\hat{Q}^c, \hat{Q}_0^c, \hat{\rho}^c)$ by contracting each SCC in G^c to a single node. Figure 9.3 shows the condensed graph $\hat{G}^c$ of G^c shown in Figure 8.2 of Section 8.1.

We use the surjective function $\hat{f} : Q^c \to \hat{Q}^c$ to describe the mapping of nodes from G^c to its DAG embedding $\hat{G}^c$. We say a node $\hat{q} \in \hat{Q}^c$ is *failure-prone* if a node $q \in Q^c$ is either *failure-imminent* or *failure-doomed* and $\hat{f}(q) = \hat{q}$.

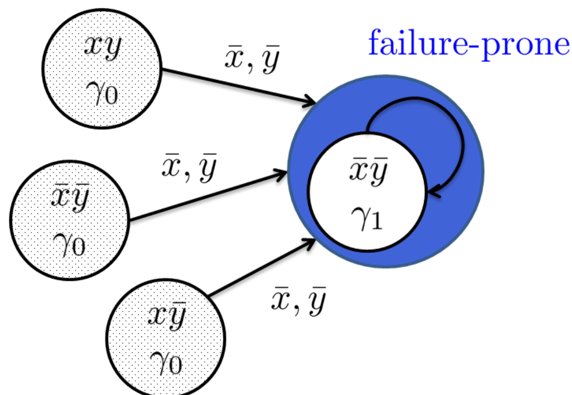


Figure 9.3: Condensed graph $\hat{G}^c$ for G^c (Figure 8.2) after contracting all SCCs.

Recall from Section 9.3 that the notion of *minimally-intervening* requires the minimization of a cost function $\mathcal{C}$, which involves the *probability* that *auto* is set to `false`, Thus far, we have not associated any probabilities with transitions taken by the environment or the system. While our approach can be adapted to work with any assignment of probabilities, for ease of presentation, we make a particular choice in this thesis. Specifically, we assume that at each step, the environment picks a next-input uniformly at random from the set of possible *legal* actions (next-inputs) – those that do not violate any (safety) assumption. In Example 1 and correspondingly Figure 8.2, this means that it is equally likely for *env* to choose $\bar{x}$ or x from any of the states. We use $c(q)$ to denote the total number of legal actions that the environment can take from a state q.

In addition, we need to take into account of the cost of having the human operator perform the maneuver instead of the auto-controller. In general, this cost increases with longer human engagement. Based on these two notions, we define ϖ, which assigns a weight to an edge $e \in \hat{Q}^c \times \hat{Q}^c$ in $\hat{G}^c$, recursively as follows. For an edge between $\hat{q}_i$ and $\hat{q}_j$,

$$\varpi(\hat{q}_i, \hat{q}_j) = \begin{cases} 1 & \text{if } \hat{q}_j \text{ is } \textit{failure-prone} \\ \frac{pen(\hat{q}_i) \times len(\hat{q}_i))}{c(\hat{q}_i} & \text{Otherwise} \end{cases}$$

where $pen : \hat{Q}^c \rightarrow \mathbb{Q}^+$ is a user-defined penalty parameter, and $len : \hat{Q}^c \rightarrow \mathbb{Z}^+$ is the length (number of edges) of the shortest path from a node $\hat{q}_i$ to any failure-prone node in $\hat{G}^c$. Intuitively, a state far away from any failure-prone state is less likely to cause a failure since the environment would need to make multiple consecutive moves all in an adversarial way. However, if we transfer control at this state, the human operator will have to spend more time in control, which is not desirable for a HuIL controller. In the next section, we describe how to use this edge-weighted DAG representation of a counterstrategy graph to derive a HuIL controller that satisfies the criteria established in Section 9.3.

9.4.2 Counterstrategy-Guided Synthesis of HuIL Controllers

Following the counterstrategy-guided strategy presented in Chapter 8, if we can find an assumption to eliminate the counterstrategy represented by G^c such that the resulting specification is realizable, then we can obtain an auto-controller that is correct-by-construction. However, not all assumptions are *monitorable*. For example, we cannot determine whether the formula $\mathbf{G}\ (\mathbf{F}\ x)$ holds for some input x by monitoring the traces of a FST. In the rest of this section, we describe the type of assumptions we mine, such that we can synthesize the auto-controller, the advisory controller, as well as their interaction with the human operator. In addition, we show that the counterstrategy-guided approach can be adapted to handle response time constraint as well.

Consider an outgoing edge from a non-failure-prone node $\hat{q}$ in $\hat{G}^c$, this edge encodes a particular condition where the environment makes a next-move given some last move made by the environment and the system. If some of these next-moves by the environment are disallowed, such that none of the failure-prone nodes are reachable from any initial state (or source node), then we have effectively eliminated the counterstrategy. This means that if we assert the negation of the corresponding conditions as additional ψ_t^{env} (environment transition assumptions), then we can obtain a realizable specification, as described in Chap 8.

Formally, we mine assumptions of the form $\phi = \bigwedge_i (\mathbf{G}\ (a_i \rightarrow \mathbf{X}\ \neg b_i))$, where a_i is a Boolean formula describing a set of assignments over variables in $X \cup Y$, and b_i is a Boolean formula describing a set of assignments over variables in X (see also, Section 8.1).

Under the assumption ϕ, if $(\phi \wedge \psi^{env}) \rightarrow \psi^{sys}$ is realizable, then we can automatically synthesize an auto-controller that satisfies ψ. In addition, the key observation here is that mining ϕ is *equivalent to* finding a set of edges in $\hat{G}^c$ such that, if these edges are removed from $\hat{G}^c$, then none of the failure-prone nodes is reachable from any initial state (assuming the initial states are not failure-prone; this condition can be checked separately). We denote such set of edges as E^ϕ, where each edge $e \in E^\phi$ corresponds to a conjunct in ϕ. For example, if we remove the three outgoing edges from the source nodes in Figure 9.3, then the failure-prone node is not reachable. Removing these three edges correspond to adding the following environment assumption, which can be monitored at runtime.

$$(\mathbf{G}\ ((x \wedge y) \rightarrow \mathbf{X}\ \neg\bar{x})) \wedge (\mathbf{G}\ ((\bar{x} \wedge \bar{y}) \rightarrow \mathbf{X}\ \neg\bar{x})) \wedge (\mathbf{G}\ ((x \wedge \bar{y}) \rightarrow \mathbf{X}\ \neg\bar{x}))$$

Human factors play an important role in the design of a HuIL controller. The criteria established for a HuIL controller in Section 9.3 also require it to be *prescient* and *minimally intervening*. Hence, we want to mine assumptions that reflect these criteria as well. The notion of *prescient* essentially requires that none of the failure-prone nodes is reachable from a non-failure-prone node with less than T steps (edges). The weight function ϖ introduced earlier can be used to characterized the cost of a failing assumption resulting in the advisory controller prompting the human operator to take over control (by setting *auto* to `false`). Formally, we seek E^ϕ such that the total cost of switching control $\sum_{e \in E^\phi} \varpi(e)$ is minimized.

We can formulate this problem as a *s*-*t* min-cut problem for directed acyclic graphs. Given $\hat{G}^c$, we first compute the subset of nodes $\hat{Q}_T^c \subseteq \hat{Q}^c$ that are backward reachable within

$T-1$ steps from the set of failure-prone nodes (when $T=1$, $\hat{Q}^c_T$ is the set of failure-prone node). We assume that $\hat{Q}^c_0 \cap \hat{Q}^c_T = \emptyset$. Next, we remove the set of nodes $\hat{Q}^c_T$ from $\hat{G}^c$ and obtain a new graph $\hat{G}^c_T$. Since $\hat{G}^c_T$ is again a DAG, we have a set of source nodes and a set of terminal nodes. Thus, we can formulate a s-t min-cut problem by adding a new source node that has an outgoing edge (with a sufficiently large weight) to each of the source nodes and a new terminal node that has an incoming edge (with a sufficiently large weight) from each of the terminal nodes. This s-t min-cut problem can be easily solved by standard techniques [CLR05]. The overall approach is summarized in Algorithm 6.

Algorithm 6 Counterstrategy-Guided HuIL Controller Synthesis

Input: GR(1) specification $\psi = \psi^{env} \rightarrow \psi^{sys}$.
Input: T : parameter for minimum human response time.
Output: $\mathcal{AC}$ and $\mathcal{VC}$. $\mathcal{HIL}$ is then a composition of $\mathcal{AC}$, $\mathcal{VC}$ and the human operator as described in Section 9.3.

if ψ is *realizable* **then**
  Synthesize transducer $M \models \psi$ (using standard GR(1) synthesis);
  $\mathcal{HIL} := M$ (fully autonomous).
else
  Generate G^c from ψ (assume a single G^c; otherwise the algorithm is performed iteratively as in Algorithm 5);
  Generate the DAG embedded $\hat{G}^c$ from G^c.
  Reduce $\hat{G}^c$ to $\hat{G}^c_T$;
  Assign weights to $\hat{G}^c$ using φ; by removing $\hat{Q}^c_T$ – nodes that are within $T-1$ steps of any failure-prone node;
  Formulate a s-t min-cut problem with $\hat{G}^c_T$;
  Solve the s-t min-cut problem to obtain E^ϕ;
  Add assumptions ϕ to ψ to obtain the new specification $\psi_{new} := (\phi \wedge \psi^{env}) \rightarrow \psi^{sys}$;
  Synthesize $\mathcal{AC}$ so that $M \models \psi_{new}$;
  Synthesize $\mathcal{VC}$ as a (stateless) monitor that outputs $auto = \texttt{false}$ iff the assumption ϕ is violated.
end if

Theorem 9.1. *Given a GR(1) specification ψ and a response time parameter T, Algorithm 6 is guaranteed to either produce a fully autonomous controller satisfying ψ, or a HuIL controller, modeled as a composition of an auto-controller $\mathcal{AC}$, a human operator and an advisory controller $\mathcal{VC}$, that is* monitoring, prescient *with parameter T,* minimally intervening *with respect to the cost function $f_C = \sum_{e \in E^\phi} \varpi(e)$, and* conditionally correct.[3]

Proof. When ψ is realizable, a fully autonomous controller is synthesized and unconditionally satisfies ψ.

[3]We assume that all failure-prone nodes are at least T steps away from any initial node. This condition can be checked easily and if it is not satisfied, human should take control immediately.

Now consider that case when ψ is not realizable.

The HuIL controller is *monitoring* as ϕ only comprises a set of environment transitions up to the next environment input.

It is *prescient* by construction. The *auto* flag advising the human operator to take over control is set to `false` precisely when ϕ is violated. When ϕ is violated, it corresponds to the environment making a next-move from the current state according to some edge $e = (\hat{q}_i, \hat{q}_j) \in E^\phi$. Since $\hat{q}_i \notin \hat{Q}^c_T$ by the construction of $\hat{G}^c_T$, $\hat{q}_i$ is at least T transitions away from any *failure-prone* state in $\hat{G}^c$. A failure-prone state in $\hat{G}^c$ corresponds to either a *failure-imminent* state in G^c or a set of *failure-doomed* states in G^c. Hence, when *auto* is set of `false`, it takes at least T number of transitions to reach a state that is *unsafe*.

The HuIL controller is also *conditionally correct*. The algorithm produces an assumption ϕ and an auto-controller $\mathcal{AC}$ that satisfies $\psi_{new} := (\phi \wedge \psi^{env}) \rightarrow \psi^{sys}$. Hence, if ϕ is satisfied by the environment, $\mathcal{AC}$ satisfies ψ and the signal *auto* is set to `true`. By the definition of a counterstrategy to ψ, for any state q in $\mathcal{AC}$, $\mathcal{F}(q) =$ `false`. Now suppose ϕ is first violated when $\mathcal{AC}$ is at a state q. Since $\mathcal{F}(q) =$ `true` if and only if there exist some $q^c \in Q^c$ such that $\theta^c(q^c) = x_k y_k$ and q^c is either *failure-imminent* or *failure-doomed*, q is at least T transitions aways from a state q' where $\mathcal{F}(q') =$ `true`. In addition, $\mathcal{F}(q') =$ `false`, $\forall q' \in R_T(q)$.

Finally, since *auto* is set to `false` precisely when ϕ is violated, and ϕ in turn is constructed based on the set of edges E^ϕ, which minimizes the cost function $f_C = \sum_{e \in E^\phi} \varpi(e)$, the HuIL controller is *minimally-intervening* with respect to the cost function f_C.

□

9.4.3 Switching from Human Operator to Auto-Controller

Once control has been transferred to the human operator, when should the human yield control to the autonomous controller again? We outline here an approach one can take to address this question, while noting that alternative approaches may exist.

The basic idea in our approach is for the HuIL controller to continue to monitor the environment after the human operator has taken control, checking if a state is reached from which the auto-controller can ensure that it satisfies the specification under assumption ϕ, and then take back control.

Recall that the original specification ψ is of the form $\psi^{env} \rightarrow \psi^{sys}$ where ψ^{env} is the original set of assumptions on the environment, including on initial states. Algorithm 6 augments ψ^{env} with an additional assumption ϕ, to obtain the combined environment assumption $\phi \wedge \psi_{env}$. Under this combined assumption, the GR(1) synthesis algorithm is able to extract a winning strategy for the system, which forms the auto-controller. While extracting the winning strategy, we record the set of all states from which the "system" (autonomous controller) can win the game no matter what the environment does for the next T steps (where T is the human response time, as before). This set can be recorded in terms of its characteristic Boolean function, denoted W.

Thus, the HuIL controller continues to monitor the environment and check whether W becomes `true`. If so, the auto-controller notifies the human operator that it is ready to take

back control. As long as W remains `true`, the human operator then may return control to the autonomous system, and the auto-controller executes as before.

9.5 Experimental Results

In this section, we present several scenarios in the context of autonomous driving and demonstrate the usefulness of our approach in synthesizing human-in-the-loop controllers that satisfy the criteria established in Section 9.3. Our algorithm is implemented as an extension to the temporal logic synthesis and diagnosis tool RATSY [BCG+10].

9.5.1 Car-Following

Recall the car-following example shown in Section 9.2. We describe the formalization of the requirements as LTL specifications below.
Input Variables:

1. Position of car B: $p_B \in \{1 \dots 10\}$.
2. Position of car C: $p_C \in \{1 \dots 10\}$.

Output Variables:

1. Position of car A: $p_A \in \{1 \dots 10\}$.
2. Visibility of car A: $v_A \subseteq \{1 \dots 10\}$.
3. $follow =$ `true` iff car A can see the region where car B is in.

Environment Assumptions:

1. Initially, car B is in region 6 and car C is in region 1.

$$p_B = 6 \wedge p_C = 1$$

2. Car B can only move at most one square up at each time step. For example, starting at 6, car B can move to 7 or 8, or stay at 6.

$$\mathbf{G}\ \big((p_A = 6 \rightarrow \mathbf{X}\ (p_A = 6 \vee p_A = 7 \vee p_A = 8)\big)$$

3. Car C can move at most two squares up at each time step. For example, starting at 1, car C can move to 3, 4 or 5, or stay at 1.

$$\mathbf{G}\ \big((p_C = 1 \rightarrow \mathbf{X}\ (p_C = 3 \vee\ p_C = 4\ \vee p_C = 5 \vee p_C = 1)\big)$$

4. Car C does not purposely collide into car A. For example, if car A is at 4, then car C should not move to 4 in the next cycle.

$$\mathbf{G}\big((p_A = 4 \rightarrow \mathbf{X}\ (p_C \neq 4)\big)$$

System Guarantees:

1. Initially, car A is in region 4.

$$p_A = 4$$

2. Car A must not collide with car B or C.

$$\mathbf{G}\ (p_A \neq p_B \wedge p_A \neq p_C)$$

3. Car A can move at most two squares up at each time step. For example, starting at 4, car A can move to 5, 6 or 8, or stay at 4.

$$\mathbf{G}\ \big((p_A = 4 \rightarrow \mathbf{X}\ (p_A = 5 \vee\ p_A = 6\ \vee p_A = 8 \vee p_A = 4)\big)$$

4. Constraints on the visibility regions of car A. When the view of car A is not obstructed by another vehicle directly in front, car A can see squares ahead include the current position on both lanes. This specification simulates the potential limitation on the sensing capabilities on the vehicle. For example, starting at 4, car A is supposed to be able to see 3, 4, 5, 6, 7 and 8, but due to car B being at 6 (thus partially blocking the view of A), car A can only see regions 3, 4, 5 and 6.

$$\mathbf{G}\ \big((p_A = 4 \wedge (p_B = 6 \vee p_C = 6)) \rightarrow (v_A = \{3, 4, 5, 6\})\big)$$

5. Constraints on $follow$. Basically, car A is said to be able to "follow" car B if it can always move into a position where it can see where car B is at.

$$\mathbf{G}\ \big((track = \texttt{true}) \leftrightarrow (p_B \in v_A)\big)$$

We further require that $\mathbf{G}\ (follow = \texttt{true})$.

6. Car A cannot change lane if another car parallel to it is changing lane as well. For example, if car A is at 4 and car C is at 3, and car C is moving to 6, then car A cannot move to 5.

$$\mathbf{G}\ \big((p_A = 4 \wedge p_C = 3 \wedge (\mathbf{X}\ p_C = 6)) \rightarrow (\mathbf{X}\ p_A \neq 5)\big)$$

Observe that car C can in fact force a violation of the system guarantees in one step under two situations – when $p_C = 5, p_B = 8$ and $p_A = 4$, or $p_C = 5, p_B = 8$ and $p_A = 6$. Both are situations where car C is blocking the view of car A, causing it to lose track of car B. The second failure scenario is illustrated in Figure 9.2b.

Applying our algorithm to this (unrealizable) specification with $T = 1$, we obtain the following ϕ.

$$\phi = \mathbf{G}\left(((p_A = 4) \wedge (p_B = 6) \wedge (p_c = 1)) \rightarrow \mathbf{X}\, \neg((p_B = 8) \wedge (p_C = 5))\right) \bigwedge$$

$$\mathbf{G}\left(((p_A = 4) \wedge (p_B = 6) \wedge (p_c = 1)) \rightarrow \mathbf{X}\, \neg((p_B = 6) \wedge (p_C = 3))\right) \bigwedge$$

$$\mathbf{G}\left(((p_A = 4) \wedge (p_B = 6) \wedge (p_c = 1)) \rightarrow \mathbf{X}\, ((p_B = 6) \wedge (p_C = 5))\right) \bigwedge$$

In fact, ϕ corresponds to three possible evolutions of the environment from the initial state. In general, ϕ can be a conjunction of conditions at different time steps as *env* and *sys* progress. The advantage of our approach is that it can produce ϕ such that we can synthesize an auto-controller that is guaranteed to satisfy the specification if ϕ is not violated, together with an advisory controller that prompts the driver (at least) T ($T = 1$ in this case) time step ahead of a potential failure when ϕ is violated.

9.5.2 Gridworld Hallway

Figure 9.4: Gridworld hallway example.

This example is a modified version of the grid world hallway example used in [LMB12]. The controllable vehicle (car A) starts from position I as shown in Figure 9.4. We use p_A the position of car A. We would like to guarantee that car A eventually visits the two goal states G.

$$\mathbf{F}\,(p_A = 2) \wedge \mathbf{F}\,(p_A = 14)$$

We restrict car A to only move in one direction to the right side of the grid, so as it approaches to the right, it won't be able to back up. For example if car A is at position 4, it cannot backup to position 3:

$$\mathbf{G}\,((p_A = 4) \rightarrow \mathbf{X}\,(p_A = 4 \vee p_A = 5 \vee p_A = 11))$$

The uncontrollable vehicle (car B) can only move in the shaded area (regions $\{4, 5, 6, 11, 12, 13\}$) and must leave position E infinitely often. We use p_B to denote position of car B.

$$\mathbf{G}\ (\mathbf{F}\ (p_B \neq 4))$$

There is also a fixed obstacle at position 10.

We list the details of the LTL specifications below.

Input Variables:

1. Position of car B: $p_B \in \{4, 5, 6, 11, 12, 13\}$.
2. Position of the obstacle O: $p_O \in \{1 \ldots 14\}$.

Output Variables:

1. Position of car A: $p_A \in \{1 \ldots 14\}$.

Environment Assumptions:

1. The obstacle O is at a fixed position.

$$\mathbf{G}\ (p_O = 10)$$

2. Car B can only move at most one square in each direction in the shaded area. For example, starting at 12, car B can move to 11, 13 or 5, or stay at 12.

$$\mathbf{G}\ \big((p_A = 12 \rightarrow \mathbf{X}\ (p_A = 12\ \vee\ p_A = 11\ \vee\ p_A = 13\ \vee\ p_A = 5)\big)$$

3. Car B must leave position E infinitely often.

$$\mathbf{G}\ (\mathbf{F}\ (p_B \neq 4))$$

System Guarantees:

1. Initially, car A is in region 1.

$$p_A = 1$$

2. Car A must eventually visit the goal states.

$$\mathbf{F}\ (p_A = 2) \wedge \mathbf{F}\ (p_A = 14)$$

3. Car A must not collide with car B.

$$\mathbf{G}\ (p_A \neq p_B)$$

4. Car A must not collide with the obstacle O.

$$\mathbf{G}\ (p_A \neq p_O)$$

5. Car A can move at most one squares to the right side of the grid at each time step, or move one square up or down. For example, given that car A is at 5, it can move to 6, 12, or stay at 5.

$$\mathbf{G}\ \big((p_A = 5 \rightarrow \mathbf{X}\ (p_A = 5 \vee\ p_A = 6\ \vee p_A = 12)\big)$$

The controllable vehicle cannot synthesize a controller for this example as it has to pass the shaded region to visit its goal at position 14, and the uncontrollable vehicle can mirror the controllable vehicle's movement. For instance, if car A moves into the shaded area at position 4, car B already located at 5 can block its path by staying at 5. If car A moves to 11, then car B can move to 12. The different scenarios in ϕ are depicted in Figure 9.5. Notice that car B can stay in 6 to block car A's path later as car A moves closer to the goal.

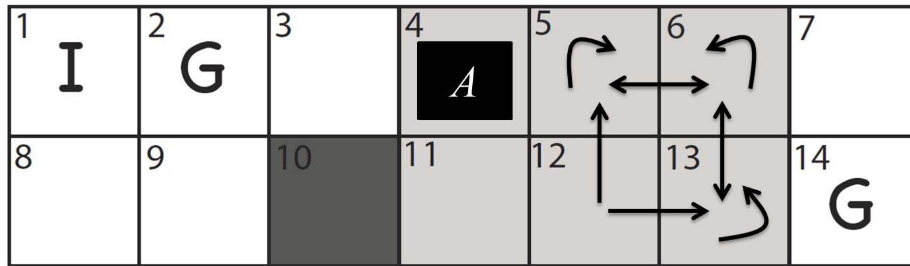


Figure 9.5: Illustration of ϕ: arrows indicate all possible movements of car B from the current position to the next position.

9.6 Additional Related Work

In recent years, there has been an increasing interest in human-in-the-loop systems in the control systems community. The current human-in-the-loop control paradigm studies and learns human models and takes action whenever it concludes that the human user is not capable of controlling the system. Anderson et al. [APPI10] study obstacle avoidance and lane keeping for semi-autonomous cars, which is a common example of human-in-the-loop control. The control input of the semi-autonomous vehicle is a weighted sum of control input of driver and control input of the autonomous system with weights representing threat functions that only depend on side-slip angles of the vehicle. The autonomous control in this work is a model predictive control which is an iterative, finite-horizon optimization of the plant model. Our approach, unlike this one, seeks to provide correctness guarantees in the form of temporal logic properties.

Verma et al. [VDV11, VDV12] consider safety and collision avoidance in multi-vehicle systems, where a human driven vehicle is not communicating with a fully autonomous system. In this scenario, the autonomous vehicle finds the least restrictive safe control action by modeling the human driven car as a hidden mode hybrid system (HMHS) and estimating the state of the human driver. The control of the autonomous vehicle is less conservative and least restrictive since this model does not assume an adversarial human driven car. This approach finds a safe control for a fully autonomous vehicle in a multi-vehicle situation with non-communicating human driven cars, which is a different situation from the HuIL control in this work.

Vasudevan et al. [VSG+12] focus on learning and predicting a human model based on prior observations (past states of the driver, past vehicle trajectories and past generated optimal vehicle trajectories) and current state of the environment. The learned human model is then used in a "vehicle intervention function". Based on the measured level of threat, the controller intervenes and overwrites the driver's input. However, we believe that this paradigm, where the autonomous controller is capable of overriding the human inputs is unsafe in scenarios where the environment is not fully modeled. For that reason, we propose a different paradigm where we allow the human to take control of the vehicle if the autonomous system predicts failure, and for the autonomous vehicle to request back control when it is certain of safe operation.

9.7 Summary

In this chapter, we propose a formalism of human-in-the-loop controllers. These controllers are motivated by the need to safely and soundly integrate human operators in the ongoing drive towards automation, especially in the automotive industry. We study a setting where such controllers are automatically synthesize from their temporal logic specifications. We show that, by extending the assumption mining algorithm described in Chapter 8, we can obtain these controllers in a correct-by-construction manner.

Chapter 10

Mining Assumptions from Natural Language Specifications

> Natural language will always remain the basic interpretation of, and reservoir for, the development of the artificial formalized languages of science.
>
> – Doris Bradbury

The *lingua franca* of formal methods is *logic*, which provides an unambiguous semantics to the (formal) language describing a design, and the means to reason with it. However, most people who experience or interact with computers today are “end-users”. “End-users” are not expert logicians, and their way of describing their usage to others is through natural language. In many cases, even domain experts, such as circuit designers, resort to natural language as the main vehicle for communicating their mental model of a design to consumers of such model, as evidenced by the large proportion of design documents still written in natural language today. Hence, formal methods encapsulated in NLP layers can bring greater accessibility to the engineering discipline, especially at the requirement stage, by liberating “end-users” from the burden of learning formal logic. Figure 10.1 illustrates our vision, where a non-expert user of formal analysis can still use it to reason about problems of natural language requirement.

In this chapter, we propose that assumption mining, in the context of requirement engineering, can be useful to “end-users” when encapsulated inside a NLP layer. We first review related work in Section 10.1. We then describe our natural language to LTL conversion workflow in Section 10.2. In Section 10.3, we present a detailed case study on applying our approach to a set of requirements taken from a publicly available document released by the Federal Aviation Administration. Finally, we conclude in Section 10.4.

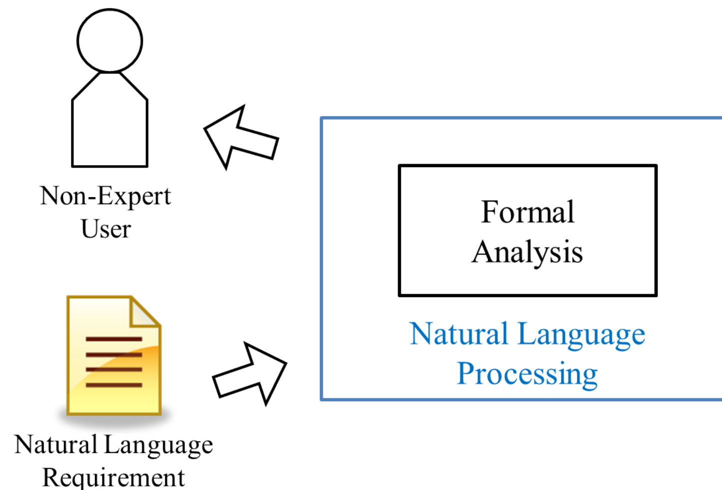


Figure 10.1: Non-expert user uses formal methods to analyze problems in a design document, facilitated by a NLP layer.

10.1 Related Work in NLP

There is a rich body of work related to requirement engineering. In this section, we focus on efforts that aim that connecting (semi-)natural language input to formal methods.

Kress-Gazit et al. [KGFP08], Smith et al. [SACO02] and Shimizu et al. [Shi02] propose grammars for representing requirements in controlled natural language. However, these grammars are very application-specific and rather restrictive. In addition, they can pose steep learning curves for new users. In this work, we try to tackle a much wider spectrum of requirements written in natural language directly.

Zowghi et al. [ZGM01], Gervasi et al. [GZ05], Scott et al. [SCK04], and Xiao et al. [XPTX12] propose to process constrained natural language text using NLP tools, such as CFG and Cico, and perform various kinds of checks, such as consistency and access control, on the requirements. Our work is similar in spirit. We combine a general NLP front-end with domain-specific ontology to generate meaningful temporal logic formulas. We also extend beyond consistency (satisfiability and realizability) checks by finding recommended fixes in the form of additional environment assumptions.

Behavior-driven development is a framework where natural language specifications are used during the software testing phase. Drechsler et al. [DDG+12], Soeken et al. [SWD12], and Harris [Har12] show different ways of translating high-level natural language requirements to tests in this framework, which can then be used by tools like Cucumber [cuc] or RSpec [rsp]. Our work focus on the problem of *specification*, which can also occur early in a design cycle, and uses NLP to mediate the interaction between formal analysis (e.g., assumption mining) and non-expert users of these methods.

More recently, Keszocze et al. [KSKD13] proposed an Eclipse-based IDE, called Lips, that

leveraged state-of-the-art NLP algorithms to extract structural models as Unified Modeling Language diagrams and specifications as Object Constraint Language expressions. While our NL to formal specification translation is similar in spirit, our target specification (i.e. LTL) is different. In addition, we use this translation to further demonstrate the usefulness of assumption mining when applied to natural language requirements.

10.2 Natural Language to LTL Formula

In this section, we give an overview of our technique for translating an English requirement to its corresponding LTL formula. The techniques presented here are based on the work in [GEL+13]. The details of the NLP techniques are outside the scope of this thesis, which is really about mining specifications (specifically, environment assumptions in this chapter). For ease of understanding, we describe the key steps below.

Our choice of LTL is motivated, in part, by the increasing adoption of Accellera's Property Specification Language (PSL) for specifying assertions about hardware designs. A thorough introduction to PSL can be found in [EF06]. In particular, PSL is based on LTL but also extends it to cover the full expressiveness of ω-regular languages. We also observe that design documents, especially those for hardware designs, are mostly written in stylized or semi-natural languages, which often look akin to LTL formulas.

Mapping from a natural language sentence to a temporal logic formula is not trivial. We highlight the two main challenges below.

- Syntactical difference: Natural language requirements, especially those used in the Engineering discipline, are arguably more structured, say than a sentence that would appear on the Wall Street Journal. However, these are still sentences governed by a much richer and more complicated syntax than that of LTL. Hence, the reconciliation of the huge difference in syntax cannot be addressed simply by a domain-specific front-end.

- Association of semantics: In addition to handling sentences with much less structure, we need to map them to LTL formulas in a way that the semantics are correctly preserved. Further, this translation should be done with minimal assistance from the user. Hence, this entails recognizing named-entities (variables and their values), and recognizing Boolean as well as temporal relations.

Figure 10.2 illustrates our step-by-step solution to the challenges mentioned above.

10.2.1 Preprocessor

Given the following requirement in English,

If the Regulator Mode equals INIT, the Heat Control shall be set to Off. (10.1)

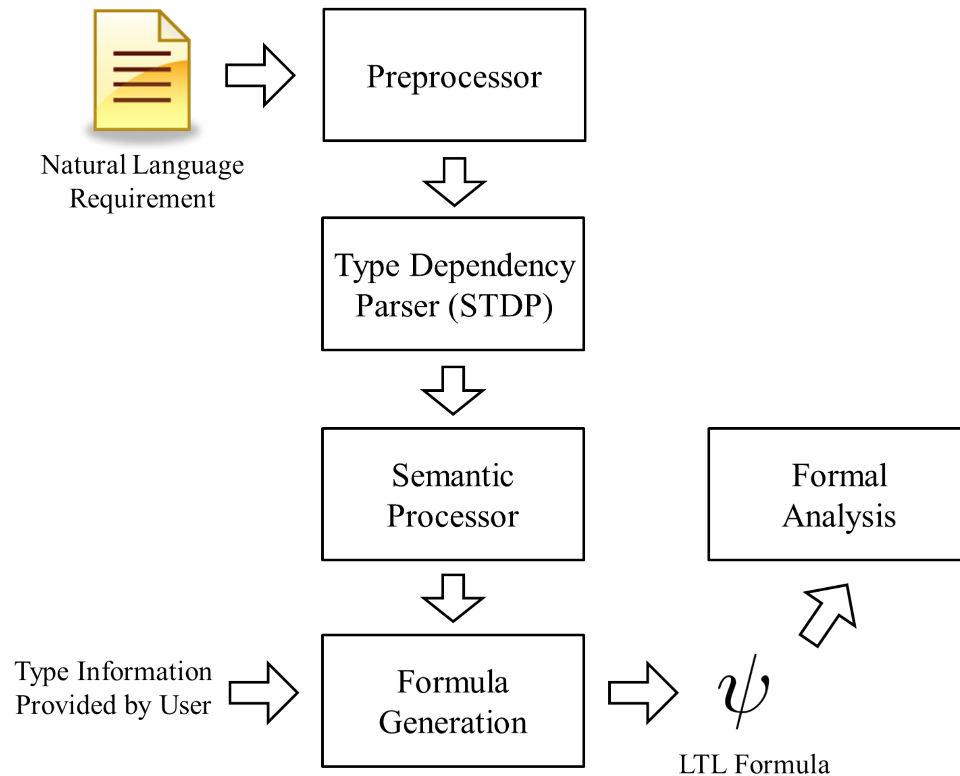


Figure 10.2: Workflow: NL → LTL → formal analysis.

a preprocessor is first used to identify n-grams that correspond to entities specific to the domain. In this example, "Regulator Mode", "INIT", "Heat Control" and "Off" are instances of these, called *terms*. The sentence is then rewritten with each of these n-grams converted to a single word. For example, "Regulator Mode" will be replaced by "Regulator_Mode". Below is the modified sentence.

If the Regulator_Mode equals INIT, the Heat_Control shall be set to Off. (10.2)

This simple step helps the Stanford Typed Dependency Parser (STDP) [dMMM06], which we will use next, to reliably produce a correct parsing of grammatical relations between words in the requirement sentence.

10.2.2 Stanford Type Dependency Parser (STDP)

The syntactic parser in STDP parses the requirements text to obtain unique entities called *mentions*, while the dependency parser generates *grammatical relations* between the mentions. The output of STDP is a set of dependencies in the form of type dependency (TD) triplets each representing grammatical relations between two mentions. In this thesis, we

use *grammatical relations* and *type dependencies* (or simply, *dependencies*) interchangeably. Specifically, the TD triplets have the form:

$$\langle \text{name of the relation} \rangle (\langle \text{governor} \rangle, \langle \text{dependent} \rangle)$$

For example, "*prep_to*(set-11, Off-13)" is one such triplet indicating that "set" is related to "Off" via the prepositional connective "to". The number suffices in the triplet indicate the positions of the mentions in the sentence. Figure 10.3 shows the dependencies generated for Requirement 10.2 in the form of a directed graph, with "set-11" being the root of the graph.

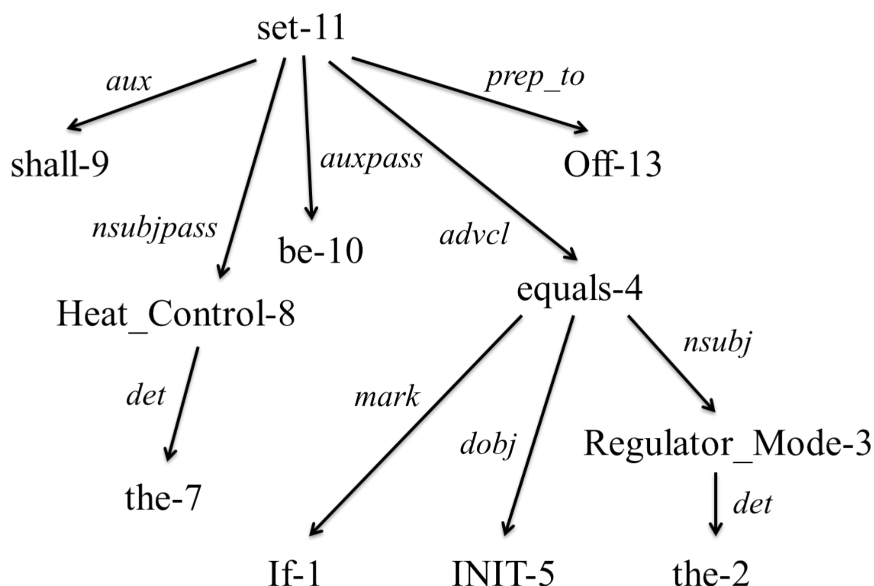


Figure 10.3: Dependencies generated using STDP.

10.2.3 Semantic Processor

Our semantic processor uses the output from STDP and systematically applies a set of *type rules* to the mentions and dependencies to associate specific meanings to them. Each type rule specifies a mapping from a set of dependencies (grammatical relations between mentions) to a set of predicates with built-in "semantics".

For example, the following type rule simply maps a grammatical relations over the mention $?g$ to a unary predicate specifying that the mention is a unique term. For simplicity, we omit the curly brackets "{" and "}" and separate the predicates in the same set by "," on both sides of "→".

$$det(?g,\texttt{the}) \rightarrow unique(?g)$$ [1]

Note that the here refers to an actual mention of the word "the" in the sentence. Applying this rule to the dependencies generated for Requirement 10.2 produces two unary predicates *unique*(Heat_Control-8) and *unique*(Regulator_Mode-3).

A slightly more complicated example is the following type rule that maps two grammatical relations *advcl*($?g$,$?d$) and *mark*($?d$,if) for any two mentions $?g$ and $?d$ to the binary predicate *impliedBy*($?g$,$?d$).

$$advcl(?g,?d),\ mark(?d,\texttt{if}) \rightarrow impliedBy(?g,?d)$$ [2]

Note that here the predicate *impliedBy*($?g$,$?d$) means $?d$ *logically implies* $?g$. Applying the above rule to Requirement 10.2 we obtain *impliedBy*(set-11,equals-4).

The type rules we use for Requirement 10.2 and the result of applying them to the dependencies generated by STDP are listed below.

- *det*($?g$,the) → *unique*($?g$):
  unique(Regulator_Mode-3)
- *det*($?g$,the) → *unique*($?g$):
  unique(Heat_Control-8)
- *nsubj*($?g$,$?d$)[3] → *ARG1*($?g$,$?d$)[4], *unique*($?d$):
  ARG1(equals-4, Regulator_Mode-3), *unique*(Regulator_Mode-3)
- *dobj*($?g$,$?d$)[5] → *ARG2*($?g$,$?d$)[6], *unique*($?d$):
  ARG2(equals-4, INIT-5), *unique*(INIT-5)
- *prep_to*($?g$, $?d$) → *ARG2*($?g$,$?d$), *unique*($?d$):
  ARG2(set-11, Off-13), *unique*(Off-13)
- *nsubjpass*($?g$, $?d$)[7] → *ARG1*($?g$,$?d$), *unique*($?d$):
  ARG1(set-11, Heat_Control-8), *unique*(Heat_Control-8)
- *advcl*($?g$,$?d$), *mark*($?d$,if) → *impliedBy*($?g$,$?d$):
  impliedBy(set-11, equals-4)

[1] *det* stands for determiner, and *unique* is a unary predicate describing that its argument is a unique term.
[2] *advcl* stands for adverbial clause modifier and *mark* stands for marker.
[3] *nsubj* stands for nominal subject.
[4] *ARG1* is a predicate stating that the first argument of $?g$ is $?d$.
[5] *dobj* stands for direct object.
[6] *ARG2* is a predicate stating that the second argument of $?g$ is $?d$.
[7] *nsubjpass* stands for passive nominal subject.

Remark 10.1. *In essence, the type rules are used to extract domain-specific terms (e.g., "Regulator_Mode") and domain-independent phrases that are indicative of simple mathematical expressions (e.g., "equals") as well as logical and temporal relations (e.g., "If"). They also help to filter out natural language artifacts (e.g., "shall") that are not important to expressing the natural language requirement in LTL.*

Figure 10.4 shows the resulting predicate graph after applying the type rules to type dependency graph shown in Figure 10.3. Similar to the type dependency graph, the edges represent binary predicates. The unary predicate *unique* is indicated by boxes with blue borders. Additionally, mentions containing indicative words such as "equals" and "set" (shown in red in Figure 10.4) are associated with predefined predicates *equal* and *set*.

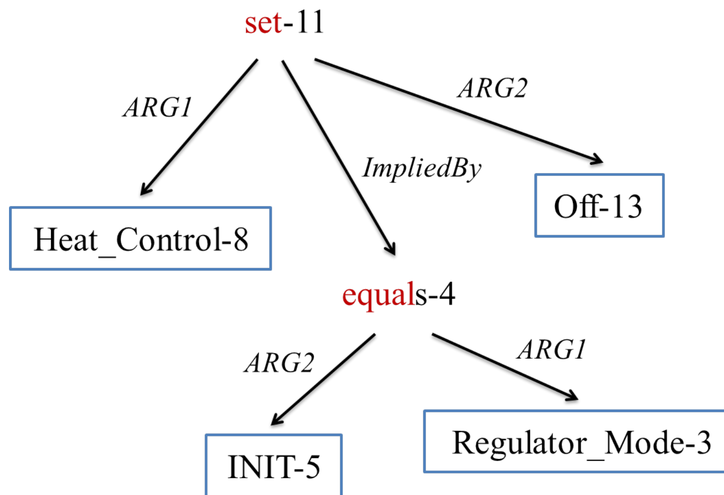


Figure 10.4: Predicate graph after the application of type rules.

Remark 10.2. *The set of type rules we use in this work in by no means complete. However, this decomposition of syntax parsing and semantic association offers the flexibility of mapping a natural language sentence to potentially different target logics. It also allows a user to use a general-purpose parser without restricting herself to a grammar that is domain or application-specific.*

10.2.4 Formula Generation

Syntactic translation. We use a set of *expression translation rules* to translate the generated predicates and mentions to an LTL formula. The expression rules trs are categorized into four types of rules – tr^t for temporal operators, tr^l for logical operators, tr^m for simple arithmetic (e.g., equality) expressions and tr^u for terms. We use $e(X)$ to denote the expression associated with mention X, which we will repeatedly rewrite during the translation

Table 10.1: Formula Translation Rules

Predicate	Expression Translation
unique(X)	$tr^u(e(X)) : e(X)$
set(X) ∧ *ARG1*(X, Y) ∧ *ARG2*(X, Z)	$tr^m(e(X)) : tr^u(e(Y)) = tr^u(e(Z))$
equal(X) ∧ *ARG1*(X, Y) ∧ *ARG2*(X, Z)	$tr^m(e(X)) : tr^u(e(Y)) = tr^u(e(Z))$
less(X) ∧ *ARG1*(X, Y) ∧ *ARG2*(X, Z)	$tr^m(e(X)) : tr^u(e(Y)) < tr^u(e(Z))$
greater(X) ∧ *ARG1*(X, Y) ∧ *ARG2*(X, Z)	$tr^m(e(X)) : tr^u(e(Y)) > tr^u(e(Z))$
impliedBy(X, Y)	$tr^l(e(X)) : tr^m(e(Y)) \to tr^m(e(X))$
and(X, Y)	$tr^l(e(X)) : tr^m(e(X)) \wedge tr^l(e(Y))$[8]
or(X, Y)	$tr^l(e(X)) : tr^m(e(X)) \vee tr^l(e(Y))$[9]
not(X)	$tr^l(e(X)) : \neg tr^m(e(X))$

process. If mention X is associated with a unique term, i.e. $unique(X)$, its expression is simply the English word in the mention, e.g., e(Heat_Control-8) = Heat_Control. Given a graph representing the predicates over the mentions, we recursively apply the translation rules starting from the root (e.g., set-11 in Figure 10.4). When multiple rules are applicable to the same mention, they are applied in the order of tr^l followed by tr^t, tr^m and then tr^u. The translation rules used for Requirement 10.2 are given in Table 10.1.

The resulting LTL formula, translated from Requirement 10.2, is shown below.

$$\mathbf{G}\ \big((\text{Regulator_Mode} = \text{INIT}) \to (\text{Heat_Control} = \text{Off})\big)^{10} \tag{10.3}$$

Variables and values. An LTL formula is defined over atomic propositions. Hence, we need to further distinguish variables and values among terms. This procedure involves two phases. First, the left argument of an arithmetic expression is considered as a variable and the right argument is considered as a possible value of that variable, unless it has already been categorized as another variable. Second, the domain of a variable (of enumerated type) is computed by aggregating all the possible values of that variable across all the sentences. In this thesis, we assume each variable either has an enumerated type containing a finite number of elements or has an integer type.

Additional type information. Observe that we do not have predicates and translation rules corresponding to the temporal operator **X**. Currently, we rely on additional user input to generate this information. We elaborate this below.

The semantics of LTL are usually interpreted over Kripke structures [Kri63], which are labeled transition systems commonly used in model checking [CGP00]. However, *transitions* may not be described explicitly in a natural language requirement. Consider the following

[8] If $tr^l(e(Y))$ cannot be applied, $tr^m(e(Y))$ is attempted next.

[9] Similar to the translation rule above, if $tr^l(e(Y))$ cannot be applied, $tr^m(e(Y))$ is attempted next.

sentence.

If the Regulator_Status equals True, the Regulator_Mode shall be set to NORMAL. (10.4)

Suppose that "Regulator_Mode" is a state variable in the transition system and "Regulator_Status" is an input, then this sentence is in fact describing a guarded transition and should be more appropriately translated to

$$\mathbf{G}\ \big((\text{Regulator_Status} = \texttt{true}) \rightarrow \mathbf{X}\ (\text{Regulator_Mode} = \text{NORMAL})\big) \tag{10.5}$$

Hence, we require the user to provide additional type information for each of the variables to resolve this ambiguity. A variable can either be an *input*, a *state and output*, or a *pure output*. We intentionally enforce internal variables (state or non-state) to be made *output* so that we can check *realizability* of the resulting LTL formula.

10.3 Case Study

In this section, we describe a focused study on requirements from an Isolette design to further showcase the usefulness of assumption mining when applied to NL sentences.

An Isolette is an incubator for infants that provides controlled temperature, humidity and oxygen. Our example is an Isolette Thermostat that regulates the air temperature inside an Isolette such that it is maintained within a desired range. The Thermostat is composed of two interacting modules – the "Regulate Temperature" module and the "Monitor Temperature" module. We focus on the "Regulate Temperature" module, which receives input from the "Operator Interface" and the "Monitor Temperature" module, and produces output to the "Heat Source". The requirements are taken from Appendix A of the "Requirement Engineering Management Handbook" released by the Federal Aviation Administration, which was intended to serve as an example of the "best practices" advocated in this handbook [LM09]. The English sentences describing the requirements, as well as their sources in the document, are tabularized in Table 10.2.

As described in Section 10.2, we require the user to provide additional type information for each variable. For this FAA-Isolette example, we assume the following type information is given.

- *Input:*

[11] Due to the ambiguity of "is" and STDP having difficulty parsing it, we preprocessed the sentences by replacing "is" by "equals".

[12] Requirement 10 and 11 were written based on the definition in Table A-10 and before any formal analysis was conducted.

[13] Requirement 11-15 are written based on Figure A-4 in the document and and before any formal analysis was conducted.

Table 10.2: Isolette requirements in English

	Requirement in English[11]	Source
1	If the Regulator Mode equals INIT, the Output Regulator Status shall be set to Init.	REQ-MRI-1
2	If the Regulator Mode equals NORMAL, the Output Regulator Status shall be set to On.	REQ-MRI-2
3	If the Regulator Mode equals FAILED, the Output Regulator Status shall be set to Failed.	REQ-MRI-3
4	If the Status attribute of the Lower Desired Temperature or the Upper Desired Temperature equals Invalid, the Regulator Interface Failure shall be set to True.	REQ-MRI-6
5	If the Status attribute of the Lower Desired Temperature and the Upper Desired Temperature equals Valid, the Regulator Interface Failure shall be set to False.	REQ-MRI-7
6	If the Regulator Mode equals INIT, the Heat Control shall be set to Off.	REQ-MHS-1
7	If the Regulator Mode equals NORMAL, and the Current Temperature is less than the Lower Desired Temperature, the Heat Control shall be set to Control On.	REQ-MHS-2
8	If the Regulator Mode equals NORMAL, and the Current Temperature is greater than the Upper Desired Temperature, the Heat Control shall be set to Control Off.	REQ-MHS-3
9	If the Regulator Mode equals FAILED, the Heat Control shall be set to Control Off.	REQ-MHS-5
10	If the Regulator Interface Failure is set to False, and the Regulator Internal Failure is set to False, and the Status attribute of the Current Temperature is set to Valid, the Regulator Status shall be set to True.	Table A-10[12]
11	If the Regulator Interface Failure is set to True or the Regulator Internal Failure is set to True or the Status attribute of the Current Temperature is not set to Valid, the Regulator Status shall be set to False.	Table A-10
12	The Regulator Mode shall be initialized to INIT.	Req MRM 1[13]
13	If the Regulator Mode equals INIT and the Regulator Status equals True, the Regulator Mode shall be set to NORMAL.	Req MRM 2
14	If the Regulator Mode is set to NORMAL and the Regulator Status is set to False, the Regulator Mode shall be set to FAILED.	Req MRM 3
15	If the Regulator Mode is set to INIT and the Regulator Init Timeout is set to True, the Regulator Mode shall be set to FAILED.	Req MRM 4

  - Upper_Desired_Temperature_Status
  - Lower_Desired_Temperature_Status
  - Regulator_Init_Timeout
  - Current_Temperature
  - Lower_Desired_Temperature
  - Upper_Desired_Temperature
  - Regulator_Internal_Failure
  - Current_Temperature_Status
  - Regulator_Interface_Failure
  - Regulator_Status

- *State and output:* Regulator_Mode
- *Pure output:* Output_Regulator_Status, Heat_Control

Using this information, we are now ready to generate LTL [GR(1)] formulas. The generated LTL formulas corresponding to the English sentences are listed below.

1. $\mathbf{G}\ \big((\text{Regulator_Mode} = \text{INIT}) \rightarrow (\text{Output_Regulator_Status} = \text{Init})\big)$
2. $\mathbf{G}\ \big((\text{Regulator_Mode} = \text{NORMAL}) \rightarrow (\text{Output_Regulator_Status} = \text{On})\big)$
3. $\mathbf{G}\ \big((\text{Regulator_Mode} = \text{FAILED}) \rightarrow (\text{Output_Regulator_Status} = \text{Failed})\big)$
4. $\mathbf{G}\ \big((\text{Upper_Desired_Temperature_Status} = \text{Invalid} \vee \text{Lower_Desired_Temperature_Status} = \text{Invalid}) \rightarrow (\text{Regulator_Interface_Failure} = \texttt{true})\big)$
5. $\mathbf{G}\ \big((\text{Upper_Desired_Temperature_Status} = \text{Valid} \wedge \text{Lower_Desired_Temperature_Status} = \text{Valid}) \rightarrow (\text{Regulator_Interface_Failure} = \texttt{false})\big)$
6. $\mathbf{G}\ \big((\text{Regulator_Mode} = \text{INIT}) \rightarrow (\text{Heat_Control} = \text{Control_Off})\big)$
7. $\mathbf{G}\ \big((\text{Regulator_Mode} = \text{NORMAL} \wedge \text{Current_Temperature} < \text{Lower_Desired_Temperature} = \texttt{true}) \rightarrow (\text{Heat_Control} = \text{Control_On})\big)$
8. $\mathbf{G}\ \big((\text{Regulator_Mode} = \text{NORMAL} \wedge \text{Current_Temperature} > \text{Upper_Desired_Temperature} = \texttt{true}) \rightarrow (\text{Heat_Control} = \text{Control_Off})\big)$
9. $\mathbf{G}\ \big((\text{Regulator_Mode} = \text{FAILED}) \rightarrow (\text{Heat_Control} = \text{Control_Off})\big)$
10. $\mathbf{G}\ \big((\text{Regulator_Interface_Failure} = \texttt{false} \wedge \text{Regulator_Internal_Failure} = \texttt{false} \wedge \text{Current_Temperature_Status} = \text{Valid}) \rightarrow (\text{Regulator_Status} = \texttt{true})\big)$

11. $\mathbf{G}\ \big((\text{Regulator_Interface_Failure} = \texttt{true} \ \vee\ \text{Regulator_Internal_Failure} = \texttt{true} \ \vee\ \neg(\text{Current_Temperature_Status} = \text{Valid})) \rightarrow (\text{Regulator_Status} = \texttt{false})\big)$

12. Regulator_Mode = INIT

13. $\mathbf{G}\ \big((\text{Regulator_Mode} = \text{INIT} \ \wedge\ \text{Regulator_Status} = \texttt{true}) \rightarrow \mathbf{X}\ (\text{Regulator_Mode} = \text{NORMAL})\big)$

14. $\mathbf{G}\ \big((\text{Regulator_Mode} = \text{NORMAL} \wedge \text{Regulator_Status} = \texttt{false}) \rightarrow \mathbf{X}\ (\text{Regulator_Mode} = \text{FAILED})\big)$

15. $\mathbf{G}\ \big((\text{Regulator_Mode} = \text{INIT} \wedge \text{Regulator_Init_Timeout} = \texttt{true}) \rightarrow \mathbf{X}\ (\text{Regulator_Mode} = \text{FAILED})\big)$

As described in Section 10.2, the domain of each variable is gathered and computed across all sentences. For example, the variable Regulator_Mode has taken one of the three values INIT, NORMAL or FAILED. Thus, Regulator_Mode is given an *enumerated type* with elements INIT, NORMAL and FAILED. This is useful when we need to use a Boolean encoding of the domains in the formal analysis stage[14].

Assumption mining: In general, both *satisfiability* and *realizability* can be used to reason about the consistency of a set of LTL formulas. In this case, we verified that the conjunction of the generated formulas are indeed satisfiable. The next step is to check if the requirement is realizable, i.e. there exists an implementation (e.g. Mealy transducer) that satisfies the generated formulas.

Observe that each of the generated formulas complies with the GR(1) syntax. Hence, given the input and output type information of the variables, we can group them into $\psi_i^{env}, \psi_t^{env}, \psi_f^{env}, \psi_i^{sys}$, $\psi_t^{sys}, \psi_f^{sys}$ and check if the formula $\psi = \psi^{env} \rightarrow \psi^{sys}$ is realizable. Specifically, Formula $4, 5, 10, 11$ are environment assumptions (ψ^{env}) and the rest are system guarantees (ψ^{sys}).

Using our counterstrategy-guided approach described in Chapter 8, we can also find additional assumptions as recommended fixes to the specification in case it is not realizable. In the FAA-Isolette example, ψ is not realizable and the following assumption ϕ is produced by our algorithm, such that $\phi \wedge \psi^{env} \rightarrow \psi^{sys}$ is realizable.

$$\phi := \mathbf{G}\ \neg(\text{Regulator_Status} = \texttt{true} \wedge \text{Regulator_Init_Timeout} = \texttt{true}) \tag{10.6}$$

To better understand why this assumption is necessary for realizability, observe that there is in fact unresolved nondeterminism between Requirement 13 and Requirement 15, as illustrated in Figure 10.5.

[14]For this example, Boolean encoding was performed manually on the generated LTL formulas so that they could be used directly by a synthesis tool like RATSY. We note that further automation of this process requires predicate abstraction for equality and inequality constraints over integers.

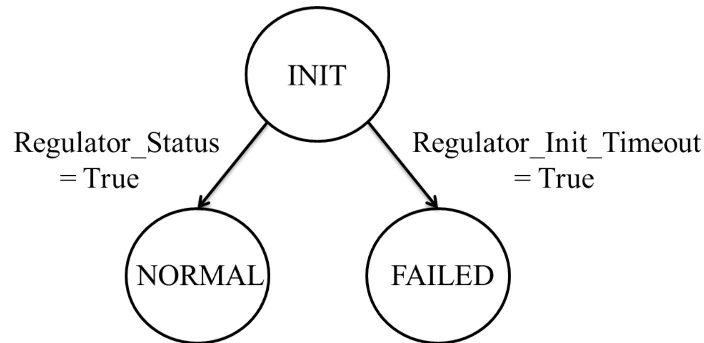


Figure 10.5: Transition of Regulator_Mode based on Requirement 13 and 15.

Specifically, when both Regulator_Status = `true` and Regulator_Init_Timeout = `true`, it is not clear what the intended transition of Regulator_Mode should be. The mined assumption, on the other hand, puts a constraint on the environment by prohibiting this input condition. While this assumption is not necessarily true in the Isolette design environment, it clearly pinpoints the source of the problem. The user can either decide if the assumption is valid or use it as a hint to modify the existing requirements. We also envision that presenting an English translation of ϕ to the user can help ease the difficulty of understanding LTL formulas. An example is shown below, with words corresponding to temporal and Boolean operators highlighted in Italic.

Assumption: It is *always not* the case that Regulator Status equals true *and* Regulator Init Timeout equals true *at the same time*.

Nondeterminism is not the only culprit of unrealizability. In general, the counterstrategy explains why a specification is not realizable. However, the counterstrategy graph can be too large to comprehend by a user visually. In addition, it is difficult to expect a non-expert user of formal methods to first formalize the specification and then reason with it using the counterstrategy. Thus, we propose to embed assumption mining inside a NLP layer to broaden the applicability of this analysis.

10.4 Summary

We have presented a workflow for automatically converting natural language requirements to their corresponding LTL formulas. This workflow allows us to lift formal requirement analysis, particular assumption mining, to the natural language level. Our case study on the FAA-Isolette example shows that our technique is useful for not only identifying bugs but also recommending fixes at the requirement stage of a design. In the future, we aim to extensively evaluate and improve the robustness of the NLP part of the workflow.

Chapter 11

Conclusion and Future Work

This chapter summarizes the main contributions of this thesis and suggests avenues for future work.

11.1 Closing Remarks

Specification is a critical step in ensuring the correctness of a design. However, the process of writing a good formal specification remains largely a manual and error-prone process. This thesis proposes to mitigate this problem by mining likely specifications of a design given evidence on how the design should or should not behave. In prior work, these evidences are often treated as the end products of automation. In this thesis, the frontier of automation is pushed further by employing the proposed specification mining techniques which summarize the evidences intelligently and give recommendations to the users. The proposed techniques are supported by novel formalisms of specification (and specification mining), such as the generalization of the template-based approach presented in Section 2.4 and the sparse-coding approach presented in Chapter 5. For each of these formalisms, we present algorithms and optimizations that are tailored to the formalism and the application domain. For instance, the *counterstrategy-guided* assumption mining approach described in Chapter 8 makes novel use of the information produced when a synthesis process fails, and enables a systematic framework that closes the loop between under-specification by a user and synthesis from specification by a mechanized tool. Another example is the synthesis of human-in-the-loop controllers, presented in Chapter 9, which adapts the counterstrategy-guided algorithm to handle domain-specific constraints (e.g., human response time). In addition to studying formalisms and algorithms, this thesis also makes an endeavor on broadening the scope of specification mining. In Chapter 6, we propose a crowdsourced specification mining game called CrowdMine, which gamifies the process of mining behavioral patterns in subtraces in a way that non-experts can collectively contribute to solving the problem. In Chapter 10 of the thesis, we lift the proposed technique of mining environment assumptions to the direct treatment of requirements written in natural language, by suitably incorporating NLP

techniques. Experimental results demonstrate the practical utility of the proposed techniques to a wide variety of applications, ranging from bug localization in hardware to the synthesis of controllers in semi-autonomous cars.

We envisage a new paradigm where specification mining becomes an integral part of a verification or synthesis flow. In this new paradigm, specification mining is used to bridge multiple gaps between the labor-intensive tasks such as specification, environment modeling and error localization and the largely automated procedures such as model checking and realizability checking. Moreover, this thesis postulate that, by leveraging human inputs in an intelligent way, the process of specification mining not only can be mediated by quick expert inspection but also can be driven by non-expert inputs, with the support of technologies such as natural language processing or crowdsourced games.

11.2 Future Work

In this section, we discuss potential avenues for future work.

11.2.1 Combining Sparse Coding and Automata-Based Specification Mining

The sparse-coding approach, described in Chapter 5, can be viewed as a technique that extracts bounded temporal properties. The drawback, however, is that it becomes inefficient as the window size becomes large. The automata-based approach, described in, Chapter 4, on the other hand, can mine simple temporal properties across a large span of cycles very quickly. Its drawback though lies in the fact that memory requirement of the algorithm grows geometrically with the size of the template alphabet. While the post-processing step described in Section 4.3 aims to mitigate this problem by merging patterns that contain events always occurring at the same time, it cannot handle complex events that span a small time window, e.g., an event comprising two smaller events always occurring in sequence.

The complementary strengths and weaknesses of the two approaches motivate a combined technique by executing the two approaches in stages. First, the sparse-coding approach is used to compute a set of bases, for some small numbers of p (window size). These basis subtraces then form the constituent events for which the automata-based approach will operate on. The result is then a set of likely temporal properties over these composite events (basis subtraces). We hypothesize that this combination can be useful in finding more complex patterns and interactions and we plan to investigate it further in future work.

11.2.2 Compositional Analysis

Compositional analysis is the only way to scale both verification and synthesis to the ever-increasing complexity of designs today. One central component in compositional analysis is *interface*, which specifies the inputs and outputs of each component and how any two

components may interact with each other. *Contracts*, which are pairs of properties representing the assumptions on the environment and guarantees that a component should fulfill under these assumptions, are often used in verification, such as the assume-guarantee rule in hardware verification [Mcm99], and more recently synthesis of analog circuits [NSVSP12]. We envision our specification mining technique to be applicable to both of these domains, in a setting where the contracts are specified as temporal properties.

Assumption Mining for Verification

Modular verification, which is the proposition that one can verify individual components of a design in isolation, and then as a whole, also reason about the end-to-end properties of the design, is one of the holy-grails of verification. The key challenge, even when interfaces of the components are given, lies in finding the contracts, and particularly, the environmental assumptions. The result of an missing assumption on the environment often leads to spurious counterexamples, and in turn waste of efforts in trying to identify and fix false bugs. In fact, it is one of the main factors that hinders the adoption of formal verification in large-scale industrial settings. Recent efforts, such as automating assumption generation via learning [CGP03, GMF] and finding correlation between real counterexamples and mined assumptions [MBD12], have been made to mitigate this problem. Our suite of specification mining techniques further strengthens this portfolio. In particular, our counterstrategy-guided approach shares much in spirit with the work by Mitra et al. [MBD12] which mines simple patterns from counterexamples. We plan to explore this direction further in future work.

Contract-Based Synthesis

Our counterstrategy-guided assumption mining approach produces assumptions necessary for synthesis. Hence, we hypothesize that it can also be leveraged in an iterative fashion for synthesizing compositional designs. Consider two interacting components M_1 and M_2, each specified with contracts (A_1, G_1) and (A_2, G_2) respectively, and the contracts are given as GR(1) specifications. We want to synthesize M_1 and M_2 automatically from their contracts. Assume that A_i comes directly from G_j for $i \neq j$, and for the sake of argument, we start with both A_1 and A_2 as `true`. This means we only know what each component is supposed to guarantee, but not what assumptions on their inputs are sufficient to ensure those guarantees. In the context of synthesis from temporal logic, the specifications $A_1 \rightarrow G_1$ and $A_2 \rightarrow G_2$ are often not realizable because lack of constraint on the component's environment (missing assumptions).

In Chapter 8, we describe a technique for computing an assumption ϕ such that $(\phi \wedge A_1) \rightarrow G_1)$ is realizable. In the compositional setting, ϕ is also added to the guarantee G_2 of M_2. If $A_2 \rightarrow (G_2 \wedge \phi)$ is realizable, then we are done. If not, we can iterate, but this time finding an assumption for M_2. In each iteration, we are essentially finding a refinement of one of the contracts. The iteration goes on until both contract are realizable, and at that point, we can

synthesize M_1 and M_2. This style of compositional synthesis, based on assume-guarantee contracts, is a subject of future work.

11.2.3 Improving Sparse Coding

We plan to improve our sparse-coding inspired approach for specification mining on multiple fronts. First, the current interpretation of a subtrace as superimposition of multiple basis subtraces may not be fitting for all systems. One alternative is to consider the full algebra over the Boolean ring (instead of the semi-ring in our case). Second, solving the Boolean matrix factorization problem and its sparse variants can be computationally expensive. In this context, it would be interesting to use slightly different definitions of a basis (for example, using the field of rationals rather than the semi-ring we consider) so that the problem of computing a sparse basis is polynomial-time solvable. Lastly, the ideas introduced in this paper can be extended beyond digital circuits to software, distributed systems, analog/mixed-signal circuits, and other domains, providing many interesting directions for future work.

11.2.4 Other Application Domains

Recently, Jin et al. [JDDS13] proposed to mine specifications in the form of Parametric Signal Temporal Logic (STL) to capture requirements of closed-loop control systems. This work is also an instance of template-based specification mining, where a specification template is now is STL formula with concrete signal or time values replaced by parameters. The advantage of this logic formalism is that it can capture real-valued and time-varying behaviors. We envision that the techniques proposed in this dissertation are also applicable to their setting. For example, the mined STL specifications may be used to diagnose bugs of a control system in a similar way as described in Section 4.4. In the future, we plan to investigate other application domains, such as distributed systems, cyber-physical systems, analog circuits and software programs, where the lack of good specifications and the difficulty of localizing bugs are also prevalent.

Bibliography

[AAC+04] Gabriela Alexe, Sorin Alexe, Yves Crama, Stephan Foldes, Peter L. Hammer, and Bruno Simeone. Consensus algorithms for the generation of all maximal bicliques. *Discrete Applied Mathematics*, 145:11–21, December 2004.

[ABL02] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'02)*, pages 4–16. ACM, 2002.

[AEY00] Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. Inference of message sequence charts. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE'00)*, pages 304–313. ACM, 2000.

[AKB12] Rawan Abdel-Khalek and Valeria Bertacco. Functional post-silicon diagnosis and debug for networks-on-chip. In *Proceedings of the 30th International Conference on Computer-Aided Design (ICCAD'12)*, pages 557–563, 2012.

[AKT+06] Roy Armoni, Dmitry Korchemny, Andreas Tiemeyer, Moshe Y. Vardi, and Yael Zbar. Deterministic dynamic monitors for linear-time assertions. In *Proceedings of International ICSC Symposium on Multi-Agents and Mobile Agents in Virtual Organizations and E-Commerce (MAMA'2006)*, pages 1–20. Springer, 2006.

[ALT04] Rajeev Alur and Salvatore La Torre. Deterministic generators and games for ltl fragments. *ACM Transaction on Computational Logic*, 5(1):1–25, January 2004.

[AMT13] Rajeev Alur, Salar Moarref, and Ufuk Topcu. Counter-strategy guided refinement of gr(1) temporal logic specifications. In *Proceedings of the 13th Conference on Formal Methods in Computer-Aided Design (FMCAD'13)*, pages 26–33, 2013.

[APPI10] Sterling J Anderson, Steven C Peters, Tom E Pilutti, and Karl Iagnemma. An optimal-control-based framework for trajectory planning, threat assessment, and semi-autonomous control of passenger vehicles in hazard avoidance scenarios. *International Journal of Vehicle Autonomous Systems*, 8(2):190–216, 2010.

[AS87] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 1987.

[AS95] Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In *Proceedings of the 11th International Conference on Data Engineering (ICDE'95)*, pages 3–14, 1995.

[AvMN05] Rajeev Alur, Pavol Černý, P. Madhusudan, and Wonhong Nam. Synthesis of interface specifications for java classes. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05)*, pages 98–109. ACM, 2005.

[BAM03] Pradip Bose, David H. Albonesi, and Diana Marculescu. Guest editors' introduction: Powerand complexity aware design. *IEEE Micro*, 23(5):8–11, September 2003.

[BBDER97] Ilan Beer, Shoham Ben-David, Cindy Eisner, and Yoav Rodeh. Efficient detection of vacuity in actl formulas. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 279–290. Springer, 1997.

[BCG+10] Roderick Bloem, Alessandro Cimatti, Karin Greimel, Georg Hofferek, Robert Könighofer, Marco Roveri, Viktor Schuppan, and Richard Seeber. Ratsy – a new requirements analysis tool with synthesis. In *Proceedings of 32nd Computer Aided Verification Conference (CAV'10)*, pages 425–429. Springer, 2010.

[BCK09] David Baneres, Jordi Cortadella, and Mike Kishinevsky. A recursive paradigm to solve boolean relations. *IEEE Transactions on Computers*, 58(4):512–527, 2009.

[BGHS04] H. Barringer, A. Goldberg, K. Havelund, and Koushik Sen. Program monitoring with ltl in eagle. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, 2004.

[BGJ+07a] Roderick Bloem, Stefan Galler, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Martin Weiglhofer. Automatic hardware synthesis from specifications: A case study. In *Proceedings of the Conference on Design, Automation Test in Europe (DATE'07)*, pages 1–6, April 2007.

[BGJ+07b] Roderick Bloem, Stefan Galler, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Martin Weiglhofer. Specify, compile, run: Hardware from psl. *Electronic Notes in Theoretical Computer Science*, 190:3–16, November 2007.

[BGL+00] Saddek Bensalem, Vijay Ganesh, Yassine Lakhnech, César Mu noz, Sam Owre, Harald Rueß, John Rushby, Vlad Rusu, Hassen Saïdi, N. Shankar, Eli Singerman, and Ashish Tiwari. An overview of SAL. In *Proceedings of the 5th NASA Langley Formal Methods Workshop*, pages 187–196, June 2000.

[BL69] J. Richard Büchi and Lawrence H. Landweber. Solving sequential conditions by finite-state strategies. *Transactions of the American Mathematical Society*, 138:295–311, 1969.

[BNR03] Thomas Ball, Mayur Naik, and Sriram K. Rajamani. From symptom to cause: localizing errors in counterexample traces. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'03)*, pages 97–105, 2003.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transaction on Computers*, 35(8):677–691, 1986.

[BSL04] Yves Bontemps, Pierre-Yves Schobbens, and Christof Löding. Synthesis of open reactive systems from scenario-based specifications. *Fundamenta Informaticae*, 62:139–169, February 2004.

[BW96] Beate Bollig and Ingo Wegener. Improving the variable ordering of obdds is np-complete. *IEEE Transaction on Computers*, 45(9):993–1002, 1996.

[Cap75] Michel Caplain. Finding invariant assertions for proving programs. In *Proceedings of the International Conference on Reliable Software*, pages 165–171. ACM, 1975.

[CBP+11] Chia Yuan Cho, Domagoj Babić, Pongsin Poosankam, Kevin Zhijie Chen, Edward XueJun Wu, and Dawn Song. Mace: Model-inference-assisted concolic exploration for protocol and vulnerability discovery. In *Proceedings of the 20th USENIX Security Symposium*, August 2011.

[CES86] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8:244–263, 1986.

[CGP00] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, 2000.

[CGP03] JamiesonM. Cobleigh, Dimitra Giannakopoulou, and CorinaS. Păsăreanu. Learning assumptions for compositional verification. In *Proceedings of the 9th Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, volume 2619 of *Lecture Notes in Computer Science*, pages 331–346. Springer, 2003.

[CGR$^+$12] Chih-Hong Cheng, Michael Geisinger, Harald Ruess, Christian Buckl, and Alois Knoll. Mgsyn: Automatic synthesis for industrial automation. In *Proceedings of the 24th Conference on Computer Aided Verification (CAV'12)*, volume 7358 of *Lecture Notes in Computer Science*, pages 658–664. Springer, 2012.

[CHJ08] Krishnendu Chatterjee, Thomas A. Henzinger, and Barbara Jobstmann. Environment assumptions for synthesis. In *Proceedings of the 19th International Conference on Concurrency Theory (CONCUR'08)*, pages 147–161. Springer, 2008.

[CHJR10] Krishnendu Chatterjee, Thomas Henzinger, Barbara Jobstmann, and Arjun Radhakrishna. A solver for probabilistic games. In *Proceedings of the Seventeenth Conference on Computer Aided Verification (CAV'10)*, volume 6174 of *Lecture Notes in Computer Science*, pages 665–669. Springer, 2010.

[Chu57] Alonso Church. Applications of recursive arithmetic to the problem of circuit synthesis. In *Summaries of Talks Presented at the Summer Institute of Symbolic Logic*, pages 3–50. Communications Research Division, Institute for Defense Analysis, 1957.

[Chu62] Alonso Church. Logic, arithmetic and automata. In *Proceedings of the International Congress of Mathematicians*, August 1962.

[CJK07] Mihai Christodorescu, Somesh Jha, and Christopher Kruegel. Mining specifications of malicious behavior. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (FSE'07)*, pages 5–14. ACM, 2007.

[CLR05] Marie-Christine Costa, Lucas Léocart, and Frédéric Roupin. Minimal multicut and maximal integer multiflow: A survey. *European Journal of Operational Research*, 162(1):55–69, 2005.

[CRST08] Alessandro Cimatti, Marco Roveri, Viktor Schuppan, and Andrei Tchaltsev. Diagnostic information for realizability. In *Proceedings of the 9th international conference on Verification, model checking, and abstract interpretation (VMCAI'08)*, pages 52–67. Springer, 2008.

[csf] Crowd sourced formal verification (csfv). http://www.darpa.mil/Our_Work/I2O/Programs/Crowd_Sourced_Formal_Verification_(CSFV).aspx.

[CT91] Luca Console and Pietro Torasso. A spectrum of logical definitions of model-based diagnosis. *Computational Intelligence*, 7(3):133–141, Auguest 1991.

[cuc] Cucumber. http://cukes.info.

[CWKK09] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Kruegel, and Engin Kirda. Prospex: Protocol specification extraction. In *Proceedings of the 30th IEEE Symposium on Security and Privacy (SP'09)*, pages 110–125. IEEE, 2009.

[DB09] Andrew DeOrio and Valeria Bertacco. Human computing for EDA. In *Proceedings of the 46th Design Automation Conference (DAC'09)*, pages 621–622, 2009.

[DDG+12] Rolf Drechsler, Melanie Diepenbeck, Daniel Große, Ulrich Kühne, Hoang M. Le, Julia Seiter, Mathias Soeken, and Robert Wille. Completeness-driven development. In *Graph Transformations*, volume 7562 of *Lecture Notes in Computer Science*, pages 38–50. Springer, 2012.

[DH01] Werner Damm and David Harel. Lscs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[dKMR92] Johan de Kleer, Alan K. Mackworth, and Raymond Reiter. Characterizing diagnoses and systems. *Artificial Intelligence*, 56(2-3):197–222, 1992.

[DLE03] Nii Dodoo, Lee Lin, and Michael D. Ernst. Selecting, refining, and evaluating predicates for program analysis. Technical Report MIT-LCS-TR-914, MIT Laboratory for Computer Science, 2003.

[dMMM06] Marie-Catherine de Marneffe, Bill MacCartney, and Christopher D. Manning. Generating typed dependency parses from phrase structure parses. In *Proceedings of the 5th International Conference on Language Resources and Evaluation (LREC'06)*, pages 449–454, 2006.

[dPGH+08] Flavio M. de Paula, Marcel Gort, Alan J. Hu, Steven J. E. Wilton, and Jin Yang. Backspace: Formal analysis for post-silicon debug. In *Proceedings of the 8th International Conference on Formal Methods in Computer-Aided Design (FMCAD'08)*, pages 1–10, 2008.

[ECH+01] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, pages 57–72. ACM, 2001.

[EF06] Cindy Eisner and Dana Fisman. *A Practical Introduction to PSL.* Springer, 2006.

[EPG+07] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45, 2007.

[Erl00] Len Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, 2000.

[Ern03] Michael D. Ernst. Static and dynamic analysis: Synergy and duality. In *Proceedings of the 2003 International Workshop on Dynamic Analysis (WODA'03)*, pages 24–27, May 2003.

[FD04] Gèorschwin Fey and Rolf Drechsler. Improving simulation-based verification by means of formal methods. In *Proceedings of the 2004 Conference on Asia South Pacific Design Automation (ASP-DAC'04)*, pages 640–643, 2004.

[Fed95] Federal Aviation Administration (FAA). The interfaces between flight crews and modern flight systems, 1995.

[Fro95] Véronique Froidure. *Rangs des relations binaires, semigroupes de relations non ambigues*. PhD thesis, June 1995.

[GCKS06] Alex Groce, Sagar Chaki, Daniel Kroening, and Ofer Strichman. Error explanation with distance metrics. *Software Tools for Technology Transfer*, 8(3):229–247, 2006.

[GEL+13] Shalini Ghosh, Daniel Elenius, Wenchao Li, Patrick Lincoln, Natarajan Shankar, and Wilfried Steiner. Automatically extracting requirements specifications from natural language. Technical Report Number SRI-CSL-13-01, Computer Science Laboratory, SRI International, Menlo Park, CA, December 2013.

[GKL08] Serge Gaspers, Dieter Kratsch, and Mathieu Liedloff. On independent sets and bicliques in graphs. In *Graph-Theoretic Concepts in Computer Science*, volume 5344 of *Lecture Notes in Computer Science*, pages 171–182. 2008.

[GMF] Anubhav Gupta, Kenneth L. Mcmillan, and Zhaohui Fu. Automated assumption generation for compositional verification. *Formal Methods in System Design*, 32(3):285–301.

[Gol78] E. Mark Gold. Complexity of automatic identification from given data. *Information and Control*, 37:302–320, 1978.

[GS08a] Mark Gabel and Zhendong Su. Javert: fully automatic mining of general temporal properties from dynamic traces. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'08)*, pages 339–349. ACM, 2008.

[GS08b] Mark Gabel and Zhendong Su. Symbolic mining of temporal specifications. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, pages 51–60, 2008.

[GZ05] Vincenzo Gervasi and Didar Zowghi. Reasoning about inconsistencies in natural language requirements. *ACM Transaction on Software Engineering Methodology*, 14:277–330, July 2005.

[Har12] Ian G. Harris. Extracting design information from natural language specifications. In *Proceedings of the 49th Annual Design Automation Conference*, pages 1256–1257, 2012.

[HKSY09] Shigeki Hagihara, Yusuke Kitamura, Masaya Shimakawa, and Naoki Yonezaki. Extracting environmental constraints to make reactive system specifications realizable. In *Proceedings of the 16th Asia-Pacific Software Engineering Conference (APSEC'09)*, pages 61–68, December 2009.

[HLR93] Nicolas Halbwachs, Fabienne Lagnier, and Pascal Raymond. Synchronous observers and the verification of reactive systems. In *Proccedings of the 3rd International Conference on Algebraic Methodology and Software Technology (AMAST'93)*. Springer, 1993.

[HNCC05] Sudheendra Hangal, Sridhar Narayanan, Naveen Chandra, and Sandeep Chakravorty. Iodine: a tool to automatically infer dynamic invariants for hardware designs. In *Proceedings of the 42nd Design Automation Conference (DAC'05)*, pages 775–778, 2005.

[How] Jeff Howe. Crowdsourcing: A definition. http://crowdsourcing.typepad.com.

[HRS+00] Mary Jean Harrold, Gregg Rothermel, Kent Sayre, Rui Wu, and Liu Yi. An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing, Verification and Reliability*, 10(3):171–194, 2000.

[HSV13] Samuel Hertz, David Sheridan, and Shoba Vasudevan. Mining hardware assertions with guidance from static analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(6):952–965, 2013.

[IB06] Beth Isaksen and Valeria Bertacco. Verification through the principle of least astonishment. In *Proceedings of the 24th International Conference on Computer-Aided Design (ICCAD'06)*, pages 860–867, 2006.

[JDDS13] Xiaoqing Jin, Alexandre Donzé, Jyotirmoy Deshmukh, and Sanjit A. Seshia. Mining requirements from closed-loop control models. In *Proceedings of the International Conference on Hybrid Systems: Computation and Control (HSCC'13)*, April 2013.

[JGWB07] Barbara Jobstmann, Stefan Galler, Martin Weiglhofer, and Roderick Bloem. Anzu: a tool for property synthesis. In *Proceedings of the 19th international conference on Computer aided verification (CAV'07)*, pages 258–262. Springer, 2007.

[JLH09] Jie-Hong R. Jiang, Hsuan-Po Lin, and Wei-Lun Hung. Interpolating functions from large boolean relations. In *Proceedings of the 27th International Conference on Computer-Aided Design (ICCAD'09)*, pages 779–784, 2009.

[Joh08] R. Colin Johnson. The future according to freescale: 1,000 embedded devices per person. http://www.eetimes.com/document.asp?doc_id=1168780, June 2008.

[KCD00] Linda T. Kohn, Janet M. Corrigan, and Molla S. Donaldson. To err is human: Building a safer health system. Technical report, A report of the Committee on Quality of Health Care in America, Institute of Medicine, Washington, DC, 2000. National Academy Press.

[KGFP07] Hadas Kress-Gazit, Georgios E. Fainekos, and Geroge J. Pappas. Where's waldo? sensor-based temporal logic motion planning. In *Proceedings of the 2007 IEEE International Conference on Robotics and Automation (ICRA'07)*, pages 3116–3121, 2007.

[KGFP08] Hadas Kress-Gazit, Georgios E. Fainekos, and George J. Pappas. Translating structured english to robot controllers. *Advanced Robotics*, pages 1343–1359, 2008.

[KHB09] R. Könighofer, G. Hofferek, and R. Bloem. Debugging formal specifications using simple counterstrategies. In *Proceedings of the 9th International Conference on Formal Methods in Computer-Aided Design (FMCAD'09)*, pages 152–159, November 2009.

[KHB10] Robert Könighofer, Georg Hofferek, and Roderick Paul Bloem. Debugging unrealizable specifications with model-based diagnosis. In *Proceedings of the 6th International Conference on Hardware and Software: Verification and Testing (HVC'10)*, pages 29–45. Springer, 2010.

[Kri63] Saul A. Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16(1963):83–94, 1963.

[KSKD13] O. Keszocze, M. Soeken, E. Kuksa, and R. Drechsler. Lips: An ide for model driven engineering based on natural language processing. In *Proceedings of the 1st International Workshop on Natural Language Analysis in Software Engineering (NaturaLiSE'13)*, pages 31–38, 2013.

[KYV01] Orna Kupferman and Moshe Y. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, October 2001.

[LAZJ03] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. volume 38, pages 141–154. ACM, May 2003.

[LBRN07] Honglak Lee, Alexis Battle, Rajat Raina, and Andrew Y. Ng. Efficient sparse coding algorithms. In *Proceedings of the 21st Annual Conference on Neural Information Processing Systems (NIPS'07)*, pages 801–808, 2007.

[LCGM10] Greg Little, Lydia B. Chilton, Max Goldman, and Robert C. Miller. Turkit: Human computation algorithms on mechanical turk. In *Proceedings of the 23nd ACM Symposium on User Interface Software and Technology (UIST'10)*, pages 57–66, 2010.

[LCH+09] David Lo, Hong Cheng, Jiawei Han, Siau-Cheng Khoo, and Chengnian Sun. Classification of software behaviors for failure detection: A discriminative pattern mining approach. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'09)*, pages 557–566. ACM, 2009.

[LDS11] Wenchao Li, Lili Dworkin, and Sanjit A. Seshia. Mining assumptions for synthesis. In *Proceedings of the 9th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'11)*, pages 43–50, July 2011.

[LFS10] Wenchao Li, Alessandro Forin, and Sanjit A. Seshia. Scalable specification mining for verification and diagnosis. In *Proceedings of the 47th Design Automation Conference (DAC'10)*, pages 755–760, 2010.

[Lib04] Benjamin R. Liblit. *Cooperative Bug Isolation*. PhD thesis, University of California, Berkeley, 2004.

[LKL08] David Lo, Siau-Cheng Khoo, and Chao Liu. Mining past-time temporal rules from execution traces. In *Proceedings of the 2008 International Workshop on Dynamic Analysis (WODA'08)*, pages 50–56. ACM, 2008.

[LKLH11] David Lo, Siau-Cheng Khoo, Chao Liu, and Jiawei Han. Specification mining: A concise introduction. 2011.

[LM08] David Lo and Shahar Maoz. Mining scenario-based triggers and effects. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE'08)*, pages 109–118. IEEE, 2008.

[LM09] David L. Lempia and Steven P. Miller. Requirements engineering management handbook. Final Report DOT/FAA/AR-08/32, Federal Aviation Administration, June 2009.

[LMB12] Scott C Livingston, Richard M Murray, and Joel W Burdick. Backtracking temporal logic synthesis for uncertain environments. In *Proceedings of the 2012 IEEE International Conference on Robotics and Automation (ICRA'12)*, pages 5163–5170. IEEE, 2012.

[LMK07] David Lo, Shahar Maoz, and Siau-Cheng Khoo. Mining modal scenario-based specifications from execution traces of reactive systems. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07)*, pages 465–468. ACM, 2007.

[LMP08] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. Automatic generation of software behavioral models. In *Proceedings of the ACM/IEEE 30th International Conference on Software Engineering (ICSE'08)*, pages 501–510, 2008.

[LNRB09] Benjamin Livshits, Aditya V. Nori, Sriram K. Rajamani, and Anindya Banerjee. Merlin: Specification inference for explicit information flow problems. *SIGPLAN Notices*, 44(6):75–86, June 2009.

[LPJM13] Scott C. Livingston, Pavithra Prabhakar, Alex B. Jose, and Richard M. Murray. Patching task-level robot controllers based on a local μ-calculus formula. pages 4588–4595, 2013.

[LS12] Wenchao Li and Sanjit A. Seshia. Sparse coding for specification mining and error localization. In *Proceedings of the 3rd International Conference on Runtime Verification (RV'12)*, pages 64–81, September 2012.

[LSJ12] Wenchao Li, Sanjit A. Seshia, and Somesh Jha. Crowdmine: Towards crowdsourced human-assisted verification. In *Proceedings of the 49th Design Automation Conference (DAC'12)*, pages 1250–1251, June 2012.

[LSSS14] Wenchao Li, Dorsa Sadigh, S. Shankar Sastry, and Sanjit A. Seshia. Synthesis for human-in-the-loop control systems. In *Proceedings of the 20th Conference on Tools and Algorithms for the Construction and Analysis of System (TACAS'14)*, April 2014.

[LSTV12] Lingyi Liu, David Sheridan, William Tuohy, and Shobha Vasudevan. A technique for test coverage closure using goldmine. *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, 31(5):790–803, 2012.

[LZ05] Zhenmin Li and Yuanyuan Zhou. Pr-miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 306–315. ACM, 2005.

[MBD12] Srobona Mitra, Ansuman Banerjee, and Pallab Dasgupta. Formal methods for ranking counterexamples through assumption mining. In *Proceedings of the Conference on Design, Automation Test in Europe (DATE'12)*, pages 911–916, 2012.

[Mcm99] Kenneth L. Mcmillan. Circular compositional reasoning about liveness. In *Advances in Hardware Design and Verification: IFIP WG10.5 International Conference on Correct Hardware Design and Verification Methods (CHARME'99)*, pages 342–345. Springer, 1999.

[Mie10] Pauli Miettinen. Sparse boolean matrix factorizations. In *Proceedings of the 2010 IEEE International Conference on Data Mining (ICDM'10)*, pages 935–940. IEEE, 2010.

[Mit79] Tom M. Mitchell. *Version Spaces: An Approach to Concept Learning*. PhD thesis, Stanford, CA, USA, 1979.

[MMG+06] Pauli Miettinen, Taneli Mielikäinen, Aristides Gionis, Gautam Das, and Heikki Mannila. The discrete basis problem. In *Proceedings of the 10th European conference on Principle and Practice of Knowledge Discovery in Databases (PKDD'06)*, pages 335–346. Springer, 2006.

[MP92] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer, 1992.

[MSN10] Subhasish Mitra, Sanjit A. Seshia, and Nicola Nicolici. Post-silicon validation: Opportunities, challenges and recent advances. In *Proceedings of the 47th Design Automation Conference (DAC'10)*, pages 12–17, June 2010.

[Muk96] Madhavan Mukund. Finite-state automata on infinite inputs, 1996.

[Nat13] National Highway Traffic Safety Administration. Preliminary statement of policy concerning automated vehicles, May 2013.

[NSVSP12] Pierluigi Nuzzo, Alberto Sangiovanni-Vincentelli, Xuening Sun, and Alberto Puggelli. Methodology for the design of analog integrated interfaces using contracts. *Sensors Journal, IEEE*, 12(12):3329–3345, 2012.

[OF97] Bruno A. Olshausen and David J. Fieldt. Sparse coding with an overcomplete basis set: a strategy employed by v1. *Vision Research*, 37:3311–3325, 1997.

[ope] Opencores benchmarks. http://opencores.org.

[Org07] Accellera Organization. Accellera standard ovl v2. www.accellera.org, 2007.

[PBWM10] Sung-Boem Park, Anne Bracy, Hong Wang, and Subhasish Mitra. Blog: Post-silicon bug localization in processors using bug localization graphs. In *Proceedings of the 47th Design Automation Conference (DAC'10)*, pages 368–373, 2010.

[Pee03] René Peeters. The maximum edge biclique problem is NP-complete. *Discrete Applied Mathematics*, 131(3):651–654, 2003.

[Peh01] Li-Shiuan Peh. *Flow control and micro-architectural mechanisms for extending the performance of interconnection networks*. PhD thesis, 2001.

[PLF06] Richard Neil Pittman, Nathaniel Lee Lynch, and Alessandro Forin. emips, a dynamical extensible processor. Technical Report MSR-TR-2006-143, Microsoft Research, 2006.

[PM08] Sung-Boem Park and Subhasish Mitra. Ifra: Instruction footprint recording and analysis for post-silicon bug localization in processors. In *Proceedings of the 45th Design Automation Conference (DAC'08)*, pages 373–378, 2008.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[PPS06a] Nir Piterma, Amir Pnueli, and Yaniv Sa'ar. Synthesis of reactive(1) designs. In *Proceedings of the 7th Verification, Model Checking, and Abstract Interpretation (VMCAI'06)*, pages 364–380. Springer, 2006.

[PPS06b] Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of reactive(1) designs. In *Proceedings of the 7th Verification, Model Checking, and Abstract Interpretation (VMCAI'06)*, pages 364–380. Springer, 2006.

[PR89] Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'89)*, pages 179–190. ACM, 1989.

[QB11] Alexander J. Quinn and Benjamin B. Bederson. Human computation: A survey and taxonomy of a growing field. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI'11)*, pages 1403–1412. ACM, 2011.

[Rab72] Michael O. Rabin. Automata on infinite objects and church's problem. In *Proceedings of the Regional Conference Series in Mathematics*. American Mathematical Society, 1972.

[RGJ07] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. Path-sensitive inference of function precedence protocols. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, pages 240–250. IEEE, 2007.

[RKF+08] Frank Rogin, Thomas Klotz, Görschwin Fey, Rolf Drechsler, and Steffen Rülke. Automatic generation of complex properties for hardware designs. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'08)*, pages 545–548. ACM, 2008.

[RKG11] Vasumathi Raman and Hadas Kress-Gazit. Analyzing unsynthesizable specifications for high-level robot behavior using ltlmop. In *Proceedings of 33th Computer Aided Verification Conference (CAV'11)*, volume 6806 of *Lecture Notes in Computer Science*, pages 663–668. Springer, 2011.

[Ros92] Roni Rosner. Modular synthesis of reactive systems. *Ph.D. dissertation, Weizmann Institute of Science*, 1992.

[rsp] Rspec. http://en.wikipedia.org/wiki/RSpec.

[Rus12] John Rushby. The versatile synchronous observer. In *Formal Methods: Foundations and Applications*, volume 7498 of *Lecture Notes in Computer Science*. Springer, 2012.

[SACO02] Rachel L. Smith, George S. Avrunin, Lori A. Clarke, and Leon J. Osterweil. PROPEL: An approach supporting property elucidation. In *Proceedings of the 24th International Conference on Software Engineering (ICSE'02)*, pages 11–21, 2002.

[San] Michael Sanie. Solving modern verification challenges for todays industry leaders. http://chipdesignmag.com/display.php?articleId=4503.

[SCK04] William Scott, Stephen Cook, and Joseph Kasser. Development and application of context-free grammar for requirements. In *Proceedings of the System Engineering Test and Evaluation Conference (SETE'04)*, 2004.

[SE10] Todd W. Schiller and Michael D. Ernst. Rethinking the economics of software engineering. In *Proceedings of the Workshop on the Future of Software Engineering Research*, pages 325–330, 2010.

[Ses12] Sanjit A. Seshia. Sciduction: Combining induction, deduction, and structure for verification and synthesis. In *Proceedings of the 49th Design Automation Conference (DAC'12)*, pages 356–365, June 2012.

[Sha79] Adi Shamir. How to share a secret. *Communications of the ACM*, 22:612–613, November 1979.

[SHB12] Matthias Schlaipfer, Georg Hofferek, and Roderick Paul Bloem. Generalized reactivity(1) synthesis without a monolithic strategy. In *Hardware and Software: Verification and Testing. 7th International Haifa Verification Conference*

(HVC'11), volume 7261 of *Lecture Notes in Computer Science*, pages 20–34. Springer, 2012.

[Shi02] Kanna Shimizu. *Writing, Verifying, and Exploiting Formal Specifications for Hardware Designs*. PhD thesis, Department of Electrical Engineering, Stanford University, August 2002.

[Sie00] Daluss J. Siewert. *Biclique covers and partitions of bipartite graphs and digraphs and related matrix ranks of 0,1 matrices*. PhD thesis, 2000.

[smv] Cadence smv. http://www.kenmcmil.com/smv.html.

[Sto02] Richard Stolzman. Understanding assertion-based verification. http://www.eetimes.com/document.asp?doc_id=1275847, Auguest 2002.

[SWD12] Mathias Soeken, Robert Wille, and Rolf Drechsler. Assisted behavior driven development using natural language processing. In *Proceedings of the 50th International Conference on Objects, Models, Components, Patterns (TOOLS'12)*, volume 7304 of *Lecture Notes in Computer Science*, pages 269–287. 2012.

[Tea08] The Amazon S3 Team. Amazon s3 availability event: July 20, 2008. http://status.aws.amazon.com/s3-20080720.html, July 2008.

[TK01] Serdar Tasiran and Kurt Keutzer. Coverage metrics for functional validation of hardware designs. *IEEE Design and Test of Computers*, 18(4):36–45, 2001.

[tur] Amazon's mechanical turk. www.mturk.com/mturk/welcome.

[UKM01] Sebastian Uchitel, Jeff Kramer, and Jeff Magee. Detecting implied scenarios in message sequence chart specifications. In *Proceedings of the 8th European Software Engineering Conference Held Jointly with the 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 74–82. ACM, 2001.

[vA05] Luis von Ahn. *Human Computation*. PhD thesis, Carnegie Mellon University, December 2005.

[VAG07] Jaideep Vaidya, Vijayalakshmi Atluri, and Qi Guo. The role mining problem: Finding a minimal descriptive set of roles. In *Proceedings of the 12th ACM Symposium on Access Control Models and Technologies (SACMAT'07)*, pages 175–184, 2007.

[VDV11] Rajeev Verma and Domitilla Del Vecchio. Semiautonomous multivehicle safety. *IEEE Robotics Automation Magazine*, 18(3):44–54, 2011.

[VDV12] Rajeev Verma and Domitilla Del Vecchio. Safety control of hidden mode hybrid systems. *IEEE Transactions on Automatic Control*, 57(1):62–77, January 2012.

[VSG+12] Ram Vasudevan, Victor Shia, Yiqi Gao, Ricardo Cervera-Navarro, Ruzena Bajcsy, and Francesco Borrelli. Safe semi-autonomous control with enhanced driver modeling. In *Proceedings of the 2012 American Control Conference (ACC'12)*, pages 2896–2903, June 2012.

[VSP+10] Shoba Vasudevan, David Sheridan, Sanjay Patel, David Tcheng, Bill Tuohy, and Daniel Johnson. Goldmine: Automatic assertion generation using data mining and static analysis. In *Proceedings of the Conference on Design, Automation Test in Europe (DATE'10)*, pages 626–629, 2010.

[WB91] Yosinori Watanabe and Robert K. Brayton. Heuristic minimization of multiple-valued relations. In *Proceedings of the 9th International Conference on Computer-Aided Design (CAV'91)*, pages 126–129, 1991.

[WB11] Ilya Wagner and Valeria Bertacco. *Post-Silicon and Runtime Verification for Modern Processors*. Springer, 2011.

[Weg74] Ben Wegbreit. The synthesis of loop predicates. *Communications of the ACM*, 17(2):102–113, 1974.

[WHT03] Nico Wallmeier, Patrick Hütten, and Wolfgang Thomas. Symbolic synthesis of finite-state controllers for request-response specifications. In *Implementation and Application of Automata*, volume 2759 of *Lecture Notes in Computer Science*, pages 11–22. Springer, 2003.

[WN05] Westley Weimer and George C. Necula. Mining temporal specifications for error detection. In *Proceedings of the 11th Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*, pages 461–476. Springer, 2005.

[Won10] Tichakorn Wongpiromsarn. *Formal methods for design and verification of embedded control systems: application to an autonomous vehicle*. PhD thesis, 2010.

[WTM09] Tichakorn Wongpiromsarn, Ufuk Topcu, and Richard M. Murray. Receding horizon temporal logic planning for dynamical systems. In *Proceedings of the 48th IEEE Conference on Decision and Control (CDC'09)*, pages 5997–6004, 2009.

[WTM12] Tichakorn Wongpiromsarn, Ufuk Topcu, and Richard M. Murray. Receding horizon temporal logic planning. *IEEE Transactions on Automatic Control*, 57(11):2817–2830, 2012.

[XPTX12] Xusheng Xiao, Amit Paradkar, Suresh Thummalapenta, and Tao Xie. Automated extraction of security policies from natural-language software documents. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE'12)*, pages 12:1–12:11. ACM, 2012.

[XTM12] Huan Xu, U. Topcu, and R.M. Murray. A case study on reactive protocols for aircraft electric power distribution. In *Decision and Control (CDC), 2012 IEEE 51st Annual Conference on*, pages 1124–1129, 2012.

[YEB+06] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Perracotta: mining temporal api rules from imperfect traces. In *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*, pages 282–291, 2006.

[ZGM01] Didar Zowghi, Vincenzo Gervasi, and Andrew McRae. Using default reasoning to discover inconsistencies in natural language requirements. In *Proceedings of the 8th Asia-Pacific Software Engineering Conference (APSEC'01)*, pages 133–140, 2001.

[ZHT04] Hui Zou, Trevor Hastie, and Robert Tibshirani. Sparse principal component analysis. *Journal of Computational and Graphical Statistics*, 15:2006, 2004.

[ZWM11] Charlie Shucheng Zhu, Georg Weissenbacher, and Sharad Malik. Post-silicon fault localization using maximum satisfiability and backbones. In *Proceedings of the 11th International Conference on Formal Methods in Computer-Aided Design (FMCAD'11)*, 2011.

[ZWSM11] Charlie Shucheng Zhu, Georg Weissenbacher, Divjyot Sethi, and Sharad Malik. Sat-based techniques for determining backbones for post-silicon fault localisation. In *Proceedings of the 2011 IEEE International High Level Design Validation and Test Workshop (HLDVT'11)*, pages 84–91, 2011.