מכון ויצמן למדע

WEIZMANN INSTITUTE OF SCIENCE

Thesis for the degree
Doctor of Philosophy

עבודת גמר (תזה) לתואר
דוקטור לפילוסופיה

Submitted to the Scientific Council of the
Weizmann Institute of Science
Rehovot, Israel

מוגשת למועצה המדעית של
מכון ויצמן למדע
רחובות, ישראל

By
Guy Katz

מאת
גיא כץ

על ניבי מקביליות והשפעתם על ניתוח תכניות

On Concurrency Idioms and their Effect on Program Analysis

Advisor:
Prof. David Harel

מנחה:
פרופ. דוד הראל

December 2015

כסלו תשע"ו

On Concurrency Idioms and their Effect on Program Analysis

Ph.D. Thesis

Submitted by: Guy Katz
Advisor: Prof. David Harel
Department of Computer Science and Applied Mathematics
The Weizmann Institute of Science

December 9, 2015

Acknowledgements

I would like to thank my advisor, Prof. David Harel, for his guidance and support during these past four years. I am particularly grateful to him not only for his insightful contributions to my research, but also for the academic and geographical freedom that he has provided me with.

Next, I would like to thank my fellow students, colleagues and collaborators: Dr. Amir Kantor, Dr. Gera Weiss, Prof. Clark Barrett, Dr. Rami Marelly, Dr. Robby Lampert, Dr. Guy Wiener, Smadar Szekely, Guy Weiss and Yaarit Goel. My special thanks to Dr. Assaf Marron, with whom I have collaborated on a large portion of this work, and to whose guidance I am indebted.

Thanks are due also to the members of my thesis committee, Prof. Orna Kupferman and Prof. David Peleg, for their insights and valuable feedback.

Last but not least I would like to thank my family and friends for their love and support, without which this work would not have been possible.

Abstract

The development of large reactive software systems is an expensive and error-prone undertaking. Deliverables will often fail, resulting in unintended software behavior, exceeded budgets and breached time schedules. One of the key reasons for this difficulty is the growing complexity of many kinds of reactive systems, which increasingly prevents the human mind from managing a comprehensive picture of all their relevant elements and behaviors.

In recent decades, a prominent approach for tackling this issue has been that of formal analysis. There, one relies on automatic tools that methodically explore the state space of the system and perform various tasks. The main hindrance to the applicability of formal analysis is the *state explosion* phenomenon: the size of the state space of a system can be exponential in the size of its constituent components. This makes it highly difficult for analysis techniques to scale up to real-world systems. Although the research community has put a great deal of work into improving the scalability of analysis tools, resolving the great difficulties in developing reliable reactive systems remains a major, and critical, moving target.

In this thesis we focus on an aspect of the problem which, we feel, has much untapped potential: the effect that the computational model (e.g., the programming language) in use has on the complexity of analyzing the resulting software. We advocate a *design for analysis* approach: our hypothesis is that by carefully choosing the programming idioms to be used in the development of a particular system, and in particular by preferring simpler idioms, one can render the resulting software more amenable to analysis. We demonstrate that this hypothesis applies to several program analysis tasks, such as verification, optimization and repair.

Given this approach, a natural question to ask is whether it is possible to find programming models that are simple enough to facilitate program analysis on the one hand, but which are expressive and convenient enough to be appealing to programmers on the other hand. Indeed, we study this delicate trade-off, and show that certain programming idioms satisfy both requirements in certain cases. We take a modular approach, studying the contribution and costs of individual idioms; our ultimate goal is to provide engineers with a pool of programming idioms, from which they will be able to tailor a programming framework to their specific needs.

We believe that the results presented in this thesis are compelling evidence that the careful selection of a programming model can indeed render program analysis easier. We hope that it will serve to draw more attention to this important research direction.

תקציר

פיתוחן של מערכות תגובתיות מורכבות הינה מלאכה יקרה וסבוכה. לעיתים קרובות מערכות שכבר נפרסו בשטח מתפקדות בצורה שונה מזו שיועדה להן, וגורמות בכך לחריגות בעלויות ובזמני פיתוח. אחת הסיבות העיקריות לקושי זה היא רמת המורכבות הגבוהה של מערכות תגובתיות בימינו – וכתוצאה, חוסר יכולתם של מהנדסים אנושיים לראות בעיני רוחם תמונה מקיפה וכוללת של התנהגות המערכת ורכיביה.

בעשורים האחרונים, אחת הגישות העיקריות שדרכה ניסו חוקרים להתמודד עם הבעיה הזאת הינה *ניתוח פורמלי*. בניתוח פורמלי משתמשים בכלים אוטומטיים שסורקים את מרחב המצבים של מערכת תוכנה ומנתחים אותה. המכשול העיקרי להפעלת שיטה זו בפועל הוא בעיית *"התפוצצות המצבים"* – מרחב המצבים של תוכנה נוטה לגדול בצורה מעריכית בגודל קוד המקור שלה, ועל כן כלי ניתוח מתקשים להתמודד עם מערכות תוכנה גדולות. למרות שהקהילה המדעית השקיעה מאמצים מרובים בהתמודדות עם בעיית התפוצצות המצבים, פתרון לקושי הרב שבפיתוח תוכנה אמינה עדיין אינו נראה לעין.

בתזה זו אנו מתמקדים בפן אחר של הבעיה, אשר לדעתנו טמון בו פוטנציאל בלתי מנוצל רב: השפעתו של המודל החישובי, או שפת התכנות שבה משתמשים, על רמת הסיבוכיות שבניתוח התוכנה המתקבלת בעזרתם. במילים אחרות, אנו מקדמים כאן גישה של *"תכנות לאימות"*: אנו סבורים שע"י בחירה מושכלת של ניבי תכנות שיכללו במודל החישובי, נוכל להפוך את התוכנה המתקבלת לקלה יותר לניתוח. אנו מדגימים מקרים רבים שבהם שימוש בניבי תכנות פשוטים אכן גורם להקלה משמעותית במשימות ניתוח תוכנה כגון אימות, אופטימיזציה ותיקון.

לאור גישתנו זו, עולה השאלה הבאה: האם ניתן למצוא מודלים תכנותיים שהם מחד פשוטים דיים כדי לפשט את משימת ניתוח התוכנה, אך מאידך גם מורכבים וחזקים דיים כדי למשוך מתכנתים ומהנדסים להשתמש בהם. בפרקים הבאים נדון בסוגיה הזו, ונראה שבמקרים מסוימים אכן קיימים ניבי תכנות שעונים לשתי הדרישות. אנו נוקטים בגישה מודולרית, ומנתחים את תועלתו ועלותו של כל ניב תכנות בנפרד. בטווח הארוך, שאיפתנו היא לאפשר למהנדסים לבחור ולהשתמש בדיוק באותם ניבי תכנות הדרושים לצורך משימתם הנוכחית, ובכך לשמר את פשטותו של המודל ככל הניתן ולהקל על משימות הניתוח.

לעניות דעתנו התוצאות המוצגות בתזה זו מהוות תימוכין מהימנים לכך שבחירה מושכלת של ניבי תכנות יכולה להקל על מלאכת ניתוח התוכנה. אנו תקווה שמסקנותינו יגבירו את תשומת הלב המוקדשת לכיוון מחקר זה.

Contents

I Introduction 15

1 Constructing Reliable Concurrent Reactive Systems 17

1.1 Concurrency Bugs and Formal Analysis 17

1.2 Design for Analysis 18

2 The Request-Wait-Block Model 21

2.1 Behavioral Programming 21

2.2 The Underlying *RWB* Model 23

3 Summary of Contributions 25

3.1 Automatic, Non-Intrusive Repair of *RWB* Programs 25

3.2 Module-Based Abstraction and Repair of *RWB* Programs 26

3.3 On the Succinctness of *RWB* Programs 27

3.4 Compositional Verification of *RWB* Programs 29

3.5 Theory-Aided Compositional Verification of Concurrent Transition Systems . 30

3.6 Distributed *RWB* Programs 31

3.7 Scaling-Up *RWB*: Steps from Basic Principles to Application Architectures . 32

3.8 An Initial Wise Development Framework for *RWB* 33

II Formal Analysis of *RWB* Programs 35

4 Automatic, Non-Intrusive Repair of *RWB* Programs 37

4.1 Introduction 37

4.2 Outline of the Repair Approach 38

4.3 Definitions 39

4.4 Extending the Model Checking of Invariants and Deadlocks 41

4.5 Safety Patches for Loopless Programs 43

4.5.1 Generating Linear Safety Patches 43

4.5.2 Patching for a Specific Event Selection Mechanism 45

4.5.3 Example: Patching Tic-Tac-Toe 46

4.6 Safety Patches for Programs with Cycles 48
4.6.1 Generating Safety Patches for Cycles 48
4.6.2 Subgraph Representation 50
4.6.3 Example: Patching a Coffee Machine 51
4.7 Dealing with Liveness 52
4.7.1 Classifying Hot States 54
4.7.2 Handling Hot-Trap States 56
4.7.3 Hot-Escapable States and Transition Fairness 56
4.7.4 Liveness Patches 58
4.7.5 The Liveness Patching Algorithm 59
4.7.6 Minimal Fairness Enforcement 61
4.7.7 Example: Liveness Patching for the Dining Philosophers 63
4.8 Limited-Depth Repair 64
4.8.1 Automatic Repair from Field Error Reports 64
4.8.2 Example: Limited-Depth repair of the Dining Philosophers 64
4.9 Related Work 65
4.10 Conclusion and Next Steps 66

5 Module-Based Abstraction and Repair of *RWB* Programs 67
5.1 Introduction 67
5.2 Abstractions for Behavioral Programming 68
5.2.1 Abstracting a Behavioral Thread 68
5.2.2 Abstracting a Behavioral Program 70
5.3 Counterexample Guided Abstraction-Refinement 72
5.3.1 Determining if an Execution is Spurious 72
5.3.2 Refining in order to Eliminate a Spurious Execution 74
5.4 Repair using Abstractions 76
5.5 Experimental Results 82
5.6 Related Work and Conclusion 83

6 On the Succinctness of *RWB* Programs 85
6.1 Introduction 85
6.2 Definitions 87
6.2.1 Request-Wait-Block Automata 87
6.2.2 Finite Parallel Automata 88
6.2.3 Succinctness Gaps 90
6.3 *RWB* and Parallel Automata 90
6.3.1 *RWB*-Automata and $\mathcal{C}$-Automata 91

6.3.2 Counting with Succinct *RWB*-Automata 93
6.3.3 Combining *RWB* with $\mathcal{E}$- and $\mathcal{A}$-Automata 94
6.4 Contributions of the Request, Wait, and Block Idioms 96
6.5 Comparing the Request, Wait, and Block Idioms 101
6.6 Related Work 105
6.7 Conclusion and Future Work 106

7 Compositional Verification of *RWB* Programs 109
7.1 Introduction 109
7.2 Formulating BP's Idioms in Z3 111
7.3 Examples 113
7.3.1 Counting with Small Orthogonal Modules 113
7.3.2 Simulating Constrained Movement 115
7.3.3 A Job Scheduler 118
7.3.4 Dining Philosophers 121
7.3.5 Tic-Tac-Toe 122
7.4 Relationship to Other Work 125
7.5 Discussion and Conclusion 126

8 Theory-Aided Compositional Verification of *RWB* Programs 129
8.1 Introduction 129
8.2 Definitions 131
8.3 The Theory of Transition Systems 132
8.4 Automatic Analysis of Transition Systems 135
8.5 Verifying Periodic Programs 138
8.6 Verifying Programs With Shared Arrays 140
8.7 Experimental Results 143
8.8 Related Work and Discussion 145
8.9 Appendix 146
8.9.1 Derivation Rules for the $\mathcal{TS}$ Solver 146
8.9.2 Backward Reachability 151

III The *RWB* Model: A Software Engineering Point-of-View 155

9 Distributed *RWB* Programs 157
9.1 Eager Synchronization 159
9.1.1 Static Analysis 160
9.1.2 Dynamic Analysis 160

9.1.3 Spanning State Graphs 161
9.2 A Distributed Implementation using Eager Execution 162
9.3 Distributed Execution Formalized 163
9.4 Further Relaxing the Distributed Execution Mechanism 166
9.5 Eager Execution and B-Nodes 166
9.6 Conclusion and Future Work 168

10 Scaling-Up *RWB*: From Basic Principles to Application Architectures 169
10.1 Introduction 169
10.2 Handling Input in BP 172
10.3 Support for Time Constraints 173
10.4 Customizable Event Selection 175
10.5 Dynamic Thread Creation 177
10.6 Parameterized Input 179
10.7 Formal Semantics 180
10.7.1 Event Sets 181
10.7.2 Behavior Threads 181
10.7.3 Configurations 181
10.7.4 Behavioral Programs 185
10.8 The BPC Framework 186
10.8.1 User Interface 186
10.8.2 The Underlying Mechanism 187
10.9 Case-Study: a Web-Server 188
10.9.1 The Implementation's Layout 188
10.9.2 Features and Evaluation 192
10.9.3 Discussion: Incremental Development 192
10.10 Related Work 193
10.11 Conclusion and Future Work 194

11 An Initial Wise Development Framework for *RWB* 199
11.1 Introduction 199
11.2 A Simple Example 200
11.3 Explaining the Framework: The Three "Sisters" 204
11.4 A Case-Study: A Cache Coherence Protocol 208
11.5 Related Work 210
11.6 Conclusion 212

IV Conclusion **215**

12 Discussion and Next Steps **217**

12.1 Discussion 217

12.2 Next Steps 219

List of Abbreviations **221**

Statement Regarding Collaboration **222**

List of Publications Included in the Thesis **223**

References **224**

Part I

Introduction

Chapter 1

Constructing Reliable Concurrent Reactive Systems

1.1 Concurrency Bugs and Formal Analysis

Nowadays, reactive software systems are key components in many aspects of our lives. They appear, e.g., in airplanes, elevators, smartphones and cars, and we have come to rely greatly on their smooth operation.

A common problem in modern software systems is *software errors*, or *bugs*. These bugs lead to undesired software behavior which can have dramatic effects, sometimes much worse than mere inconvenience to users. For example, in one case, erroneous software conversions between the metric and US measurement systems caused the Mars Climate Orbiter spacecraft to crash. In another case, six patients died due to faulty dosage calculations by the Therac-25 radiation therapy machine. Thus, devising error-free software is a highly desirable goal.

But where do software bugs even come from? Modern concurrent reactive systems are typically characterized by a myriad of threads and services running in parallel, continuously interacting with each other and with their environment. Errors in these systems often do not originate from single threads or components, but are the result of unexpected interleaving of sets thereof [119]. Consequently, some bugs cannot be discovered by looking at just one component of the system; rather, a more "global" view is required. Since these systems tend to be highly complex, it is difficult for human engineers to obtain this global point of view, making concurrency related bugs hard to predict, understand and prevent.

In recent decades, a prominent approach for tackling this issue and creating more reliable software has been that of formal analysis. There, one relies on automatic tools that methodically explore the state space of the system and perform various analysis tasks, such as detecting bugs or even repairing them. In many cases, this automatic and systematic exploration can detect subtle bugs that would otherwise go unnoticed. Analysis techniques can be used during

development, allowing the engineers to correct any discovered bugs prior to deployment.

Unfortunately, formal analysis has not been quite able to eradicate software bugs altogether. The main hindrance in applying formal analysis to large systems is the *state explosion* phenomenon: the size of a system's state space tends to grow exponentially in the size of its constituent components. Thus, even medium-sized software system can have many billions of states. This makes it difficult for analysis techniques to scale up to real-world systems.

In its ongoing attempts to improve the scalability of analysis tools, the research community has directed a great deal of effort into finding more efficient ways to explore the state spaces of large systems. Specifically, research has focused on finding exploration methods that do not entail explicitly enumerating and visiting every composite state of the system. A few notable examples include the efficient traversal of state graphs using *binary decision diagrams* (*BDDs*) [38, 40] — a data structure that can detect and eliminate similarities in a program's state graph, reducing its size; a technique called *partial order reduction* [13], aimed at ignoring redundant thread interleavings; *compositional verification* [76], which allow one to reason about individual parts of the system and then deduce global correctness; and *abstraction-refinement* based techniques [51], which allow us to analyze and reason about abstract, more compact models of the system. These often harness advanced theorem provers and SMT solvers as their underlying engines [61, 72]. Still, existing tools and techniques are quickly reaching their limits, and the great difficulties in developing reliable reactive systems remain a major, and critical, moving target.

1.2 Design for Analysis

In this thesis we focus on an aspect of analysis tasks, which, we feel, has not received sufficient attention in the literature: the effect that the selected computational model (e.g., the programming language idioms) has on the complexity of software analysis.

It is now widely accepted that many of the errors in concurrent software result from the "unconstrained" concurrency that characterizes modern programming languages [119]. Furthermore, advanced programming language features, such as pointers and aliasing, are very difficult for analysis tools to handle. Thus, since analyzing arbitrary programs poses such a difficulty, one reasonable approach is to trade generality for effectiveness — i.e., to limit the programming idioms available to the programmer, effectively narrowing down the scope of programs that he/she is allowed to write, in order to obtain better analysis performance for programs that remain within that scope. This is the approach that we take, showing that various analysis tasks may indeed become significantly easier for programs written using specific concurrency models. In other words, we advocate a *design for analysis* approach: by carefully choosing the programming idioms to be used in the development of a particular system, one

can render the resulting programs easier to analyze, ultimately increasing software reliability. Observe that this direction is mostly orthogonal to the advanced analysis techniques mentioned in Section 1.1, and that those techniques can still be applied to software written using the more constrained programming models that we advocate.

In Part II of the thesis we demonstrate several ways in which using simple programming idioms is beneficial to program analysis. This is not entirely surprising — as mentioned before, advanced programming features (e.g., pointers, aliasing) are known to be problematic for modern analysis tools. Taken to the extreme, it is not difficult to believe, for example, that a C++ program is harder to analyze than a finite state automaton. This naturally gives rise to the question: is it possible to find programming models that are simple enough to facilitate program analysis on the one hand, but which are expressive and convenient enough to be appealing to programmers on the other hand? Indeed, if we only advocate trivial programming models they will never be adopted by programmers. Consequently, a great deal of our efforts in this thesis is devoted to studying the delicate trade-off between the simplicity needed for analysis and the richness required by programmers. In particular, in Part III we show that the simple models studied in Part II are sufficiently rich and expressive to be used for various real-world purposes.

We stress that we are not looking for a "magical" programming model that would constitute a silver bullet for solving the analysis problem in general: indeed, we do not think that such a model exists. Instead, our approach is *modular*: we often focus on individual programming idioms instead of whole programming models, attempting to characterize for each idiom the programming instances in which it is required, as well as the analysis costs that it entails. Our ultimate goal is to provide software engineers with a catalog of programming idioms, from which they would be able to tailor a programming framework to their specific needs — including enough concurrency idioms to make programming convenient, but also adding "just enough" concurrency to keep the model simple and amenable to analysis.

In this thesis we focus primarily on three fundamental concurrency idioms (and combinations thereof): the *requesting*, *waiting-for* and *blocking* of discrete events. These idioms, and the motivation for focusing on them, are discussed in Chapter 2, and in the subsequent chapters we provide an in-depth study of their properties, analysis- and engineering-wise. We hope that our results, which we find promising, will trigger additional research in this direction.

Chapter 2

The Request-Wait-Block Model

In this thesis we demonstrate our design-for-analysis approach using the *Request-Wait-Block* ($\mathcal{RWB}$) model for concurrent programs. We chose to focus on this model because (i) it is simple; and (ii) it offers several software-engineering advantages. As explained in Section 1.2 it is of critical importance for us to have both of these properties in models that we study, in order to make our design for analysis approach viable. In this chapter we explain the principles of this model and the motivation behind it.

2.1 Behavioral Programming

Behavioral Programming (*BP*), is a programming language recently proposed by Harel, Marron and Weiss [90]. It is an extension and a generalization of scenario-based programming, which was introduced with the language of live sequence charts (*LSC*s) [57, 87]. A behavioral program consists of independent threads of behavior that are interwoven at run time. Each *behavior thread* (abbr. *b-thread*) specifies events and event sequences which, from its own point of view must, may, or must not occur. As shown in Figure 2.1, the infrastructure synchronizes and interweaves all behaviors, selecting events that constitute integrated system behavior without requiring direct communication between b-threads. Specifically, all b-threads declare events that should be considered for triggering (called *requested events*) and events whose triggering they forbid (*block*), and then synchronize. An *event selection mechanism* (*ESM*) then triggers one event that is requested and not blocked, and resumes all b-threads that requested the event. B-threads can also declare events that they simply “listen-out for”, and they too are resumed when these waited-for events occur.

One of BP’m main source of appeal is that it enables and facilitates *incremental modularity*. In particular, it is possible in BP to iteratively add new threads to the program, each representing an additional scenario or a requirement that the system needs to handle; and the event selection mechanism then interweaves these threads together, producing cohesive system

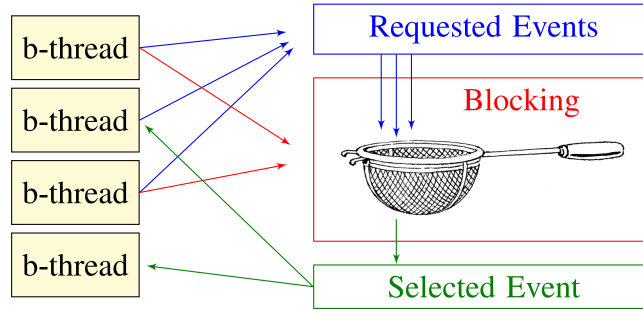


Figure 2.1: The behavioral programming execution cycle: all b-threads synchronize, declaring requested and blocked events; a requested event that is not blocked is selected and b-threads waiting for it are resumed.

behavior. Often, it is enough to add new threads without changing existing code — a small illustrative example appears in Figure 2.2.

More detailed examples showing the power of incremental modularity in behavioral programming appear in [89, 91]. One example is a behavioral program for playing Tic-Tac-Toe [89]. There, each game-rule is implemented in a dedicated b-thread; e.g. *"block X moves when it is O's turn"* or *"block marking of already-marked squares"*. Similarly, player-strategy modules are oblivious of other strategies; e.g., *"wait for two X marks in the same line, and then request marking O in that line"*. A similar technique is used to control a robot performing simultaneous missions, such as vehicle operation and route management.

In [91], a program is presented for stabilizing a quadrotor — an unmanned flying vehicle with four rotors. There, four b-threads each control a particular orientation angle, or the quadrotor's altitude, solely by changing rotor speeds. Each b-thread repeatedly requests and blocks events representing possible increases or decreases of rotor RPM, which could contribute to its own goal. The triggering of an event that is requested by one or more b-threads and blocked by none allows at least one b-thread to progress. Affected b-threads can then recalculate their declarations of requested and blocked events, and the process repeats.

Additional motivation for using behavioral programming is that its strict and simple mechanism for inter-object communication, e.g. its request-wait-block interface, results in b-threads that are often aligned with how humans often describe behavior [74, 90], and that they foster abstract programming skills [12]. We point out that the requesting, waiting-for and blocking idioms are common and appear in various additional models, such as *publish-subscribe* architectures [69], *live sequence charts* [57, 87], and *supervisory control* [136].

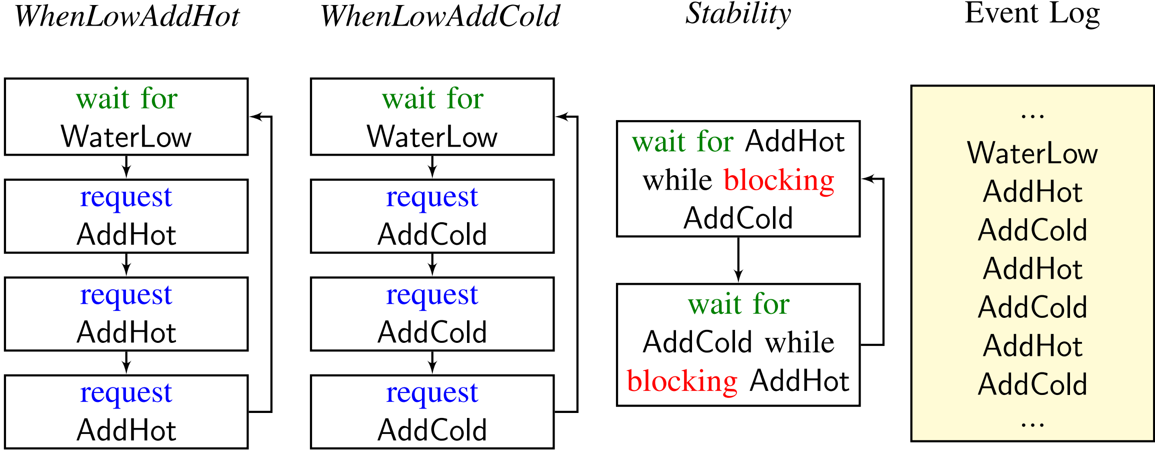


Figure 2.2: The Incremental development of a system for controlling water level in a tank with hot and cold water sources. Whenever the water level in the tank is too low, we assume a WaterLow event is triggered. The b-thread *WhenLowAddHot* repeatedly waits for WaterLow events and requests three times the event AddHot. *WhenLowAddCold* performs a similar action with the event AddCold, reflecting a separate requirement, which was introduced when adding three water quantities for every sensor reading proved to be insufficient. When *WhenLowAddHot* and *WhenLowAddCold* run simultaneously, the six water addition events — three AddHot events and three AddCold events — can be triggered in any order. A new requirement is then introduced, to the effect that water temperature should be kept stable. We add the b-thread *Stability*, to interleave AddHot and AddCold events.

2.2 The Underlying *RWB* Model

While behavioral programming is geared towards natural and intuitive development using almost any programming language, its underlying infrastructure can be conveniently described and analyzed in terms of transition systems in the $\mathcal{RWB}$ model. There are several ways to define these transition systems; we give here the definition proposed in [7].

A b-thread BT over event set E is a tuple $BT = \langle Q, \delta, q_0, R, B \rangle$, where Q is a set of states (one for each synchronization point), q_0 is the initial state, $R : Q \to 2^E$ and $B : Q \to 2^E$ map states to the sets of events requested and blocked at these states (respectively), and $\delta \subseteq Q \times E \times Q$ is a transition relation. When δ is a deterministic, i.e. when

$$\langle q, e, q_1 \rangle \in \delta \wedge \langle q, e, q_2 \rangle \in \delta \quad \Rightarrow \quad q_1 = q_2$$

we say that BT is a deterministic thread. When $\langle q, e, \tilde{q} \rangle \in \delta$ we sometimes write $\tilde{q} \in \delta(q,e)$; and if δ is also deterministic, we sometimes abuse notation and write that $\delta(q,e) = \tilde{q}$.

Behavioral programs are created by *composing* b-threads. The parallel composition of threads $BT^1 = \langle Q^1, \delta^1, q_0^1, R^1, B^1 \rangle$ and $BT^2 = \langle Q^2, \delta^2, q_0^2, R^2, B^2 \rangle$ over the common event set E yields the b-thread defined by $BT^1 \parallel BT^2 = \langle Q^1 \times Q^2, \delta, \langle q_0^1, q_0^2 \rangle, R^1 \cup R^2, B^1 \cup B^2 \rangle$, where

$\langle \tilde{q}^1, \tilde{q}^2 \rangle \in \delta(\langle q^1, q^2 \rangle, e)$ if and only if $\tilde{q}^1 \in \delta^1(q^1, e)$ and $\tilde{q}^2 \in \delta^2(q^2, e)$. The union of the labeling functions is defined in the natural way; i.e. $e \in (R^1 \cup R^2)(\langle q^1, q^2 \rangle)$ if and only if $e \in R^1(q^1) \cup R^2(q^2)$. A *behavioral program* P comprised of b-threads $BT^1, BT^2, \ldots, BT^n$ is the composite thread $P = BT^1 \parallel \ldots \parallel BT^n$. We sometimes refer to this composite thread as the *state graph* of P. If threads $BT^1, \ldots, BT^n$ are all deterministic, we say that P is deterministic.

Let $P = \langle Q, \delta, q_0, R, B \rangle$. An execution of P starts from q_0, and in each state q along the run an enabled event is chosen for triggering, if one exists (i.e., an event $e \in R(q) - B(q)$). Then, the execution moves to state $\tilde{q} \in \delta(q, e)$, and so on. An execution can be infinite, or finite if it ends in a state with no successors (a *deadlock* state); and it can be formally recorded as a (possibly infinite) sequence of states and triggered events, $\varepsilon = q_0 \overset{e_1}{\rightarrow} q_1 \overset{e_2}{\rightarrow} \cdots$. The matching set of events, without states, is called a *run*. The set of all runs of the program is denoted by $\mathfrak{L}(P)$.

Note that when implemented in a standard programming language, we assume that b-threads do not share data, and rely solely on events for input and output. This results in the abstraction that a behavior thread is "in a state" only when synchronized with others, and that the state transition caused by executing program instructions between synchronization points is atomic.

In Chapter 9 we describe how the individual transition systems underlying b-threads can be automatically extracted from high level code and then composed, in order to formulate a program's underlying transition system. This procedure is highly useful for analysis purposes.

Chapter 3

Summary of Contributions

In this chapter we present a summary of the results that we have obtained so far and explain how they connect to the general theme of our thesis. These results span eight papers. For each result we provide here only a brief overview; they are each discussed in depth in the following chapters. Our results are logically partitioned into two groups: those appearing in Part II of the thesis discuss how writing programs using the $\mathcal{RWB}$ model can facilitate analysis, whereas those in Part III show how this relatively simple model can be applied in the development of large systems.

3.1 Automatic, Non-Intrusive Repair of *RWB* Programs

This work, which appeared in [6] and is discussed in Chapter 4, studies the amenability of the $\mathcal{RWB}$ model to program repair.

Software maintenance is a difficult and error prone task, which poses a serious burden on engineers. As errors (bugs) are discovered and requirements are added or changed, the developers work hard to modify existing code without introducing new errors. They are often constrained by limited knowledge of possible side-effects, since undocumented interdependencies might have been forgotten or might be known only to different people (usually, the original developers) who are unavailable. Research on automated program aims to address these and related challenges.

Several approaches to program repair have emerged over the years. Some focus on error localization — identifying the module responsible for the problem, and then removing it and synthesizing a replacement module [104, 143, 142]. Another approach called *genetic programming* involves repeatedly creating various mutations the code, comparing them against a specification, and at each phase selecting the mutation that does best [150]. This approach is sometimes extended by mutating the specification too, not just the program [21]. See [111] for a survey. The approach we take here is complementary: we seek to study whether certain

concurrency idioms could make program repair easier.

As it turns out, the presence of the *blocking* idiom in a program's underlying concurrency model can serve to facilitate its repair. Consider, for instance, a program that typically performs well, but occasionally malfunctions (i.e., it violates a *safety* property). Intuitively, we could use the blocking idiom to block event sequences that lead to violations, retaining just the good behavior of the program. Further, consider cases in which a program gets stuck in an infinite loop without achieving its goals (i.e., it violates a *liveness* property). Blocking could likewise be used to force the execution out of the infinite loop and direct it towards fulfilling its specification. In both cases, a major advantage in applying this form of repair (made possible due to the blocking idiom) is that it results in a *non-intrusive patch*: a piece of code that is added to the program automatically, without making changes to the original code. We show that these patches do not introduce new bugs and do not affect the rest of the program's functionality.

3.2 Module-Based Abstraction and Repair of *RWB* Programs

In this work, which appeared in [7] and is discussed in Chapter 5, we continue our study of the automatic repair of $\mathcal{RWB}$ programs. Specifically, we study the compatibility of our repair approach with abstraction/refinement capabilities. Our work in [6] presented repair techniques that are useful for $\mathcal{RWB}$ programs, but which require traversing the input program's state graph — thus limiting the scalability of the approach. To mitigate this difficulty, we turned to abstraction techniques, which are a widely employed strategy in combating state explosion.

Intuitively, an abstract program forms an over-approximation of the original program: it contains all the behaviors of the original program and perhaps additional, spurious behaviors (there also exist under-approximation techniques, but this is beyond the scope of this thesis). Abstract programs are useful because their representations are typically smaller. Thus, in cases where analysis tools can operate on the abstract program instead of the concrete one, their scalability is improved. In this work we thus set out to combine abstraction techniques with our program repair techniques.

To demonstrate how abstraction techniques may be applied to the $\mathcal{RWB}$ model, we define an abstraction operator that transforms an $\mathcal{RWB}$-thread into a new thread, that has fewer states and is more *permissive* than the original one. Intuitively, this is straightforward to perform, given the $\mathcal{RWB}$ idioms: we simply combine sets of states (which correspond to sets of synchronization points) into a single state that requests additional events or blocks fewer events (or both) compared to any of the original states. We formally prove that abstracting one or more of the program's threads in this way results in an over-approximation of the entire program, and propose a methodology (and tool support) for producing these abstract programs in practice.

Having defined the abstraction operator, we propose an abstraction-based repair scheme

for safety violations in $\mathcal{RWB}$ programs, which generalizes our previous work [6] and, again, utilizes the blocking idiom. At first glance, one might be tempted to simply apply the original technique to an abstract program, which is, itself, an $\mathcal{RWB}$ program. However, due to the fact that states in the abstract program represent multiple concrete states, doing so may result in blocking behaviors that are in fact permitted in the original program. We resolve this by introducing a novel *counter-example guided abstraction/refinement* (*CEGAR*) [51] approach to program repair.

In CEGAR-based model checking, one begins with a coarse abstract program and model checks it to see if a given property holds. If the model checker replies in the affirmative, the property is known to hold for the concrete program as well. Alternatively, whenever a counter-example is produced, it is tested on the concrete system to see if it is genuine; and if it is not, the abstract program is refined in a way that eliminates the spurious run. The process is then repeated iteratively, until a conclusion is reached

Our adaptation of CEGAR to program repair follows in this spirit: our repair algorithm is run on the abstract program and returns a patch. If we determine that this patch would also block good runs of the system because the abstraction is too coarse, we refine the program in a way that causes the repair algorithm to not generate this patch again; otherwise, we apply the patch and are done. To the best of our knowledge, this is the first application of CEGAR-based techniques to program repair. We evaluated our technique on a large $\mathcal{RWB}$ implementation of a TCP/HTTP protocol stack, and obtained encouraging results.

3.3 On the Succinctness of *RWB* Programs

In this work [4], also discussed in Chapter 6, we take a more rigorous view of the $\mathcal{RWB}$ model in comparison to other models of concurrency. In particular, we study the model's *descriptive succinctness* — a criterion for comparison of models that has been used ever since the Rabin-Scott work on nondeterministic automata [135]. Intuitively, this criterion measures the size of the smallest program in a model that can produce a certain language. The significance of using succinct models lies in the strong connection between succinctness and *software reliability* [129], indicating that succinct software is easier to develop, maintain and reuse. Further, the descriptive succinctness of a model is often connected to the complexity of various decision problems in it [100], and may be relevant to various verification problems — as we later demonstrate in Chapter 7.

Here, we compare the $\mathcal{RWB}$ computational model to three well-known types of automata: automata with classical nondeterminism [135], automata with universal ("and") nondeterminism, and bounded concurrent automata [66]. Each of these models is known to be exponentially more succinct than deterministic, non-parallel automata [66]. Further, all three types are *or-*

thogonal, meaning that their descriptive succinctness is independent of the others and additive with respect to them. In particular, [66] shows that the model with all three idioms (existential nondeterminism, universal nondeterminism and bounded concurrency) affords a tight triple-exponential gap in succinctness compared to non-parallel automata.

The first issue we address is the classification of $\mathcal{RWB}$ with respect to these fundamental models. We show that any $\mathcal{RWB}$ program is polynomially reducible to a bounded concurrent automaton — and that hence, $\mathcal{RWB}$ can be regarded as a fragment of languages that are based on these automata, such as Statecharts [78]. Next, we observe that, succinctness-wise, $\mathcal{RWB}$ is a proper subset of bounded concurrent automata: there exist languages that have a succinct description using bounded concurrent automata, but that incur an exponential blowup when described using an $\mathcal{RWB}$ program. However, we show that not all is lost, as $\mathcal{RWB}$ programs may still be exponentially more succinct than deterministic, non-parallel automata.

Having classified $\mathcal{RWB}$ as a fragment of bounded concurrent automata, we examine its relationship with the other two models — nondeterministic and universal automata. We are able to prove that when the three idioms are combined, the resulting model is triple-exponential more succinct than the deterministic non-parallel model. This strengthens the result of [66], by showing that one can achieve the same succinctness even when limited to the $\mathcal{RWB}$ fragment of Statecharts.

In the last part of this work we explore the contribution of each of the $\mathcal{RWB}$ idioms (requesting, waiting-for and blocking of events) to the succinctness of the framework as a whole. We show that the contribution of each of the idioms is exponential; i.e., their removal results in an exponential blowup for some languages.

This research is highly pertinent to the general theme of our thesis, for two reasons:

1. By showing that the $\mathcal{RWB}$ model is succinct, and that this succinctness is orthogonal to that provided by nondeterministic and universal automata, we establish the power of the $\mathcal{RWB}$ idioms. The importance of this result is not by showing that concurrency leads to succinctness — indeed, this is widely known — but that even very simple idioms, like requesting, waiting-for and blocking, suffice in this respect. Combined with the usefulness of these idioms, as discussed in, e.g., Chapters 4 and 7, this demonstrates the benefits of models that employ the $\mathcal{RWB}$ idioms.

2. Our results include an attempt at characterizing the cases in which each of the individual idioms leads to succinct programs. This could help programmers tailor a model to their specific needs, by picking only those idioms that are useful for the problem at hand — thus retaining "just enough concurrency" to program effectively and succinctly, while also keeping the programs simple and maintainable.

3.4 Compositional Verification of *RWB* Programs

One analysis task that has received a great deal of attention is that of *formal verification* [52]. In formal verification, one takes as input a program P and a specification ϕ, and attempts to prove that P satisfies ϕ — or provide evidence that it does not (typically, a faulty execution). Like many other analysis techniques, verification is prone to the state explosion problem, and is notoriously difficult.

The difficulty in verifying large systems has been studied extensively over the years. One of the prominent techniques for tackling it is that of compositional verification: a divide-and-conquer approach, in which smaller portions of the system are verified in isolation, and in a manner that implies system-wide correctness (see, e.g., [128, 133, 97, 53], among many others). Unfortunately, compositional verification, and in particular automated versions thereof [67], is a difficult challenge in its own right [54].

In this work [1], also discussed in Chapter 7, we adopt a software-engineering based approach to compositional verification: if compositionally verifying an existing program is difficult, perhaps one can design the program in the first place in a way that makes it more amenable to such techniques. In particular, we attempt to exploit the properties of programs written in the $\mathcal{RWB}$ model (primarily the fact that components therein communicate through a strict protocol) to facilitate compositional verification.

The first step in compositional verification is to divide the program in question into components. In the context of $\mathcal{RWB}$, the natural approach is to have each thread form a component, and indeed this is the path we take. Then, one must define the desired properties of each of the threads. This part, which is the only non-automated part in our proposed approach, is performed by the programmer, who most probably already has the properties of the thread in mind when he or she defines it. Here, the $\mathcal{RWB}$ infrastructure plays a key role: the fact that inter-thread interfaces are strict and well defined simplifies the process of characterizing the properties of single modules.

The next step is to verify that each of the implemented threads satisfies its respective properties. This is performed by directly model checking the implemented threads, using the technique of [86]. By checking each module in isolation, as opposed to the composite program, we significantly reduce the number of states that need be explored. This statement is supported by the fact that $\mathcal{RWB}$ modules may be significantly more succinct than the composite program [4] (see Chapter 6).

Finally, one must show that the thread properties imply the desired system-wide property. For this task we employ Z3 [60] — a state-of-the-art SMT solver and theorem prover. When the program in question deals with theories known to Z3, such as integer arithmetic or bit vectors, run times may be exponentially shorter than those of a traditional model checker.

Our experimental results were inconclusive. On some examples our approach performed very well, sometimes greatly out-performing direct model checking of the composed program. On other examples, however, the results were worse than those of direct model checking. Still, we consider the approach to be advantageous: indeed, it imposes very little overhead upon the programmer, and in some cases it significantly speeds up verification; and whenever it fails to do so, the programmer can always default to traditional model checking.

3.5 Theory-Aided Compositional Verification of Concurrent Transition Systems

In this work [8], discussed in Chapter 8, we improve and generalize the direction presented Chapter 7. In particular:

1. We strengthen the interfaces with the SMT solver used in proving global correctness. This allows us to both improve performance in many instances, and also to quickly recognize instances for which the approach will not do well, informing the users that they had better use direct model checking.

2. We automate the generation of thread properties, the only part of the original approach that was manual, thus making the technique fully automatic.

After the completion of [1], we attempted to gain a deeper understanding of the difficulties we encountered. Our research showed that the need to translate $\mathcal{RWB}$ programs into the language of the SMT solver was a major bottleneck. In particular, modern SMT solvers "speak the language" of e.g., linear arithmetic, bit-vectors and uninterpreted functions, and are ill-equipped to handle transition systems such as $\mathcal{RWB}$ programs. In order to address these difficulties, we introduce in this work a rigorous formalization and implementation of an SMT theory solver for a *theory of transition systems* ($\mathcal{TS}$) within the context of CVC4 [26] — a lazy, DPLL(T) based SMT solver [130]. The $\mathcal{TS}$ solver takes as input formulas describing a program's concurrent threads (given as transition systems) and the assertion that a certain safety property is violated, and it answers UNSAT if the program is safe, or SAT if it is not. Thus, the $\mathcal{TS}$ solver enables the SMT solver to operate directly and more effectively on $\mathcal{RWB}$ inputs, resulting in significant performance improvement.

In its basic form, the $\mathcal{TS}$ solver performs explicit model checking: it explores the space of reachable states of the given system in order to determine its safety, and this exploration is driven by the SMT solver's underlying SAT engine. However, the larger portion of our work here is devoted to enabling the $\mathcal{TS}$ solver to analyze the input threads and look for pre-supplied *patterns*: structural properties of the threads that may be expressed as assertions in

the languages of other theories supported by the SMT solver, such as arithmetic or arrays. Through these assertions, other theories can "understand" the input program and efficiently discover, e.g., that a certain branch of the search space cannot lead to a violation. This allows the $\mathcal{TS}$ solver to quickly prune the search space, greatly reducing the number of states that need to be explored. A key fact here is that each thread/transition system is analyzed separately — and hence the compositionality of our approach: the analysis complexity is proportional to the size of the program and not to that of its state space. This process can be regarded as the automation of the manual property derivation described in [1].

A critical aspect of this approach is the discovery of useful thread properties: properties that can imply the global correctness, or prune large portions of the search space, but which are easy to discover in individual threads. Here the simple idioms of the $\mathcal{RWB}$ model again come into play, as the strict synchronization mechanism of the model makes it easier to formulate the properties of individual threads. Indeed, our empirical results are quite promising [8], and show that even a small number of stored patterns can already apply to a large variety of programs, accelerating verification.

While our work here focuses solely on the $\mathcal{RWB}$ models, we believe that the technique may be applied to discrete event models with other concurrency idioms.

3.6 Distributed *RWB* Programs

Heretofore, we have shown how the simple idioms of the $\mathcal{RWB}$ model render it amenable to various forms of analysis. The beneficial traits of $\mathcal{RWB}$ that we discussed originated in part from the model's highly synchronous nature — and in particular, from the fact that all threads must repeatedly synchronize globally throughout the run. However, while this synchronization is clearly beneficial for analysis tasks, it entails high costs, time- and communication-wise, which could harm the performance of $\mathcal{RWB}$ software. It also seems to limit $\mathcal{RWB}$'s applicability in distributed settings and in settings where various threads operate on different time scales. In [2], discussed also in Chapter 9, we develop techniques through which these difficulties may be mitigated.

A key observation made in [79] is that although formally $\mathcal{RWB}$ requires every thread to synchronize at every synchronization point, this is often unnecessary. For instance, if all threads but one have synchronized and event e is requested and is not blocked (by the synchronized threads), and the remaining, unsynchronized thread is known to never block e, then e may be triggered immediately — without violating the semantics of $\mathcal{RWB}$. The unsynchronized thread is then notified that e has been triggered so that it may process this information at some point in the future (when it reaches its synchronization point). In some cases, e.g., when the unsynchronized thread is known to "not care" about e, it need not even be notified, thus reducing

communication costs. In the meantime, the rest of the system may continue with its execution. This form of execution is called *eager synchronization*.

In Chapter 9 we demonstrate how eager synchronization can help in implementing distributed $\mathcal{RWB}$ programs. In particular, we show how this process can be mechanized, by automatically analyzing threads and discovering, e.g., the events they never request or block. This information is gathered, and is used in determining whether events may be triggered when threads are not yet synchronized. We also discuss the automatic discovery of logical modules in the program. Finally, we prove that applying the relaxed synchronization mechanism in distributing $\mathcal{RWB}$ programs does not change the language that the programs generate. In other words, all previously discussed techniques, such as program repair and verification, remain sound; they may be safely applied to a fully centralized version of the program, even if a distributed version thereof is used in practice, and the results will remain valid.

3.7 Scaling-Up *RWB*: Steps from Basic Principles to Application Architectures

In Section 1.2 we mentioned that there is a certain trade-off between amenability to analysis and ease of programming: the more concurrency idioms we add to a model the more attractive programmers are likely to find it, but this will likely make the model harder to analyze. Thus, in the line of work discussed in [3] and in Chapter 10, we set out to check whether the simple $\mathcal{RWB}$ idioms suffice for the programming of a real world system — in this case, a functioning web-server, comprised of TCP and HTTP stacks. This case-study exposed a few limitations of the $\mathcal{RWB}$ model, which we addressed by adding extensions to it. However, these extensions were small and in line with the general approach of the $\mathcal{RWB}$ model. Hence, we conclude that the $\mathcal{RWB}$ principles are indeed suitable for the programming of large systems.

The most significant issue that we had to address, where we felt the traditional $\mathcal{RWB}$ idioms were not entirely adequate, was in dealing with *time*. Indeed, the eager synchronization extension to $\mathcal{RWB}$ [2] allows one to handle timing constraints to some extent, but we felt that a more fundamental solution was in order. In this work we solve this difficulty by introducing a *timeout* idiom to the thread synchronization interface, by which, if a synchronization point has not been resolved in the specified amount of time, a thread may change states without waiting for the triggering of an event. Other, smaller changes that we propose, aimed mainly at providing programmers with more convenient interfaces, include the addition of parametrized events, the ability to define a program-specific event selection strategy, and allowing the dynamic creation of threads. We show how each of the proposed idioms plays a role in the programming of our case-study, and also provide a formal semantics for the resulting extended model.

3.8 An Initial Wise Development Framework for *RWB*

In Chapter 11 we seek to bring together several of our aforementioned results, by providing an interactive and proactive framework for developing $\mathcal{RWB}$ programs [5] — which utilizes our previously developed analysis tools.

As discussed in Section 1.1, one of the key reasons for the great difficulty in developing reliable reactive software is the growing complexity of many kinds of reactive systems, which increasingly prevents the human mind from managing a comprehensive picture of all their relevant elements and behaviors. Over the years it has been proposed, in various contexts, e.g., [138, 137, 43, 82], that a possible strategy for mitigating these difficulties could lay in changing the role of the computer in the development process. Instead of having the computer serve as a tool, used only to analyze or check specific aspects of the code as instructed by the developer, one could seek to actually transform it into a member of the development team — a proactive participant, analyzing the entire system and making informed observations and suggestions. This way, the idea goes, the computer's superior capabilities of handling large amounts of code could be manifested. Combined with human insight and understanding of the system's goals, this synergy could produce more reliable and error-free systems.

In this work we follow this spirit, and present a methodology and an interactive framework for the modeling and development of complex reactive systems, in which the computer plays a proactive role. Following the terminology of [82], and constituting a very modest initial effort along the lines of the Wise Computing vision outlined there, we term this framework a *wise* framework. Intuitively, a truly *wise* framework should provide the developer with an interactive companion for all phases of system development, "understand" the system, draw attention to potential errors and suggest improvements and generalizations; and this should be done via two-way communication with the developer, which will be very high-level, using natural (perhaps natural-language-based) interfaces. The framework presented here is but a first step in that direction, and focuses solely on providing an interactive development assistant capable of discovering interesting properties and drawing attention to potential bugs; still, it can already handle non-trivial programs, as we later demonstrate through a case-study.

A main novel aspect of our approach is in the coupling of the notion of a proactive and interactive framework with the $\mathcal{RWB}$ model. It is now widely accepted that a key aspect in the viability of analysis tools and environments is that they are sufficiently lightweight to be integrated into the developer's workflow without significantly slowing it down [139, 56]. We attempt to achieve this by leveraging the simplicity of the $\mathcal{RWB}$ model. As demonstrated in Chapter 11, the proactiveness of our approach and its tight integration into the development cycle can lead to early detection of bugs during development, when they are still relatively easy and cheap to fix.

Part II

Formal Analysis of *RWB* Programs

Chapter 4

Automatic, Non-Intrusive Repair of *RWB* Programs

4.1 Introduction

Software maintenance is a difficult and error prone task. As errors (bugs) are discovered and requirements are added or changed, developers work hard to modify existing code without introducing new errors. They are often constrained by limited knowledge of possible side-effects, since undocumented interdependencies might have been forgotten or might be known only to different people (usually, the original developers) who are unavailable. Research on automated program repair and, more generally, program synthesis from specifications, aims to address these and related challenges. Such automation may prove particularly valuable for handling failure/bug reports from users who simply press the *"Send to Software Vendor"* button. In such cases, the software engineer cannot discuss with the user the context of the problem, or possible generalizations thereof.

In this chapter we focus on programs written using the $\mathcal{RWB}$ model, and our work is centered on the idea of repairing by carefully forbidding existing faulty execution paths. This technique is highly suitable for (a) non-intrusive incremental repair; i.e., large parts of the system are already developed and are not modified by the repair process; (b) methodological integration of the repair process with standard, ongoing development during and after the repair activity; and (c) practical techniques for dealing with the complexity of the use of model checking when creating local patches in the repair process.

4.2 Outline of the Repair Approach

In the present chapter we utilize the model checker of [86] to automate elements of manual program-repair processes, using a principle that can be summarized as "taking the road not taken". For illustration, assume that a system was tested, or even model checked, to satisfy its specification, and a new requirement was then introduced, or a bug reported, highlighting a required property not previously articulated, and thus neither tested nor model checked. Our method calls for first adding the new property to the specification. We then model check the program to find distinct violating runs. In the case of violated *safety* properties ("bad things never happen"), for each such run we add a special b-thread that waits for the sequence of all events in the run, up to the last one requested by the program (rather than by the environment). The repair b-thread then blocks this event. Some other pending requests might then be triggered. Violated *liveness* properties ("good things eventually happen") are handled similarly: when the system is traversing a loop in which "good things do not happen", the repair b-thread applies blocking to steer the run in another direction. In the liveness case blocking is only performed with some small probability, thus injecting bias towards certain desirable execution paths without forbidding other paths which are also permitted.

For example, consider a faulty game-strategy b-thread, whose event request leads to a loss. When this event is blocked, another b-thread, perhaps one that requests a set of default moves, comes into play (so to speak), offering an alternative. The elimination process continues until "the right" default move is the choice at that state. The new corrective wait-and-block behavior is non-intrusive, in that its implementation does not require changing the existing program code.

We refer to such a repair b-thread as *a patch*, and to the process as *patching*, or simply, *repairing*. We hope that combined with the behavioral-programming principles, our approach will help make the concept of patching seem less a "necessary evil" and more a useful, mainstream software maintenance practice.

As the full repair algorithm may not scale up to large programs due to the state explosion problem, we also discuss the case where patching can be limited to a bounded "neighborhood" of a specific operation scenario; for example, when we are provided with a bug report sent from a user.

We formally prove correctness and analyze the method, characterize the programs on which it can be used, and exemplify its usage with our proof-of-concept tool — implemented within the *BPJ* framework for behavioral programming in Java [89].

The rest of this chapter is organized as follows. Basic definitions regarding the model checking of behavioral programs are given in Sections 4.3 and 4.4, respectively. The repair of safety violations of loopless programs is discussed in Section 4.5, followed by a repair algo-

rithm for safety violations in general programs in Section 4.6. Next, in Section 4.7 we extend the algorithm to also handle liveness violations. A method for handling large programs, called limited-depth patching, is described in Section 4.8. Related work is discussed in Section 4.9, and we conclude with Section 4.10.

4.3 Definitions

We use here a slightly extended version of the definitions of an $\mathcal{RWB}$ program given in Section 2.2. These definitions are needed to discuss the model checking of behavioral programs.

A deterministic behavior thread over a set of atomic propositions AP is a tuple $BT = \langle Q, \delta, q_0, R, B, L \rangle$. The definitions of Q, δ, q_0, R and B are as in Section 2.2. L is a labeling function, $L: Q \to 2^{AP}$, that maps each state to a set of atomic propositions that hold in that state.

The composition of behavior threads with atomic proposition is, again, performed similarly to the composition define in Section 2.2. The only addition is that in the composition of threads $BT^1 = \langle Q^1, \delta^1, q_0^1, R^1, B^1, L^1 \rangle$ and $BT^2 = \langle Q^2, \delta^2, q_0^2, R^2, B^2, L^2 \rangle$, over the common event set E and common atomic proposition set AP, the atomic propositions assigned to a composite state are given by:

$$L^1 \cup L^2(\langle q^1, q^2 \rangle) = L^1(q^1) \cup L^2(q^2)$$

For a deterministic behavior program over a set of atomic propositions AP, we associate each run $\rho \in \mathfrak{L}(P)$ with a *trace* of atomic propositions. Specifically, for a run $\rho = e_1, e_2, \ldots$, such that the execution corresponding to ρ is $q_0 \xrightarrow{e_1} q_1 \xrightarrow{e_2} q_2 \ldots$, we define $\mathrm{Tr}(\rho) = L(q_0)L(q_1)L(q_2)\ldots$ and define the set of all traces of P to be

$$\mathrm{Tr}(P) = \{\mathrm{Tr}(\rho) \mid \rho \in \mathfrak{L}(P)\}$$

In the context of repair, it is useful to distinguish between *system events*, which are events controlled by the system, and *environment events* which are controlled by the system's environment (for example, the user). We denote these two sets of events as E_{sys} and E_{env}, respectively. The set of all program events E is the disjoint union of E_{sys} and E_{env}.

Repairing a program is always done with respect to a violated *specification*:

Definition. *A* specification *for a behavioral program P is a linear time (LT) property Φ (i.e. a subset of $(2^{AP})^{\omega}$). We say that P satisfies Φ, denoted $P \models \Phi$, iff* $\mathrm{Tr}(P) \subseteq \Phi$.

Since this definition assumes infinite runs, when dealing with systems of finite runs we pad any finite run with the trace $\varnothing^{\omega}$.

It is important to note, that the same set of b-threads can satisfy Φ with one event selection mechanism, and not with another. We adopt a wider perspective here, and ensure that the

patched set of b-threads satisfies Φ with *all* event selection mechanisms. Such patching immediately detects and fixes any bugs that could have remained hidden with a certain mechanism, but which may emerge later. An approach that takes a specific event selection mechanism into account may also be useful for some applications.

In this context we focus on two major types of LT properties: safety properties and liveness properties. We define safety properties first, and give also the related definitions of invariants and deadlocks.

Definition. *An LT property Φ over AP is called a* safety *property if for all $\sigma \in (2^{AP})^{\omega} - \Phi$ there exists a finite prefix $\bar{\sigma}$ of σ such that*

$$\Phi \cap \left\{ \sigma' \in \left(2^{AP}\right)^{\omega} \mid \bar{\sigma} \text{ is a finite prefix of } \sigma' \right\} = \phi.$$

Intuitively, a safety property states that no "bad" sequences of events may happen. Any run that causes such a sequence has a *bad prefix*; after it the run does not satisfy the property no matter how it continues.

The notion of *invariants* plays a key role in the model checking of safety properties:

Definition. *An LT Φ property over AP is an* invariant *if there is a propositional logic formula φ over AP such that $\Phi = \left\{ A_0 A_1 A_2 \ldots \in (2^{AP})^{\omega} \mid \forall j \geq 0, A_j \vDash \varphi \right\}$.*

Intuitively, invariants are properties of the current state of the system, and do not reflect the history of events leading to it.

Through invariant checking one can handle *regular safety properties*: those safety properties for which the associated bad prefixes are recognizable by some finite automaton [23], or, in our case, there is a b-thread that marks its state as bad when the bad prefix is recognized. By applying the invariant model checker to a program with these threads added, we can effectively handle general regular safety properties.

Definition. *We say that a (finite) run $\rho = (e_1, e_2, \ldots, e_n)$* causes a deadlock *if it leads to a state s that has no enabled events (all requested events are also blocked).*

Much like invariants, deadlocks too are properties of states in the system, and not of runs.

When patching against safety violations, we will receive as input a program P and an invariant Φ. We will implicitly check that the system has no deadlocks; if it does, the patching algorithm will try to remove them. In particular, we will make sure that no new deadlocks are created while patching; otherwise we could "patch" a system by simply blocking all enabled events at its initial state.

The other type of properties we consider is liveness properties. The following is adopted from [23]:

Definition. *An LT property* Φ *over AP is called a* liveness *property if any finite word can be extended such that the resulting infinite trace satisfies* Φ*. Formally, let* $pref(\sigma) = \{\bar{\sigma} \in (2^{AP})^* \mid \bar{\sigma}$ *is a finite prefix of* $\sigma\}$ *and* $pref(\Phi) = \bigcup_{\sigma \in \Phi} pref(\sigma)$*. Then* Φ *is a liveness property if and only if* $pref(\Phi) = (2^{AP})^*$.

In the case of regular safety properties, invariant checking plays a key role. When it comes to liveness properties, a similar role is played by *persistence* checking:

Definition. *An LT property* Φ *over AP is called a* persistence *property if it states that a certain condition holds forever, from some point in* Φ*. Formally,* Φ *is a persistence property if there exists a propositional logic formula* φ *such that* $\Phi = \{A_0A_1A_2\ldots \in (2^{AP})^\omega \mid \exists_i$ *such that* $\forall_{j \geq i}, A_j \models \varphi\}$*. Formula* φ *is called the persistence (or state) condition of* Φ*.*

As discussed in, e.g., [23], the model checking of regular liveness properties is reducible to persistence checking. The latter is performed by portioning the states of the system into two sets: states in which φ holds, termed “cold” states, and states in which it does not hold, termed “hot” states. Then, the property holds if and only if there are no reachable cycles consisting strictly of hot states (which we refer to as reachable “hot cycles”). This can be checked, for instance, using a nested DFS algorithm.

When patching against liveness violations, we will receive as input a program P and a persistence property Φ. In practice, this property is given by an indicator thread that marks the system’s states as either hot or cold.

4.4 Extending the Model Checking of Invariants and Deadlocks

In order to prepare the ground for the correction of various safety and liveness violations, we begin by describing how to check that a behavioral program satisfies an invariant and is deadlock-free. We follow the algorithm in [23, Section 3.3.1], and the implementation in [86].

Any state that violates the invariant or is deadlocked is marked as “bad”. We construct the state graph of the program, traverse it using DFS (trimming when arriving at a previously visited state), and check that all states reachable from the initial state are not bad. From each state we explore all enabled events (which reflects our decision to cater for all possible event selection mechanisms).

The runtime complexity of this algorithm, implemented as in [86], is as follows. Let $G = (V_G, E_G)$ denote the state graph constructed, and let n be the number of threads and $e = |E|$ the number of events in the original program. $|V_G| + |E_G|$ operations have to be performed to

traverse the graph. Further, for each $state \in V_G$ we have to perform $n \cdot e$ operations in order to find all its enabled events. This yields:

$$T_{mc} = O\left(|E_G| + |V_G| \cdot (n \cdot e)\right).$$

This complexity is the minimum price one has to pay for running a model checker on a behavioral program. Since our technique is based on model checking, it will necessarily be forever linked in complexity to that of model checking [146, 23], and the progress made there, for better or for worse. T_{mc} thus serves a base point with which to compare the complexity of our patching algorithms, and we are interested in how much additional overhead they incur above it.

We actually use a slightly different algorithm. For our purposes, the usual model checking that returns a single violating run does not suffice: we want to explore *all* runs that violate the invariant or cause a deadlock.

This is achieved as follows: we traverse the state graph using the same DFS, but whenever we reach a bad state we store that information in its predecessor states. Each state already visited in the graph will thus contain information on all its bad successors. If the state is reached again, through another route from the root, we need not traverse its subtree again: we simply update the relevant states using the data already stored (see Figure 4.1).

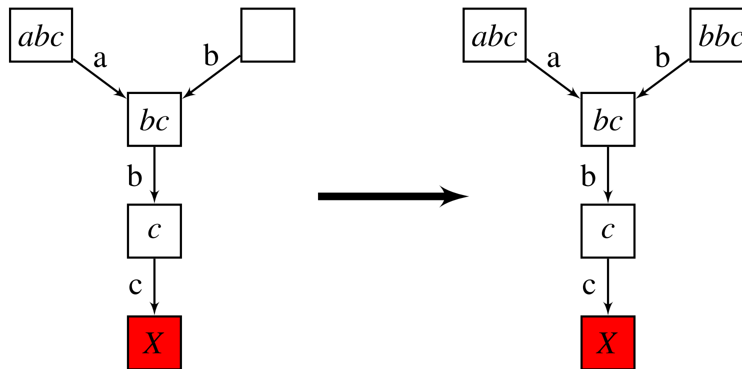


Figure 4.1: When a "bad" state is reached, all its predecessors store the relative path from that point to the violation. When a node in this path is reached through a different path, the data is propagated. The DFS continues until the root stores all violating paths.

The added complexity of this algorithm is measured using the number of violating runs, Υ (OOPSilon: pun intended), and the depth of the state graph D. For each violating run we propagate at most D events to the predecessors, causing an overhead of $1+2+\ldots+D$ per violating run. The total runtime complexity is thus:

$$T = T_{mc} + \Upsilon \cdot (1+2+\ldots+D) = T_{mc} + O(\Upsilon \cdot D^2).$$

Finally, if all direct successors of a state are bad, then the state itself can be considered bad; this is because the patching technique we discuss will cut off the violating children, rendering the state a deadlock. We thus add the following modification: if, during the DFS, all of a state's successors are violating or deadlocked, the state itself is marked as violating; thus its successors can be ignored. The runtime worst-case complexity remains unchanged.

4.5 Safety Patches for Loopless Programs

4.5.1 Generating Linear Safety Patches

Before discussing the safety violation patching of general programs, we begin with the simpler case of finite programs that are *loopless*: their state graph contains no cycles. In a loopless program, every run is finite.

Definition. *A* linear safety wait-block patch *for event sequence* $(e_1, e_2, \ldots, e_n, e_{last})$*, such that* $e_{last} \in E_{sys}$*, is a b-thread with the following properties:*

- *The patch waits for events* $e_1, \ldots, e_n$*, blocks* e_{last} *once and then terminates.*
- *If the run deviates from the sequence* $e_1, \ldots, e_n$*, the patch terminates.*
- *The patch never requests events and does not label states* ($R(q) = L(q) = \varnothing$ *for all* q*).*

Intuitively, the patch is designed to prevent one bad run from occurring. Events $e_1, \ldots, e_n$ will be chosen according to violating runs found by the model checker. The patch will intervene before the last event, causing another event to be triggered, thus preventing the violation.

The patch only interferes with runs starting with events $e_1, \ldots, e_n$; other runs remain unchanged. Formally:

Lemma 1 (The Locality Lemma)**.** Let P be a collection of b-threads, let p be a linear safety wait-block patch for event sequence $(e_1, \ldots, e_n)$, and let $P' = P \cup \{p\}$ denote the patched program. Then for any run ρ of P that *does not* start with events $e_1, \ldots, e_n$, the events of ρ constitute a valid run ρ' of P', and $\mathrm{Tr}(\rho) = \mathrm{Tr}(\rho')$.

Proof. To prove that ρ' is a valid run of P', we need to show that at each synchronization point during ρ', the triggered event is also enabled; namely, it is requested and not blocked. By definition, if a run does not start with events $(e_1, \ldots, e_n)$, then the patch never requests or blocks events. Further, the original b-threads will reach the same states during ρ' as they did during ρ, consequently requesting and blocking the same events. It follows that the program P' has the same requested and blocked events as P in each state during the run. Thus, the events triggered by ρ' are enabled, and the run is indeed valid.

Finally, since the original b-threads reach the same states during ρ' as during ρ, they will have the same atomic propositions associated with them. Since the patch has no atomic propositions associated with its states, we get that $\text{Tr}(\rho) = \text{Tr}(\rho')$. □

The Locality Lemma is our motivation for patching: it states (in this case, for linear patches) that when we add a patch to negate a single bad run, other runs remain unharmed, meaning that the patch is local. This is an advantage of our method as compared to traditional, manual, patching: our patches do not create new errors in unexpected parts of the code.

The distinct bad runs representing the bug or emanating from the new safety requirement are found by model checking:

Algorithm 1 Linear Safety Patching(P, Φ)

1: Run the model checker on (P, Φ)
2: **if** $P \models \Phi$ **then**
3:     **return** P
4: P' ← P
5: **for** each violating run $(e_1, \ldots, e_n)$ **do**
6:     **if** $\forall i,\ \ e_i \in E_{env}$ **then**
7:         **return Failure**
8:     **else**
9:         Find the largest k such that $e_k \in E_{sys}$
10:         Create a linear safety wait-block patch p for $(e_1, \ldots, e_k)$
11:         $P' \leftarrow P' \cup \{p\}$
12: **return** P'

The idea is straightforward: the model checker finds all runs violating Φ and we add a patch per run to prevent them. Because Φ is a safety property and P is loopless, there are only finitely many violating runs. The algorithm guarantees that the blocking performed by the patches creates no deadlocks, by first recursively marking as "bad" any state that has only "bad" children. Furthermore, because the model checker works with respect to all possible event selection mechanisms, any bugs that emerged after the patching are fixed. The Locality Lemma guarantees that no good runs "far away" from the patch are harmed. If the algorithm returns a patched program, we thus know that it satisfies the specification Φ and causes no deadlocks.

There is also the case where the algorithm returns a failure notice, as a result of the model checker returning a violating run in which there were no program-requested events. This, of course, means that the program cannot be repaired through wait-block patching. Formally:

Lemma 2 (The Patchability Lemma)**.** Let P be a loopless program with state graph $G = (V_G, E_G)$ and let Φ be a safety property. Then the following three statements are equivalent:

1. The algorithm succeeds in returning a patched program P'.

2. There exist linear safety wait-block patches $p_1, \ldots, p_k$, such that $P \cup \{p_i\} \models \Phi$.

3. There exists a graph $G' = (V_G, E_{G'})$ with $E_{G'} \subseteq E_G$ and $E_G - E_{G'} \subseteq E_{sys}$, such that no states violating Φ or causing deadlocks are reachable from the initial state in G'.

Proof. $(1) \Rightarrow (2)$ is trivial.

For $(2) \Rightarrow (3)$: Take the original state graph G, and for each p_i remove the edge corresponding to the event it blocks. Since the patched program satisfies Φ and does not deadlock, all reachable states in the graph obtained in this way satisfy Φ and do not cause deadlocks. Furthermore, by the definition of a wait-block patch, all edges removed are in E_{sys}, as needed.

For $(3) \Rightarrow (1)$: Without loss of generality, assume that P starts with an initialization event $e_{init} \in E_{sys}$. If this does not hold we can change to a new initial state q_0' and add a thread that forces event e_{init} to be chosen before proceeding to the original program.

Suppose that G' exists but that the algorithm returned a failure notice. We conclude that it deadlocked on the very first state, q_0'. This, in turn, means that state q_0 was marked as bad, so that all paths starting in q_0 lead to bad states. This contradicts the existence of G', thus proving the claim. □

Condition (3) means that the original program was "not too far" from satisfying Φ: it contained some good runs and some bad runs, and through some blocking the bad runs could be averted. Observe that the equivalence of (1) and (2) is really the validity of the algorithm.

The worst case runtime complexity of the algorithm is just that of the modified model checker, namely $T = T_{mc} + O(\Upsilon \cdot D^2)$. This shows the dependence of our algorithm on the number of violating runs in the original program. If their number and lengths are small enough our automatic patching is not much worse than regular model checking. This also demonstrates why using this algorithm for synthesis could be costly. If the program is "far away" from satisfying Φ, as could be the case when trying to synthesize a program from scratch (say, from a general program that constantly requests all possible events), then Υ could be polynomial in the size of the state graph, greatly slowing the process.

4.5.2 Patching for a Specific Event Selection Mechanism

The above algorithm patches the program so that it satisfies Φ, regardless of the event selection mechanism used. However, it may be useful to patch the program for the specific mechanism M to be used, as it could speed up the patching process, reduce the number of generated patches, and most importantly, block less events, leaving open more options for further behavior refinements and repair, as explained in Figure 4.2.

In this case, the model checking algorithm is modified to return as output all violating runs of the original program, as well as all (and only) violating runs that would be created

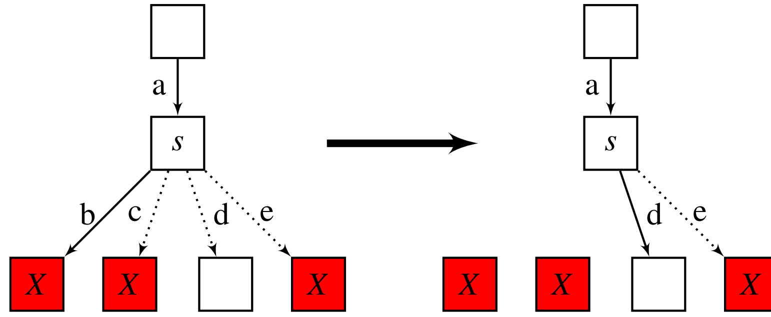


Figure 4.2: In state s, a patch that considers all event selection mechanisms will block b,c, and e. A patch that considers only, say, an ESM that chooses events alphabetically, needs to block b and c, but can leave e unblocked, relying on the selection of d.

by blocking previously discovered bad transitions. Bad runs that will not be possible in the patched program, under the specific ESM, are ignored. This technique is readily applicable also to patches for programs with cycles and for liveness patches, discussed in the sequel.

4.5.3 Example: Patching Tic-Tac-Toe

We demonstrate the use of the linear safety patching algorithm on the loopless Tic-Tac-Toe behavioral program from [86]. It is loopless since the fact that each step adds a new move to the board means that its state graph has no cycles.

Suppose that the original program is developed without a model checker. At the time of development, the programmer is convinced that the program always achieves its goal, i.e., never loses (observe that this is a safety property — bad things do not happen). Various testers support this statement. The program is then deployed. Some months later, a customer defeats it and sends in the game's trace. However, the original software engineer has long quit the firm, and it would take a long time for a new engineer to repair the code. A suitable solution would be to apply an automatic patching algorithm to the malfunctioning software.

To simulate this, we took the complete program from [86], and omitted the more complex threads — those that handle situations where our opponent creates, simultaneously, two ways to win. If the human player does not try the complex strategy that create such double attacks, the program does indeed seem to work, but a skilled player can defeat it.

The automatic proof-of-concept tool is easy to use, requiring little modifications to the original program. The input is the behavioral program and the safety property Φ, given as b-threads marking bad states (e.g., victory of the opponent). The output is code files for new thread instances which are easy to read and to integrate into the original program (see Figure 4.3).

Each such patch inherits from a parent class which implements its "main" function; see Figure 4.4.

In our example, the patched Tic-Tac-Toeprogram contains 26 different patches, one of

```
public patch1() {
  events.add( new X( 2, 2 ) );
  events.add( new O( 1, 1 ) );
  events.add( new X( 0, 0 ) );
  events.add( new O( 2, 0 ) );
}
```

Figure 4.3: Example of a wait-block patch generated by the proof-of-concept tool. The patch's code contains a sequence of events that should be waited-for — events X(2,2), O(1,1) and X(0,0). The last event in the list, O(2,0), is the one that should be blocked by the patch. The automatically generated code is legible and comprehensible, as the more complicated details are hidden away in a parent class.

```
public void runBThread() {
  for( int i = 0; i < events.size() - 1; i++ ) {
    bp.bSync( none, all, none );
    if ( !lastEventWas( events.get( i ) ) )
      disablePatch();
    }
    bp.bSync( none, all, events.getLast() );
    disablePatch();
}
```

Figure 4.4: The patch thread's main function, `runBThread()` is part of the patching library, and is not added to the actual patched program. It waits for events defined by a particular patch instance (as in Figure 4.3), blocking the last event and then terminating. If the events chosen deviate from those defined in the patch instance, it terminates.

which is demonstrated in the figure. Subsequent verification by the model checker confirms that now the specification is indeed satisfied.

4.6 Safety Patches for Programs with Cycles

4.6.1 Generating Safety Patches for Cycles

The correctness of the algorithms for linear safety patching relies on the program's state graph's having no cycles. As most reactive systems run indefinitely, periodically returning to some "idle" state, such systems cannot be patched by linear wait-block patches. For example, fixing a behavioral program that enters a bad state after a sequence of events of the form $(a)^{*}b$, will call for infinitely many linear patches.

Our solution is to extend the linear safety patch associated with a single sequence of events, into one that can keep track of an entire hierarchy of paths and cycles in the graph, blocking the violating event as needed.

Definition. *Given a state graph* $G' = (V_{G'}, E_{G'})$*, two special vertices marked* v_{init} *and* v_{end} *and an event* $e \in E_{sys}$*, a* cyclic safety wait-block patch *for* G' *is a b-thread with the following properties:*

- *It waits for all events chosen by the event selection mechanism and traverses the graph* G' *according to those events.*
- *Whenever state* v_{end} *is reached, it blocks event e once.*
- *If an event occurs such that there is no edge marked with that event, it terminates.*
- *It never requests events and does not label states.*

Intuitively, the patch is designed to prevent a family of bad runs that are similar to one another, in that they reach their bad state by transitioning from v_{end} via the event e. The graph G' will be chosen such that it contains *all* paths from v_{init} to v_{end}, thus rendering a single patch able to block that entire family of bad runs.

The Locality Lemma holds for the cyclic case as well: all runs of the original system, apart from those starting in v_{init} and ending in reaching the violating state through v_{end} and e, are valid runs of the patched system. The proof is based on the fact that in any such run, the generated patch does not request or block any events, and thus does not affect the events requested by the program.

Linear safety patches are a particular case of the cyclic ones, in which the graph G' is a path, meaning there is precisely one way to reach the violating state.

The cyclic safety patching algorithm is as follows (G denotes the full state graph traversed by the model checker):

Algorithm 2 Cyclic Safety Patching(P, Φ)

1: Run the model checker on (P, Φ)
2: **if** $P \models \Phi$ **then**
3:    **return** P
4: **for** each violating run $(e_1, \ldots, e_n)$ **do**
5:    **if** $\forall i,\ \ e_i \in E_{env}$ **then**
6:       **return Failure**
7:    **else**
8:       Find the largest k such that $e_k \in E_{sys}$
9:       Let q_{end} denote the state reached after events $e_1, \ldots, e_{k-1}$
10:       Construct the minimal subgraph G' containing all paths in G from q_0 to q_{end}
11:       Create a cyclic wait-block patch p for G' with states $v_{init} = q_0$, $v_{end} = q_{end}$, and event e_k.
12:       $P' \leftarrow P' \cup \{p\}$
13: **return** P'

Constructing the minimal subgraph G' is done using a modified BFS algorithm, in the following manner. Given the full graph and the two vertices q_0 and q_{end}, we run a modified BFS search from q_0. Unlike a regular BFS search, where each vertex stores a single predecessor (the first vertex from which it is found), here each vertex stores *all* the vertices from which it is found. When the search is over, we begin in q_{end} and backtrack through all possible predecessors of each vertex, until reaching q_0. The set of edges and vertices traversed this way forms the subgraph G' that we need.

To show that every path from q_0 to q_{end} is in G', let $p = (q_0, q_1, \ldots, q_n, q_{end})$ be a path. If p is simple, i.e., no state repeats itself, then clearly after $n+1$ iterations of the BFS search each vertex in p has its preceeding state marked as a predecessor. Therefore, the entire path will be traversed during the backtrack phase, meaning that p is in G'.

Now, suppose that p is a complex path with one cycle (the proof for the general case is an easy extension). Then p can be expressed as follows:

$$p = (q_0, q_1, \ldots, q_k, \underbrace{q'_1, q'_2, \ldots, q'_j, q_k}_{\text{the cycle}}, q_{k+1}, \ldots, q_n, q_{end})$$

The states before and after the cycle are found as before. The cycle's states, $q'_1, \ldots, q'_j$, are found at the latest during the j'th iteration after the first arrival at q_k. When the cycle ends, q'_j is marked as a predecessor of q_k. Therefore, during the backtrack phase that passes through q_k, the entire cycle will be found. Consequently, the returned subgraph contains p.

To see why G' is minimal, observe that if a state is added to the subgraph it is part of at least one path from q_0 to q_{end}, and therefore cannot be omitted from the graph.

Lemma 3. If the algorithm returns a patched program P', then $P' \models \Phi$.

Proof. Suppose that there exists a run ρ of P' violating Φ. Denote its states $q_1, \ldots, q_n$, and extract from them a violating run with no cycles. If $q_i = q_j$ for some $j > i$, delete states $q_{i+1}, \ldots, q_j$. Denote the remaining states as $q_{t_1}, \ldots, q_{t_k}$. The run corresponding to this state sequence was found by the model checker, and a patch for some subgraph G' which contains this run was created. Since G' contains all paths from q_1 to q_n, it also contains ρ. Therefore, the patch would have blocked the last program-requested event of ρ, causing a contradiction. □

As with the linear case, it is possible for the algorithm to return a failure notice. The Patchability Lemma, which characterized programs that could be fixed in the linear case, holds for the cyclic case as well; its proof is analogous.

The complexity of the algorithm is as follows: The exploration of violating runs costs, as before, $O(T_{mc} + \Upsilon \cdot D^2)$. Constructing the relevant subgraph for each violating run costs another $|V_G| + |E_G|$ times Υ runs, yielding:

$$T = O\left(T_{mc} + \Upsilon \cdot D^2 + \Upsilon(|V_G| + |E_G|)\right).$$

Again, this shows our dependence on the number of violating runs, Υ. The smaller that number, the closer our complexity is to that of the model checker; the higher it is, the closer we are to the notorious, worst-case complexity of the synthesis problem.

4.6.2 Subgraph Representation

The generated code for a linear safety patch contains only the list of events to be waited for, followed by the event to be blocked. This list can be readily understood and possibly manipulated by a human, say, for documentation or analysis. Further, the developer may simplify or generalize the patch; e.g., skip waiting for certain guaranteed events or consolidate patches into fewer "symbolic" one, using BPJ's event filters. However, when a patch traverses a complex subgraph, gaining such insights is harder. Thus, we propose to represent the subgraph as a collection of easily readable linear event scenarios, amenable to human manipulation. The operation of the cyclic safety patch will be as before.

Specifically, We use the term *line* for a finite sequence of events that occur along some contiguous path in the state graph, and along which no state is visited twice. We use the term *tail* for a line whose last event would lead to a bad state in the state graph. The program's state graph, or parts thereof, are stored as a collection of lines, each containing its sequence of

events, and links to other lines that are reachable by a single event from the last event in the line. See Figure 4.5.

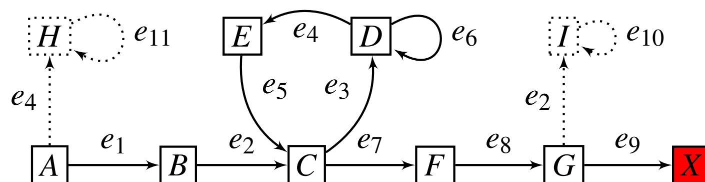


Figure 4.5: A state graph of a buggy program. The model checker returns the violating run with events e_1, e_2, e_7, e_8, e_9. The subgraph of all paths from state A to state G (see solid states and edges) is decomposed into: $line_1 = e_1, e_2$ (successors $tail, line_2$); $line_2 = e_3$ (successors $line_3$, $line_4$); The self-loop $line_3 = e_6$ (successors $line3, line_4$); $line_4 = e_4, e_5$ (successors $line_2$, $tail$) ; $tail = e_7, e_8$ (with event to be blocked, e_9). In addition to the run found by the model checker, the patch prevents other runs, e.g., $e_1, e_2, \underline{e_3, e_6, e_6, e_4, e_5, e_3, e_4, e_5}, e_7, e_8, e_9$, where the underlined events correspond to cycles.

Thus, each patch,

- begins by activating lines containing the initial state;
- waits for all events and traverses active lines;
- deactivates active lines when they are deviated from;
- deactivates a line and activates its successors when the line's last event occurs;
- in a tail, prior to the event leading to the bad state, blocks that event, waits for one more event, and deactivates the tail.

The line representation can be implemented in a data structure or in separate patch b-threads, each beginning with waiting for a unique activation event. This results in a number of small patches and is readily implementable in all implementations of behavioral programming.

4.6.3 Example: Patching a Coffee Machine

We demonstrate cyclic safety patching with a simple coffee vending machine example, which is expected to repeatedly wait for a coin, wait for a coffee request, and prepare the coffee. The main requirement is that coffee is never prepared unless a coin is first inserted. However, if immediately after power-up the user requests coffee, the machine incorrectly allows coffee to be requested and prepared infinitely many times without a coin. When the first coin is inserted, the machine enters normal operation. The machine's state graph is depicted in Figure 4.6.

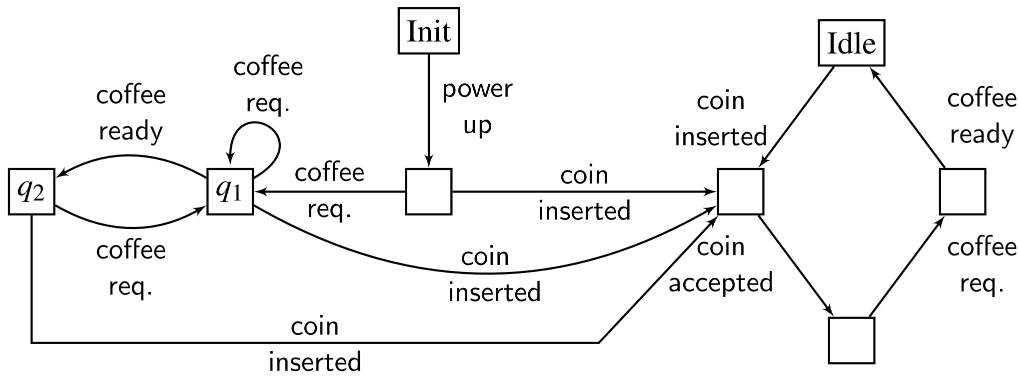


Figure 4.6: The buggy coffee machine's state graph. After the PowerUp event, if a CoffeeRequested event occurs (before a coin is inserted), free coffee can be obtained infinitely many times, until a coin is inserted. The loop on the right-hand side of the graph represents the desired operation. The problematic state (marked q_1) has two enabled events: CoffeeReady, which is immediately requested (and selected), and the environment event CoffeeRequested. We expect the patch to block the CoffeeReady event.

Observe that the bug is a safety bug — coffee is served without first inserting a coin. When it is discovered and automatic patching is attempted, the first step is to have a new b-thread identify and mark bad states (namely, q_2).

The automatic patching algorithm generates a single patch, corresponding to the subgraph depicted in Figure 4.7.

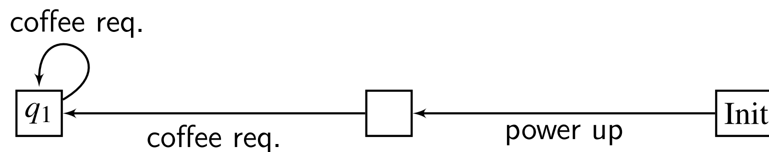


Figure 4.7: The subgraph of the program's state graph for which a patch is created. It shows all paths from the graph's initial state to state q_1, in which event CoffeeReady must be blocked to prevent violations.

Finally, the graph of the patched program is depicted in Figure 4.8, and the code generated by the proof-of-concept tool is shown in Figure 4.9.

4.7 Dealing with Liveness

Up to this point, we dealt with safety properties — those that assert that "nothing bad happens". Another important class of properties is those involving liveness, asserting that "good things eventually happen". In this section we show how wait-block patches can be applied in order to fix liveness violations too.

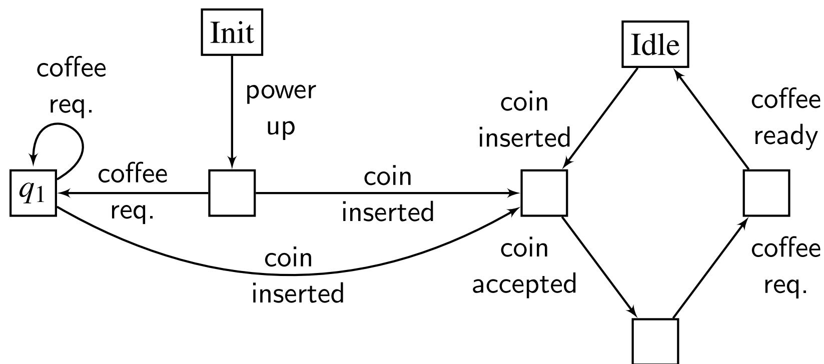


Figure 4.8: The patched program's state graph (states of the patches themselves are omitted for clarity). The violating CoffeeReady event has been blocked, and the bad state no longer exists in the state graph.

```
1  public cyclicPatch1() {
2    line1Events.add( new PowerUp() );
3    line1Events.add( new CoffeeRequested() );
4    line1 = new LineComponent( line1Events );
5
6    line2Events.add( new CoffeeRequested() );
7    line2 = new LineComponent( line2Events );
8
9    tailEvents.add( new CoffeeReady() );
10   tail = new TailComponent( tailEvents );
11
12   line1.addSuccessor( tail );
13   line1.addSuccessor( line2 );
14   line2.addSuccessor( line2 );
15   line2.addSuccessor( tail );
16
17   this.addActiveComponent( line1 );
18 }
```

Figure 4.9: The automatically-generated Java code for representation of the subgraph in Figure 4.7. The first line contains events PowerUp and CoffeeRequested, and the second line contains CoffeeRequested. The tail contains only the event to be blocked, CoffeeReady. The code is readily understandable.

In the case of safety properties, ensuring that a property holds is reducible to rendering all "bad" states unreachable, and so it was straightforward to use blocking in order to correct malfunctioning programs. Recall that in Section 4.3 we mentioned that liveness violations correspond to reachable cycles of "hot" states (i.e., "hot cycles") in the program's state graph, and so it is less clear how to apply blocking. One natural approach might be to identify when the system is traversing a hot cycle, and then block one of the cycle's transitions (when an alternative exists), forcing the run to leave the cycle. This has several drawbacks:

1. Unlike in the safety case, where a bad state was never to be visited, in the liveness case it is legal to traverse the hot cycle any finite number of times. Consequently, safety-like patching would destroy good runs, which is highly undesirable.

2. Naïvely forcing the run to leave a hot cycle does not guarantee that it reaches a cold state; it could enter another hot cycle.

3. We would need to keep track of the hot cycles in the graph — the number of which could be very large.

To overcome these difficulties, we adopt a different perspective. Instead of considering runs and the hot cycles they traverse, we consider the hot states themselves. We show how, using wait-block patches, one can enforce a state-based policy that forces every run to visit cold states infinitely often, thus ensuring that the liveness property in question holds.

Our technique works by distinguishing between two types of hot states: *hot-trap* states and *hot-escapable* ones. Hot-trap states have the property that once they are visited, a liveness violation cannot be prevented; i.e., the system can never force the run into a cold state again. Consequently, hot-trap states are considered as "bad" states, and we use safety wait-block patches to render them unreachable. The hot-escapable states are those from which the system could force the run to visit a cold state, via some transitions; however, we cannot assume that these transitions may ever be traversed. In particular, it is possible for the system to continuously choose transitions that keep the run in hot states, although transitions to cold states are always enabled. We handle the hot-escapable states by enforcing *fairness*: we make sure that if a transition is enabled infinitely often, it will eventually be traversed. This type of fairness can be enforced using *probabilistic wait-block patches*, which we also call *liveness patches*. Through their use we can ensure that any liveness violations are effectively eliminated.

In the remainder of the section we discuss the liveness patching process more thoroughly.

4.7.1 Classifying Hot States

The first step in our repair algorithm is partitioning the hot states in the state graph into the two types mentioned. These two sets are formally defined by the algorithm below, which takes

as an input the state graph of the program $G = (V_G, E_G)$, and returns the sets of hot-escapable and hot-trap states. For each hot-escapable state the algorithm also outputs its *escape-distance*, denoted δ: this is the length of a path from the hot-escapable state that reaches a cold state, and which the system can enforce regardless of the environment's behavior. See Figure 4.10 for an illustration.

Algorithm 3 Classify Hot States(V_G, E_G)

1: $A \leftarrow ColdStates(V_G)$, $B \leftarrow HotStates(V_G)$, $iteration \leftarrow 1$
2: $continue \leftarrow true$
3: **while** $continue$ **do**
4: $continue \leftarrow false$, $New \leftarrow \phi$
5: **for** each state $s \in B$ **do**
6: **if** at least one outgoing edge (internal or external) from s leads to a state in A, and no outgoing external edge from s leads to a state in B **then**
7: $continue \leftarrow true$
8: $New \leftarrow New \cup \{s\}$
9: $\delta(s) \leftarrow iteration$
10: $iteration{+}{+}$
11: $B \leftarrow B - New$, $A \leftarrow A \cup New$
12: $HotEscapable \leftarrow A \cap HotStates(V_G)$
13: $HotTrap \leftarrow B$
14: **return** $(HotEscapable, HotTrap)$

The algorithm performs a fixpoint computation of the set A of states that are either cold or from which the system can force the execution to reach a cold state. When the algorithm terminates, this set contains the hot-escapable states.

The key point in the algorithm is line 6, which contains the condition based on which a new hot state enters A. For a state to become hot-escapable, all its external edges must lead into A, which expresses the fact that these events are beyond our control, and are controlled by the environment. Since we cannot prevent (block) them, we require that they cause no problem in the first place — namely, that they lead to states that have already been classified as hot-escapable by their being in the set A. Another condition, which handles the case where a state only has internal events enabled, is that there be at least one edge going into a state of A. The key fact is that if either condition holds, the blocking idiom can be applied to block all edges that do not lead to A.

The *escape-distance* value, δ, of a hot-escapable state indicates the number of the iteration in which it joined A. It measures the shortest guaranteeable distance to a cold state — that is, the length of the shortest such path that can be enforced by blocking.

Observe that while this algorithm serves to define hot-escapable and hot-trap states, it is not efficient — primarily because of the loop in line 5. By considering, at every iteration, only

nodes that have successors that were determined hot-escapable in the previous iteration, the run time complexity can be reduced to $O(|V_G| + |E_G|)$.

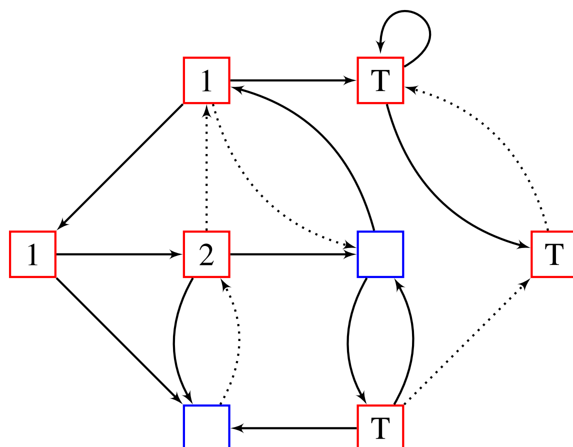


Figure 4.10: Hot-trap and hot-escapable states. Hot states are marked red and contain either a number or the letter T; cold states are marked blue and are empty. Solid edges correspond to internal events, and dotted edges correspond to external events. A number inside a hot state designates the state as hot-escapable and indicates the escape-distance. The letter T designates the state as hot-trap.

4.7.2 Handling Hot-Trap States

As discussed previously, once the run enters a hot-trap state the system cannot guarantee that it ever reaches a cold state. Consequently, we are forced to block that entrance in the first place. This is done by applying safety wait-block patches, using the technique discussed in Section 4.6, which renders all hot-trap states unreachable.

Observe that this may remove potentially good runs too — namely, runs that go through a hot trap state yet still visit a cold state eventually. This can happen, for example, when an external event leading to another hot trap state is not triggered and, instead, an internal event that leads to a cold state is triggered. However, since we cannot depend on external events being triggered or not, our only way to ensure that no violations occur is to make hot-trap states unreachable.

4.7.3 Hot-Escapable States and Transition Fairness

The criterion used in determining the set of hot-escapable states ensures that careful use of the blocking idiom can force the run from a hot-escapable state into a cold state. The actual technique we propose is aimed at harming as few good runs as possible, and is based on *fairness*.

The notion of *fairness assumptions* [121] is used widely in formal verification, typically in order to rule out violating runs of the system because they are not realistic. Here, we discuss a special kind of fairness, called *transition fairness* [18]: if a transition is enabled infinitely often (i.e., its state of origin is visited infinitely often), then it is traversed infinitely often. We also allow a set of transitions originating from the same state to form a single constraint: if the state is visited infinitely often, then at least one of the transitions in the set is traversed infinitely often. Note that, unlike in the traditional setting where fairness is *assumed* for verification purposes, here we aim to *enforce* it within a malfunctioning system.

Intuitively, hot-escapable states have the property that if the event selection mechanism were to choose the triggered events uniformly at random, a run that visits them would eventually lead to cold states. It turns out that one can also settle for assumptions that are weaker than truly random event selection. We express these required assumptions as transition fairness constraints, and then discuss how to enforce them. Formally:

Definition. *Let P be a behavioral program with state graph G. A* transition fairness constraint *c on G is a set of one or more transitions (edges) in the graph, $\{e_1, \ldots, e_n\}$, all originating from the same node v. We say that P* satisfies *c, denoted $P \models c$, if it has the following property: if a run ρ of P visits v infinitely often, transitions from c are traversed infinitely often.*

Let $C = \{c_1, c_2, \ldots, c_k\}$ be a set of transition fairness constraints. We say that P satisfies *C, denoted $P \models C$, if $\forall_{1 \leq i \leq k} P \models c_i$.*

We now define a set of specific transition constraints for each of the hot-escapable states in the graph, and then show that they suffice for guaranteeing the liveness property in question.

Definition. *Let P be a behavioral program with state graph $G = (V_G, E_G)$, and let $V_{hot-escapable} \subseteq V_G$ be its set of hot-escapable states with respect to some liveness property Φ. For each $v \in V_{hot-escapable}$, the* transition fairness constraint of *v, $\tau(v)$, is defined as follows:*

- *if v has transitions corresponding to external events, $\tau(v)$ is the set of these transitions.*
- *otherwise, v has a neighbor, u, such that u is a cold state or $\delta(u) < \delta(v)$. In this case, we define $\tau(v)$ to be the edge leading from v to u.*

Observe that for every hot-escapable state v, $\tau(v)$ can be found during the hot state classification algorithm at no additional cost. We define the set of transition fairness constraints of the entire program to be the set of transitions fairness constraints on all its hot-escapable states, namely $\tau(P) = \bigcup_{v \in V_{hot-escapable}} \tau(v)$. The following proposition justifies our choice of constraints:

Lemma 4. Let P be a behavioral program and let Φ be a liveness property. If P has no hot-trap states with respect to Φ and $P \models \tau(P)$, then $P \models \Phi$.

Proof. Suppose, towards contradiction, that $P \not\models \Phi$. Then there exists a run ρ of P and a hot state $v_0 \in V_{hot}$, such that v_0 appears infinitely often in ρ. Since P has no hot-trap states, v_0 is hot-escapable.

By our assumption that the constraints of $\tau(P)$ hold, there exists a neighbor of v_0, denoted v_1, that also appears infinitely often in ρ, and this v_1 is either a cold state or a hot-escapable state with $\delta(v_1) < \delta(v_0)$. If the former holds, then $\rho \models \Phi$ and we are done. If the latter holds, we reapply the same logic iteratively. Clearly, this produces a chain of hot-escapable states $v_0, v_1, \ldots, v_n$, all appearing infinitely often in ρ, with $\delta(v_0) > \delta(v_1) > \ldots > \delta(v_n)$. Since $\delta(v_0)$ is finite, this process ends in visiting a cold state infinitely often, again implying that $\rho \models \Phi$. □

Note that the lemma assumes that P has no hot-trap states. However, this is not a real limitation, since, as previously explained, we can first apply safety patching to make such states unreachable.

4.7.4 Liveness Patches

We have characterized fairness constraints that are sufficient for correcting the liveness violation. Unfortunately, behavioral programs are not guaranteed to be fair. This is an intrinsic property of the event selection mechanisms commonly used in behavioral prgramming. For example, in arbitrary or priority-based selection certain transitions might be enabled infinitely often but never triggered. Consequently, we introduce a new type of patch, termed a *liveness wait-block patch*, aimed at enforcing a transition fairness constraint on the program.

Definition. *Given a state graph $G = (V_G, E_G)$, a probability η, a hot-escapable state $v \in V$ and its transition fairness constraint $\tau(v)$, a* liveness wait-block patch *for v is a b-thread with the following properties:*

- *It waits for all events chosen by the event selection mechanism and traverses the graph G according to them.*
- *It keeps track of the present state and notes when the execution reaches v.*
- *Whenever in v, with probability $1 - \eta$ the patch does nothing. With probability η, it blocks all transitions except those in $\tau(v)$.*
- *It does not request events and does not label states.*

Intuitively, liveness wait-block patches are a way of incrementally injecting fairness into specific states of an already existing program, without modifying existing code. When the patch is applied to a hot-escapable state, it enforces the fairness constraint of that state; in runs in which the state is visited infinitely often, at least one of the transitions specified by

the constraint will be triggered infinitely often. Indeed, the probability that edges in $\tau(v)$ are not traversed after after m visits to v approaches 0 as m tends to infinity, and this is true even for very small values of η. Note that, despite their probabilistic nature, these are essentially wait-block patches: they wait for a sequence of events, and then apply blocking to steer the run in the right direction.

Observe that it is indeed always possible to block all the transitions except those in $\tau(v)$. The only events that cannot be blocked are the external ones; and if there are external transitions in v, they are all in $\tau(v)$ by definition. Further, observe that by their definition liveness patches cannot cause deadlocks in states that were deadlock-free before the patching — as the patch always leaves unblocked at least one event that was already enabled.

Our motivation for using probability-based blocking is the desire to leave good runs unaffected. Choosing η to be small still guarantees that the fairness constraint holds, but makes it likely that runs that scarcely visit the state remain unaffected.

As in the case of cyclic safety patches (Section 4.6.2), liveness patches can be represented as a collection of *lines* and *tails* to make them more comprehensible.

We point out that so far we have dealt strictly with deterministic behavioral programs. Our probabilistic liveness patches, however, introduce nondeterminism into the system. This nondeterminism is not “against the grain” of behavioral programming, and indeed, extending behavioral programming definitions to support nondeterminism is straightforward, and is omitted.

4.7.5 The Liveness Patching Algorithm

Based on the discussion in the previous sections, we now present the patching algorithm itself:

Observe that, as in the safety patching algorithm of Section 4.6, it may be impossible to create safety patches for hot-trap states in certain cases. One extreme example is when the entire state graph consists of hot-trap states only, so that attempting to render these states unreachable produces a trivial program that deadlocks in its initial state. In such cases, the algorithm returns a failure notice.

The correctness of the algorithm is established by the following lemma:

Lemma 5. Let P' be a patched program returned by the algorithm, and let ρ be a run of P'. Then with probability 1, $\rho \vDash \Phi$.

Proof. By the previously proved correctness of safety patching (Lemma 3), the algorithm ensures that there are no reachable hot-trap states in P'. By Lemma 4, it suffices to show that P' satisfies the constraints in $\tau(P)$ with probability 1. This claim is immediately derived from the definition of a liveness wait-block patch (Definition 4.7.4) and the discussion following it. □

Algorithm 4 Liveness Patching(P, Φ)

1: $P' \leftarrow P$
2: Run the model checker on (P, Φ)
3: **if** $P \models \Phi$ **then**
4: **return** P
5: Run *Classify Hot States* on the state graph
6: **for** each hot state $v_h \in V$ **do**
7: **if** v_h is a hot-trap **then**
8: **if** creating a safety patch to prevent runs from reaching v_h is impossible **then**
9: **return Failure**
10: Create a safety patch $p^s_{v_h}$ that prevents runs from reaching v_h
11: $P' \leftarrow P' \cup \{p^s_{v_h}\}$
12: **else**
13: Create a liveness patch $p^\ell_{v_h}$ for v_h
14: $P' \leftarrow P' \cup \{p^\ell_{v_h}\}$
15: **return** P'

Part of our motivation for using wait-block patches in repairing violated safety properties was the Locality Lemma, which stated that any good runs remain unchanged. Unfortunately, that lemma cannot be proved for the liveness case; in fact, any liveness wait-block patch, by definition, might affect good runs as well as bad ones. We settle for the following:

Lemma 6 (The Weak Locality Lemma (Liveness))**.** Let P be a collection of b-threads, let p be a liveness wait-block patch for hot-escapable state s_h, and let P' denote the patched program $P \cup \{p\}$. Any run ρ of P that *does not* reach s_h constitutes a valid run ρ' of P', and $\text{Tr}(\rho) = \text{Tr}(\rho')$.

The proof is similar to that of the safety case and is omitted. The result is weaker, in the sense that if s_h is hot-escapable then good runs that pass through it might, with some probability, become invalid in the patched program. That probability increases the more times they pass through s_h. However, the effect on good runs can be reduced by decreasing the patches' blocking probability η.

The complexity of the liveness patching algorithm is as follows. The model checking phase costs $O(T_{mc})$. Classifying the hot states is linear in the size of the state graph. Each hot-trap state is then handled as a safety violation, adding $O(|V_{hot-trap}| \cdot (D^2 + |V_G| + |E_G|))$. Finally, for every hot-escapable state, we must construct the sub-graph needed to check when it is visited, yielding another $O(|V_{hot-escapable}| \cdot (|V_G| + |E_G|))$. Combining these, the total worst-case complexity becomes:

$$T = O\big(T_{mc} + (|V_{hot}| + 1) \cdot (|V_G| + |E_G|) + |V_{hot-trap}| \cdot D^2\big).$$

The runtime complexity shows the algorithm's dependence on the number of hot states: the smaller it is, the closer our complexity is to that of regular model checking. In the next section we discuss a heuristic-based approach for reducing this in practice.

4.7.6 Minimal Fairness Enforcement

In the algorithm just discussed, we first rendered all the hot-trap states in the state graph unreachable and then enforced a set of transition fairness constraints in the hot-escapable states. By Lemma 5, we are guaranteed that this repairs any liveness violations in the program. However, the number of enforced fairness constraints is rather large — it is approximately the number of hot-escapable states in the program. Despite this, in some cases one can settle for fairness constraints that are far less extensive. For instance, consider the graph in Figure 4.11.

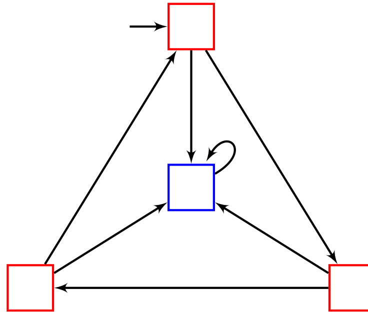


Figure 4.11: An instance where the liveness repair algorithm would enforce more fairness than is required. The graph above has three hot-escapable states, and the algorithm would enforce transition fairness on the three edges leading from them into the cold state. Clearly, it suffices to settle for just one of these three constraints in order to guarantee that the cold state is eventually reached. Similar, larger constructions show that our algorithm might enforce any number of fairness constraints where just one would suffice.

Decreasing the number of fairness constraints being enforced is highly desirable, for two reasons. First, as we mentioned earlier, we wish to perform as few modifications to the original program as possible, and enforcing fewer constraints clearly serves this goal. Second, the size of the automatically generated code module is in correlation with the number of constraints that it enforces. Hence, fewer constraints means shorter modules, which are easier to maintain.

A natural question thus arises: can one identify a minimal-size set of fairness constraints that need be enforced on a given behavioral program in order to ensure that a given liveness property holds? Formally, we define the minimal fairness problem MF_{opt}, as follows: Given a behavioral program P with state graph $G = (V_G, E_G)$ and a liveness property Φ, such that G has no hot-trap states with respect to Φ, find a minimal-size set C of transition fairness constraints such that $P \models C \Rightarrow P \models \Phi$.

Unfortunately, it turns out that this problem is NP-complete. In [47], the authors study the

problem of synthesis in the face of incomplete knowledge about the system's environment. In particular, they show that the problem of finding a minimal fairness assumption on the environment in order to make a given specification realizable is NP-complete. It is straightforward to show that this problem is reducible to MF_{opt} and that MF_{opt} is in NP, rendering it NP-complete.

Given this fact, we propose a greedy algorithm for approximating MF_{opt} in practice. The algorithm starts with an empty constraint set, and adds new constraints iteratively, in a manner similar to the way algorithm *Classify Hot States* finds the set of hot-escapable states.

Throughout its iterations, the algorithm maintains a growing set of already "handled" hot-escapable states. This is the set of states for which enforcing the current set of fairness constraints guarantees that they partake in no liveness violations. In other words, a run that visits any of these states infinitely often will reach a cold state infinitely often too.

The set of handled states is increased in each iteration. There are two ways for a state v to become handled:

1. By direct fairness enforcement: this happens when the algorithm chooses to enforce a fairness constraint leading from v into the set of already handled states.

2. By indirect domination: if, due to previous fairness constraints, all of v's successors are already handled, then v itself can be immediately marked as handled.

At each iteration, the algorithm imposes one fairness constraint, meaning that precisely one vertex becomes handled through method 1. Our criteria in choosing this particular vertex is trying to maximize the number of states that will become handled through method 2. The actual choice is performed by looking at all the candidates, namely nodes that can become handled through the enforcement of a single constraint. Each candidate is then assigned a value, which is its number of hot-escapable predecessors that are not yet handled (observe that these predecessors are precisely the vertices with potential to become dominated by choosing this vertex). Finally, the highest valued candidate is selected, and the corresponding fairness constraint is enforced. Here is pseudo-code outline of the algorithm:

Algorithm 5 Approximate $MF_{opt}(V,E)$

1: $A \leftarrow HotStates(V)$, $handled \leftarrow \phi$, $constraints \leftarrow \phi$
2: **while** $A \neq \phi$ **do**
3:     $candidates \leftarrow$ `FindAllCandidates()`
4:     $max \leftarrow$ `MaxValuedCandidate()`
5:     Add a constraint that handles max to $constraints$
6:     Move max to from A to $handled$
7:     **while** there are nodes in A dominated by $handled$ **do**
8:         Move dominated nodes from A to $handled$
9: **return** $constraints$

The two subroutines, `FindAllCandidates` and `MaxValuedCandidate`, are omitted. As with *Classify Hot States*, an efficient implementation of the algorithm and its subroutines runs in time that is linear in the size of the program's state graph.

4.7.7 Example: Liveness Patching for the Dining Philosophers

We implemented our liveness patching algorithm (including the greedy approximation algorithm) within our proof-of-concept tool. For evaluation, we used the dining philosophers problem [63]. A behavioral implementation thereof includes the events of a philosopher picking up and putting down a given fork, a b-thread for the behavior of each philosopher and a b-thread for each fork. Each philosopher's b-thread is subject to a strict event sequence: pick up one fork, pick up the other, put down one fork, put down the other. Each fork's b-thread waits for events that change its state, and blocks illegal events (e.g., a second picking up, or, a putting down by the "wrong" philosopher). In [86] we model checked this problem and variations thereof for safety and liveness properties.

For our experiment, we used a variant where the first $n-1$ philosophers are left-handed and the last one is right handed, which prevents deadlocks. All events in the program are internal, and so no hot-trap states exist. Finally, the liveness property used was this: "Philosopher #1 eats infinitely often". The results are shown in Table 4.1.

Table 4.1: Comparing the results of the naïve repair algorithm and the greedy approximation repair algorithm for the dining philosophers problem, with 9 - 12 philosophers. The *States* column shows the total number of states in the program. The *Patches (Naïve)* and *Patches (Greedy)* columns show the number of patches generated by the naïve and greedy algorithms, respectively. Observe that since the naïve algorithm generates one fairness constraint per hot-escapable state, the *Patches (Naïve)* column reflects the number of hot-escapable states as well. Finally, the *Reduction* column shows the percentage of patches saved by using the greedy version.

#Philosophers	**#States**	**#Patches (Naïve)**	**#Patches (Greedy)**	**Reduction**
9 Philosophers	19682	17495	9913	43%
10 Philosophers	59048	52487	30760	41%
11 Philosophers	177146	157463	93989	40%
12 Philosophers	531440	472391	287283	39%

Each fairness constraint is translated into the actual code that enforces it, using the same mechanism as for safety patches. Although there may be many patches (as the example demonstrates), each of them is fairly comprehensible. The possibly high number of patches was part of our motivation for using the greedy algorithm; coming up with better algorithms to further reduce this number remains a topic for future work.

4.8 Limited-Depth Repair

4.8.1 Automatic Repair from Field Error Reports

Many facilities exist for end-users to send reports of software failures to the software vendor (see, e.g., Figure 4.12). Typically, these reports correspond to violated safety properties (e.g., "the system never crashes").

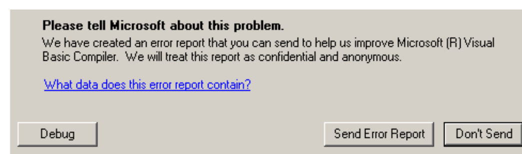


Figure 4.12: Event logs from bug reports are used in patch construction.

For behavioral programs, we propose a methodology for using such failure reports in order to cope with the state-explosion problem inherent to model checking, and to patch programs with many violating runs:

- The failure report contains an event log.
- Using the fact that the effect of a patch is local, we constrain the model checking depth to a neighborhood of the path of the failure (the bad run), followed by a limited fan-out of possible continuations, past the blocked transition.
- This is enforced by a dedicated b-thread, which monitors all events, and when an event occurs that is not along the reported bad path, it starts counting the distance from the bug report. When the distance is greater than a given parameter, the b-thread calls a model checker API to prune the search.
- Finally, the safety patch is generated as above.

Such patching prevents the failure reported by the end-user, along with any other failures "not far" from it, and can help when full model checking and patching consumes too much resources. As with bounded model checking techniques [32], the search-depth parameter is key and needs to be adjusted per repaired program; higher depth means repairing more violations, but poorer performances. It is up to the user to use knowledge of the program's state graph, or run tests, in order to come up with the best choice.

4.8.2 Example: Limited-Depth repair of the Dining Philosophers

Again consider the dining philosophers problem [63], this time where all philosophers are left handed. The reported bug fixed is the classical deadlock where all philosophers pick up the

fork on their left. Table 4.2 shows the results of patching for the single bad run that we gave the patcher.

Table 4.2: Patching the dining philosophers problem using bounded depth patching. Receiving a bug report (e.g., each philosopher picked up a single fork), the algorithm searches for event sequences that deviate from, or continue, the event trace in the bug report by no more events than the search depth parameter. The patches handle cycles discovered within the search depth (e.g., one of the philosophers completing a full cycle of picking up and putting down her two forks, while the others do not proceed). The tests were carried out on a PC with a Intel Quad Core Q6600 CPU @ 2.40GHz.

Search Depth	3 Philosophers	6 Philosophers	9 Philosophers
3	3 patches 3 loops 0.5 seconds	1 patch 2 loops 4.2 seconds	1 patch 2 loops 30 seconds
4	15 patches 30 loops 1.2 seconds	2 patches 4 loops 22 seconds	3 patches 6 loops 4.5 minutes
5	20 patches 380 loops 3.2 seconds	12 patches 1200 loops 2 minutes	12 patches 2580 loops 45 minutes

4.9 Related Work

The research in [143, 142, 104] presents fault localization and automatic repair of programs, where a set of software components that are suspected to cause a fault is replaced by a set of synthesized components, such that the resulting system is guaranteed to meet the full specification. Automatic repair of concurrency bugs (e.g., accessed to shared memory), is presented in [103]. The detection mechanism uses bad runs associated with bug reports, and the analysis involves actual execution. The repair is manifested in modification to existing code. Genetic-programming-based repair of legacy C programs is demonstrated in [150]. The repair relies on changes to existing code in order to correct problems that were assumed to be local in nature. In [21], genetic-programming is combined with co-evolution of the test cases against which the program is evaluated. Naturally, any work on automatic-repair would be considered a particular case of program synthesis [134, 36].

As mentioned in Section 4.7.6, our work on repairing liveness violations relates to that of [47], where the authors show how to synthesize fairness assumptions the environment must uphold in order for a specification to be realizable. Our work tackles similar difficulties, but in a different setting: the environment is fixed, and fairness constraints on the system are synthesized. Our repair algorithm then imposes those constraints on the previously designed system.

As for other approaches for coordinating simultaneous behaviors, such as Esterel, BIP or Linda (see related work in [90, 89] for a comparison of behavioral programming with these approaches), we believe that comparable localized repair mechanisms would be possible. The key would be implementing the equivalent of blocking which, combined with ability to subscribe to all events, is central to our solution. This, of course, is possible, as it was in Java and Erlang, and could also benefit other aspects of incremental development in these environments.

4.10 Conclusion and Next Steps

The contribution of this chapter is in the proposed automated approach, in which faulty components are neither identified nor modified. Instead, the system is non-intrusively augmented with additional components, to yield desired overall system behaviors. The entire approach is made possible by the incrementality and modularity of behavioral programs. The new components are readily understandable by humans, and can be documented, enhanced, or generalized as part of standard development. The generated patches can then be distributed to users without re-distributing the original software. Finally, contributing to the on-going and up-hill battle with state explosion, we propose a methodology and a practical technique for constructing local patches using limited-depth model checking.

This research is a step in the direction of developing methodologies and tools for the repair of behavioral programs. An important next step is to enrich the tool with interactive capabilities, allowing the developer to examine the state graph and enhance the proposed repairs: consolidating similar patches, generalizing or constraining patch functionality, or perhaps changing existing code after all.

Future research problems include repairing the program with regard to time-related properties, as well as integration with other formal methods tools and techniques, including other synthesis algorithms, symbolic model checking, and compositional verification. Our tool could be combined with Java Pathfinder [148] or other tools to explore support of richer inter-process communication beyond solely behavioral events, and possibly solving concurrency problems among b-threads, as in [103].

We hope that with further developments in incremental, non-intrusive development, supported by powerful repair automation, the task of software maintenance may eventually shed its present (often lackluster) image, becoming a rewarding undertaking, allowing software engineers to quickly address customer needs in a productive, satisfying manner.

Chapter 5

Module-Based Abstraction and Repair of *RWB* Programs

5.1 Introduction

Explicit model checking algorithms operate by spanning a program's state graph and comparing it to a given specification. This method becomes infeasible for large systems, as the state graphs tend to grow exponentially in the size of the program (the *state explosion* problem). Abstraction techniques [52] are among the most important methods for coping with state explosion and increasing the scalability of model checking algorithms.

The key idea underlying abstraction techniques is to replace the concrete system model (i.e., the program's state graph) with a smaller abstraction thereof. Typically, the abstraction constitutes an *over-approximation* — it includes the behaviors of the concrete system, and may also include other behaviors. In the case of model checking, proving that a given property holds for the abstract model implies that it holds for the concrete model as well. Since the abstract model is more succinct, the state explosion problem is hopefully mitigated.

We study the application of abstraction techniques to the BP framework, and to $\mathcal{RWB}$ models in general. Recall that in BP, programs consist of *behavioral threads* — threads of code that run in parallel, each designed to affect a specific behavior of the system. In the first part of our work, we present a formulation of BP's semantics that supports the notion of *modules*, which are logically related threads grouped together, and discuss abstracting these modules. We then demonstrate how the composition of module abstractions yields an over-approximation of the entire behavioral program.

In the second part of this chapter we discuss model checking abstract behavioral programs, and propose a *counterexample guided abstraction refinement* (*CEGAR*) [51] scheme for BP. When model checking over-approximations, counterexamples found by the model checker may prove *spurious*, i.e. nonexistent in the concrete system. In CEGAR, one validates each coun-

terexample against the concrete system and, if it is spurious, refines the abstract model in a way that eliminates it. The process is then repeated iteratively until the property is proven or a genuine counterexample is found. Based on our module-based abstraction of behavioral programs, we propose a two layer abstraction-refinement scheme, similar to that of [49], in which spurious counterexamples of the composed system are used to refine module abstractions. In our setting, module interdependencies make it impossible to resolve spurious counterexamples by examining modules individually; our algorithm compensates by considering these interdependencies and refining multiple modules simultaneously when needed.

In the third part of the chapter, we combine our abstraction techniques with a program repair algorithm. In Chapter 4 we demonstrated how safety violations can be eliminated from behavioral programs by adding separate, non-intrusive behavioral threads to the program. Since that repair technique included spanning the program's concrete state graph, it was susceptible to the state explosion problem. Here, we modify the technique to work on abstract state graphs instead of concrete ones, without affecting the algorithm's correctness and soundness. We observe that a given abstraction might not allow finding a correct repair even if one exists, in which case we use the desired repair as a means for refining the abstraction further. We believe that similar repair-driven refinement techniques may also be applicable to other frameworks, besides BP.

The rest of this chapter is organized as follows. We begin by defining abstract behavioral programs in Section 5.2. We then discuss applying CEGAR to BP in Section 5.3, and suggest an abstraction-based repair algorithm in Section 5.4. Our experimental results appear in Section 5.5. Discussion of related and future work appears in Section 5.6.

5.2 Abstractions for Behavioral Programming

Given behavioral programs P and $\overline{P}$, we say that $\overline{P}$ is an over-approximation of P if and only if $\text{Tr}(P) \subseteq \text{Tr}(\overline{P})$. Thus, for any LTL formula Φ over AP, $\text{Tr}(\overline{P}) \vDash \Phi$ implies $\text{Tr}(P) \vDash \Phi$, and so verifying that $\text{Tr}(\overline{P}) \vDash \Phi$ shows that the original program is correct (for an introduction to LTL see, e.g., [23]). In this section we focus on constructing a suitable program $\overline{P}$ that is smaller than P, so that checking whether $\text{Tr}(\overline{P}) \vDash \Phi$ is easier than checking whether $\text{Tr}(P) \vDash \Phi$.

5.2.1 Abstracting a Behavioral Thread

We begin by defining abstractions of b-threads. Let $BT = \langle Q, \delta, q_0, R, B, L \rangle$ be a thread over event set E and propositions AP, and let π be a AP-preserving partition of Q, i.e., $q_1 \equiv_\pi q_2 \implies L(q_1) = L(q_2)$. Let $\eta_\pi : Q \to Q/\pi$, termed the abstraction function induced by π, be a function that maps each state to its equivalence class under π. η_π gives rise to a b-thread

$\overline{BT} = \langle \overline{Q}, \overline{\delta}, \overline{q_0}, \overline{R}, \overline{B}, \overline{L} \rangle$, called the abstraction thread of BT induced by π, defined in the following manner. The states of $\overline{BT}$ are the equivalence classes $\overline{Q} = Q/\pi$, and its initial state is $\overline{q_0} = \eta_\pi(q_0)$. For every state $\overline{q} \in \overline{Q}$, the mapping functions are given by $\overline{R}(\overline{q}) = \bigcup_{q \in \eta_\pi^{-1}(\overline{q})} R(q)$, $\overline{B}(\overline{q}) = \bigcap_{q \in \eta_\pi^{-1}(\overline{q})} B(q)$ and $\overline{L}(\overline{q}) = L(q)$ for (every) $q \in \eta_\pi^{-1}(\overline{q})$. The transitions relation $\overline{\delta}$ is derived from δ by:

$$\frac{q \xrightarrow{e} \tilde{q}}{\eta_\pi(q) \xrightarrow{e} \eta_\pi(\tilde{q})}$$

Note that for every $\overline{q}$, $\overline{R}(\overline{q}) \cap \overline{B}(\overline{q}) = \emptyset$, and that $\overline{q}$ has a transition for every $e \notin \overline{B}(\overline{q})$. Hence, $\overline{BT}$ is a valid b-thread. The definition is designed to make $\overline{BT}$ *more permissive* than BT — that is, to ensure that replacing BT with $\overline{BT}$ within a given program results in an over-approximation of that program. In particular, the abstraction preserves atomic proposition of states, and abstract states request at least as much and block no more than their matching concrete states. Formally, we present the following Lemma

Lemma 7. Let $P = [BT^1 \parallel \ldots \parallel BT^n]$ be a behavioral program. Let π be an AP-preserving partition of the states of BT^1, and let $\overline{BT^1}$ be the abstraction of BT^1 induced by π. Finally, let $\overline{P} = [\overline{BT^1} \parallel BT^2 \parallel \ldots \parallel BT^n]$. Then $\text{Tr}(P) \subseteq \text{Tr}(\overline{P})$.

Proof. Observe that, without loss of generality, we may assume that $n = 2$; otherwise, we would first calculate the composition $BT' = BT_2 \parallel \ldots \parallel BT^n$, and then deal with $P = [BT^1 \parallel BT']$.

In order to prove the lemma, we look at an execution ε of P, and prove that there exists an execution $\overline{\varepsilon}$ of $\overline{P}$ such that $\text{Tr}(\varepsilon) = \text{Tr}(\overline{\varepsilon})$ — and hence, $\text{Tr}(P) \subseteq \text{Tr}(\overline{P})$.

Let $\varepsilon = q_0 \xrightarrow{e_1} q_1 \xrightarrow{e_2} \ldots$ be an execution of P. Each state q_i is comprised of two components, q_i^1 and q_i^2, denoted $q_i = \langle q_i^1, q_i^2 \rangle$, such that q_i^j is a state of thread BT^j. We look at an execution $\overline{\varepsilon} = \overline{q_0} \xrightarrow{e_1} \overline{q_1} \xrightarrow{e_2} \ldots$, with same events as ε. The states are set to $\overline{q_i} = \langle \eta_\pi(q_i^1), q_i^2 \rangle$, where η_π is the abstraction function mapping each state to its equivalence class under partition π. We next show that this $\overline{\varepsilon}$ is a valid execution of $\overline{P}$, and that it has the same trace as ε.

By definition, every state $\overline{q_i}$ is indeed a state of $\overline{P}$. Further, $L(\overline{q_i}) = L(\eta_\pi(q_i^1)) \cup L(q_i^2) = L(q_i^1) \cup L(q_i^2) = L(q_i)$, and consequently $\text{Tr}(\overline{\varepsilon}) = \text{Tr}(\varepsilon)$. It only remains to prove that for each $\overline{q_i}$, the transition to $\overline{q_{i+1}}$ is legal — namely, that event e_{i+1} is enabled at state $\overline{q_i}$ and that the transition $\overline{q_i} \xrightarrow{e_{i+1}} \overline{q_{i+1}}$ exists in $\overline{P}$.

To see why event e_{i+1} is enabled, recall that by definition $\overline{R}(\eta_\pi(q_i^1)) \subseteq R(q_i^1)$. Hence:

$$e_{i+1} \in R(q_i^1) \cup R(q_i^2) \implies e_{i+1} \in \overline{R}(\eta_\pi(q_i^1)) \cup R(q_i^2)$$

And so, if e_{i+1} is enabled in state $\langle q_i^1, q_i^2 \rangle$ then it is also enabled in state $\langle \eta_\pi(q_i^1), q_i^2 \rangle$. Finally, by the abstraction's definition, the transition $q_i^1 \xrightarrow{e_{i+1}} q_{i+1}^1$ in BT^1 implies the transition $\eta_\pi(q_i^1) \xrightarrow{e_{i+1}} \eta_\pi(q_{i+1}^1)$ in $\overline{BT^1}$; and, in turn, the transition $\langle \eta_\pi(q_i^1), q_i^2 \rangle \xrightarrow{e_{i+1}} \langle \eta_\pi(q_{i+1}^1), q_{i+1}^2 \rangle$ in $\overline{P}$. Thus, $\overline{\varepsilon}$ is a valid execution of $\overline{P}$; the claim follows. □

By definition, a thread's abstraction is determined by the AP-preserving partition π in use. Clearly, an abstraction of a minimal number of states is achieved when π is the AP-partition itself, i.e. $q_1 \equiv_\pi q_2 \iff L(q_1) = L(q_2)$. As our goal is to minimize the number of states of the composed program, this partition is of special interest. We refer to this abstraction as the coarsest abstraction of BT, and denote it by $\widehat{BT}$.

5.2.2 Abstracting a Behavioral Program

Due to BP's composite nature — where sets of composed threads are threads themselves — thread abstraction can be applied at various points throughout the composition process. In choosing when to apply it, our goal is to end up with an over-approximation that is neither too concrete (to mitigate state explosion), nor too abstract (so that it is meaningful). In our experiments, the best results were achieved by first grouping threads that are logically related and composing them into modules. Intuitively, this entails clustering threads that assign similar atomic propositions to their states into the same module. Each module is then abstracted individually, effectively ignoring threads that deal with other atomic propositions. Finally the abstractions are composed, generating the desired over-approximation. In this section we provide motivation for this approach, and propose an automated way for grouping together logically related threads.

To illustrate the benefits of using modules, we first discuss two of the more natural alternatives. One approach is to apply abstraction at the last step of the composition process: i.e., to compute $BT = BT^1 \parallel \ldots \parallel BT^n$ and then set $\overline{P} = [\widehat{BT}]$. While this method produces meaningful abstractions, it entails calculating the very large b-thread BT, which has at least as many states as P. Hence, this technique suffers from the state explosion problem that we have been trying to avoid. Another natural approach is to abstract each of the basic threads, i.e. calculate $\overline{P} = [\widehat{BT^1} \parallel \ldots \parallel \widehat{BT^n}]$. While this method does indeed circumvent the state explosion problem, our experiments show that the abstractions it tends to produce are too coarse to be of any practical use. Specifically, behavioral programming promotes writing threads that are small and specific, and tend to contain a single atomic proposition. Thus, early abstraction usually collapses the threads into a couple of states each, abstracting away most implementation details. Later, during verification tasks, multiple rounds of refinement are needed until a meaningful model is obtained.

The module based method can be seen as a middle ground between these two extreme alternatives. On one hand, as abstraction is applied during the early phases of the composition process, the state explosion problem is averted. On the other hand, as it is applied to threads that are sufficiently complex, the resulting over approximation is more likely to be meaningful.

The rationale behind grouping together logically related threads, as opposed to just using an

arbitrary partitioning of the threads, is the desire to generate small modules: logically related threads tend to share atomic propositions, and request and block similar events. Consequently, the resulting abstractions tend to contain fewer states, and the approximation labeling functions $\overline{R}$ and $\overline{B}$ tend to be tighter, reducing the number of edges in the final over-approximation.

We conclude this section by discussing an automated method for grouping together logically related threads. As the above discussion suggests, such threads tend to share atomic propositions and requested/blocked events, and indeed this is how we attempt to group them. Let BT be a thread with states $q_1,\ldots,q_m$, and let $ap \in AP$. We define the correlation between BT and ap as:

$$cor(BT,ap) = \frac{|\{i \mid ap \in L(q_i)\}|}{m}$$

A thread's correlation to an atomic proposition is thus the fraction of states to which the labeling function assigns the proposition. Intuitively, threads that have high correlation to the same atomic proposition may be logically related. Setting a threshold M, say 0.5, induces a partitioning of the threads into modules, denoted $\equiv_M$. At first each thread is considered to reside in a separate module, and then pairs of modules are iteratively joined by the rule:

$$cor(BT^i,ap) \geq M \bigwedge cor(BT^j,ap) \geq M \implies BT^i \equiv_M BT^j$$

Analogous correlation can be defined between threads and events, by considering the fraction of states in which a thread requires or blocks the event. These correlations are easy to compute using static analysis of the threads, and are supported by the BPC framework.

Further information that can be taken into account when looking for related threads includes various string distance metrics applied to their respective names and locations in the directory structure — as programmers tend to group similar threads together and give them similar names. These measures are also straightforward to compute using automated methods. Finally, any or all of the above measures can be combined into a single metric, yielding the desired partition into logically related modules.

We summarize the resulting module-based abstraction algorithm:

Algorithm 6 Module-Based Abstraction

1: Partition the threads into modules $BT^{M_1},\ldots,BT^{M_m}$
2: For each module BT^{M_i}, calculate $\widehat{BT^{M_i}}$
3: **return** $\overline{P} = [\widehat{BT^{M_1}} \parallel \ldots \parallel \widehat{BT^{M_m}}]$

By iteratively applying Lemma 7, we get the following corollary:

Corollary. *Let $BT^1,\ldots,BT^n$ be threads over event set Σ and atomic propositions AP. Let $P = [BT^1 \parallel \ldots \parallel BT^n]$, and let $\overline{P}$ be the program returned by Algorithm 6. Then* $\mathrm{Tr}(P) \subseteq \mathrm{Tr}(\overline{P})$.

5.3 Counterexample Guided Abstraction-Refinement

Given a behavioral program P and an LTL property Φ, we attempt to prove that $P \models \Phi$ by calculating an over-approximation $\overline{P}$ and proving that $\overline{P} \models \Phi$. However, it may be the case that $P \models \Phi$ but $\overline{P} \not\models \Phi$, because $\overline{P}$ is too abstract (see an illustration in Figure 5.1). Model checking $\overline{P}$ then results in a *spurious* counterexample, i.e. one that exists in $\overline{P}$ but not in P. A standard technique for handling this problem, known as *counterexample guided abstraction refinement* (*CEGAR*) [51], uses such spurious counterexamples in order to refine $\overline{P}$ in a way that eliminates them. The process is repeated until a genuine counterexample is found, or until the property is shown to hold.

In this section, we describe an implementation of CEGAR in the context of BP. The two main phases of the technique — determining whether a counterexample is spurious or genuine and refining the abstraction in order to eliminate spurious executions — are discussed in Sections 5.3.1 and 5.3.2, respectively.

For simplicity, we limit the discussion to safety properties, for which counterexamples are finite executions. The method can be extended to liveness properties and the associated loop counterexamples through *loop unwinding*; see [51].

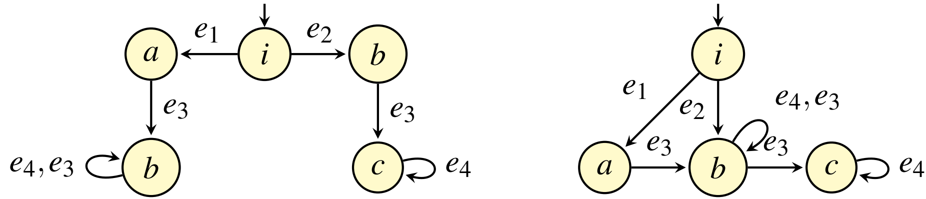


Figure 5.1: A concrete state graph (on the left), and a matching abstraction (on the right). The atomic proposition labeling appears inside the states. The two states with identical labeling (b) are abstracted into a single state. The abstract state graph contains fewer states, but it also allows spurious executions. While some properties, such as $G(a \rightarrow X\neg a)$, hold for both graphs, the property $G(a \rightarrow G\neg c)$ holds in the concrete case but not in the abstract one, because of the spurious execution fragment $i \stackrel{e_1}{\rightarrow} a \stackrel{e_3}{\rightarrow} b \stackrel{e_3}{\rightarrow} c$.

5.3.1 Determining if an Execution is Spurious

Suppose that on checking whether $\overline{P} \models \Phi$, the model checker replies in the negative, providing a finite counterexample $\overline{\varepsilon}$. We wish to determine whether $\overline{\varepsilon}$ is a valid execution of the original system. The idea, based on [51], is to simulate $\overline{\varepsilon}$ on the concrete program in order to check if it constitutes a genuine execution. During this simulation, we must take into account the two layer structure of our abstraction scheme, as well as the role of *requested* and *blocked* events, in determining whether runs are valid.

Let $\overline{P} = [\widehat{BT^{M_1}} \parallel \ldots \parallel \widehat{BT^{M_m}}]$ be an abstract program, composed of m abstract modules, and let $\overline{\varepsilon} = \overline{q_0} \overset{e_1}{\to} \overline{q_1} \overset{e_2}{\to} \ldots \overset{e_n}{\to} \overline{q_n}$ be a finite execution of $\overline{P}$. It is tempting to say that $\overline{\varepsilon}$ is a valid execution of the concrete system if and only if its projections onto the modules form valid executions of the modules; indeed, a similar technique is used in [49]. However, in our context, this approach does not suffice. Consider, for instance, the case where the transition labeled e_1 in $\overline{\varepsilon}$ exists in each of the concrete modules, but that none of them requests event e_1. In this case, looking at each module separately, we would have no way of knowing whether event e_1 is indeed enabled on the program level. Thus, our scheme must take into account the mutual effect modules have on each other.

We begin with some notation. For a set of states S, we denote by $R(S,e)$ the subset of states of S in which event e is requested. We use $\text{Post}(S,e)$ to denote the set of successors of states in S when event e is triggered. Finally, let $\overline{q} = \langle \overline{q^1}, \overline{q^2}, \ldots, \overline{q^m} \rangle$ denote an abstract state, and let η_j denote the abstraction function of module BT^{M_j}. We use η to denote the global abstraction function, i.e. $\eta(\langle q^1, \ldots, q^m \rangle) = \langle \eta_1(q^1), \ldots, \eta_m(q^m) \rangle$. This function and its inverse function are not stored explicitly, as doing so for every state in $\overline{P}$ would entail enumerating all states of P — negating the advantages offered by our two layered approach. Instead, η is only computed locally for specific states, on demand, by invoking the module abstraction functions.

Our technique follows the idea of [51], and defines a series of sets $\{S_i\}$, representing the concrete states the system can actually reach in each step of $\overline{\varepsilon}$. These sets are computed by using the concrete module state graphs. The definition of S_i is given by $S_0 = \{\langle q_0^1, q_0^2, \ldots, q_0^m \rangle\}$ for the concrete initial states and $S_i = \text{Post}(R(S_{i-1}, e_i), e_i) \cap \eta^{-1}(\overline{q_i})$ for $1 \leq i \leq n$.

The idea behind this definition is to walk on the abstract graph according to the execution, and for each abstract state identify the concrete states that are truly reachable along this specific execution, using the S_i sets. As we later prove, a run is genuine if and only if it corresponds to a series of non-empty sets. Each set is derived from its predecessor by looking only at states in which the next event is requested, and calculating their successor states. Out of these successors we only keep those that are abstracted to the next state of the abstract execution, as expressed by intersecting with $\eta^{-1}(\overline{q_i})$.

The actual algorithm for checking whether an execution is spurious is thus:

Algorithm 7 Check If Spurious

```
1: for i := 0 to n do
2:     Calculate S_i; if it is empty, return True
3: return False
```

The algorithm's correctness is established via Lemma 8:

Lemma 8. Let $\overline{\varepsilon}$ be an execution of $\overline{P}$. Then $\overline{\varepsilon}$ is spurious, i.e. is not a valid execution of P, if and only if Algorithm 7 returns *True*.

Proof. We show that the algorithm answers *False* if and only if the run is genuine.

First Direction: A Genuine Run.

Suppose that $\overline{\varepsilon} = \overline{q_0} \overset{e_1}{\to} \overline{q_1} \overset{e_2}{\to} \dots \overset{e_n}{\to} \overline{q_n}$ is a genuine execution of $\overline{P}$; i.e., there exists an execution $\varepsilon = q_0 \overset{e_1}{\to} q_1 \overset{e_2}{\to} \dots \overset{e_n}{\to} q_n$ of P, such that for every $q_i = \langle q_i^1, q_i^2, \dots, q_i^m \rangle$ and $\overline{q_i} = \langle \overline{q_i}^1, \overline{q_i}^2, \dots, \overline{q_i}^m \rangle$ it holds that $\eta_j(q_i^j) = \overline{q_i}^j$ for every j. Further, in every concrete state q_i (for $0 \le i < n$), event e_{i+1} is enabled.

A straightforward inductive argument on $i = 1 \dots, n$ shows that for the i'th step of ε, set S_i contains the concrete state $\langle q_i^1, q_i^2, \dots, q_i^m \rangle$, and is thus non-empty. As this state requests and does not block the next event of the execution, it follows that $q_{i+1} \in S_{i+1}^j$. Hence, for all i we get $S_i \neq \emptyset$, which in turn implies that the algorithm returns *False*, as needed. □

Second Direction: Algorithm returns *False*.

Suppose that on execution $\overline{\varepsilon} = \overline{q_0} \overset{e_1}{\to} \overline{q_1} \overset{e_2}{\to} \dots \overset{e_n}{\to} \overline{q_n}$, the algorithm answers *False*. We show that this implies the existence of a genuine run ε that corresponds to $\overline{\varepsilon}$.

By the algorithm's answer, we know that the computed sets S_i are not empty for all $0 \le i \le n$ and. We use these sets, backtracking from $i = n$ to $i = 0$, reconstructing the genuine run as we go.

For $i = n$, we pick an arbitrary $q_n = \langle q_n^1, \dots, q_n^m \rangle \in S_n$. Then, for state q_{n-1}, we pick a state $q \in S_{n-1}$ such that $e_n \in R(q)$ and $q_n \in \mathrm{Post}(q, e_n)$; such a state exists by the way the S_i sets are defined. This process continues iteratively, until $\varepsilon = q_0 \overset{e_1}{\to} q_1 \overset{e_2}{\to} \dots \overset{e_n}{\to} q_n$ is constructed. It is straightforward to see that it constitutes a valid run of P. The claim follows. □

Observe that computing the S_i sets is performed using the concrete state graphs of the modules, and does not entail constructing the explicit state graph of P. Every state $q \in S_i$ is stored as the set of module states to which it corresponds. The sets $R(q,e)$ and $Post(R(q,e),e)$ can be computed locally from these states. Further, there is no need to actually compute $\eta^{-1}(\overline{q_i})$, which is costly; instead, for every $q \in \mathrm{Post}(R(S_{i-1}, e_i), e_i)$, we check whether $\eta(q) = \overline{q_i}$ by applying the module abstraction functions to its components, which is substantially cheaper.

5.3.2 Refining in order to Eliminate a Spurious Execution

We now discuss refining $\overline{P}$ in order to eliminate a spurious counterexample, thus allowing another round of model checking. The iteration on which Algorithm 7 halted indicates where the refinement should occur. Indeed, this is where the abstract and concrete graphs diverge, and so splitting the previous abstract state into multiple states could render the spurious execution invalid.

Suppose that the *Check If Spurious* algorithm stopped because $S_{i+1} = \varnothing$. This indicates a problem with transition $\overline{q_i} \overset{e_{i+1}}{\to} \overline{q_{i+1}}$ of the execution: either the concrete system can only reach states that are not mapped to abstract state $\overline{q_{i+1}}$, or event e_{i+1} is not even enabled in the concrete program — although it is enabled in the abstract one. Each case is characterized and handled differently:

Case 1.

For all concrete states in S_i, transitions labeled e_{i+1} do not lead to abstract state $\overline{q_{i+1}}$, i.e. $\text{Post}(S_i, e_{i+1}) \cap \eta^{-1}(\overline{q_{i+1}}) = \varnothing$. In this case, we split $\overline{q_i}$ into 2 abstract states: state $\overline{q_i'}$ that corresponds to the concrete states S_i, and state $\overline{q_i''}$ that corresponds to the remaining states, $\eta^{-1}(\overline{q_i}) - S_i$. By definition, execution $\overline{\varepsilon}$ would visit abstract state $\overline{q_i'}$ instead of $\overline{q_i}$, from which there would be no transitions to $\overline{q_{i+1}}$. Thus, $\overline{\varepsilon}$ would no longer be a valid execution of the abstract program. This case corresponds to the technique used in [51].

Case 2.

There exists a state $q \in S_i$ such that $\text{Post}(q, e_{i+1}) \in \eta^{-1}(\overline{q_{i+1}})$. However, $e_{i+1} \notin R(q)$; if that were not so, we would get $S_{i+1} \neq \varnothing$. In this case, state q is *waiting* for event e_{i+1} without requesting it. The request for e_{i+1} is made by a different state in $\eta^{-1}(\overline{q_i})$. As both states are mapped into the same abstract state, the outcome is the edge $\overline{q_i} \overset{e_{i+1}}{\to} \overline{q_{i+1}}$.

In this case, performing refinement as in Case 1 might not suffice, as the state requesting event e_{i+1} might also be in S_i. We thus resort to two rounds of refinement: first, we split state $\overline{q_i}$ into $\overline{q_i'}$ and $\overline{q_i''}$, as before. Then, we further refine state $\overline{q_i'}$, in order to separate states requesting event e_{i+1} from those that do not. Formally, we split $\overline{q_i'}$ into state $\overline{q_i^R}$ corresponding to concrete states $q \in S_i$ such that $e_{i+1} \in R(q)$, and state $\overline{q_i^{NR}}$ corresponding to concrete states $q \in S_i$ such that $e_{i+1} \notin R(q)$. By definition, execution $\overline{\varepsilon}$ would visit abstract state $\overline{q_i^{NR}}$ instead of $\overline{q_i}$, from which there would be no transitions to $\overline{q_{i+1}}$, making it an invalid execution of the abstract program.

The following Lemma immediately follows from the above discussion:

Lemma 9. Let $\overline{\varepsilon}$ be a spurious execution of $\overline{P}$, and let $\overline{P'}$ be the refined program obtained by the above refinement step. Then $\overline{\varepsilon}$ is not a valid execution of $\overline{P'}$.

Observe that the iterative verification process entails explicitly computing $\eta^{-1}(\overline{q})$ once per each refinement step. While this step is expensive, hopefully the number of iterations is small. Reducing the number of iterations is part of our motivation for using logically related modules — see discussion in Section 5.2.2.

We note that the resulting refinement is defined in terms of a global abstract state that should be split into smaller states. However, as η is not stored explicitly, this refinement cannot be

applied directly. Constrained by our two layered setting, we may only perform refinements on the module abstraction functions $\eta_1, \ldots, \eta_m$, indirectly refining η. Thus, a set of refinements for the $\eta_1, \ldots, \eta_m$ functions needs to be derived from the desired η refinement. This can be performed by separating (within the modules) any pair of concrete states that do not always appear simultaneously in the new global abstract states. However, as not every refinement of η can be expressed as refinements of $\eta_1, \ldots, \eta_m$, the resulting global refinement may be finer (i.e., produce more states) than the desired one.

5.4 Repair using Abstractions

In this section we discuss how our proposed abstraction scheme can prove useful in the context of repairing violated safety properties. Repairing safety violations in behavioral programs was discussed in Chapter 4; we provide a short recap here.

Our scope includes fixing safety violations in existing programs. Finding these violations can be reduced to invariant checking [23]. Thus, without loss of generality, a program is correct if its state graph has no reachable "bad" states. This, along with the event blocking idiom of BP, enables an elegant method of repair by trimming: correcting the program by removing edges from its state graph using the blocking idiom, so that bad states become unreachable.

The repair is non-intrusive, i.e. performed strictly by adding new threads to the program (termed "wait-block patches"), and without modifying existing code. The patch threads are passive, in the sense that they never request any events or assign any atomic propositions to states, thus keeping the repaired program as close to the original as possible. Only when the execution gets dangerously close to a bad state does the patch block events that would cause a violation, forcing the system to choose a different execution path. In Chapter 4 it is shown that, for programs with deterministic threads, this method does not eliminate correct executions, as events are blocked only when they are guaranteed to lead to a violation. Further, no deadlocks are created as a result of such patching.

This repair technique is adequate for systems that are capable of generating the desired ("good") behavior but may, in some scenarios, produce erroneous output. For instance, patching may be applied to a variety of bugs resulting from race conditions between parallel components — fixing them by temporarily blocking one of the components, forcing it to yield to its counterpart. However, not all systems can be repaired in this way, and the repair algorithm fails gracefully in this case. A soundness result shows that if a correct patch exists, it will indeed be found by the repair algorithm.

The algorithm operates by analyzing a program's state graph and looking for the smallest fixpoint set of states that can be removed from the graph in order to render q_b, the single bad state, unreachable. Specifically, the algorithm backtracks from q_b, attempting to isolate it by

trimming edges without creating deadlocks. Whenever all the successors of a state are bad, it is marked as bad itself; see Figure 5.2. Below is the repair algorithm's pseudo-code; *Pre* denotes the predecessor states of a given set of states.

Algorithm 8 Concrete Safety Patching

1: $BAD \leftarrow \{q_b\}$
2: $PRE \leftarrow Pre(BAD)$
3: **while** $\exists q \in PRE$ such that $\forall e, \mathrm{Post}(q,e) \in BAD$ **do**
4: Move q from PRE to BAD
5: **if** q is the initial state **then return** *Failure*
6: $PRE \leftarrow Pre(BAD)$
7: **return** a patch that blocks edges from PRE to BAD

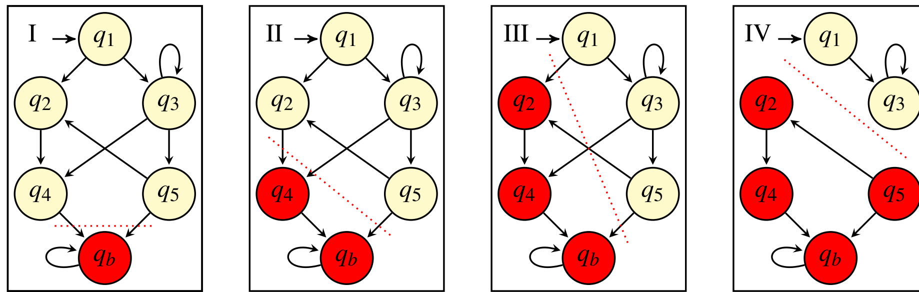


Figure 5.2: The algorithm for trimming the concrete state graph of a program in order to correct a safety violation. Graph I depicts the initial configuration, with the only bad state, q_b, marked in red. The edges from states in *PRE* to states in *BAD* cross the dotted red line, and are candidates for blocking. In the first iteration, blocking these edges would cause a deadlocked in state q_4. Thus, in graph II state q_4 is also marked as bad, and q_2 joins *PRE*. Unfortunately, now a deadlock would be caused in state q_2, and the algorithm iterates again, putting q_2 in *BAD*. The next iteration puts q_5 in *BAD*. Only then, in graph IV, can edges crossing the dotted line be safely removed without causing deadlocks. The states in *BAD* are thus rendered unreachable, fixing the safety violation.

As this algorithm uses the program's concrete state graph, it does not scale to large programs. We thus seek to adjust it so it can use an over-approximation instead. Unfortunately, directly applying the concrete patching algorithm to an abstract graph yields erroneous results. In particular, the algorithm might fail when a correct answer exists, or the resulting patches might also eliminate good executions — traits that did not exist in the concrete version. See Figure 5.3.

Intuitively, the reason for these failures is the fact that *patch-incompatible* concrete states are abstracted into the same abstract states. By patch-incompatible, we mean that the concrete algorithm would block a different set of events in each of the concrete states. In the abstract graph, however, such blocking becomes impossible, resulting in the algorithm's undesired be-

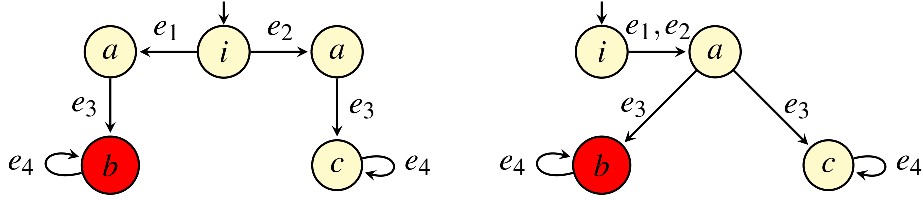


Figure 5.3: A concrete state graph on the left, and its abstraction on the right. The atomic propositions appear inside the states. The safety property in question is the invariant $G\neg b$, which is violated when the states in red are reached. In the concrete graph, a simple patch can fix the problem: by blocking e_1 in the initial state, the red state is made unreachable, and no deadlocks are caused. On the abstract graph, however, no repair is possible without causing a deadlock somewhere in the program. As a result of the nondeterminism in state a, where two edges correspond to the same event, we are unable to block one edge while leaving the other enabled.

havior. In order to overcome this difficulty, we incorporate a refinement phase into the repair algorithm; however, instead of using counterexamples as means of guiding the refinement, the driving force is the need to create abstract states that correspond only to patch-compatible concrete states.

The algorithm uses an over-approximation of the state graph, in which $\overline{q_b}$ is the single abstract bad state, corresponding to q_b. As in the concrete case, we assume the concrete b-threads are deterministic. Here is the pseudo-code:

Algorithm 9 Abstract Safety Patching

1: $BAD \leftarrow \{\overline{q_b}\}$
2: **while** $True$ **do**
3: $\quad PRE \leftarrow Pre(BAD)$
4: $\quad$ **if** $\exists \overline{q} \in PRE$ such that $NeedToRefine(\overline{q})$ **then**
5: $\quad\quad Refine(\overline{q})$
6: $\quad$ **else if** $\exists \overline{q} \in PRE$ such that $\forall e, \mathrm{Post}(\overline{q}, e) \subseteq BAD$ **then**
7: $\quad\quad$ Move $\overline{q}$ from PRE to BAD
8: $\quad\quad$ **if** $\overline{q}$ is the initial state **then return** $Failure$
9: $\quad$ **else**
10: $\quad\quad$ **return** a patch that blocks edges from PRE to BAD

The core of the algorithm remains the same as in the concrete case: we start at the bad state $\overline{q_b}$, backtracking and marking states that only lead to bad states as bad themselves. Once we reach a setting in which all states in PRE also have edges leading to good states (as to not create deadlocks), we return a patch trimming the edges from PRE to the bad states. The refinement phase prevents good executions from being likewise trimmed:

In order to determine if an abstract state $\overline{q}$ needs to be refined, we look at the events that we would like to block in it (set E). If there exists a concrete state in $\eta^{-1}(\overline{q})$ for which $e \in E$

Algorithm 10 NeedToRefine($\overline{q}$)

1: $E \leftarrow \{e \in \Sigma \mid \text{Post}(\overline{q}, e) \cap BAD \neq \emptyset\}$
2: **if** exists $q \in \eta^{-1}(\overline{q}), e \in E$ such that $e \in R(q)$ and $\eta(\text{Post}(q,e)) \notin BAD$ **then**
3:    **return** $True$
4: **if** exists $q \in \eta^{-1}(\overline{q})$ such that $R(q) \subseteq E$ **then**
5:    **return** $True$
6: **return** $False$

is requested and leads to a good state, refinement is needed to prevent good executions from being eliminated. Similarly if there exists a state in $\eta^{-1}(\overline{q})$ that has no requested events that would remain unblocked, refinement is needed in order to avoid causing a deadlock. The actual refinement is performed as follows:

Algorithm 11 Refine($\overline{q}$)

1: For every $q \in \eta^{-1}(\overline{q})$ calculate $\mathfrak{B}(q) = \{e \in R(q) \mid \eta(Post(q,e)) \in BAD\}$
2: Form a partition $\eta^{-1}(\overline{q}) = C_1 \cup C_2 \cup \ldots \cup C_k \cup C_{deadlock}$ such that if $\mathfrak{B}(q) = R(q)$, then $q \in C_{deadlock}$; else, $q_1, q_2 \in C_i \iff \mathfrak{B}(q_1) = \mathfrak{B}(q_2)$.
3: Split abstract state $\overline{q}$ into $k+1$ new states $\overline{q_1}, \ldots, \overline{q_{k+1}}$ such that $\eta^{-1}(\overline{q_i}) = C_i$ for $1 \leq i \leq k$, and $\eta^{-1}(\overline{q_{k+1}}) = C_{deadlock}$.

Set $\mathfrak{B}(q)$ contains the events to be blocked in q. The refinement splits the problematic abstract state into multiple abstract states, each representing concrete states in which the same events need to be blocked. Observe that state $\overline{q_{k+1}}$, in which the necessary blocking will introduce a deadlock, will be put in *BAD* in one of the following iterations of the main algorithm.

For correctness and soundness, we present the following theorem, which is analogous to the one for the concrete algorithm presented in [6]; hence, it demonstrates that the improved scalability does not come at the expense of the concrete version's desirable qualities.

Theorem. *For a behavioral program P and a violated safety property* Φ*,*

1. *A patch returned by Algorithm 9 eliminates all bad executions of the program, does not eliminate good executions, and does not create deadlocks.*

2. *If there exists a wait-block patch that corrects P with respect to* Φ*, such a patch will be found by Algorithm 9. Otherwise, the algorithm will issue a* Failure *notice.*

Proof. We begin with a side note about the meaning of a patch eliminating executions. As the patch is intended to be integrated into the program as a thread, it will change the program's underlying state graph. Hence, it is not immediate that executions of the original system have any meaning in the context of the patched program.

We resolve this issue by making the following observation. Due to the special structure of the patch — namely, that it follows the program's state graph and only blocks events, without requesting events or assigning atomic propositions — the program graph of the patched program is *isomorphic* to that of the original program, except for the edges being removed. Hence, any execution of the original program corresponds to a unique execution of the patched program, and it makes sense to discuss such executions being eliminated. For simplicity, for the rest of the proof we ignore this issue, regarding patches as eliminating transitions in the original state graph without modifying its states.

The theorem's proof relies mainly of the following invariant of the algorithm, which we prove as a separate proposition:

Proposition 1. Let $\overline{q}$ denote an abstract state that the algorithm puts in set *BAD*. Then for any concrete state abstracted into $\overline{q}$, i.e. for every $q \in \eta^{-1}(\overline{q})$, any execution ε of P that visits q must violate Φ.

Proof. We prove the proposition using induction on the algorithm's iteration index. Observe iteration i, the first iteration in which some state $\overline{q}$ is about to enter set *BAD*. At the beginning of this iteration, set *BAD* contains only the abstract state $\overline{q_b}$. Since $\overline{q}$ is about to enter set *BAD*, it must be that $\overline{q} \in PRE$. Further, the *NeedToRefine* subroutine returned *False* on $\overline{q}$ — meaning that any event that is not blocked in $\overline{q}$ leads to $\overline{q_b}$. If there existed a concrete state $q \in \eta_{i_1}^{-1}(\overline{q})$ and an event $e \in R(q)$ such that $Post(q,e) \neq \{q_b\}$, a matching transition would also appear in the abstract graph, and $\overline{q}$ would not be put in *BAD*. Hence, $Post(q) = \{q_b\}$. In other words, any concrete execution passing through any concrete state associated with $\overline{q}$ is bound to visit q_b and cause a violation.

Now, suppose that the claim holds for the first i iterations, and observe iteration $i+1$. Suppose a new state $\overline{q}$ joins *BAD* in this iteration. The reasoning is the same as before: $\overline{q}$ is put in *BAD* only if for every $q \in \eta^{-1}(\overline{q})$ and every event $e \in R(q)$, $q \xrightarrow{e} q'$ implies that $\eta(q') \in BAD$. By the inductive hypothesis, an execution that visits q' is thus bound to cause a violation. Since this applies to every successor of every concrete state $q \in \eta^{-1}(\overline{q})$, the claim follows. □

A second observation that we prove separately is that the algorithm always halts:

Proposition 2. The *Abstract Safety Patching* algorithm always halts.

Proof. Observe the algorithm's main loop. If the algorithm does not stop, it must make infinitely many iterations of this loop. Each iteration that does not lead to termination is devoted to either performing a single refinement of the abstract program, or to moving an abstract state into the growing set *BAD*. We show that both types of iterations can only be performed a finite number of times, proving the proposition.

Begin with iterations dedicated to refinement. Any refinement step splits an abstract state into at least two states; hence, each such step increases the number of states of the abstract program by at least one. Since this number is bound from above by the number of states of the original program, only a finite number of refinements can be performed. Once the abstract and concrete program coincide, the *NeedToRefine* subroutine will return *False* on every state, and the algorithm will cease attempting to refine the program.

We now turn to iterations in which states are moved to *BAD*. Observe the set of concrete states mapped to *BAD* in iteration i, denoted $\eta_i^{-1}(BAD)$. These sets start with $\eta_1^{-1}(BAD) = \{q_b\}$, and for each iteration i that puts a new state in *BAD* we have $|\eta_i^{-1}(BAD)| > |\eta_{i-1}^{-1}(BAD)|$. Since the size of $|\eta_i^{-1}(BAD)|$ is also upper bounded by the number of states in the concrete program, we get that the number of such iterations is also finite. We thus conclude that the algorithm always halts. □

We now use these propositions to prove part 1 of the theorem. Consider a patch BT_P produced by the repair algorithm. This patch eliminates transitions leading to all states in set *BAD*, effectively disconnecting them from the state graph. In particular, all executions leading to state $\overline{q_b}$ are eliminated. Since the existence of a concrete execution leading to q_b implies the existence of an abstract execution leading to $\overline{q_b}$, it follows that the patch indeed eliminates all bad executions in the concrete system.

Next, we show that no good executions are eliminated. All transitions that were removed from the state graph lead to states in *BAD*. By Proposition 1, any execution that visits these states is bound to cause a violation; hence, none of the affected executions are good.

Finally, we show that no deadlocks are created by the algorithm. A deadlock is created if and only if there exists a state in $q \in \eta^{-1}(PRE)$ for which the set of requested events, $R(q)$, coincides with the events to be blocked. Hence, when the state graph is finalized, state q would have no outgoing transitions.

Observe state $\overline{q} = \eta(q)$. This state is in *PRE*, and is not moved to *BAD*; hence, it has outgoing transitions that lead to good states. These transitions cannot originate in q; hence, there is another state, $q' \neq q$, such that $\eta(q') = \overline{q}$ and q' would not become deadlocked when the patch is applied. This contradicts the fact that $\overline{q} \in PRE$ at the time the algorithm halts, as subroutine *NeedToRefine* would return *True* for state $\overline{q} = \eta(q)$, leading to its being refined. This refinement would cause states q and q' to be mapped into separate abstract states; and in the algorithm's next iteration, the abstract state of q would be put in *BAD*. Hence, no deadlocks can occur as a result of patching, and the first part of the theorem is proven. □

We now turn to part 2 of the theorem. Here, we must show that the algorithm does not return a *Failure* when a correct patch exists. Suppose, then, that a correct patch BT_P exists. This patch corresponds to a set of transitions that are to be blocked, cutting off some of the concrete program's states. Also, this patch does not create deadlocks. We mark the set states

to be cut off by S. Again we observe the series of sets $\eta_i^{-1}(BAD)$ that our algorithm grows through its iterations. By Proposition 1, for every i the set $\eta_i^{-1}(BAD)$ consists only of states that must lead to a violation of Φ. Since BT_P is correct, it follows that it, too, cannot allow executions to reach states in $\eta_i^{-1}(BAD)$. In other words, for every i we have $\eta_i^{-1}(BAD) \subseteq S$.

Our algorithm only issues a *Failure* notice if it reaches a state where the initial state of the concrete system, q_0, is in $\eta_i^{-1}(BAD)$. However, by the correctness of BT_P, set S cannot contain q_0, or else it would create deadlocks. Hence, our algorithm will not return a *Failure* notice. As Proposition 2 establishes that the algorithm must halt, we conclude that it will return some patch. Finally, by the first part of the theorem, this patch will be correct. We conclude that our algorithm will indeed output a correct patch if such a patch exists, as needed. □

In this algorithm, the inverse global abstraction $\eta^{-1}(\overline{q})$ is computed multiple times; indeed, this is an expensive step. However, for programs that are "close to being correct", the repair algorithm may only need to perform a few refinements, hopefully terminating in reasonable time. As discussed in Section 5.3.2, not every refinement is obtainable in our two layered structure; see discussion therein.

5.5 Experimental Results

For our experiments we used the *BPC* framework for BP in C++, available online [9]. We implemented the algorithms presented in the previous sections, namely thread abstraction, partitioning into modules, CEGAR verification and abstraction based patching, as a proof-of-concept tool on top of BPC. Since our goal was to show the improved scalability offered by the abstraction techniques, we also implemented concrete versions of the same algorithms in BPC. All implementations are explicit; symbolic implementation is left for future work.

We tested our algorithms on a BP based web-server application. The server, a work in progress, implements basic TCP and HTTP protocol stacks and is compatible with the Firefox browser. Due to the server's size of several million states, BPC ran out of memory when attempting to verify it concretely.

In contrast, the abstraction based methods were able to produce an initial abstraction of the system within 22 seconds. The automated module partitioning algorithm successfully divided the threads into logically related modules along the lines of the TCP and HTTP layers, grouping the HTTP threads into a single module and dividing the TCP threads between a few modules. The resulting over-approximation contained 800 states and some 12500 transitions.

We then used this over-approximation to identify and repair a bug where the TCP stack would, under certain conditions, acknowledge a FIN message for already closed connections. Identifying this bug using the CEGAR-based verification algorithm took 9.5 minutes, and in-

cluded 3 refinement phases, at the end of which a genuine counterexample was produced. Producing a patch that fixes the bug using Algorithm 9 then took 38 minutes.

Our experiments were run on a 2.66 GHz T500 laptop. The model and some of the properties used for our tests are available from [107].

5.6 Related Work and Conclusion

The main contribution of our work is in applying abstraction techniques to behavioral programming. In particular, we propose a technique for efficiently generating over-approximations of programs, which can later be used in analysis algorithms. We demonstrate two such algorithms: a CEGAR based method for model checking behavioral programs, and an abstraction based algorithm for the repair of safety violations. We regard this research as a step in the direction of developing more scalable methodologies and tools for formal analysis of BP.

Another contribution of our work is in the field of program repair, where we show an abstraction based algorithm that uses repair-guided refinement. Program repair is closely related to the synthesis problem, where various abstraction-refinement schemes have been proposed (e.g., [95, 59]); thus, we feel that this is a useful concept that could potentially improve the scalability of existing repair methods, not necessarily restricted to BP.

The use of abstraction-refinement based techniques to expedite model checking has been extensively studied (e.g., [51, 52, 125, 19]) and has been implemented in several frameworks, such as SLAM [25] and BLAST [96]. Among these, the work most closely related to ours is the MAGIC framework [44, 49]. There, the authors similarly propose a two layer CEGAR approach, in which modules are abstracted separately and their abstractions then composed. However, the setting of [44, 49] allows spurious counterexamples to be checked against each module separately — whereas in the setting of BP, checking involves all modules simultaneously. Analogously, refinements may not be confined to a single module.

In the area of program repair, recent work has focused on locating faulty components and then using synthesis to alter or replace them. In [104, 142], the authors seek corrections in the form of strategies that may be implemented without introducing new states (memoryless strategies), in order to alter the original program as little as possible. We address the same need by only adding code, leaving the original program unmodified. The work of [75] discusses repairing boolean programs by using abstractions of these programs. This approach is similar to ours, but does not include a refinement phase in case spurious executions in the abstract program prevent finding a repair. In [109], the authors tackle state explosion by maintaining an under-approximation of a repair candidate, at each iteration adding more constraints that it must fulfill. New constraints are produced by checking the candidate against the concrete faulty system. This technique appears orthogonal to our own, in which the program is abstracted and

the repair candidate is calculated explicitly. Attempting to combine the two methods seems promising, and is left for future work.

A different repair approach includes using genetic and co-evolutionary programming [21, 150], where a set of candidate programs is iteratively evaluated against the specification. Programs with high *fitness* survive, and are *mutated* to produce the next iteration's candidates, until a correct program is obtained. This approach handles more general bugs than ours (as it is not limited to trimming), but may extensively alter the original program's code.

In the future, we plan to extend our abstraction-based repair algorithm to handle violated liveness properties, as well safety ones. Indeed, some preliminary work we have done shows promising results. Another direction we hope to pursue is improving the performance of BPC by enhancing it with symbolic capabilities. Finally, another interesting line of work is strengthening our module-partitioning algorithm: we feel the programmer-created b-threads contain currently untapped meta data about the structure of the system, which could be utilized in making "smarter" partitions. We hope that tapping this meta data will also prove useful in the context of automated compositional verification.

Chapter 6

On the Succinctness of *RWB* Programs

6.1 Introduction

As is well known, many measures of computational complexity are used to compare solutions to algorithmic and software development problems. However, when it comes to comparing the methods, languages and tools that are used to construct those solutions, one needs quite different criteria for comparison. One of the main approaches to this, which has been used ever since the Rabin-Scott work on nondeterministic automata [135], is the size of the description. Size comparisons are usually carried out on the finite automata level of detail, and the most common metric, often called *descriptive succinctness* or *state complexity*, is the total number of states needed by the automata to express certain languages.

A large amount of work has been dedicated to descriptive succinctness in recent decades. A few notable models whose succinctness has been studied in detail are nondeterministic and universal automata, alternating automata, reverse automata, unary automata, and also various kinds of grammars and language formalisms (see, e.g., [100] for a survey). These studies have been motivated by the strong connection between succinctness and *software reliability* [129], indicating that succinct software is easier to develop, maintain and reuse. Further, the descriptive succinctness of a model is often connected to the complexity of various decision problems in it [100], and hence can be relevant also to verification problems.

In this chapter, we set out to analyze the descriptive succinctness of various idioms used in concurrent programming, seeking, as in most previous studies, exponential gaps in descriptive power. In particular, we study whether the addition of certain idioms to a programming model exponentially improves that model's succinctness, and in what cases. In addition to the considerations mentioned above and to our desire to better understand the fundamental nature of these concurrency idioms, our motivation has another aspect: a careful selection of concurrency idioms may make resulting programs more amenable to formal analysis. Thus, a better characterization of concurrency idioms and of the types of problems which they are suitable for

solving could allow programmers to more carefully tailor the programming model used to the problem at hand — on the one hand retaining "just enough" concurrency to efficiently solve the problem, while on the other hand keeping the model simple and amenable to analysis [83].

Here, we focus on the three fundamental concurrency idioms defined in Section 2.2: requesting, blocking and waiting for events. As previously mentioned, the requesting and waiting-for idioms are fairly common in discrete-event programming languages, with versions thereof appearing as first-class citizens in, e.g., *publish-subscribe* architectures [69]; whereas the blocking idiom is somewhat less common, appearing, e.g., in the *live sequence charts* (LSCs) formalism [57]. All three idioms can, of course, be implemented in any high level language. Combined, they also form the *behavioral programming* (*BP*) model [90]. Research suggests that using these idioms may lead to simple code modules that are aligned with the specification [90].

Following the required definitions presented in Section 6.2, this chapter's contributions appear in Sections 6.3, 6.4 and 6.5. In Section 6.3 we study a model containing the requesting, waiting-for and blocking idioms (which we call the $\mathcal{RWB}$ model), and position it in comparison to other well known models. Specifically, we show that $\mathcal{RWB}$ is polynomially expressible as automata with cooperative concurrency *a la* statecharts [66], but that cooperative concurrency can be exponentially more succinct than $\mathcal{RWB}$. We then show that despite this gap, the $\mathcal{RWB}$ model, which affords greater encapsulation, shares some of the cooperative model's strength and offers considerable advantages when compared to non-parallel automata. Next, we show that the succinctness of $\mathcal{RWB}$ is additive to that of classical nondeterminism and universal ("and") nondeterminism, and that a combination of all three features yields a triple-exponential improvement in succinctness. This last result establishes a hierarchy of succinctness relations indicating, e.g., that the (more practical) nondeterministic or universal $\mathcal{RWB}$ models are double-exponentially more succinct than non-parallel automata.

Next, in Section 6.4, we study the separate contribution of each of $\mathcal{RWB}$'s idioms to the model's descriptive succinctness. We define variants of $\mathcal{RWB}$ in which each of these idioms is omitted, and show that the full $\mathcal{RWB}$ model has exponential succinctness advantages over each of the variants.

Finally, in Section 6.5 we show that each of the downgraded versions of $\mathcal{RWB}$ has succinctness advantages over one or both of the other downgraded versions and over non-parallel models. This establishes the fact that each of the idioms makes its own unique contribution to succinctness, and is not subsumed by its counterparts. Notable among these results is the fact that event blocking, which is less common as a first-class concurrency idiom, provides exponential savings in succinctness. Further, we show that the succinctness afforded by each of these three idioms is not of equal power: for instance, the waiting-for idiom is weaker than the requesting one.

Related work appears in Section 6.6, and we conclude with Section 6.7.

6.2 Definitions

6.2.1 Request-Wait-Block Automata

In this chapter we use a slightly modified (although equivalent) version of the definitions given in 2.2. The reason for this is our desire to study the $\mathcal{RWB}$ idioms on the automata level, in order to maintain compatibility with the large existing body of work in this field.

An *$\mathcal{RWB}$-automaton* consists of orthogonal components called $\mathcal{RWB}$-threads:

Definition. *A* Request-Wait-Block-thread *($\mathcal{RWB}$-*thread*) is a tuple $\langle Q, E, \delta, q_0, R, B\rangle$, where Q is a finite set of states, E is a finite set of events, $\delta \subseteq Q \times E \times Q$ is a transition relation and q_0 is an initial state. We require that δ be deterministic, i.e. $\langle q,e,q_1\rangle \in \delta \wedge \langle q,e,q_2\rangle \in \delta \implies q_1 = q_2$. For simplicity of notation, we use $\bar{\delta}$ to indicate the effect event e has in state q (or its absence):*

$$\bar{\delta}(q,e) = \begin{cases} q' & ; \text{if exists } q' \in Q \text{ such that } \langle q,e,q'\rangle \in \delta \\ q & ; \text{otherwise}. \end{cases}$$

The mapping functions $R, B : Q \to 2^E$ associate a state with the set of events requested *and* blocked*, respectively, by the $\mathcal{RWB}$-thread in that state.*

Observe that there is no labeling function for waited-for events: the notion of waiting is expressed via the transitions between states. If state q has a transition labeled with event e that was not requested at q, the thread is considered to be waiting for event e in state q.

A composition of $\mathcal{RWB}$-threads yields an $\mathcal{RWB}$-automaton, defined as follows:

Definition. *An $\mathcal{RWB}$-*automaton *(*RWBA*) A over a finite event set E is a finite tuple of $\mathcal{RWB}$-threads $\langle T_1, \ldots, T_n\rangle$, denoted $T_i = \langle Q^i, E^i, \delta^i, q_0^i, R^i, B^i\rangle$, such that $E^i \subseteq E$ for all i, and the Q^i state sets are pairwise disjoint.*

A configuration *of an RWBA is the state of its threads, i.e. an element of $Q^1 \times \ldots \times Q^n$. A configuration $\hat{c} = \langle \hat{q}^1, \ldots, \hat{q}^n\rangle$ is a* successor *of configuration $c = \langle q^1, \ldots, q^n\rangle$ with respect to an event $e \in E$, denoted $c \xrightarrow{e} \hat{c}$, whenever*

$$\underbrace{e \in \bigcup_{i=1}^{n} R^i(q^i)}_{e \text{ is requested}} \bigwedge \underbrace{e \notin \bigcup_{i=1}^{n} B^i(q^i)}_{e \text{ is not blocked}} \bigwedge_{i=1}^{n} \Big(\underbrace{(e \in E^i \implies \hat{q}^i = \bar{\delta}^i(q^i,e))}_{\substack{\text{affected threads read the event} \\ \text{and change state if needed}}} \wedge \underbrace{(e \notin E^i \implies \hat{q}^i = q^i)}_{\substack{\text{unaffected threads} \\ \text{stay in the same state}}} \Big).$$

Observe that, since the threads have deterministic transition functions, each configuration can have at most one successor with respect to a specific event. It may, however, have multiple successors, each with respect to a different event.

An *execution* of A is a sequence of configurations $c_0c_1c_2\dots$ such that, for all i, c_{i+1} is a successor (with respect to some event) of c_i and $c_0 = \langle q_0^1,\dots,q_0^n\rangle$ is the initial configuration. An execution may be an *infinite* sequence of successive configurations, or a *finite* sequence that ends in a *terminal configuration*, i.e., a configuration with no successors. Every execution $\varepsilon = c_0c_1c_2\dots$ of an RWBA induces a set $runs(\varepsilon) = \{\rho \in E^* \cup E^\omega : \forall_{0\le i<|\varepsilon|}, c_i \xrightarrow{\rho[i]} c_{i+1}\ \}$. These runs are also sometimes referred to as the *words* associated with ε. Note that $runs(\varepsilon) \subseteq E^*$ or $runs(\varepsilon) \subseteq E^\omega$, depending on ε being finite or infinite, respectively. We say that a run $\rho \in E^* \cup E^\omega$ is *accepted* by an RWBA A if there is an execution ε of A such that $\rho \in runs(\varepsilon)$. The *language* of A, denoted $\mathfrak{L}(A)$, is the set of all runs accepted by A.

The acceptance condition in this definition is simple — all valid runs are accepted. Of course, the formalism can be modified to cater for more elaborate acceptance conditions, such as conventional accepting states or the various acceptance conditions for ω-automata. The motivation for the present choice is that we regard $\mathcal{RWB}$ as representing the underlying models of programming approaches. As such, languages are seen as generated by, rather than accepted by, a program; indeed, we use these two terms interchangeably.

Next, we define our notion of size, to be used in the analysis of the descriptive succinctness of various variants of RWBAs and other models.

Definition. *The size of an $\mathcal{RWB}$-automaton A with threads $\{\langle Q^i,E^i,\delta^i,q_0^i,R^i,B^i\rangle\}_{i=1}^n$ is $|A| = \sum_{i=1}^n |Q^i| + |\{\langle q,e,\hat{q}\rangle \in \delta^i\}|$, namely the total number of states and transitions in the threads. For simplicity, the requested and blocked events in every state are omitted from the calculation. They contribute no more than $|E|\cdot|Q^i|$ to the size of each thread, and have no effect on the size's order of magnitude as $|E|$ is considered constant.*

6.2.2 Finite Parallel Automata

In order to measure the advantages of $\mathcal{RWB}$ and of other parallel models, we define the following non-parallel model to serve as a reference point:

Definition. *A* deterministic looping automaton *(*DLA*) A is a tuple $\langle Q,E,\delta,q_0\rangle$, where Q is a set of states, E is an alphabet, $\delta \subseteq Q\times E\times Q$ is a deterministic transition relation and $q_0 \in Q$ is an initial state. As it reads an input word, A traverses its states according to δ, in the usual manner. A accepts infinite words, as well as finite words that end in terminal states (states with no successors). A word is rejected if it contains a letter for which there is no matching transition, or if it ends in a non-terminal state. The* language $\mathfrak{L}(A)$ *is the set of words accepted by A, and the* size *of A is $|A| = |Q| + |\{\langle q,e,\hat{q}\rangle \in \delta\}|$, namely the number of states plus the number of transitions in A.*

We now discuss other parallel models, focusing on the three fundamental notions: *nondeterminism* [135] ($\mathcal{E}$-automata) and its dual, *pure parallelism* ($\mathcal{A}$-automata), which when combined

yield *alternating automata* [46], and *cooperative* concurrency ($\mathcal{C}$-automata) [66]. The first two notions take the form of $\exists$- and $\forall$-states in alternating automata, whereas cooperative automata play a role in formalisms and languages such as statecharts [78].

All three features — $\mathcal{E}$, $\mathcal{A}$ and $\mathcal{C}$ — may co-exist. Further, it is shown in [66] that each feature contributes exponentially to the succinctness of the model, independently and additively, so that, e.g., $(\mathcal{E},\mathcal{A},\mathcal{C})$-automata allow for triple-exponentially more succinct representations than is possible without these features. Below we give the definition of $(\mathcal{E},\mathcal{A},\mathcal{C})$-automata; the other models are regarded as restrictions thereof.

Definition. *An* alternating cooperative automaton *(an $(\mathcal{E},\mathcal{A},\mathcal{C})$-*automaton*) over a finite alphabet E is a tuple $M = \langle M^1, M^2, \ldots, M^n, \Phi\rangle$ where each M^i is a triple $\langle Q^i, \delta^i, q_0^i\rangle$. Q^i are pairwise-disjoint state sets and q_0^i are the initial states. $\delta^i \subseteq Q^i \times E \times \Gamma \times Q^i$ are transition relations, where Γ is the set of propositional formulas over the states of all components, $\bigcup_{i=1}^n Q^i$. Elements from Γ serve as* guards*: a transition can be applied only if its guard evaluates to true. For example, for $q_1 \in Q^1$ and $q_2 \in Q^2$, the guard $q_1 \wedge \neg q_2$ evaluates to true precisely when component M^1 is in state q_1 and component M^2 is not in state q_2. Finally, $\Phi \in \Gamma$ is the $\mathcal{E}$-condition — a condition that, when true, implies that the configuration is existential (an $\mathcal{E}$-*configuration*); otherwise, the configuration is universal (an $\mathcal{A}$-*configuration*). In [66], these automata include a termination condition as well, but as we deal with the simple variant of looping automata, we may omit it.*

A configuration *of M is an element of $Q^1 \times Q^2 \times \ldots \times Q^n \times (E^* \cup E^\omega) \times \mathbb{N}$, indicating the state of each component, the (finite or infinite) input word, and the position of M in that word. A configuration c satisfies a guard condition $\gamma \in \Gamma$ if γ evaluates to true when assigned the states of c. Let $\rho = \rho_0\rho_1 \ldots \in E^* \cup E^\omega$ and let $t = \langle q, a, \gamma, p\rangle$ be a transition in δ^i. We say that t is applicable to a configuration $c = \langle q^1, \ldots, q^n, \rho, j\rangle$ if $\rho_j = a$, $q^i = q$ and c satisfies γ. A configuration $\langle p^1, \ldots, p^n, \rho, m\rangle$ is a successor of c if for each i there is a transition $\langle q^i, \rho_j, \gamma^i, p^i\rangle \in \delta^i$ that is applicable to c, and $m = j + 1$.*

A computation of M on input word ρ can be described as a tree. It starts at the initial configuration $\langle q_0^1, q_0^2, \cdots, q_0^n, \rho, 1\rangle$, and reads a letter. If the state has multiple successors, the computation "splits", and progresses in parallel for all possible successor states. The process then continues. Any infinite path in this tree is said to be accepting. *A finite path is accepting iff it ends in a terminal configuration (a configuration with no successors). An $\mathcal{E}$-configuration is accepting iff there* exists *an accepting path starting at that state, whereas an $\mathcal{A}$-configuration is said to be accepting iff* every *path starting at that state is accepting. Word ρ is accepted by M iff the root of its computation tree is accepting.*

If each configuration of M has a single successor (i.e., all transitions are deterministic), we have a $\mathcal{C}$-automaton, which we might call a cooperative automaton. When $n = 1$ it is in fact an $(\mathcal{E},\mathcal{A})$-automaton: an alternating looping automaton. When $n = 1$ and $\Phi = true$, M is

a nondeterministic looping automaton; and when $n = 1$ and $\Phi = false$ it is a universal looping automaton. Finally, when both $n = 1$ and every configuration has a single successor, M is simply a deterministic looping automaton — a DLA.

Definition. *The* size *of an $(\mathcal{E}, \mathcal{A}, \mathcal{C})$-automaton M is defined to be the sum of the sizes of its condition and components; i.e. $|M| = |\Phi| + \sum_{i=1}^{n} |M^i|$, where $|M^i| = |Q^i| + \sum_{\langle q,a,\gamma,p\rangle \in \delta^i} |\gamma|$. A condition's size is defined as the length of the formula that represents it.*

6.2.3 Succinctness Gaps

We next lay out the method of comparing the succinctness of two models. Informally, we say that a computational model $\mathcal{M}_1$ is more succinct than model $\mathcal{M}_2$ if there are programs that have descriptions in $\mathcal{M}_1$ that are significantly smaller than the smallest possible descriptions for those programs in $\mathcal{M}_2$. In this chapter we consider a gap to be significant if it is at least exponential. Following [66], we define upper and lower bounds on gaps in succinctness:

Definition. *Let $\mathcal{M}_1, \mathcal{M}_2$ denote two computational models. We write $\mathcal{M}_1 \stackrel{p}{\to} \mathcal{M}_2$ (resp., $\mathcal{M}_1 \stackrel{\cdot}{\to} \mathcal{M}_2$) if there is a polynomial p (resp., a polynomial p and a constant $k > 1$) such that for any automaton $M_1 \in \mathcal{M}_1$ of size m there is an automaton $M_2 \in \mathcal{M}_2$ such that $\mathfrak{L}(M_1) = \mathfrak{L}(M_2)$, and M_2 is of size no more than $p(m)$ (resp., $k^{p(m)}$). In this case, we say that $\mathcal{M}_1$ is* at most polynomially *(resp.,* exponentially*)* more succinct than $\mathcal{M}_2$.

We write $\mathcal{M}_1 \underset{\cdot}{\to} \mathcal{M}_2$ if there is a family of ω-regular languages L_n, a polynomial p and a constant $k > 1$, such that L_n is accepted by an automaton $M_1 \in \mathcal{M}_1$ of size $p(f(n))$ for some monotonically-increasing function f, but the smallest $M_2 \in \mathcal{M}_2$ accepting it is at least of size $k^{f(n)}$. In this case, we say that $\mathcal{M}_1$ is at least exponentially more succinct than $\mathcal{M}_2$.

6.3 *RWB* and Parallel Automata

In this section, we investigate how $\mathcal{RWB}$-automata fare when considered in the context of $\mathcal{E}$-, $\mathcal{A}$- and $\mathcal{C}$-automata; that is, how the special $\mathcal{RWB}$ idioms relate to the conventional idioms of and- and or-nondeterminism and bounded concurrency. We observe that, of the three models, $\mathcal{RWB}$ seems most closely related to $\mathcal{C}$ — as the threads of an RWBA constitute cooperating components running in parallel — although this cooperation is more limited than in the $\mathcal{C}$ model. The first part of this section validates this observation, by proving that $\mathcal{RWB} \stackrel{p}{\to} \mathcal{C}$, but that $\mathcal{C} \underset{\cdot}{\to} \mathcal{RWB}$. This establishes a firm succinctness relationship between $\mathcal{C}$ and $\mathcal{RWB}$: the former is strictly stronger.

The proof that $\mathcal{C} \underset{\cdot}{\to} \mathcal{RWB}$ revolves around *counting* — a task for which the $\mathcal{C}$ model is particularly suited, as it allows one to count to n using automata of size only $O(\log^2 n)$ [66].

As we prove, in the general case of counting, $\mathcal{RWB}$-automata must be of size n, which is exponentially worse. This result gives rise to the question: does $\mathcal{RWB}$ retain any of $\mathcal{C}$'s power, i.e. is it succinctness-wise better than non-parallel automata?

We answer the question in the affirmative, in two parts. First, we show that $\mathcal{RWB}$ shares some of the power of $\mathcal{C}$ automata; e.g., in certain cases it is possible to count to n with $\mathcal{RWB}$-automata of size $O(\log^2 n \cdot \log\log n)$, and so $\mathcal{RWB} \overset{\cdot}{\rightarrow}$ DLA. Second, we study the relationship between $\mathcal{RWB}$ and the $\mathcal{E}$ and $\mathcal{A}$ models, and show that $\mathcal{RWB}$ can sometimes replace $\mathcal{C}$ in $(\mathcal{E},\mathcal{A},\mathcal{C})$-automata, while preserving that model's descriptive succinctness.

The relationship between $\mathcal{E},\mathcal{A}$ and $\mathcal{C}$ has been extensively studied in [66], where it is shown that they are *orthogonal*, i.e. that their descriptive succinctness is independent and additive. In particular, [66] shows that the $(\mathcal{E},\mathcal{A},\mathcal{C})$ model offers a tight triple-exponential gap in succinctness compared to non-parallel automata. Our proof that the $(\mathcal{E},\mathcal{A},\mathcal{RWB})$ model affords the same triple-exponential gap thus strengthens the original result of [66], as it shows that a model in which components cannot freely observe other components' states, and is thus more encapsulated than $\mathcal{C}$, suffices for obtaining the triple exponential gap.

6.3.1 *RWB*-Automata and $\mathcal{C}$-Automata

Of the three models $\mathcal{E},\mathcal{A}$ and $\mathcal{C}$, it is natural to define $\mathcal{RWB}$ programs in terms of $\mathcal{C}$-automata, as the underlying parallel components of both make transitions that depend on other components. $\mathcal{C}$-automata take the most general form, allowing components to query the internal states of other components. This is established in the following proposition.

Proposition 3. $\mathcal{RWB} \overset{p}{\rightarrow} \mathcal{C}$

Proof. Consider an $\mathcal{RWB}$-automaton A with threads $T_1,\dots,T_n$, each described by a tuple $T_i = \langle Q^i, E^i, \delta^i, q_0^i, R^i, B^i\rangle$. This naturally gives rise to a $\mathcal{C}$-automaton M, in the following manner. The alphabet E of M is the set of all thread events, $E = \cup E^i$. M has n components $M^1,\dots,M^n$, each corresponding to a single thread. Component M^i has the same states and initial state as its corresponding thread, Q^i and q_0^i.

The transition relation of M^i is defined as follows: For every $e \in E$, let

$$\mathfrak{R}(e) = \{q \mid \exists i, q \in Q^i, e \in R^i(q)\}, \quad \mathfrak{B}(e) = \{q \mid \exists i, q \in Q^i, e \in B^i(q)\}$$

denote the set of states in which individual threads request/block event e. For every event $e \in E$ and every thread T_i, for each state $q \in Q^i$ we define a transition in M^i by $\langle q, e, (\bigvee_{u\in\mathfrak{R}(e)} u) \wedge (\neg \bigvee_{v\in\mathfrak{B}(e)} v), \hat{q}\rangle$, where if $e \in E^i$ then $\hat{q} = \bar{\delta}^i(q,e)$, and if $e \notin E^i$ then $\hat{q} = q$. Thus, the transitions imitate the operation of the RWBA, while their guards guarantee that they are applicable iff the event is requested by at least one thread and is blocked by none. Recall that for $e \in E^i$, q_i and

e uniquely determine $\hat{q}_i$ — and so the definition is sound. It is clear that both automata accept the same language. Further, the resulting $\mathcal{C}$-automaton is of size polynomial in the size of A. In particular, we introduce at most $|A|^2$ edges and at most $|A|$ guards, each of size at most $|A|$. □

We next show that the converse does not hold; i.e., that there exists a family of languages that can be expressed succinctly using $\mathcal{C}$-automata, but that the smallest RWBAs that can express them are exponentially larger.

Proposition 4. $\mathcal{C} \not\to \mathcal{RWB}$

Proof. For $n \in \mathbb{N}$, consider the language $L_n = (0+1)^n 0^\omega$. For every n, there exists a $\mathcal{C}$-automaton of size $O(\log^2 n)$ that accepts L_n, as follows. The automaton consists of $\log n$ components, each representing a single bit of a $(\log n)$-bit counter that counts to n. Carries are performed using the guards: bit number $i+1$ moves from state 0 to 1 if and only if all previous bits $1 \ldots i$ are in state 1. A final transition occurs when the counter reaches n, into a state that only allows 0s. As the $\log n$ components can have size $\log n$ because of the transition guards, the automaton is of size $O(\log^2 n)$. See [66] for details.

Now, let us consider the same language in the $\mathcal{RWB}$ model. Suppose that an $\mathcal{RWB}$-automaton A with threads $T_1, \ldots, T_k$ accepts L_n. We show that at least one of these threads has to have $\Omega(n)$ states, thus proving the claim. Intuitively, the proof relies on the fact that while A reads the n-bit prefix of the word the threads cannot use events to communicate between themselves, and so a single thread has to handle the counting up to n.

Suppose, contrary-wise, that all threads have fewer than n states, and consider the word $\rho = 0^{n-1} \cdot 1 \cdot 0^\omega \in L_n$. Examine an arbitrary thread T_i as it reads the $\sigma = 0^{n-1}$ prefix of ρ. By our assumption, thread T_i has fewer than n states. Consequently, by the pigeonhole principle, it has a state s_1 that it will visit at least twice as it reads σ. The portion of the path of states that it traverses between these two visits, denoted $s_1 \xrightarrow{0} s_2 \xrightarrow{0} \ldots \xrightarrow{0} s_{\alpha_i} \xrightarrow{0} s_1$, constitutes a *cycle* of length α_i in the thread's state graph. This holds for every thread T_i, and so all the threads must traverse cycles of lengths $\alpha_1, \ldots, \alpha_n$ as they read σ.

We now use a pumping argument to show that A accepts a word that is not in L_n. Let $\beta = \prod_{i=1}^{n} \alpha_i$. Consider the word $\rho' = 0^{n-1} \cdot 0^\beta \cdot 1 \cdot 0^\omega$, and its prefix $\sigma' = 0^{n-1} \cdot 0^\beta$. The word 0^ω is in L_n, and σ' is a prefix of this word; hence, the automaton cannot reject the input word after reading σ'. However, as the threads are traversing cycles of lengths that divide β, they will each be in the same state after reading σ' as they would be after reading σ. Thus, as they read the $1 \cdot 0^\omega$ suffix of ρ', they would accept the word — just as they would accept ρ. Since $\rho' \notin L_n$, this is a contradiction. □

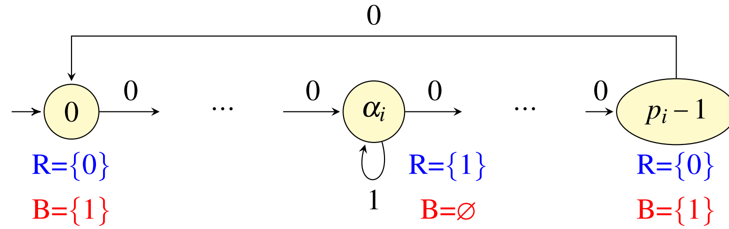
We note that the gap shown by Proposition 4 is tight, in the sense that $\mathcal{C}$-automata are at most (single) exponentially more succinct than $\mathcal{RWB}$-automata. See Proposition 7 in Section 6.4 for the proof.

6.3.2 Counting with Succinct $\mathcal{RWB}$-Automata

Proposition 4 implies that perhaps the $\mathcal{RWB}$ model is not much stronger than non-parallel automata; indeed, for the task of counting, an RWBA requires as many states as a DLA — exponentially many more than a $\mathcal{C}$-automaton requires. However, the main difference in power between $\mathcal{C}$ and $\mathcal{RWB}$ is in the ability of one component in a $\mathcal{C}$-automaton to observe the state of another without any restrictions, whereas in $\mathcal{RWB}$ a marker event (a *sentinel*) must be triggered for such an observation to be made. Thus, when a sentinel is present, the difference in succinctness between $\mathcal{C}$-automata and $\mathcal{RWB}$-automata diminishes greatly:

Proposition 5. For every $n \in \mathbb{N}$, there exists an $\mathcal{RWB}$-automaton A_n that accepts the language $L_n = 0^n 1^\omega$, such that A_n is of size $O(\log^2 n \cdot \log\log n)$.

Proof. We use the first appearance of 1 to mark the end of the counting phase. Let $k \in \mathbb{N}$ be the smallest number such that the first k prime numbers $p_1, \ldots, p_k$ satisfy $\prod_{i=1}^k p_i > n$, and let $\langle \alpha_1, \ldots, \alpha_k \rangle$ be defined by $\alpha_i = n \bmod p_i$ for all $1 \le i \le k$. By the Chinese Remainder Theorem, n is the only integer in the range $[1..\prod_{i=1}^k p_i]$ that has these remainders. Consider the $\mathcal{RWB}$-automaton A_n that has k threads $T_1, \ldots, T_k$, where thread T_i is given by:



The sets of events requested (R) and blocked (B) in each state are listed by that state. In state α_i the thread requests 1 and blocks nothing, and in the other states it requests 0 and blocks 1. To see that this automaton accepts L_n, note that if even one of the threads is not in its respective α_i state, the next event in any accepted word has to be 0, because that thread requests 0 and blocks 1 and no thread ever blocks 0. On the other hand, once all threads are in their α_i states the only requested event is 1, resulting in a 1^ω suffix. Finally, The Chinese Remainder Theorem guarantees us that the first time the threads are all in their α_i states is precisely at the nth step, as required.

Since we chose the smallest k for which $\prod_{i=1}^k p_i > n$, it follows that $k = O(\log n)$. By the Prime Number Theorem we have $p_i = O(i \log i)$. Combining the two, we get that the total size of A_n is indeed $O(\log^2 n \cdot \log\log n)$. □

From Proposition 5 it follows that $\mathcal{RWB} \overset{\cdot}{\dashrightarrow}$ DLA. Further, because $\mathcal{RWB} \overset{p}{\rightarrow} \mathcal{C}$ and $\mathcal{C} \overset{\cdot}{\rightarrow}$ DLA [66], we get that $\mathcal{RWB} \overset{\cdot}{\rightarrow}$ DLA, i.e. that the bound is tight.

6.3.3 Combining *RWB* with $\mathcal{E}$- and $\mathcal{A}$-Automata

One of the main results of [66] establishes a tight triple-exponential gap in succinctness between $(\mathcal{E},\mathcal{A},\mathcal{C})$-automata and DLA. Specifically, there exists a family of languages L_n expressible by $(\mathcal{E},\mathcal{A},\mathcal{C})$-automata of size $O(\log^2 n)$, but that require at least 2^{2^n} states when expressed by a DLA. In this section we quantify the succinctness gap between the $(\mathcal{E},\mathcal{A},\mathcal{RWB})$ model — where $\mathcal{C}$ is replaced by $\mathcal{RWB}$ — and the DLA model.

The semantics of an $(\mathcal{E},\mathcal{A},\mathcal{RWB})$-automaton is as follows: as before, the threads run in parallel, and a transition may occur if the event is requested by at least one thread and is blocked by none. In this model, unlike in $\mathcal{RWB}$, we allow nondeterministic transitions in threads, and so a state may have multiple outgoing transitions labeled with the same event. We also adapt the $\mathcal{E}$-condition to operate in an $\mathcal{RWB}$-like fashion, by allowing threads to request/block that a configuration be universal. Thus, a configuration is existential by default, but becomes universal if this was requested by at least one thread and blocked by none (this $\mathcal{E}$-condition is somewhat arbitrary — other definitions could be used as well). Observe that this form of $\mathcal{E}$-condition is a restriction (i.e., a special case) of the $\mathcal{E}$-condition of [66]. The acceptance criteria is the same as for $(\mathcal{E},\mathcal{A},\mathcal{C})$ (see Definition 6.2.2).

Having shown in Proposition 3 that $\mathcal{RWB} \overset{p}{\to} \mathcal{C}$, it follows that the upper bound of [66] holds; that is, a program in the $(\mathcal{E},\mathcal{A},\mathcal{RWB})$ model will incur at most a triple-exponential blowup when transformed appropriately into a DLA. Next, we show that this bound is tight, by establishing a corresponding lower bound. The family of languages that we use is an adaptation of a similar family from [66]:

$$L_n = \{(0+1+\#)^*\#w\#(0+1+\#)^*\#\$w\bot 0^\omega \mid w \in \{0,1\}^n\} \cup \{(0+1+\#)^\omega\}$$

over the alphabet of $\{0,1,\#,\$,\bot\}$. Intuitively, an automaton that accepts L_n encounters a sequence of words, separated by #s. Then, it encounters a \$, followed by a word w, terminated by $\bot$. The automaton must then decide if this w is of size n, and whether it was encountered before, in the initial sequence of words. If the answer is *yes*, the automaton accepts the word if it ends in an infinite sequence of 0s; otherwise, it rejects the word. The automaton also accepts all words in which the \$ and $\bot$ signs never appear.

Pigeonhole and pumping arguments show that a non-parallel automaton that recognizes the language has to remember, by the time it reaches the \$ sign, all the words of length n that it has encountered previously. Thus, it must have at least 2^{2^n} states [46, 127]. However, an $(\mathcal{E},\mathcal{A},\mathcal{RWB})$-automaton for L_n may be triple-exponentially smaller, as we now show:

Proposition 6. L_n is recognizable by an $(\mathcal{E},\mathcal{A},\mathcal{RWB})$-automaton of size $O(\log^2 n \cdot \log\log n)$.

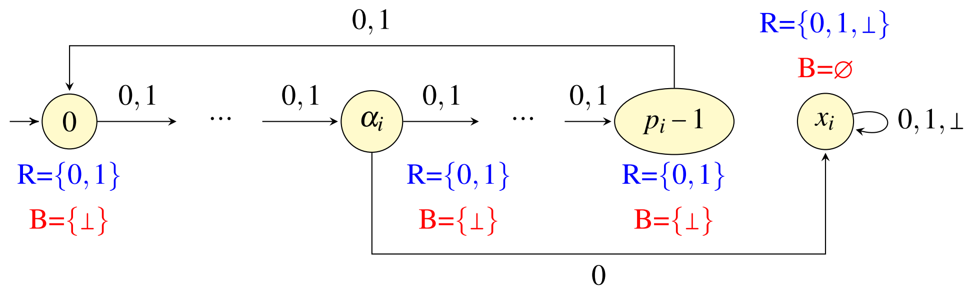
Proof. We provide here only the core of the proof. Our strategy, inspired by [66], is as follows: in words where the \$ and $\bot$ signs appear, the automaton's nondeterminism is used to "guess"

when the first instance of w is encountered. Then, universality is used to compare all n bits of the two occurrences of w simultaneously. Finally, ensuring that both copies of w are of length n, and performing the necessary counting to compare each pair of bits, is performed efficiently by the automaton's $\mathcal{RWB}$-threads.

More explicitly, the $\mathcal{RWB}$ idioms are used for 3 tasks: (1) verifying that the first occurrence of w is of size n; (2) verifying that the second occurrence of w is of size n; and (3) comparing a single pair of bits in the two occurrences of w. Because task (3) is performed universally for all n bits, it ensures that the two occurences of w are equal. For task (3) the automaton counts to n, but is suspended on # and resumed on \$. Thus, when the counting is finished, the next symbol should match the symbol on which the counting was started.

Tasks (1) and (2) can be performed succinctly by an RWBA, as both occurrences of w in L_n are terminated by a sentinel — # or $\bot$. Thus, the construction from Section 6.3.2 suffices. The automaton size these tasks require is $O(\log^2 n \cdot \log\log n)$. Task (3), however, requires counting *without* a sentinel, which — according to the proof of Proposition 4 — requires an $\mathcal{RWB}$-automaton of size $\Omega(n)$. However, we now show that in an $(\mathcal{E},\mathcal{A},\mathcal{RWB})$-automaton such counting can actually be performed succinctly, by leveraging the $\mathcal{E}$ and $\mathcal{A}$ idioms.

Let $k \in \mathbb{N}$ be the smallest number such that the first k prime numbers, $p_1,\ldots,p_k$, satisfy $\prod_{i=1}^{k} p_i > n$. Let $\langle \alpha_1,\ldots,\alpha_k \rangle$ be the tuple of remainders, i.e., $\alpha_i = n \bmod p_i$ for all $1 \le i \le k$. By the Chinese Remainder Theorem, these remainders uniquely determine n in the range $[1..\prod_{i=1}^{k} p_i]$. Suppose, without loss of generality, that the symbol in the first occurrence of w was 0. Then, our goal is to count to n and verify that we reach another 0. Consider an $\mathcal{RWB}$-automaton with k threads $T_1,\ldots,T_k$, where T_i is given by:



All states $0,\ldots,p_i-1$ request both 0 and 1 and block $\bot$; state x_i requests $0,1$ and $\bot$. Finally, thread T_i requests that the global configuration be universal if and only if it is at state x_i. The details of suspending the count on # and resuming it on \$ are omitted from the figure, to reduce clutter; this can be performed by associating each state $s \in \{1,\ldots,p_i\}$ with an auxiliary state s', and having the appearance of # send the thread to s', where it loops, until a later appearance of \$ sends it back to s. If the \$ and $\bot$ signs do not appear, then the word is accepted, as it has the form $(0+1+\#)^\omega$, which we included in L_n.

Intuitively, the automaton works as follows. All threads traverse their loops, counting to n.

While in these loops, a $\bot$ symbol causes the word to be rejected. Hence, the only way a word that has a $\bot$ sign can be accepted is if all threads escape their loops before reaching $\bot$. The only way to escape the counting loops is through the α states. If thread T_i reaches state α_i and reads a 0 symbol, it may escape its loop, assuming the transition is existential; if it is universal, one branch of the thread will remain in the loop, and will reject the word.

The escape transition remains existential until some thread has used it to escape. Afterwards, that thread will remain in its x_i state, requesting that all successive configurations be universal. Hence, all threads must traverse the transition from α_i to x_i simultaneously in order for the word to be accepted. This can only happen if all threads are in their respective α states — which, by the Chinese Remainder Theorem, only occurs at index n — and if the next symbol is the required 0. Hence, since this testing is performed universally for all symbols in w, the word is rejected if even one pair of matching symbols differs.

We stress that this solution is in line with our previous observations that $\mathcal{RWB}$ is weaker than $\mathcal{C}$, in that $\mathcal{RWB}$ cannot succinctly count without a sentinel. In this construction, the behavior threads use the ability of the $\mathcal{E}$-condition semantics to peek into the states of other threads, thus achieving some of the power of the $\mathcal{C}$-automaton guards, and enabling it to count succinctly, even without a sentinel.

As in Proposition 5, analysis shows that the automaton is of size $O(\log^2 n \cdot \log\log n)$. $\square$

We have thus established the triple-exponential succinctness gap between $(\mathcal{E},\mathcal{A},\mathcal{RWB})$ and DLA. While $(\mathcal{E},\mathcal{A},\mathcal{RWB})$ is not a practical programming model, we observe that, combined with the results of [66] and Proposition 3, this result immediately establishes a succinctness hierarchy concerning other, more practical models, such as $(\mathcal{E},\mathcal{RWB})$ and $(\mathcal{A},\mathcal{RWB})$. These corollaries are depicted and explained in Figure 6.1. In particular, these results indicate that the $\mathcal{RWB}$ idioms of requesting, blocking and waiting for events provide a succinctness advantage that is additive and independent of the succinctness provided by the $\mathcal{E}$ and $\mathcal{A}$ idioms — and that the $\mathcal{RWB}$ idioms are not just those of $\mathcal{E}$- or $\mathcal{A}$-automata in disguise. While similar results were previously shown for the $\mathcal{C}$ model [66], our results are stronger as they show that a limited version of $\mathcal{C}$ already suffices to uphold the hierarchy.

From a software-engineering point of view, $\mathcal{C}$-automata afford their succinctness by allowing each component to be aware of the internal state of each of the other components; this liberal awareness is not provided in the $\mathcal{RWB}$ model, resulting in increased module encapsulation, which is usually considered desirable (see, e.g., [132]).

6.4 Contributions of the Request, Wait, and Block Idioms

Whereas Section 6.3 was dedicated to comparing $\mathcal{RWB}$ to other parallel models succinctness-wise, in this section we focus on its internal structure. We study each of its main idioms of

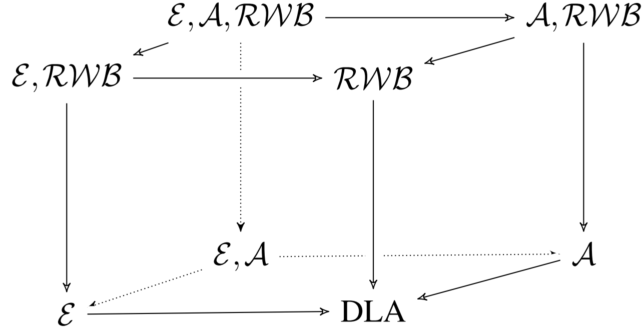


Figure 6.1: The succinctness hierarchy involving the $\mathcal{E}$, $\mathcal{A}$ and $\mathcal{RWB}$ models, and their combinations. Arrows indicate tight exponential gaps in succinctness. By Proposition 6, the $(\mathcal{E},\mathcal{A},\mathcal{RWB})$ model is at least triple exponentially more succinct than the DLA model; and, applying Proposition 3, it is also at most as succinct as the $(\mathcal{E},\mathcal{A},\mathcal{C})$ model. Combining this with the fact that $(\mathcal{E},\mathcal{A},\mathcal{C})$ is triple exponentially more succinct than DLA [66], we get that the same holds for $(\mathcal{E},\mathcal{A},\mathcal{RWB})$. Thus, any path along the edges of the depicted cube, starting at $(\mathcal{E},\mathcal{A},\mathcal{RWB})$ and ending at DLA, must include precisely 3 exponential gaps. The tight exponential gaps depicted in the figure then follow from known results regarding alternating automata and $\mathcal{C}$-automata [66], combined with Proposition 3.

requesting, waiting-for, and blocking events, and quantify their contribution to the succinctness afforded by $\mathcal{RWB}$ as a whole. Towards this end, we define the following sub-models:

1. The $\mathcal{WB}$ model: Requesting is omitted. Any event that is not blocked can be triggered. Waiting-for and blocking are allowed. This model can be viewed as having all threads request all events in each state, which, in the notation of Definitions 6.2.1 and 6.2.1, corresponds to $R^i(q^i) = E^i$ for every state $q^i \in Q^i$ of thread T_i, for every i.

2. The $\mathcal{RB}$ model: Waiting is omitted; requesting and blocking are allowed. Threads are not informed of events they did not request, and cannot change states when such events are triggered. Formally, for every T_i, if $e \notin R^i(q)$ then $\delta^i(q,e) = q$.

3. The $\mathcal{RW}$ model: Blocking is omitted. Requesting and waiting-for are allowed, and any requested event may be triggered. Formally, $B^i(q) = \emptyset$ for every state q and T_i.

We begin by establishing a simple upper bound:

Proposition 7. For any $\mathcal{M}_1,\mathcal{M}_2 \in \{\mathcal{RWB},\mathcal{WB},\mathcal{RB},\mathcal{RW}\}$, $\mathcal{M}_1 \dot{\rightarrow} \mathcal{M}_2$

Proof. We show that for any two models $\mathcal{M}_1,\mathcal{M}_2 \in \{\mathcal{RWB},\mathcal{WB},\mathcal{RB},\mathcal{RW}\}$, it holds that $\mathcal{M}_1 \dot{\rightarrow} \mathcal{M}_2$.

Every $\mathcal{RB}$- or $\mathcal{RW}$-automaton is also an $\mathcal{RWB}$-automaton, in which, at every synchronization point, the waited-for or blocked events set are empty, respectively. A $\mathcal{WB}$-automaton can

also be regarded as an $\mathcal{RWB}$-automaton, where all threads request all events at each synchronization point. Hence, $\mathcal{M}_1 \overset{p}{\rightarrow} \mathcal{RWB}$.

By Proposition 3, we know that $\mathcal{RWB} \overset{p}{\rightarrow} \mathcal{C}$. By [66], we know that $\mathcal{C} \ \dot{\rightarrow}\ $ DLA. Finally, a DLA can be translated into an $\mathcal{M}_2$-automaton with a single thread that imitates the DLA, as follows. The thread has the same states and transitions as the DLA, and:

- For $\mathcal{M}_2 = \mathcal{WB}$, in each state the thread blocks any events for which there are no transitions, and waits for the events of all outgoing transitions.
- For $\mathcal{M}_2 = \mathcal{RB}$, $\mathcal{M}_2 = \mathcal{RW}$ or $\mathcal{M}_2 = \mathcal{RWB}$, in each state the thread only requests events for which there are outgoing transitions, blocking no events.

It follows that DLA $\overset{p}{\rightarrow} \mathcal{M}_2$. Thus, putting these together yields:

$$\mathcal{M}_1 \overset{p}{\rightarrow} \mathcal{C} \ \dot{\rightarrow}\ \text{DLA} \overset{p}{\rightarrow} \mathcal{M}_2$$

And so $\mathcal{M}_1 \ \dot{\rightarrow}\ \mathcal{M}_2$, as needed. □

Next, we establish tight bounds on the difference in succinctness of every pair of these models, as depicted in Figure 6.2.

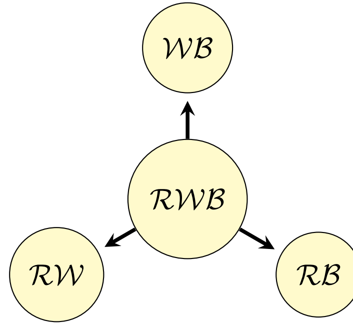


Figure 6.2: The descriptive succinctness of the $\mathcal{RWB}$ model compared to that of the $\mathcal{WB}$, $\mathcal{RW}$ and $\mathcal{RB}$ models. Each directed arrow indicates a tight exponential succinctness advantage of the source over the destination, but no such advantage in the reverse direction, i.e., a reverse translation is always possible with only a polynomial blowup.

We begin by proving that the $\mathcal{RWB}$ model is exponentially more succinct than the $\mathcal{WB}$ variant, i.e., that the event requesting idiom exponentially improves the succinctness of the model.

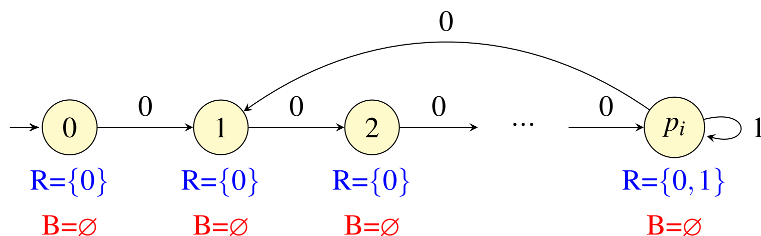
Proposition 8. $\mathcal{RWB} \ \underset{\cdot}{\rightarrow}\ \mathcal{WB}$

Proof. Let $n \in \mathbb{N}$, and let $k \in \mathbb{N}$ be the smallest number such that the first k prime numbers, $p_1, \ldots, p_k$, satisfy $\prod_{i=1}^{k} p_i > n$. Define the family of languages $L_n = \{\ell_1 \ell_2 \ldots\}$ by:

$$\ell_j = \begin{cases} 0 \text{ or } 1 & ; \quad \exists i \text{ such that } p_i \mid \#_0(\ell_1 \ldots \ell_{j-1}) \\ 0 & ; \quad \text{otherwise} \end{cases}$$

Here $\#_0(\ell_1, \ldots, \ell_{j-1})$ is the number of 0s that have appeared in the word so far. The jth event can be either 0 or 1 if there is a p_i which divides this number.

In the $\mathcal{RWB}$ model, this language is accepted by an automaton with k threads, $T_1, \ldots, T_k$, where T_i is given by:



In state p_i the thread requests 0 and 1, and in all other states it only requests 0. The size analysis is the same as in the proof of Proposition 5, and the automaton is of size $O(\log^2 n \cdot \log\log n)$.

Consider a $\mathcal{WB}$-automaton that accepts this language and the word $\sigma = 0^\omega \in L_n$. Assume that all threads have less than n states. By the pigeonhole principle they are traversing cycles in their respective transition systems as the automaton reads σ.

There are infinitely many indices in which the triggering of 1 would result in a word (either finite or infinite) that is not in L_n being accepted by the automaton. Hence, triggering 1 has to be prevented in these indices, which, in the $\mathcal{WB}$ model, means it has to be blocked infinitely often. Therefore, there is at least one thread T_j whose cycle contains a state q in which 1 is blocked. Let β denote the length of that cycle. By our assumption, $\beta < n$. Consequently, there is at least one prime p_i such that p_i and β are coprime. By the Chinese Remainder Theorem, the automaton will eventually reach a point in the run where p_i divides the number of zeroes encountered so far, and so 1 should be allowed, but in which thread T_j is in state q, blocking 1 — thus preventing a word that is in the language from being accepted by the automaton. Hence, there must be at least one thread with n or more states, proving the claim. □

This proof affords some insight into the power of the requesting idiom. Particularly, requesting allows us to succinctly express *or* conditions — i.e., that an event only be triggered if a disjunction of conditions holds.

We now prove that the waiting idiom also affords exponential succinctness:

Proposition 9. $\mathcal{RWB} \overset{\exp}{\to} \mathcal{RB}$

Proof. Note that in the $\mathcal{RB}$ model, since threads do not wait for anything unless they explicitly request it, if a thread has no requests at some synchronization point it remains in that state forever, either blocking some events forever, or doing nothing at all.

Let $n \in \mathbb{N}$, and consider the family of singleton languages $L_n = 0^n 1^\omega$. In Section 6.3.2 we saw that there exists an $\mathcal{RWB}$-automaton of size $O(\log^2 n \cdot \log\log n)$ that accepts L_n.

Now, suppose that an $\mathcal{RB}$-automaton accepts L_n, and that all its threads have less than n states. Consider the automaton after reading the input 0^n. As all threads have less than n states, they are all traversing cycles in their transition systems where all the edges are labeled 0. We call these cycles 0-cycles. In the $\mathcal{RB}$ model, these cycles can be of two kinds:

1. Single-state cycles, where the thread does not request 0; hence, a 0 event being triggered leaves the thread in that state.

2. Cycles in which all states request 0; in these cycles, the thread moves to a new state whenever 0 is triggered.

Neither kind of cycle can have a state that blocks 0 inside it; otherwise the 0^n prefix would already be rejected, although it is a prefix of a word in L_n. Also, there is at least one thread in a cycle that requests 0s; otherwise the automaton would be stuck — again, rejecting a word in L_n.

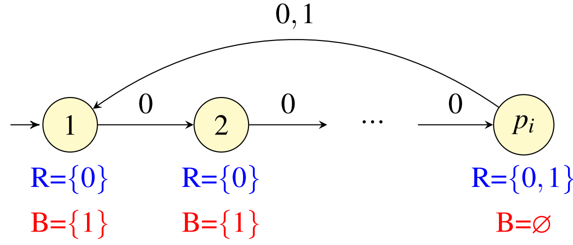
Let us see what happens when the automaton reads the word 0^ω. After reading the 0^n prefix, the threads are already traversing their cycles, in which 0 is never blocked. At least one thread is constantly requesting 0 in every state of its cycle. Hence, the threads will continue to traverse their cycles indefinitely as they read 0^ω, accepting the word although it is not in L_n. □

The language used for this proof illustrates the power of the waiting idiom. In the basic construction of Section 6.3.2, each thread would count modulo some prime number, and would, upon the correct remainder, request 1. However, that thread would also wait for a 0 event, thus letting other threads supersede it; if one of them determined that it was not yet time to trigger a 1, they would block 1 and request 0. Without the wait-for idiom, however, a thread cannot observe events it did not request, preventing this sort of inter-thread cooperation.

We now show that $\mathcal{RWB}$ is exponentially more succinct than the $\mathcal{RW}$ variant, i.e. event blocking also yields exponential succinctness. We consider this result to be particularly interesting, as blocking is perhaps the least common, or most special, idiom of $\mathcal{RWB}$.

Proposition 10. $\mathcal{RWB} \overset{\exp}{\rightarrow} \mathcal{RW}$

Proof. Let $n \in \mathbb{N}$, and let $k \in \mathbb{N}$ be the smallest number such that the first k prime numbers, $p_1, \ldots, p_k$, satisfy $\prod_{i=1}^{k} p_i > n$. We prove the claim using the family of languages $L_n = (0^{N-1}(0+1))^\omega$, where $N = \prod_{i=1}^{k} p_i$. In the $\mathcal{RWB}$ model, this language is accepted by an automaton with k threads $T_1, \ldots, T_k$, where T_i is given by:



In state p_i the thread requests both 0 and 1, and in the other states it requests 0 but blocks 1. Thus, 0 may always be triggered, but 1 may be triggered only when all threads are in their respective p_i states, as required. The size analysis is the same as in the proof of Proposition 5, and the automaton is of size $O(\log^2 n \cdot \log\log n)$.

Assume we have an $\mathcal{RW}$-automaton that accepts this language, and consider the word $0^\omega \in L_n$. Again, by the pigeonhole principle, the threads will, during this run, indefinitely traverse cycles within their respective transition systems. Clearly, event 1 has to be requested infinitely often throughout the run, in order to allow all the words of L_n to be accepted. Therefore, in at least one of the threads' cycles, there will be a state in which 1 is requested. Denote the length of that cycle by α.

In the $\mathcal{RW}$ model, threads cannot block events. Thus, every time 1 is requested it may be triggered. Since there cannot exist two words in the language that have 1s in indices that are less than N steps apart, it follows that $\alpha \geq N > n$. Thus, the total size of the $\mathcal{RW}$-automaton is at least n. □

The language used for the proof gives some intuition as to the power of the blocking idiom. Particularly, it shows that blocking can succinctly enforce *and* conditions — e.g., that an event is not blocked iff it gives the correct remainder for *all* the primes.

6.5 Comparing the Request, Wait, and Block Idioms

While Section 6.4 was dedicated to studying the contribution of the request, wait and block idioms to the $\mathcal{RWB}$ models as a whole, in this section we examine how the $\mathcal{RWB}$ idioms fare with respect to each other. For example, can requesting be replaced by blocking without having to pay with an exponential decrease in succinctness? Our results, illustrated in Figure 6.3, show that the $\mathcal{WB}$ and $\mathcal{RW}$ models, and also the $\mathcal{RB}$ and $\mathcal{RW}$ models, are incomparable — i.e., there can be exponential gains in both directions. Also, we prove that the $\mathcal{WB}$ model is weaker than the $\mathcal{RB}$ model, and so, in a way, requesting outpowers waiting. We also show that each of the $\mathcal{WB}$, $\mathcal{RB}$ and $\mathcal{RW}$ models is exponentially more succinct than DLA.

Proposition 11. $\mathcal{RB} \overset{\cdot}{\to} \mathcal{WB}$ and $\mathcal{WB} \overset{p}{\to} \mathcal{RB}$

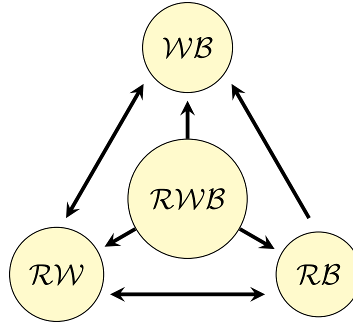


Figure 6.3: The descriptive succinctness of the $\mathcal{WB}$, $\mathcal{RW}$ and $\mathcal{RB}$ models compare to $\mathcal{RWB}$, and also compared to each other. A bi-directional arrow indicates a tight exponential gap in succinctness in both directions — either model may be more succinct than the other. As before, a directed arrow indicates a tight exponential succinctness advantage of the source over the destination, but no such advantage in the reverse direction, i.e., a reverse translation is always possible with only a polynomial blowup.

Proof. First, observe that the proof of Proposition 8, showing that $\mathcal{RWB} \underset{\cdot}{\rightarrow} \mathcal{WB}$, used an $\mathcal{RWB}$-automaton that did not utilize the waiting idiom; hence, the same proof applies here.

For the other direction, note that any $\mathcal{WB}$-automaton can be trivially translated into an $\mathcal{RB}$-automaton. In $\mathcal{WB}$, a thread can be regarded as constantly requesting all events, and waiting for only some of them. Such a thread can be converted into an equivalent thread in the $\mathcal{RB}$ model: it continues to request all events, and reacts (changes state) only to events for which the original thread waited. The resulting automaton accepts the same language as the original automaton, and has the same size. □

In light of Proposition 11, Proposition 9 is particularly surprising. It shows that although the $\mathcal{WB}$ model is a weaker than the $\mathcal{RB}$ model, their combination affords greater succinctness than either of them affords separately.

Proposition 12. $\mathcal{RW} \underset{\cdot}{\rightarrow} \mathcal{WB}$ and $\mathcal{WB} \underset{\cdot}{\rightarrow} \mathcal{RW}$

Proof. For the first direction: The proof of Proposition 8, showing that $\mathcal{RWB} \underset{\cdot}{\rightarrow} \mathcal{WB}$, used an $\mathcal{RWB}$-automaton that did not utilize the blocking idiom; hence, the same proof applies here.

For the other direction, we use the proof of Proposition 10, showing that $\mathcal{RWB} \underset{\cdot}{\rightarrow} \mathcal{RW}$. The original proof was based on an automaton in which all threads requested all the events at every state, except those that they blocked. Hence, moving them to the $\mathcal{WB}$ model would produce the same language. □

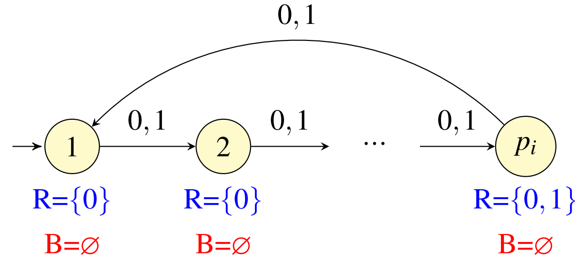
Proposition 13. $\mathcal{RB} \underset{\cdot}{\rightarrow} \mathcal{RW}$ and $\mathcal{RW} \underset{\cdot}{\rightarrow} \mathcal{RB}$

Proof. For the first direction: The proof of Proposition 10, showing that $\mathcal{RWB} \not\to \mathcal{RW}$, used an $\mathcal{RWB}$-automaton that did not wait for any events (i.e., the state-changes of each thread were caused solely by triggering events that the thread itself had requested). Hence, the same proof applies here.

We now prove that $\mathcal{RW} \not\to \mathcal{RB}$. Let $n \in \mathbb{N}$, and let $k \in \mathbb{N}$ be the smallest number such that the first k prime numbers $p_1, \ldots, p_k$ satisfy $\prod_{i=1}^{k} p_i > n$. Define the family of languages $L_n = \{\ell_1 \ell_2 \ldots\}$ by:

$$\ell_j = \begin{cases} 0 \text{ or } 1 & ; \quad \exists i \text{ such that } p_i \mid j \\ 0 & ; \quad \text{otherwise} \end{cases}$$

In the $\mathcal{RW}$ model, this language can be accepted by an automaton with k threads, $T_1, \ldots, T_k$. Thread T_i corresponds to prime p_i, and is given by:



Here, in every state the thread requests 0, and in state p_i (and only in that state) it also requests 1. Observe that 0 is always requested, and that 1 is requested if and only if the index is divisible by one of the primes. In the $\mathcal{RW}$ model there is no blocking, and so requested events can always be triggered — generating the desired language. As in previous proofs, the size of the automaton is $O(\log^2 n \cdot \log\log n)$.

Now, observe what happens in the $\mathcal{RB}$ model. Suppose towards contradiction that an $\mathcal{RB}$-automaton accepts the language L_n, and that all its threads have less than n states. Denote $N = \prod_{i=1}^{k} p_i$. We will show that there exists a sequence of N consecutive integers, $M, M+1, \ldots, M+N-1$, such that for every $M \le i \le M+N-1$, there exists a run of the automaton in which 1 is triggered at index i. This will form a contradiction to the definition of L_n, proving the claim. The rest of the proof shows how to construct these runs.

Observe the $\mathcal{RB}$-automaton as it reads the finite sequence 0^n. By the pigeonhole principle, this sequence causes all threads to traverse cycles where all the edges are labeled 0. (0-cycles). Note that this is true regardless of the starting configuration of the automaton — i.e., at any point during the run, reading a 0^n sequence causes all threads to traverse 0-cycles.

We begin with the following observation regarding 0-cycles:

Observation 1. When all threads are traversing 0-cycles, 1 cannot be blocked in any of the cycles' states.

Proof. Observe some thread T_j and the 0-cycle it is traversing. Let α denote the length of that cycle. Since T_j has less than n states, and $\prod_{i=1}^{k} p_i > n$, it follows that there exists some i such that α and p_i are coprime. Suppose the remainder of the input word is just an infinite sequence of zeroes, 0^ω. By the Chinese remainder theorem, if 1 was blocked in any of the states of the 0-cycle, it would eventually get blocked at an index divisible by p_i — and the automaton would reject a word it is supposed to accept, which is a contradiction. Hence, 1 is never blocked in any of the 0-cycles. □

Clearly, when all threads are traversing their 0-cycles, 1 has to be requested infinitely often. This means that at least one thread has, within its cycle, a state that requests 1. We now make the following observation:

Observation 2. Suppose all threads are traversing 0-cycles, and let T be one of the threads that request 1 infinitely often. Then eventually 1 will be requested by another thread when T is at a state in which it does not request 1.

Proof. Suppose towards contradiction that this is not the case; i.e. that T requests 1 whenever the index is divisible by one of the primes. Denote T's cycle length by α. Because T has less than n states, there is some i such that α and p_i are coprime. By the Chinese remainder theorem, if the stream of 0s continues, every state in T's cycle will eventually be reached when p_i divides the run index. Hence, since T must request 1 whenever the index is divisible by p_i, *every* state in its cycle must request 1. Consequently, as 1 is never blocked within the 0-cycles (Observation 1) there exists a finite index ℓ such that for every $\ell' > \ell$ the word $0^{\ell'}1$ is a prefix of a word accepted by the automaton — which is a contradiction. Hence, there must be a time in which 1 is requested, but T does not request it. □

We now use these two observations in order to construct the aforementioned N runs.

The first run we examine, denoted ρ_0, is the one associated with input word 0^ω. All threads eventually traverse 0-cycles. Observe a fixed thread T that requests 1 infinitely often. Let β denote the first index, after entering its cycle, in which T requests 1, and let α denote its cycle length. Then T will request 1 at all indices $\beta + \alpha \cdot t$, for all $t \in \mathbb{N}$.

The second run, ρ_1, is constructed as follows. The input word starts with 0^n, in order to have all threads traverse 0-cycles. We now apply Observation 2: we "feed" the automaton more 0s, until reaching an index in which 1 is requested by some thread and not by T. We then trigger 1, and afterwards continue with 0^ω. Since there is no waiting in the $\mathcal{RB}$ model, and since T did not request the triggered 1, T is oblivious to the fact that 1 has been triggered. For the remainder of the run, the thread T is *delayed* by one index position. This implies that

there is some constant T_1, such that, in run ρ_1, for every $t > T_1$, thread T requests 1s at indices $\beta + \alpha \cdot t + 1$.

We continue iteratively. Run ρ_2 is produced as follows. Up to the 1 triggered in ρ_1 the prefixes of the two runs are identical. We then feed more 0s, trigger 1 again at an index divisible by a prime when T does not request 1, and continue with 0^ω. This results in a constant T_2, such that in run ρ_2 for every $t > T_2$, thread T requests 1s at indices $\beta + \alpha \cdot t + 2$.

The process is repeated N times, where each ρ_k is identical to ρ_{k-1} up to the last 1, and a 1 is then added where T does not request it. We get a series of constants $T_{N-1} > T_{N-2} > \ldots > T_2 > T_1 > 0$, such that for every $t > T_i$, there exists a run in which T requests 1 at indices $\beta + \alpha \cdot t + i$, and 1 is not blocked.

Set $t = T_{N-1} + 1$, so that it is larger than all the constants $T_1, \ldots, T_{N-1}$ above. For every $1 \le i \le N$, there is some run of the automaton, in which thread T requests 1 at index $\beta + \alpha \cdot t + i$, and 1 is not blocked at that index. This implies that there are N consecutive integers that are each divisible by at least one of the primes $p_1, \ldots, p_n$, which is a contradiction. It follows that our initial assumption is false; i.e. that there is some thread with at least n states, proving the gap in succinctness as needed. □

Note that the above proof that $\mathcal{RW} \dashrightarrow \mathcal{RB}$ also constitutes a stronger proof for Proposition 9, i.e. that $\mathcal{RWB} \dashrightarrow \mathcal{RB}$.

From Propositions 11, 12 and 13 we also obtain the following corollary:

Corollary. *$\mathcal{WB} \dashrightarrow$ DLA, $\mathcal{RB} \dashrightarrow$ DLA and $\mathcal{RW} \dashrightarrow$ DLA.*

Proof. By using Propositions 11, 12 and 13, we get that for every $\mathcal{M}_1 \in \{\mathcal{WB}, \mathcal{RB}, \mathcal{RW}\}$ there exists a $\mathcal{M}_2 \in \{\mathcal{WB}, \mathcal{RB}, \mathcal{RW}\}$ such that $\mathcal{M}_1 \neq \mathcal{M}_2$ and that $\mathcal{M}_1 \dashrightarrow \mathcal{M}_2$. Further, the DFA model is directly embeddable in model $\mathcal{M}_2$ (see explanation in the proof of Proposition 7). The claim follows. □

6.6 Related Work

In this chapter we focused on studying the $\mathcal{RWB}$ concurrency idioms from a succinctness point of view. For a software-engineering oriented comparison between these $\mathcal{RWB}$ idioms (in the context of Behavioral Programming) and other programming models, see [90] and references therein. Below we discuss some notable related work on descriptive succinctness.

Starting with [135], extensive comparative analysis of expressiveness and succinctness in various models of computations has been carried out. Examples include Büchi, Streett, and Emerson and Lei automata [140], two-way finite automata [141, 33], sweeping automata [101],

and — most relevant to the present work — cooperative automata [66, 99]. Expressiveness and succinctness in timed automata are studied in [14].

The issue of counting to n using unary automata, which played a central role in Section 6.3, was raised in [127] and has been studied extensively. It is well known that counting requires $\Theta(n)$ states in deterministic and nondeterministic finite automata. As any deterministic unary automaton with n states has an equivalent alternating automaton with $O(\log n)$ states [113], it follows that alternating automata can count with size $O(\log n)$. [110] shows a $\Theta(\sqrt{n})$ bound for counting with universal automata, whereas cooperating automata can count to n with size $O(\log^2 n)$ [66]. Counting in other automata types has also been studied: one-switch alternating automata, for instance, count to n with $O(\log^2 n \cdot \log\log n)$ states [33].

6.7 Conclusion and Future Work

In this work we set out to analyze the descriptive succinctness afforded by various concurrent programming idioms. Our motivation was the strong connections between the succinctness of the software's description and its simplicity, maintainability, reliability, analysis and verification. We focused on three basic and common idioms — requesting, blocking and waiting for events. We began by analyzing the succinctness of the three idioms taken together, showing that the $\mathcal{RWB}$ model can be translated into cooperating automata with only a polynomial increase in size, but that the converse translation might incur an exponential blowup. Hence, the $\mathcal{RWB}$ model, in which components cannot directly query the state of other components, is strictly less succinct than the $\mathcal{C}$ model. We continued by showing that $\mathcal{RWB}$ can nevertheless succinctly perform non-trivial tasks, that its succinctness is independent and additive to that of the $\mathcal{E}$- and $\mathcal{A}$-automata, and that $(\mathcal{E},\mathcal{A},\mathcal{RWB})$-automata are triple-exponentially more succinct than DLA — making them in some cases as strong as the more general $(\mathcal{E},\mathcal{A},\mathcal{C})$-automata of [66]. This result established a succinctness hierarchy, indicating the succinctness advantages of models like $(\mathcal{E},\mathcal{RWB})$ and $(\mathcal{A},\mathcal{RWB})$. These findings show that the $\mathcal{RWB}$ model, which offers stronger encapsulation and has additional software-engineering advantages over $\mathcal{C}$-automata [90], can sometimes retain the succinctness of the more general model.

We then quantified the contribution of the requesting, waiting-for and blocking idioms to the succinctness of $\mathcal{RWB}$ as a whole. We proved that they are each vital to the succinctness of $\mathcal{RWB}$, as the removal of either may cause an exponential blowup in size, and hence that they do not subsume one another.

The contribution of the present work is thus in substantiating formally the advantages for software engineering that the $\mathcal{RWB}$ idioms, in a variety of programming languages, appear to have; and also in gaining insights into the particular tasks for which each of the idioms is particularly useful. One natural future research direction is to study the succinctness afforded

by additional idioms for concurrent programming, such as the lock-step progression idiom, by which all components process a triggered event simultaneously. Another direction is to further study the gap in succinctness between $\mathcal{RWB}$ and $\mathcal{C}$-automata; e.g., to characterize additional tasks, besides counting, in which $\mathcal{C}$'s superiority is manifested.

Chapter 7

Compositional Verification of *RWB* Programs

7.1 Introduction

The development and verification of large scale component-based software systems pose many challenges. During development, programmers working on separate modules may often be unaware of the fine details of inter-module interfaces, or these interfaces may not be well defined; and the *state explosion* problem prevents the model checking of the entire system, which could discover the resulting errors at the system level.

In recent decades, a prominent approach for tackling these issues has been that of compositional development, based on well defined interfaces, assume-guarantee contracts, and verification. This approach calls for defining the interfaces between modules, programming each module separately, and then verifying that each module guarantees certain properties under certain assumptions on its environment. The modules' verified properties are then combined in order to deduce system-wide properties. For some of the notable compositional approaches see, e.g., [128, 105, 133, 120, 53, 76, 58, 97, 35, 70, 73, 123, 64, 54, 17, 94, 65], which are reviewed in Section 7.4. A recent survey of behavioral interface specification languages [94] points to outstanding research challenges in this area, including how to deal with parallel programs, to tie module specifications to requirements specifications, and to further automate the verification process. This work aims to contribute to the pursuit of these challenges.

We present a compositional approach and a tool chain for building and verifying reactive systems. In our proposed approach, properties of individual modules are formalized and then used for automated verification of the composite system. The formalized module properties serve also as part of system documentation and can be useful for other development tasks. Two key elements of the approach are (1) a specification and programming formalism that enables programming different aspects of system behavior independently of each other; (2) means to

infer composite system properties from formally-specified properties of individual modules.

For system specification we use the formalism of *behavioral programming* (BP) [90] and for inferring system properties from module properties the Z3 SMT solver [60]. We believe, however, that the approach can be based on to other methods too as long as they cater to programming separate aspects in isolation. It can also be used with other inference tools and theorem provers.

Our goal is to improve the process of compositional development, documentation, and verification, by proposing ways that in some cases will give rise to efficient verification. Similarly important is providing tools for formal documentation of the module properties.

The methodology behind the approach consists of the following steps (which can be performed in almost any order):

Specification: Document in natural language each of the desired and undesired aspects of system behavior.

Module Properties: Design the system such that each aspect of the behavior will be implemented by its own separate program module (or a set thereof). Formalize the properties of each such module (or set of modules) as formulas in a solver or proof assistant.

Environment Properties: Similarly, formalize the description of external environment behavior and encode it in the solver or proof assistant.

Composition Properties: Specify the application-independent module-composition rules as formulas of the solver or proof assistant.

Domain Properties: Specify the application-independent domain knowledge in the solver or proof assistant.

Prove System Properties: Use the solver or proof assistant to prove that, given the module, composition and domain properties, the system will behave correctly.

System Implementation: For each independent aspect of the behavior, develop the code of the corresponding module(s). If needed for simulation purposes, also implement the external environment behavior as a separate program module (or a set thereof).

Module Verification: Verify that the individual modules satisfy their properties as stated in the Module Properties step. If the modules are small and simple, this step can be done, e.g., automatically by model checking, or, by traditional testing techniques.

We present several examples to demonstrate that this technique might yield more efficient verification in some cases, and illustrate the benefits of formally documented properties in software comprehension, reuse and maintenance.

This chapter is organized as follows. In Section 7.2 we show how BP's semantics can be formalized as a reusable application-agnostic model of the Z3 SMT solver. In Section 7.3 we apply the proposed approach to several examples, and discuss its benefits. In Section 7.4 we briefly review different approaches to compositional system specification and verification. In Section 7.5 we summarize the chapter and discuss future research directions.

7.2 Formulating BP's Idioms in Z3

In this section we show how the application-agnostic composition semantics of BP can be defined in Z3 towards serving in compositional proofs.

We begin the definition with the concepts of time and events[1]:

```
Time      = IntSort();
Event     = Datatype( 'Event' )
...
Event     = Event.create()
```

In our implementation, time is discrete and is represented by integers, and in fact refers to the sequence number of an event in a trace. Events are defined as a Z3 data type, followed by the definition and creation of the application-specific event objects.

We then define functions that model the requesting and blocking of events by b-threads, and the resulting trace of triggered events:

```
requested = Function( 'requested', Event, Time, BoolSort() )

blocked   = Function( 'blocked', Event, Time, BoolSort() )

trace     = Function( 'trace', Time, Event )
```

The first line states that the function `requested` maps any event and time instant to a boolean flag, which specifies whether or not the event was requested at that time by any b-thread. Similarly, the function `blocked` maps an event and a time instant to a boolean flag that is true if and only if the event was blocked at that time. The `trace` function associates each time instant with the event that was triggered at that instant.

We then model behavioral programming semantics as a property of these functions.

```
∀e, t: trace( t ) = e ⇒ requested( e, t ) ∧ ¬blocked( e,t )
```

This rule states that, in order to be triggered at a particular time instant, an event must be requested at that time and must not be blocked.

Since we want to establish the proof by analyzing each of the b-threads in isolation, we introduce a helper function called `requested_by` that takes the same parameters as the function

[1]Most of this work with Z3 was done using the Python API. For readability, the presentation here interchanges and mixes such Python code, formatted Z3 output, and plain mathematical formulations. The code can be found at [80].

requested, with one additional parameter that indicates which b-thread initiated the request. The function blocked_by indicates in a similar way which b-thread initiated the blocking of an event at a given time. We then define:

$$\texttt{requested(e, t)} \Leftrightarrow \bigvee_{bt \in BThreads} \texttt{requested_by(e, t, bt)}$$

and

$$\texttt{blocked(e, t)} \Leftrightarrow \bigvee_{bt \in BThreads} \texttt{blocked_by(e, t, bt)}.$$

I.e., an event is considered requested (respectively, blocked) if and only if it is requested (respectively, blocked) by some b-thread. The set *BThreads* of participating b-threads is encoded as a Z3 list.

There are various methods that can be applied in event selection, such as priority or planning-based schemes, with which it may sometimes be more convenient to program. The axioms corresponding to these models can also be formulated in Z3. For example, a priority-based scheme can be formulated as follows:

```
Priority    = IntSort();

requested   = Function( 'requested', Event,  Time, Priority, BoolSort() )

blocked     = Function( 'blocked', Event,  Time, BoolSort() )

TraceEntry = Datatype( 'TraceEntry' )
TraceEntry.declare( 'TEntry', ( 'event', Event ),  ( 'priority', Priority ) )
TraceEntry = TraceEntry.create()

trace       = Function( 'trace', Time,  TraceEntry )
```

And the axioms that describe priority-based selection are:

```
∀e, t, p : ¬requested( e, t, p ) ⇒ trace( t ) ≠ TraceEntry( e, p )

∀e, t    : blocked( e, t ) ⇒ event( trace( t ) ) ≠ e

∀e, t, pr: requested( e, t, pr ) ∧ ¬blocked( e, t ) ⇒ priority( trace( t ) ) ≥ pr
```

Finally, the requested_by helper function handles priorities:

$$\texttt{requested(e, t, pr)} \Leftrightarrow \bigvee_{bt \in BThreads} \texttt{requested_by(e, t, pr, bt)}$$

The blocked_by helper function remains unchanged.

Using the above axioms, each event is requested with an integer representing its priority, and among all events that are requested and not blocked the one of highest priority is selected for triggering. The trace function includes the priority of the event triggered at each step.

Our axiomatization presupposes that all executions of the program are infinite, since trace

is defined to be an infinite sequence of events. However, finite executions can also be dealt with, by adjusting the axioms to include a special nop event that is only triggered when no other events are enabled.

7.3 Examples

In this section we demonstrate the application of the method outlined in the introduction to several examples. In each example, we specify module and system properties, prove the latter given the former, and, when applicable, verify that the individual implemented modules indeed satisfy their properties. The source code for all examples (b-threads and Z3 code) is available online at [80]. We then discuss how the basic proof of correctness of the application also results in opening the way to possible acceleration of the proof when compared to explicit model checking, and in benefits in documentation.

7.3.1 Counting with Small Orthogonal Modules

Before going into more practical examples, we describe a small example that highlights how domain knowledge that leads to a particular design can be known to and used by the SMT solver, leading to efficient verification of a composite reactive program.

Let $p_1, p_2 \ldots, p_n$ be n large prime numbers, and let $N = \prod_{i=1}^{n} p_i$. Let E_0 and E_1 be two events, and consider the ω-regular language $L_N = \left((E_0 + E_1)E_1^{N-1}\right)^{\omega}$. Thus, in every run E_0 can only be triggered at times that are divisible by N, and E_1 may always be triggered.

Our goal is to create a behavioral program that generates L_N and prove its correctness. For $i = 1, 2, \ldots, n$ consider the b-threads $BT_1, \ldots, BT_n$ and BT_{gen} defined by the following pseudo-code:

```
1  BT_i for i ∈ {1,2,...,n}:
2    while( true ) {
3      bSync( wait for {E_0, E_1} );
4      for( j = 0; j < p_i - 1; j++ )
5        bSync( wait for E_1 while blocking E_0 );
6    }
7  }
8
9  BT_gen:
10   while( true ) {
11     bSync( request { E_0, E_1 } );
12   }
13 }
```

In words, BT_i blocks E_0 at all time instants that are not divisible by p_i, and BT_{gen} requests both E_0 and E_1 at all synchronization points. We now prove that together these b-threads generate L_N.

We first express the properties of each of the b-threads separately (shown here for $n = 2$, with $p_1 = 3$ and $p_2 = 7$):

```
∀t, e: ( ( t % 3 ≠0 ) ⇔ blocked_by( E0, t, BT1 ) ) ∧
                          ¬blocked_by( E1, t, BT1 ) ∧
                          ¬requested_by( e, t, BT1 )

∀t, e: ( ( t % 7 ≠ 0 ) ⇔ blocked_by( E0, t, BT2 ) ) ∧
                          ¬blocked_by( E1, t, BT2 ) ∧
                          ¬requested_by( e, t, BT2 )

∀t, e: requested_by( e, t, BTgen ) ∧
       ¬blocked_by( e, t, BTgen )
```

The first formula says that BT_1 blocks E_0 if and only if t is not a multiple of 3, that it never blocks E_1, and that it never requests any event. The second formula states similar properties for BT_2 with the difference that it blocks E_0 at times not divisible by 7 instead of 3. The third formula captures the properties of BT_{gen}, namely, that it always requests both events and never blocks either of them. Note that the states and transitions of the b-threads are not explicitly modeled in this case, and that the Z3 formulas also cover what the modules *do not* do, i.e., do not block or request, which is needed for our application-agnostic composition.

As these b-thread properties concentrate only on what b-threads request or do not request and what they block or do not block, and not on which events are actually triggered, each b-thread's properties can be verified by model checking the b-thread in isolation from the rest of the program. This method of model checking relies on the abstraction of a b-thread code as consisting of atomic transitions between synchronization points, in which requests and blocked events are declared. As each of the first n b-threads has p_i states and BT_{gen} has a single state, model checking the individual b-threads entails examining a total of $1 + \sum_{i=1}^{n} p_i$ states. In contrast, explicit model checking of the entire system with all b-threads would have to traverse all the $\prod_{i=1}^{n} p_i$ reachable states in the product transition system.

When we add these properties to the Z3 model described in the preceding section, we see that Z3 can indeed quickly verify that the system satisfies its desired property. Namely, that E_0 is only enabled at times divisible by N (21 in this case), and that E_1 is enabled at all times.

```
∀t: requested( E0, t ) ∧ ¬blocked( E0,t ) ⇔ t % 21 == 0
∀t: requested( E1, t ) ∧ ¬blocked( E1,t )
```

The duration it takes Z3 to verify this property is affected only negligibly by an increase in the p_i values. This illustrates the fact that the verification is performed with the aid of additional arithmetical knowledge and not by traversing the entire state space. It yields (in this case) the ultimate desired property of compositional verification – establishing correctness based on proving individual modules separately, without explicitly model checking the product transition system.

Observe that the described Z3 formulation can also be used to prove *liveness* properties. For

instance, in order to prove the property "event E_0 is triggered infinitely often" we would follow the same steps described above, formulate in Z3 the property that E_0 is enabled infinitely often (at intervals of 21 steps), and let Z3 prove it. Then, using reasonable fairness assumptions, the liveness property can immediately be deduced. Liveness property verification is also supported by verifying liveness properties of individual b-threads or group thereof using the BPJ model checker.

This example also shows the power of the blocking idiom in BP. Specifically, if we remove the ability of a b-thread to block events, we can prove (to be published separately) that one must then use at least one b-thread whose size is exponentially larger than the size of the b-threads proposed here. This shows that, in our setting, blocking may allow for an exponential saving in the size of the state space needed for verification, thus accelerating verification when appropriate compositional techniques exist.

7.3.2 Simulating Constrained Movement

Consider an application simulating movement of a particle in a two-dimensional grid, as follows. The grid has $(2n+1)^2$ points, with coordinates $\langle x,y \rangle$ where $-n \leq x,y \leq n$. Initially, the particle is at the center of the grid, at point $\langle 0,0 \rangle$. In each simulation step, the particle moves randomly from its then-current position to one of its four neighbor positions. Apart from the particle process, the system includes processes that make areas of the grid inaccessible to the particle. For example, we analyze the case where each such inaccessible area can be described by a continuous function $f(x)$ such that point $\langle x,y \rangle$ is inaccessible if and only if $y \geq f(x)$ (alternatively, $y \leq f(x)$). In other words, the area above (or below) the curve $y = f(x)$ is inaccessible.

The goal is to discover compositionally whether the inaccessible areas jointly prevent the particle from reaching the grid's boundaries.

Each process b-thread may have a rich behavior and a complex transition system unrelated to the particle movement, where the constraining of the particle movement may be only a side-effect of the behavior. This complex behavior is abstracted here by a b-thread with ℓ states, visited sequentially in a cycle, such that the constraining effect is true in all of them. The approach that we use may be applied also to more complex processes with branches in the transition system and varying sets of blocked events.

The pseudo-Java code of the particle b-thread is:

```
1  x = 0; y = 0;
2
3  while ( true ) {
4      bSync( Request the moves:
5             Move( x+1, y ), Move( x-1, y ), Move( x, y+1 ), Move( x, y-1 ) );
6
7      /* The triggered Move event is returned in bp.lastEvent */
8      x = bp.lastEvent.x;
```

```
9       y = bp.lastEvent.y;
10  }
```

Observe that the particle b-thread is "unaware" of movement constraints imposed by either the grid or the process threads.

A b-thread corresponding to a process that forbids particle movement into the region $y \geq f(x)$, and has a single cycle of ℓ states, is:

```
1  state = 0
2  while ( true ) {
3      bSync( Wait for all events, while blocking moves to all ⟨x,y⟩ s.t.  y ≥ f(x)  );
4      state = ( state + 1 ) % ℓ;
5  }
```

A direct approach to verifying this application is to span its entire state graph, and to check that there is no reachable state where x or y equals $\pm n$. For example, in explicit model checking of the composite application the number of states that will be visited is on the order of $n^2 \cdot \prod_{bt \in P} \ell_{bt}$, a quantity that grows exponentially with the size of the set P of all process b-threads.

By contrast, the compositional approach that we suggest is to model check each process b-thread separately (with all its internal dynamics), and to employ Z3 for the compositional part. For example, the relevant properties of the particle b-thread *ParticleBT* are coded in Z3 as:

```
∀e, t: ¬blocked_by( e, t, ParticleBT )

∀x, y:
    requested_by( Move( x, y ), t, ParticleBT ) ⇔
    ( trace( t - 1 ).x = x - 1 ∧ trace( t - 1 ).y = y ) ∨
    ( trace( t - 1 ).x = x ∧ trace( t - 1 ).y = y - 1 ) ∨
    ( trace( t - 1 ).x = x + 1 ∧ trace( t - 1 ).y = y ) ∨
    ( trace( t - 1 ).x = x ∧ trace( t - 1 ).y = y + 1 )
```

and the properties of each process b-thread $Process_i$ (associated with function $\mathtt{f}_i$) are coded as:

```
∀e, t   : ¬requested_by( e, t, Process_i )

∀x, y, t: blocked_by( Move( x, y ), t, Process_i ) ⇔ y >= f_i( x )
```

Observe that this formulation holds for all ℓ_i states of the process b-thread, and there is no need to articulate properties of individual states.

Another Z3 formula (not shown) specifies the initial position of the particle. Finally, to define what we want Z3 to prove, we state the (undesired) property that the particle does reach the grid boundaries:

```
∃t: trace( t ).x = n ∨ trace( t ).x = -n ∨ trace( t ).y = n ∨ trace( t ).y = -n
```

We then run Z3 to check that this model is unsatisfiable, taking advantage of Z3's knowledge of arithmetic to deduce that the inaccessible zone, as defined by the properties of the $Process_i$ b-threads, renders the edges of the grid unreachable.

It now remains to be verified that the original b-threads uphold the properties that we have encoded in Z3. Fortunately, this can be performed for each b-thread separately, without composing them — by using either static analysis or the BPJ model checker [90]. In the latter case, each process b-thread is checked, along with a b-thread that repeatedly requests all possible movements into all $(2n+1)^2$ points of the grid. The property to be verified is that the single b-thread blocks movements into coordinates where $y \geq f(x)$. The particle b-thread is also verified separately, to ensure that successive points in the movement are always connected by a single grid edge.

Table 7.1 shows the savings when model checking each process behavior and the particle behavior separately, as compared to checking the movement of the particle with all behaviors together. In this example, we set $n = 20$ (resulting in a 41×41 grid), and chose four processes with forbidden zones that prevent the particle from venturing outside the quadrilateral with vertices $\langle 15,15 \rangle$, $\langle -18,16 \rangle$, $\langle -19,-19 \rangle$, $\langle 17,-18 \rangle$. The movement-constraining processes have $2,3,5$ and 7 internal states. The run time improvement is evident.

Table 7.1: Comparing the monolithic approach to the compositional one. Rows 1 and 2 describe checking each of the b-threads separately using model checking (MC); row 3 describes the compositional step (using Z3); and row 4 summarizes the total cost of the compositional approach. The last row of the table describes model checking the entire system, as a single unit.

Checked entity	Number of states	Method of checking	Run time (sec.)
Four process b-threads	2,3,5,7	MC	160 (total)
Particle	1681	MC	4
Compositional step	—	Z3	0.03
Total compositional proof	1698	MC+Z3	164.03
MC of Entire system	119385	MC	426

For a similar setting with $n = 10^{17}$, it took Z3 approximately 7 seconds to reach a conclusion. However, a related test run, in which one of the constraints was omitted and the resulting model is satisfiable did not terminate (in a reasonable amount of time). We believe that this is a technical issue in our implementation, and not a fundamental problem in the underlying approach. Future work will include optimizing our implemented model to better fit Z3's constraints, as well as leveraging future enhancements of Z3. Note that our present model does permit one to use Z3 to verify, within a fraction of a second, that a given trace that reaches the grid boundaries is valid.

We now demonstrate how a formalization of properties of the modules in Z3 supplements the code with documentation that is useful beyond the verification process, for tasks such as module reuse or enhancement.

Consider a requirement, which arrives from the user after the system is up and running, to expand the application to a three-dimensional setting. That is, the grid is extended to 3D and the

original requirement that the particle is constrained within a box around the origin and cannot reach the grid boundaries, remains, but now is interpreted in 3D. After adding an attribute to the Move event that gives the z axis position and adding to the particle b-thread movements in the z direction, the question we ask next is how to enhance the b-threads for the processes that constrain the particle movement.

In a standard development process, without the Z3 formulation, a programmer wanting to reuse or enhance existing modules for the new requirement would need to check their code directly. While the code in our example is simple, the code of the movement-constraining b-threads may be complex, and the relevant properties may not readily emerge.

With the Z3 formulation, the contemplation of how to extend the system can be done in the context of the high-level theory. In this case, the Z3 code explicitly talks about the lines that form a closed polygon contained in the boundary of the two-dimensional grid. When we formulate in Z3 a 3D extension of the properties, we can start by checking if the current modules and formulated properties already satisfy the new requirement. If not, Z3 gives us a counterexample that we can use to guide the development. From the counterexample we may realize that, in 3D, the b-threads form infinite walls in the z dimension rising from the edges of the 2D polygon and that it is sufficient to add a “floor” and a “ceiling”. More generally, we see that the boundaries can be avoided by forming a set of planes in the 3D space that form a polytope that contains the origin and is contained within the bounding box.

The role of Z3 in this process is to help the designers identify and document all the properties of the b-threads that are relevant to the requirements. When a property is missing (e.g., if we forget to mention that the polytope contains the origin) Z3 presents a counterexample, from which the missing properties may emerge. When Z3 proves that all the requirements are satisfied, we know that we have documented all the required properties of the b-threads. The completeness of the documentation of the properties of the b-threads is important, for example, when we want to replace a b-thread.

Once a set of sufficient properties is established in Z3, the implementation can proceed in different directions: some properties may already exist in the current modules (but are not documented because they were not relevant to the 2D case), others may be implemented as changes to existing modules, and yet others may be added as new independent b-threads. The implemented modules can then be model checked to verify that they satisfy the properties and, if so, we can conclude that the application is correct.

7.3.3 A Job Scheduler

The following example demonstrates the development and compositional verification of a scheduling algorithm using behavioral programming. The program is actually incremental in

nature: when new b-threads are added, the program can be verified without rechecking existing processes.

The problem is defined as follows. A scheduler needs to assign time slots for each of k processes $P_1,\dots,P_k$. Each process P_i is associated with two parameters, m_i and n_i, meaning that it requires the assignment of m_i slots in each cycle of n_i slots. Put differently, process P_i needs to be assigned m_i slots in cycle $\{kn_i+1, kn_i+2,\dots,(k+1)n_i\}$, for all $k \in \mathbb{N} \cup \{0\}$.

A schedule that satisfies all these constraints exists if and only if $\sum_{i=1}^{k}(m_i/n_i) \le 1$, in which case an *earliest-deadline first* (abbr. EDF) policy will guarantee that none of the conditions are violated (see, e.g., [114]).

We suggest the following BP implementation. Given an instance of the problem, $\langle m_i, n_i \rangle_{1 \le i \le k}$, we program a b-thread for each process, presenting m_i requests in each cycle of n_i slots in the form of events $R(bt, j)$, where bt is the b-thread's identity and $1 \le j \le n_i$ is the number of slots (scheduling opportunities for this process) before the present cycle ends. The scheduler is implemented in BP by having all b-threads that have not yet been assigned sufficient slots in the present cycle block all event requests with a higher value of j. An additional b-thread, *Idle*, continuously requests the special event $R(idle, \infty)$ that is triggered only when no process requests any event in the slot.

At each behavioral synchronization point, one of the requested events is triggered, indicating that the requesting process is assigned the present slot. All b-threads are notified when an event is triggered and can then request to be scheduled in the next slot as needed. The b-threads for each of the processes can be modeled in BPJ as follows:

```
1  BT⟨m_i,n_i⟩ for i ∈ {1,...,k}: {
2    while ( true ) {
3      count = 0;
4      for( j = n_i; j > 0; j-- ) {
5        if( count < m_i ) {
6          bSync( request R(i,j), block all events R(s,t) such that t > j, wait for all events
             );
7          if( lastEvent == R(i,j) )
8            count++;
9        }
10       else {
11         bSync( wait for all events );
12       }
13     }
14   }
15 }
```

Without loss of generality, we assume that $\forall i, m_i = 1$ (for $m_i > 1$, as we can substitute the original process by m_i processes with parameters $\langle 1, n_i \rangle$ each). The b-thread for process P_i then has $O(n_i)$ states. Consequently, exhaustive verification of the application entails inspecting $O(\prod_{i=1}^{k} n_i)$ states (in the worst case, assuming the n_i's are pairwise mutually prime).

A compositional alternative is to verify each b-thread separately, to ensure that it constantly

blocks all requests by processes with further-away deadlines than its own — until its scheduling quota has been filled. This can be accomplished by inspecting only $O(\sum_{i=1}^{k} n_i)$ states. If these properties hold, then EDF scheduling is guaranteed, and it only remains to check that $\sum_{i=1}^{k}(m_i/n_i) \leq 1$, which can be done manually, or using a calculator.

All in all, this approach yields much shorter verification times. Further, when adding a new process $\langle m_{k+1}, n_{k+1} \rangle$ at a later time, one need not repeat the verification of the original b-threads (assuming an upper bound on the number of threads and their cycle lengths). It suffices to check that the new b-thread adheres to its responsibility in the EDF policy (by requesting and blocking events correctly), and then verify that $\sum_{i=1}^{k+1}(m_i/n_i) \leq 1$.

Another alternative solution we explored entails modeling the properties of the b-threads in Z3, and having the tool check whether a legal schedule exists in a model that includes all of them. In this approach, the properties of b-thread i with parameters $\langle 1, n_i \rangle$ are as follows:

```
∀e, t: requested_by( e, t, BTi ) ⇔
        ( e = ( BTi, ni - ( t - 1 ) % ni ) ∧ ¬already_scheduled( BTi, t ) )

∀e, t: ¬already_scheduled( BTi, t ) ⇒
        blocked_by( e, t, BTi ) ⇔ deadline( e ) > ni - ( t - 1 ) % ni

∀e, t: already_scheduled( BTi, t ) ⇒
        ¬blocked_by( e, t, BTi )
```

where, as in the BPJ implementation, events consist of the requester's identity and the time remaining until its deadline, and the functions `deadline` and `requester` are used to retrieve the respective parameters. The helper function `already_scheduled` evaluates to true if and only if b-thread BT_i has already been scheduled in the cycle to which time t belongs.

The verification is then performed by giving Z3 the undesired property that one of the processes is not scheduled in some cycle, and having it prove that the model then becomes unsatisfiable. The property is given as:

```
∃BTi, t1: ∀t2: t1·ni<t2≤( t1 + 1 ) ni ⇒ requester( trace ( t2 ) ) != BTi
```

As the property to be proved is algebraic in nature, we expected the Z3 verification process to readily display superior performance as compared with explicit model checking using the BPJ model checker. Unfortunately, that was not the case, and the running time grew exponentially with the number of processes. We believe that our implementation can be improved and the running time greatly decreased, but we leave this for future work. Despite being applicable only to programs with few processes, our current model is still useful: it demonstrates that the set of thread properties we have identified is complete, and that it suffices for proving the correctness of the system. This indicates that we have documented any hidden assumptions about the various modules, and facilitates their redesign or reuse.

7.3.4 Dining Philosophers

In this example we demonstrate a direct approach to encoding b-threads in Z3 by capturing their transition systems and the requested and blocked events in each state.

Consider for example a BP model for the famous dining philosophers problem.[2] Assume that this abstract problem is a specification for a larger BP application, e.g., a circle of industrial robots where each two adjacent ones share a tool, and each robot requires both its adjacent tools to perform its task. This behavior of the robots can be specified in BP as follows: there is a b-thread per tool (fork), with two states — "up" (fork_state is true) and "down" (fork_state is false), and a b-thread per robot (philosopher), with its four states known as the fixed cycle of "thinking", "picked up one fork", "eating", and "put down one fork". The events are of the form E(i,j,up) or E(i,j,down) and represent "philosopher i picked up (or put down) fork j" for $0 \leq i \leq n$, and $j = i$ or $j = (i+1) \bmod n$. All philosophers but one are right-handed (they first pick up the fork on their right) and one is left-handed. Each fork thread blocks events that pick it up when in the "up" state and events that put it down when in the "down" state, without ever requesting events.

We proceed to explain the transition system by formulating its properties in Z3 as part of a proof that the industrial robotic application satisfies its specification and is deadlock-free. Below we describe parts of a model for a system with eight philosophers (hence, e.g., (fo+1)%8 is the index of the fork next to fork fo (and the philosopher of same number) in cyclic order):
Fork b-threads never request events:

```
∀e, t, fo : ¬requested_by( e, t, Fork( fo ) )
```

A b-thread for a fork that is down blocks the events of putting the fork down again (and only these):

```
∀t, fo:
  ¬fork_state( fo, t ) ⇒
  (blocked_by( E( fo, fo, down ), t, Fork( fo ) ) ∧
   blocked_by( E( ( fo + 1 ) % 8, fo, down ), t, Fork( fo ) ) ∧
   (∀e1:
     blocked_by( e1, t, Fork( fo ) ) ⇒
          e1 = E( fo, fo, down ) ∨ e1 = E( ( fo + 1 ) % 8, fo, down ) ) )
```

A b-thread for a fork that is up blocks the events of picking the fork up again (and only these):

```
∀t, fo:
  fork_state( fo, t ) ⇒
  ( blocked_by( E( fo, fo, up ), t, Fork( fo ) ) ∧
   blocked_by( E( ( fo + 1 ) % 8, fo, up ), t, Fork( fo ) ) ∧
   (∀e1:
     blocked_by( e1, t, Fork( fo ) ) ⇒
          e1 = E( fo, fo, up ) ∨ e1 = E( ( fo + 1 ) % 8, fo, up ) )
```

[2]There are n philosophers sitting around a table. There is a fork between each two adjacent philosophers. To eat, a philosopher needs to hold both of her adjacent forks.

The state of the fork changes according to the pick-up/put-down actions of the philosophers on the right or left of the fork (and only these) :

```
( ∀t, fo:
  trace( t ) = E( fo, fo, up ) ∨ trace( t ) = E( ( fo + 1 ) % 8, fo, up ) ⇒
  fork_state( fo, t + 1 ) ) ∧
( ∀t, fo:
  trace( t ) = E( fo, fo, down ) ∨ trace( t ) = E( ( fo + 1 ) % 8, fo, down) ⇒
  ¬fork_state( fo, t + 1 ) ) ∧
( ∀t, fo:
  ¬( trace( t ) = E( fo, fo, up ) ∨ trace( t ) = E( ( fo + 1 ) %8, fo, up ) ∨
     trace( t ) = E( fo, fo, down ) ∨ trace( t ) = E( ( fo + 1) % 8, fo, down ) ) ⇒
  fork_state( fo, t ) = fork_state( fo, t + 1 ) )
```

Once these properties are formulated, system properties can be proven. In our case, Z3 can verify that the system does not deadlock, i.e., that there is always an event that is requested and not blocked in all executions of the program. Z3 does this in under 10 seconds. This verification is performed using a slightly modified version of the axioms presented in Section 7.2, which considers both finite and infinite executions of the program.

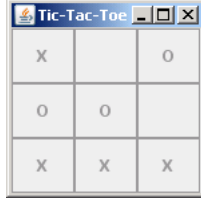
Note that the robotic implementation may be very different from the specification. Still, to verify that each property in the specification holds it should suffice to model check exhaustively only a few robots and tools.

7.3.5 Tic-Tac-Toe

In this example we demonstrate the use of Z3 to verify a slightly larger example, highlighting that the properties of the individual modules are quite independent of each other, and refer to the basic specification of the system. We illustrate our technique on the b-threads of the Tic-Tac-Toe game application presented in [86]; we briefly summarize the application's features in a description taken from [86], and refer the reader to that paper for a more detailed explanation of the application itself.

In the (classical) game of Tic-Tac-Toe, two players, X and O, alternately mark squares on a 3×3 grid whose squares are identified by $\langle row, column\rangle$ pairs: $\langle 1,1\rangle, \langle 1,2\rangle, \ldots, \langle 3,3\rangle$. The winner is the player who manages to form a full horizontal, vertical or diagonal line with three of his/her marks. If the entire grid becomes marked but no player has formed a line, the result is a draw.

In our example, player X is played by a human, and player O is played by the application. Each move (marking of a square by either player) is represented by a matching event, $\mathsf{X}_{\langle row,col\rangle}$ or $\mathsf{O}_{\langle row,col\rangle}$. The events XWin, OWin and Tie represent the respective victories and a draw. A play of the game may be described as a sequence of events. E.g., the sequence $\mathsf{X}_{\langle 1,1\rangle}$, $\mathsf{O}_{\langle 2,2\rangle}$, $\mathsf{X}_{\langle 3,2\rangle}$, $\mathsf{O}_{\langle 1,3\rangle}$, $\mathsf{X}_{\langle 3,0\rangle}$, $\mathsf{O}_{\langle 2,1\rangle}$, $\mathsf{X}_{\langle 3,3\rangle}$, XWin describes a play in which X wins, and whose final configuration is:



The BP implementation of the game as described in [86] contains two types of b-threads: game rules and strategies. Examples for game rule b-threads are the *SquareTaken* thread that blocks further marking of squares already marked by X or O, and the *EnforceTurns* thread that alternately blocks O moves while waiting for X moves, and vice versa (we assume that X always plays first).

Strategy b-threads are responsible for helping the program to play "wisely" — that is, to contribute towards ensuring that the program does not lose the game. An example for one such b-thread is *PreventThirdX*: when it notices two Xs in a line, it requests the marking of an O in the third square of this line (to prevent an immediate loss).

If neither player makes any mistakes, a Tic-Tac-Toe game ends in a draw. Therefore, we consider our game playing application to be achieving its goals if it never loses the game — namely, if the event XWin is never triggered in any run. In [86], this property was verified via explicit model checking of the Java application with concurrent execution of all b-threads. By contrast, in the present work the proof of correctness begins with the properties of the b-threads as may be verified individually, or as may be planned or designed in early development stages. Z3 is used to verify that these properties, when composed, yield the desired results.

In our proposed Z3 formulation, each event has three fields: x, y and $type$. The $type$ field can have values X,O,X_WIN,O_WIN and TIE. If the event is of one of the first two types, the x and y fields hold the row-column coordinates of the move; otherwise, these fields are meaningless.

As in the previous examples, we formulate the properties of the various b-threads as Z3 code. For instance, the *EnforceTurns* thread, BT_{et}, is formulated as:

```
∀t, e: ( t == 1 ) ⇒ blocked_by( e, t, BT_et ) ⇔ e.type() == O

∀t, e: ( t > 1 ) ⇒ blocked_by( e, t, BT_et ) ⇔
       ( e.type() == O ∧ trace( t - 1 ).type() == X ) ∨
       ( e.type() == X ∧ trace( t - 1 ).type() == O )
```

The code states that in the first move ($t == 1$) the b-thread blocks all of O's moves, and that in subsequent moves the b-thread blocks the player who played last.

Next, we see how the *PreventThirdX* b-thread, BT_{ptx}, translates into Z3 code:

```
∀t, e: requested_by( E( row, col, O ), t, BT_ptx ) ⇔
  ∃t1,t2: t1 < t ∧ t2 < t ∧
     trace( t1 ).type()==X ∧
     trace( t2 ).type()==X ∧
     ( ( trace( t1 ).x() == trace( t2 ).x() == row ) ∨
       ( trace( t1 ).y() == trace( t2 ).y() == col ) ∨
```

```
( ( trace( t1 ).x() == trace( t1 ).y() ) ∧
  ( trace( t2 ).x() == trace( t2 ).y() ) ∧
  ( row == col ) ) ∨
( ( trace( t1 ).x() + trace( t1 ).y() == 4 ) ∧
  ( trace( t2 ).x() + trace( t2 ).y() == 4 ) ∧
  ( row + col == 4 ) ) )
```

The above code indicates that the b-thread only requests an O move in square *row*, *col* if: (1) X has already marked two squares in that row, or (2) X has already marked two squares in that column, or (3) The square is part of the main diagonal ($row == col$), and X already has two squares of that diagonal, or, (4), if the square is part of the secondary diagonal ($row + col == 4$), and X has already marked the two other squares of that diagonal.

Due to the larger extent of this example, we omit the remaining b-threads. The code is available online at [80].

Once all the other rule and strategy b-threads have been translated into Z3 in a similar fashion, we had the tool prove the desired property, namely that O can never lose:

```
∀t: trace(t).type() != X_win
```

Z3 replied in the affirmative. Further, running the same test with one of O's strategy b-threads omitted resulted in a failure. Printing the Z3 model, the listing of the function *trace* reveals a counter-example scenario in which X wins:

```
[ 1 → TraceEntry( E( 3, 3, X ) ),
  2 → TraceEntry( E( 2, 2, O ) ),
  3 → TraceEntry( E( 1, 1, X ) ),
  4 → TraceEntry( E( 1, 3, O ) ),
  5 → TraceEntry( E( 3, 1, X ) ),
  6 → TraceEntry( E( 2, 1, O ) ),
  7 → TraceEntry( E( 3, 2, X ) ),
  else → TraceEntry( E (1, 1, X_WIN ) ) ]
```

Apart from enabling us to prove the desired property, we observe that formulating the b-thread's properties as Z3 axioms also provides a more precise documentation than natural-language requirements, as well as a useful abstraction of program code. For instance, the Z3 code for the the *EnforceTurns* b-thread (displayed above) states explicitly, and thus documents, the fact that player X plays first, and that neither player can make two consecutive moves. For comparison, the (pseudo) Java code of this b-thread is:

```
1 while ( true ) {
2     bSync( block O moves, wait for X moves );
3     bSync( block X moves, wait for O moves );
4 }
```

In the Z3 code, the reader can interpret each formula separately. Even if a formula is long — its scope is well defined and it is always complete. When reading program code like the above the reader has to mentally follow the flow of the *for* loop, and the instructions within it

and translate them into conditions and possible scenarios. We find that the combination of the natural scenario-oriented program code with the precise yet abstract Z3 properties complement each other in development, verification and maintenance processes. If modules from this application are to be used in another application, or in an enhanced version of the same application, the developers can readily see whether the existing code supports, e.g., more than two players, changing the order of player moves, or allowing a player two consecutive moves under some conditions.

7.4 Relationship to Other Work

Much research on compositional and modular verification has been conducted in recent years. While most proposed approaches are similar in their underlying assume-guarantee framework, they differ in several aspects: the modeling formalisms (for both program and specification), the way assumptions are inferred (manual or various automatic variants), and the type of reasoning used to deduce the desired system-wide property from the module properties. In this section we review some of these approaches.

In [128], [105] and [133], the authors study assume-guarantee proof rules for parallel programs, where communications or interference between programs are via messages (in [128]) or shared variables (in [105, 133]). Our focus is also on parallel programs, but in our work the components (b-threads) do not communicate directly with each other, but rather use the simple protocol offered by BP semantics. In addition to providing a concise interface for interweaving independent modules that represent separate facets of behavior, the protocol also allows for a reduction in the size of the state-space, as we are only interested in the state of a b-thread when it is at a synchronization point.

While BP is oriented towards programming in standard languages, composition in BP is event-based and may be formalized in terms of finite-state transition systems. System composition and modular verification in such finite-state settings were described in [120] with the introduction of I/O automata, in [76] in the context of a subset of CTL, in [58] using interface automata, and in the research on the behavior-interaction-priority formalism (BIP), see, e.g., [35]. Our work can be viewed as contributing towards applying these methods in programming contexts and towards making the application of formal methods more accessible to programmers. In line with this goal, it would be interesting to explore compositional verification of behavioral programs based on properties formalized as assertions within the code, using behavioral interface languages such as JML, SPEC#, SPARK, separation logic, and Dafny. See [94] for a survey.

The difficulty in formalizing environment behavior from the point of view of a single module is tackled in [53]. The authors verify individual parallel modules, together with interface

processes that represent a module's dependencies on its environment, but which can be simpler than the full composite behavior of the environment. The interface modules are derived from the specification of the other modules. In [97], the authors describe assume-guarantee reasoning using iterative abstraction and refinements of the assumptions. In [70], the environment assumptions of a thread are automatically inferred and are abstracted from behaviors of the other threads. In [73], the authors present techniques for automatically decomposing the verification problem and generating component assumptions based on design-level artifacts. In another approach [17], a learning algorithm is used to infer the assumptions.

In our proposed setting, the strict interface through which modules communicate (that is, the events they request, wait-for and block) facilitates integrating assumptions about the environment into the verification process. Particularly, we have shown here that it is often straightforward to represent the environment by dedicated b-threads. This process is demonstrated in Section 7.3.5, where all strategies available to the environment — the X player — are represented by a simple b-thread. From the verifier's point of view, there is no difference between that b-thread and the actual program's b-threads.

Applying model checking to the goal of establishing low-level properties and then using semi-automated high-level analysis also helps tackle the state-explosion problem. For example, in [123] the authors verify hardware systems using reasoning that is performed by a proof assistant, while the generated subgoals are verified by model checking. In [54], the authors show that finding a decomposition that yields benefits in compositional verification is not easy and may not always be possible. In this context, one of the key goals of our present work is verifying modules and composing systems based on artifacts and properties that are aligned with the specifications. Similar approaches appear in [64] for interference and cooperation of aspects and in [65] for components as part of research in the field of *component-based software engineering*.

7.5 Discussion and Conclusion

We have shown how BP and the Z3 SMT solver can be used together for composing a reactive system from relatively independent modules, while accompanying the development with a proof of system's correctness.

As mentioned in Section 7.4, a major issue in compositional verification is automatically generating component properties. We address this by using the requirements that individual modules satisfy as these properties, thus leveraging the intuition the programmers used in building the modules. Using BP to code the modules ensures that module interfaces are always well defined (per the BP semantics), and it tends to produce modules with properties that are relatively self-standing. Consequently, apart from streamlining verification, the resulting module

properties are of value for maintenance and debugging tasks. We believe that this approach has potential, as it bypasses the intricate task of looking for compositional properties in composite code — allowing the programmer to focus on module properties more than on inter-module relationships.

When the effects of module-to-module interaction are dependent on a domain theory that is known to the SMT solver, an opportunity emerges for improving the efficiency of the automated verification. This is because system properties may be inferred directly from module properties, without explicitly examining all states of the composite application.

A key issue that we have encountered is the difficulty of formulating module properties. Per our methodology, component properties have to be formulated twice: once as SMT solver axioms for the compositional part, and once as model checker properties to be proven on individual threads. In our examples, the first part was often time-consuming: it took some effort to formulate properties that appeared to us natural and aligned with system's requirements in ways that allowed Z3 to handle the proof in reasonable time. This is in line with [67], which suggests that the practical impact of compositional methods is constrained by the amount of non-trivial human input required for defining appropriate assumptions. Interestingly, we found that the second part — translating the Z3 properties into model checker properties — was almost trivial, as the properties were typically simple postulations on the states of the threads. In the future we plan to automate this transformation.

Despite its difficulties, we found that the process of refining formal properties was instrumental to our understanding and to the corresponding documentation of module behavior. Our conclusion is that, given the right tools, programmers and designers may find the property formalization and automated verification processes beneficial.

Future research directions include developing IDE support for automated proofs of behavioral applications, guidelines for formulating module properties, and possible enhancements to Z3 (or an alternative solver). Another important direction is proving that our methodology fits industry practice, by applying it to a real large-scale system.

Chapter 8

Theory-Aided Compositional Verification of *RWB* Programs

8.1 Introduction

In concurrent programming, the size of the composite program is typically exponential in the number of its constituent threads. This phenomenon, an instance of the *state explosion* problem, is a major hindrance to the verification of concurrent software. As we discussed in Chapter 7, in recent decades a prominent approach to tackling this difficulty has been that of compositional verification [76]: properties of threads are derived/verified in isolation, and are used to deduce global system correctness, without exploring the entire composite state space. When applicable, compositional verification can often significantly outperform direct verification techniques.

A key challenge in compositional verification is how to *automatically* come up with "good" thread properties — those whose verification is considerably cheaper than the verification of the global property on the one hand, but which are sufficiently meaningful to imply the desired system properties on the other. In Chapter 7 we presented a technique in which these properties were generated manually, or semi-automatically; and in this chapter we build upon these results and present an improved technique in which properties are generated fully automatically. Thus, the approach presented in this chapter offers better scalability [67], in those cases in which it is applicable.

Since the fully automatic compositional verification of arbitrary programs is difficult (and often impossible [54]), one reasonable approach is to trade generality for effectiveness — i.e., to limit the scope of programs that a scheme handles, in exchange for better performance on programs that remain within that scope. Here, we adopt this approach and propose an automatic compositional verification scheme for certain kinds of concurrent software.

This chapter has two main contributions. The first is the rigorous formalization and implementation of a solver for a *theory of transition systems* ($\mathcal{TS}$) within the context of CVC4 [26]

— a lazy, DPLL(T) based SMT solver [130]. The $\mathcal{TS}$ solver takes as input formulas describing a program's concurrent threads (given as transition systems) and the assertion that a certain safety property is violated; and it answers UNSAT if the program is safe, or SAT if it is not. As a standalone module, the $\mathcal{TS}$ solver explores the space of reachable states in order to determine a system's safety — an exploration that is driven by the SMT solver's underlying SAT engine.

Several existing approaches utilize SMT solvers in model checking (e.g., Lazy Annotation [124] and PDR [48]), but typically the process is driven by a model checker that uses an SMT solver as a black-box tool. In our approach the roles are reversed, and the SMT engine, via the $\mathcal{TS}$ solver, can be regarded as invoking a model checker. This design allows other theories within CVC4 to be seamlessly used in analyzing the input program at hand, determining which parts of the state space should be explored and which may safely be ignored. These theories may then influence the search conducted by the $\mathcal{TS}$ solver by asserting lemmas to the underlying DPLL(T) core, sometimes pruning significant portions of the search space and greatly improving performance. We term this process *theory-aided model checking*: the $\mathcal{TS}$ solver explores the state space while also looking for opportunities in which other theories may aid and direct the search.

The second contribution of this chapter is in the way other theories determine which parts of the state space may be ignored during model checking. We perform this by having the $\mathcal{TS}$ solver analyze the input threads and look for pre-supplied *patterns*: structural properties of the threads that may be expressed as assertions in the languages of other theories, such as arithmetic or arrays. It is through these assertions that other theories can "understand" the program and efficiently discover, e.g., that a certain branch of the search space cannot lead to a violation. A key fact here is that each thread/transition system is analyzed separately — and hence the compositionality of our approach: the analysis complexity is proportional to the size of the program and not to that of its state space. We thoroughly describe three of the currently implemented patterns.

The technique presented here differs in two main technical aspects from the technique discussed in Chapter 7. Here, the SMT solver operates directly on the concurrent threads of the input program, rendering it unnecessary to manually or semi-automatically translate them into SMT assertions. This also results in a major boost in performance, and the improved technique can thus handle inputs that were too large for its manual counterpart. The second difference is the automatic search for patterns, which obviates the need for the engineer to manually provide them. This makes the technique applicable to less programs, but makes it perform better and fully automatically when it indeed applies. This is in line with our approach of trading generality for effectiveness, and, as we demonstrate in later sections, our approach is capable of effectively handling broad classes of programs even with just a few stored patterns.

As in previous chapters, the type of software that we target here is the family of discrete

event systems programmed using the $\mathcal{RWB}$ model. As previously mentioned, the $\mathcal{RWB}$ concurrency idioms are widespread and appear, sometimes in related forms, in various formalisms such as *publish-subscribe* architectures [69], *supervisory control* [136] and *live sequence charts* (LSC) [57] and *behavioral programming* (*BP*). model [90]. Thus, by focusing on the $\mathcal{RWB}$ model, we hope to make our technique applicable (with appropriate adjustments) to a variety of programming formalisms. Further, we believe that the technique can be extended to cater to additional concurrency idioms and models.

The rest of the chapter is organized as follows. In Section 8.2 we recap the definitions of the DPLL(T) framework for SMT solvers and of the $\mathcal{RWB}$ model. Next, in Section 8.3 we introduce the theory of transition systems ($\mathcal{TS}$) and describe a theory solver aimed at model checking $\mathcal{RWB}$ programs. In Section 8.4 we demonstrate how the $\mathcal{TS}$ solver can cooperate with other theory solvers in order to expedite model checking. Subsequently, we apply our technique to two broad classes of problems: *periodic problems* in Section 8.5, and programs with shared arrays in Section 8.6. Experimental results appear in Section 8.7, and we conclude with a discussion and related work in Section 8.8. In order to not burden the reader with technical complexities, a few of the rigorous proofs for claims made in this chapter appear as an appendix in Section 8.9.

8.2 Definitions

The DPLL(T) Framework. DPLL(T) [130] is an extensible framework used by modern SMT solvers. It employs multiple specialized theory solvers that interact with a SAT solver. The SAT solver maintains an input formula F and a partial assignment M for F. Periodically, a theory solver is asked whether M is satisfiable in its theory; and, if it is not, the theory solver generates a conflict clause, the negation of an unsatisfiable subset of M, that is added to F. The theory solver may request case splitting by means of the splitting-on-demand paradigm [27], which allows the solver to add theory lemmas to F consisting of clauses possibly with literals not occurring in F.

Verifying $\mathcal{RWB}$ Programs. In [86, 7], the authors demonstrate how the transition systems underlying $\mathcal{RWB}$-threads can be automatically extracted from high level code and then, using abstraction techniques, be symbolically traversed in order to verify safety properties. Safety properties are themselves expressed by *marker* $\mathcal{RWB}$-threads, marking that a violation has occurred with a special API call [86]. For simplicity, we assume that marker threads signal that a violation has occurred by blocking all events, causing a deadlock. Thus, safety checking is reduced to checking for deadlock freedom.

The manual compositional verification of $\mathcal{RWB}$ programs is, of course, discussed in Chapter 7 There, it is shown how the simple $\mathcal{RWB}$ synchronization mechanism facilitates the gen-

eration of individual thread properties, which are then used for proving the system property at hand. The beneficial effect that simple concurrency idioms have on verification is also discussed in [83]. Indeed, the simplicity of the $\mathcal{RWB}$ idioms plays a key role in the pattern matching algorithm that we discuss later.

8.3 The Theory of Transition Systems

We now cast the model checking of $\mathcal{RWB}$ into a DPLL(T) setting, by defining a dedicated *theory of transition systems* ($\mathcal{TS}$). We assume familiarity with the definitions of many-sorted first order logic (see, e.g., [28]). The theory is parameterized by a set $\bar{Q} = \{Q_1, \ldots, Q_n\}$ of *state sorts* used to represent the state sets of the program's constituent threads. Let $\bar{Q}^+$ denote the composite state sorts obtained by taking the Cartesian product of one or more elements in Q. Every element $Q \in \bar{Q}^+$ is a sort in $\mathcal{TS}$. Further, every such Q is also associated with a matching transition system sort, S_Q. Finally, $\mathcal{TS}$ has an event sort, E.

For every $Q \in \bar{Q}^+$ the signature includes: the predicate $I_Q : S_Q \times Q$, indicating initial states; the predicates $R_Q, B_Q : S_Q \times Q \times E$ to indicate whether an event is requested (R_Q) or blocked (B_Q) at a given state; and the predicate $Tr_Q : S_Q \times Q \times E \times Q$ to indicate the state transition rules.

In order to reason about composite transition systems, the signature includes the following functions and predicates. For every $Q^1, Q^2 \in \bar{Q}^+$ we have the transition system composition function $\|_{Q^1Q^2} : S_{Q^1} \times S_{Q^2} \to S_{Q^1 \times Q^2}$ (Recall that $(Q^1 \times Q^2)$ is itself a sort in $\bar{Q}^+$); and also the $pair_{Q^1Q^2} : Q^1 \times Q^2 \to (Q^1 \times Q^2)$ function for composing states, which, per the $\mathcal{TS}$ semantics, is a bijection. Later we often omit the Q subscripts when clear from the context.

For each $Q^1, Q^2 \in \bar{Q}^+$, $\mathcal{TS}$ has the following axioms which enforce the $\mathcal{RWB}$ composition rules. A composite state is initial iff its components are initial states:

$$\begin{aligned}&\forall s_1 : S_{Q^1}, s_2 : S_{Q^2}, s : S_{Q^1 \times Q^2}.\ s = s_1 \parallel s_2 \implies \\ &\quad \forall q : Q_1 \times Q_2.\ (I(s,q) \iff \exists q_1 : Q_1, q_2 : Q_2.\ (I(s_1,q_1) \wedge I(s_2,q_2) \wedge q = pair(q_1,q_2))).\end{aligned}$$

Composite transitions are performed component-wise:

$$\begin{aligned}&\forall s_1 : S_{Q^1}, s_2 : S_{Q^2}, s : S_{Q^1 \times Q^2}.\ s = s_1 \parallel s_2 \implies \\ &\quad \forall q, q' : Q^1 \times Q^2, e : E.\ (Tr(s,q,e,q') \iff \\ &\quad \exists q_1, q_1' : Q^1, q_2, q_2' : Q^2.\ (q = pair(q_1,q_2) \wedge q' = pair(q_1',q_2') \wedge Tr(s_1,q_1,e,q_1') \wedge Tr(s_2,q_2,e,q_2'))).\end{aligned}$$

Requested and blocked events in a composite state are the union of those in the component

states:

$$\forall s_1 : S_{Q^1}, s_2 : S_{Q^2}, s : S_{Q^1 \times Q^2}.\ s = s_1 \parallel s_2 \implies \forall q : Q^1 \times Q^2, e : E.$$
$$(R(s,q,e) \iff \exists q_1 : Q^1, q_2 : Q^2.q = pair(q_1,q_2) \wedge (R(s_1,q_1,e) \vee R(s_2,q_2,e))) \wedge$$
$$(B(s,q,e) \iff \exists q_1 : Q^1, q_2 : Q^2.q = pair(q_1,q_2) \wedge (B(s_1,q_1,e) \vee B(s_2,q_2,e))).$$

As previously discussed, by encoding safety properties as threads of the program to be checked, safety is reduced to deadlock freedom. For each $Q \in \bar{Q}^+$, the signature includes a $deadlock_Q : S_Q \times Q$ predicate, such that:

$$\forall s : S_Q, q : Q.\ (deadlock(s,q) \iff \neg \exists q' : Q, e : E.\ Tr(s,q,e,q') \wedge R(s,q,e) \wedge \neg B(s,q,e)),$$

and the $safe_state_Q : S_Q \times Q$ predicate, with:

$$\forall s : S_Q, q : Q.\ \neg safe_state(s,q) \implies$$
$$deadlock(s,q) \vee \exists q' : S_Q, e : E.\ (Tr(s,q,e,q') \wedge R(s,q,e) \wedge \neg B(s,q,e) \wedge \neg safe_state(s,q')).$$

$\neg safe_state_Q(s,q)$ indicates that state q is unsafe, because it is (or can lead to) a deadlock state. Finally, for each $Q \in \bar{Q}^+$, $safe_Q : S_Q$ indicates that a transition system is safe:

$$\forall s : S_Q.\ \neg safe(s) \iff \exists q : Q.\ I(s,q) \wedge \neg safe_state(s,q).$$

The Theory Solver. Inputs for the $\mathcal{TS}$ solver start with a *preamble* $\mathfrak{P}$ that contains assertions that describe the program's threads. Specifically, $\mathfrak{P}$ includes variables $s_1 \ldots, s_n$, each of sort S_Q for some basic state sort $Q \in \bar{Q}$; and for every s_i it includes assertions describing its initial states, its transitions and its requested and blocked events. After $\mathfrak{P}$, the solver expects an assertion Φ about the system's safety: $s = s_1 \parallel s_2 \parallel s_3 \parallel \ldots \parallel s_n \wedge \neg safe(s)$. The solver then returns SAT iff s is determined to be unsafe.

Figure 8.1 shows derivation rules used to implement a simple explicit-state model checker.[1] Intuitively, $\mathcal{TS}$ traverses the state graph in a DFS-like manner, looking for bad states. The underlying SAT solver manages the splits by deciding which successor state to check at every point. The process ends when a deadlock state is found or when the state space has been exhausted and no derivation rules apply; an example appears in Figure 8.2. As demonstrated in the next section, this implementation allows us to seamlessly leverage other theory solvers in curtailing the state space, which may reduce the overall runtime. Additional details and proofs of correctness and termination appear in the appendix in Section 8.9.1.

[1]While we do not assume the system is finite-state, we do assume that the initial states and the successors for each state are finite and decidable.

$$\textsc{Start}\ \frac{\Gamma[\neg safe(s)]}{\Gamma, \neg safe_state(s,q_1)\ \ldots\ \Gamma, \neg safe_state(s,q_n)}\ \text{IF}\ \begin{array}{c}\Gamma \vDash_{TS} q_1,\ldots,q_n \text{ ARE THE INITIAL STATES OF } s\\ \text{AND } \forall_{1\le i\le n}.\ \neg safe_state(s,q_i) \notin \Gamma\end{array}$$

$$\textsc{Decide}\ \frac{\Gamma[\neg safe_state(s,q)]}{\Gamma, \neg safe_state(s,q_1)\ \ldots\ \Gamma, \neg safe_state(s,q_n)}\ \text{IF}\ \begin{array}{c}\Gamma \vDash_{TS} q_1,\ldots,q_n \text{ ARE THE SUCCESSORS OF } q\ (n\ge 1)\\ \text{AND } \neg deadlock(s,q) \notin \Gamma\end{array}$$

$$\textsc{Unsat}\ \frac{\Gamma[\neg safe_state(s,q)]}{\bot}\ \text{IF}\ \forall q'.\ \neg safe_state(s,q') \in \Gamma \implies \neg deadlock(s,q') \in \Gamma$$

$$\textsc{Deadlock-Lemma}: \mathfrak{P} \land \Phi \implies \neg deadlock(s,q)\ \ \text{IF}\ \ q \text{ HAS A SUCCESSOR IN } s$$

Figure 8.1: Γ represents an arbitrary set of assertions that the solver has gathered at a given state, and $\Gamma[\phi]$ indicates that ϕ appears in Γ. The *Start* rule starts the traversal of the graph: the solver initiates a forward reachability search for bad states by nondeterministically guessing an initial state that is unsafe. When a state with unvisited successors is asserted to be unsafe, the *Decide* rule is used to nondeterministically assert that one of its successors is unsafe. Splitting is handled through the splitting-on-demand feature of the DPLL(T) framework. The UNSAT rule closes branches that fail to reach a deadlock state. If all branches terminate with $\bot$, UNSAT is returned; otherwise, if a branch terminates with a state other than $\bot$ where no rule is applicable, we return SAT. The last rule, *Deadlock-Lemma*, is a lemma generation rule: the resulting lemma is theory-valid, i.e. does not depend on the context in which it was generated. These lemmas mark that a non-deadlock state has been visited, and that it does not need to be revisited in the future. As part of the proof strategy, the $\mathcal{TS}$ solver invokes the lemma generation rule for (s,q) immediately after the *Decide* rule is invoked for $\neg safe_state(s,q)$, and only then (provided that q is not a deadlock state). This strategy, together with the side-conditions on the derivation rules, ensures that no state is visited more than once. See Section 8.9.1 of the appendix for more details.

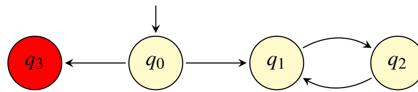


Figure 8.2: The depicted program has a reachable deadlock state, q_3. After reading the preamble, the solver uses the *Start* rule to assert $\neg safe_state(s,q_0)$. Then, it invokes the *Decide* rule for state q_0, nondeterministically asserting $\neg safe_state(s,q_1)$. This invocation of *Decide* is followed by the generation of the lemma $\neg deadlock(s,q_0)$. Next, *Decide* is invoked for state q_1, generating the assertion $\neg safe_state(s,q_2)$ — followed by the lemma $\neg deadlock(s,q_1)$. *Decide* is then invoked for state q_2, generating $\neg safe_state(s,q_1)$ and the lemma $\neg deadlock(s,q_2)$. At this point, the conditions for the UNSAT rule are met, and the solver closes this branch of the tree. The solver backtracks to the last nondeterministic split and generates the assertion $\neg safe_state(s,q_3)$. State q_3 is deadlocked, and so the *Deadlock-Lemma* rule is not invoked. No additional derivation rules apply, and so the process terminates with a SAT result, indicating that the system is unsafe.

8.4 Automatic Analysis of Transition Systems

The calculus in Section 8.3 captures the basic proof strategy of our theory solver: a forward reachability search. We next enrich this basic strategy with additional derivation rules, aimed at narrowing down the state space that needs to be explored. The idea is to include within the $\mathcal{TS}$ solver a database of *structural patterns* that characterize common/useful threads and alongside each pattern also to keep lemmas that describe these threads' behavior in the language of some other theory in CVC4. As the $\mathcal{TS}$ solver traverses the state space, it also repeatedly checks to see if any of the patterns apply to the threads at hand. When a match is found, the solver asserts the matching lemmas to the SMT framework. Sometimes, these lemmas may be contradictory to the assertion that the safety property is violated along the current search path, and another theory solver will raise a conflict: this will cause the $\mathcal{TS}$ solver to backtrack and check other areas of the state space.

We demonstrate the method on a simple example, that we have already encountered in previous chapters. Observe an $\mathcal{RWB}$ program over event set $E = \{0,1\}$ that generates the event sequence $(0^5 \cdot (0+1))^\omega$. The program has three threads, depicted in Figure 8.3. The safety property to be verified is that event 1 is never triggered (and so, the program is unsafe). Observe that direct model checking of this system requires visiting 6 composite states.

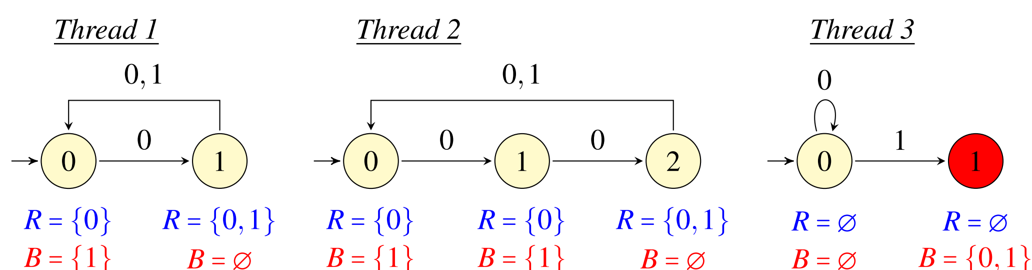


Figure 8.3: Thread 1 counts the number of events triggered so far, modulo 2. Every second event it requests both events 0 and 1; otherwise, it requests 0 but blocks 1. Thread 2 does the same, but counts modulo 3. Thread 3 is a *bad marker*: it waits for a violation to occur, i.e. for 1 to be triggered, and then goes into a "bad" state that blocks all the events, forcing a deadlock. This $\mathcal{RWB}$ program can have 0 triggered at every index, and can have 1 triggered precisely every 6 events. Consequently, it is unsafe.

For this program as input, the $\mathcal{TS}$ solver performs the following automatic compositional proof. First, it compares the transition systems to its pattern bank, and recognizes that they match the *looped* thread mold — a thread whose state is determined uniquely by the step index in the run (assuming a violation has not occurred). This is a structural property of each thread, that is checked locally and in isolation from its siblings. After determining that all threads are looped, the solver finds all individual thread states in which 1 is not blocked. In our case, this is state 1 for thread 1, state 2 for thread 2, and state 0 for thread 3. Denoting composite states

as triplets, this is state $\langle 1,2,0\rangle$. Finally, the solver uses the gathered information to generate the following lemma in order to curtail the state space:

$$\mathfrak{P} \land \Phi \implies ((\neg safe_state(s,\langle 0,0,0\rangle) \implies \neg safe_state(s,\langle 1,2,0\rangle)) \land \exists t:\mathbb{N}.\ (t \equiv 1 \pmod 2) \land (t \equiv 2 \pmod 3) \land (t \equiv 0 \pmod 1))).$$

This lemma connects the safety of the initial state $\langle 0,0,0\rangle$ with that of the only state in which 1 is not blocked, state $\langle 1,2,0\rangle$ — provided that there exists an integer t for which $t \equiv 1 \pmod 2$, $t \equiv 2 \pmod 3$ and $t \equiv 0 \pmod 1$. Because, in looped threads, the step index determines the state, this last part captures the fact that state $\langle 1,2,0\rangle$ is reachable.

Upon generation of this lemma, CVC4 asserts the lemma's arithmetical clauses to the arithmetic solver. If the latter determines that there is no solution for t, CVC4 answers UNSAT on the entire query. This signifies that the system is safe, which is indeed the case if state $\langle 1,2,0\rangle$ cannot be reached. However, if the arithmetic solver manages to solve for t, as is the case here, the $\mathcal{TS}$ solver continues exploring the successors of state $\langle 1,2,0\rangle$ and discovers that it has a bad successor. Then, SAT is returned for the query.

The key observation is that through the automatically generated lemma, the 4 intermediate states between state $\langle 0,0,0\rangle$ and $\langle 1,2,0\rangle$ did not need to be explored. Because the threads matched the looped pattern, CVC4 was able to deduce that these intermediate states would be safe iff state $\langle 1,2,0\rangle$ was safe. Further, because the arithmetic solver can solve for t more quickly than the intermediate states can be traversed (especially when generalizing to $(0^n \cdot (0+1))^\omega$ for a large n), the solver's performance is improved.

Pattern Matching. The $\mathcal{TS}$ solver's pattern database consists of *pattern matchers*. A pattern matcher P is comprised of a family of *recognizer predicates* $\{R_n\}_{n\geq 1}$, where R_n is defined over n transition system variables $s_1,\ldots,s_n$, and a lemma generating function f (described later). For input system $s = s_1 \parallel \ldots \parallel s_n$, we say that pattern P applies to s if $R_n(s_1,\ldots,s_n)$ evaluates to true. The R_n predicates can encode various facts about the transition systems: e.g., that threads always or never block certain events, that they have a certain state that must always be revisited, that certain events always send threads into a deadlock state, etc. For example, in the previously discussed looped pattern, R_n evaluates to true iff each of the threads' states has precisely one successor state.

In our proof-of-concept C++ implementation, recognizer predicates are coded as Boolean methods that take as input a list (of arbitrary length) of transition systems. Upon receiving a query, the $\mathcal{TS}$ solver passes the input program's threads to the recognizer predicates of each of the patterns, to determine which patterns apply in this case. Recognizer implementations may traverse the given transition systems, compute strongly connected components, etc. The only restriction, needed for the method to be efficient, is that recognizers do not compute the composite transition systems of the system; they are restricted to (polynomial) operations on

the individual threads. Thus, the complexity of pattern matching is polynomial in the size of the individual threads — and because these threads are typically exponentially smaller than the composite program [4], we can quickly test multiple patterns.

The second component in a pattern matcher is the lemma generating function, f. When pattern P applies to an input program, its lemma generating function is invoked repeatedly during state space traversal, in order to allow P to generate lemmas that affect the search. Specifically, f is invoked whenever $\mathcal{TS}$ visits a new state q (i.e., after the *Decide* rule generates the assertion $\neg \mathit{safe_state}(s,q)$), and returns a (possibly empty) list of lemmas concerning the safety of state q. The $\mathcal{TS}$ solver then asserts these lemmas to the underlying SAT engine, and other theories may use them in trimming the search space. In practice, the generated lemmas may depend on parameters extracted from the input threads by the pattern recognizers. For example, in the looped pattern, the size of the loop is extracted by the recognizer and is then used in generated lemmas.

Limitations. The above example demonstrates our method's potential advantages, but also raises a question regarding its generality: can the pattern database be sufficiently extensive, i.e. apply to a sufficient range of programs, so as to make our approach worthwhile? Indeed, if one needed to "teach" the solver new patterns for every new input program, the method would boil down to a manual compositional proof.

We believe that the answer to this question is affirmative: our findings show that even a small set of patterns included within the $\mathcal{TS}$ solver may already apply to broad classes of interesting programs. We demonstrate two such cases, *periodic programs* and *programs with arrays*, in Sections 8.5 and 8.6. Still, adding new patterns is not a trivial task, and so we store them in a central repository — amortizing the cost of adding additional patterns over future applications.

The $\mathcal{TS}$ Solver vs. Model Checking. In the simple example given above, our theory-aided approach could also be implemented by a more standard design: a model checker that issues queries to a black-box SMT solver. Our motivation for conducting model checking within the $\mathcal{TS}$ solver is in handling more elaborate examples, in which SMT theories partake in directing the state space traversal (see, e.g., Section 8.5). While such cases can still be accommodated by a model checker that is "running the show" and an SMT solver that exposes proper callbacks, we feel that a DPLL(T)-based solution is cleaner, and also more extensible and robust. By encoding the state traversal engine as a few axioms and lemma generation rules, and by having the pattern matching mechanism likewise generate lemmas, the complexity of integrating and synchronizing the two is automatically and seamlessly handled by CVC4's DPLL(T) core — simplifying the implementation of the $\mathcal{TS}$ solver. Further, this enables the $\mathcal{TS}$ solver to be plugged into any other SMT solver that adheres to the DPLL(T) framework.

8.5 Verifying Periodic Programs

In this section, we discuss the theory-aided verification of *periodic programs* [116] — a class of single processor scheduling problems that have been widely studied over the last decades. A periodic program consists of a finite set of *tasks* $T_1, \ldots, T_n$, which are processes that repeatedly need to be scheduled for execution on a single processor. Each task T_i is characterized by its period time P_i and an execution time C_i (for simplicity, we ignore here other parameters such as relative deadlines and initial offsets). From task T_i's point of view, the execution of the program is divided into time cycles of length P_i each, and in each such cycle the task must be alloted C_i time slots on the processor. The least common multiple of the tasks' period times is called the program's *hyper-period*. Tasks may have *priorities*: a task with a higher priority will preempt another if both need to be scheduled at a specific point in time. A periodic program is said to be *schedulable* if there exists a task scheduling in which no deadlines are violated.

For example, consider a periodic program with 3 tasks, T_1, T_2, T_3, each with execution time $C_1 = C_2 = C_3 = 1$. The tasks' periods are 2, 3 and 6, respectively. As depicted in Figure 8.4, every 2 consecutive time slots must include a scheduling of T_1; every 3 consecutive slots include a scheduling of T_2; and every 6 consecutive slots include a scheduling of T_3. As the figure shows, this program is schedulable, and its schedule repeats after every 6 steps (the hyper-period).

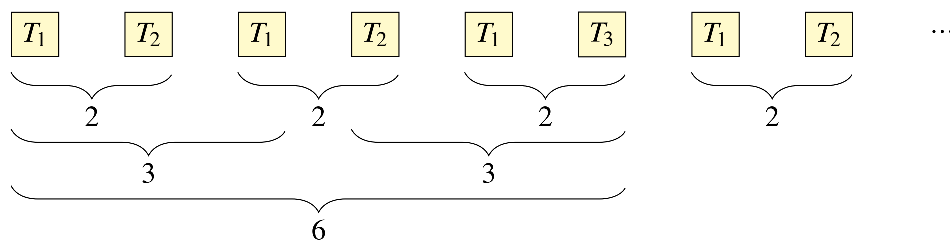


Figure 8.4: A scheduling for a periodic program with 3 tasks, T_1, T_2, T_3, with execution time $C_1 = C_2 = C_3 = 1$ and periods $P_1 = 2$, $P_2 = 3$ and $P_3 = 6$.

Here, we study the verification of safety properties in periodic programs: we assume that the input program is schedulable, and check whether it can violate a given property. This is typically done by transforming the periodic program into an equivalent sequential program and then verifying it using standard model checking [45]. Our approach is similar, but we seek to leverage the program's special structure in order to explore only a portion of its state space.

As we saw in Chapter 7, periodic programs may be programmed in the $\mathcal{RWB}$ model by expressing each task as a thread that requests an event whenever the task needs to be scheduled. Priorities are expressed using blocking: a thread (task) may block events belonging to other threads with lesser priorities. Figure 8.5 illustrates the structure of task threads and describes their pattern matcher.

Whenever all input threads are identified as *tasks*, the pattern recognizer reports that the

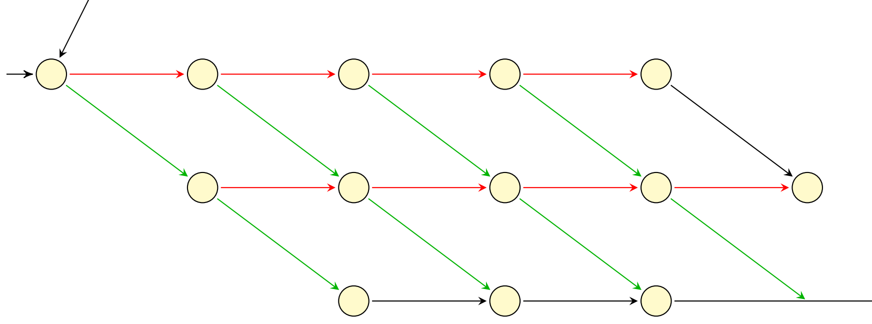


Figure 8.5: An $\mathcal{RWB}$ implementation of a *task* thread with period time $P = 5$ and execution time $C = 2$. The thread's underlying transition system can be regarded as a $(C+1)\times P$ matrix, where the columns represent the time passed since the beginning of the period and the rows represent the number of times the task has been scheduled so far. Green edges in the figure represent the task being successfully scheduled (i.e., its requested event was triggered) and red edges represent the task not being scheduled (an event requested by some other task was triggered). Thus, with every time unit the state moves to the right, and if the task was scheduled it also moves one row down. If the task's deadline is violated, it enters a deadlock state (the rightmost state in the figure). The task pattern matcher traverses the state graph of each input thread and checks whether it has these structural properties. If so, it also extracts the task's P and C parameters and its sequence of requested events (not illustrated). If blocking is used to prioritize tasks, the matcher also extracts the prioritization hierarchy.

program is periodic. This causes the pattern's lemma generation function to be repeatedly invoked during state space traversal, so that it may generate lemmas aimed at curtailing the search space. For this purpose, we extend the signature of $\mathcal{TS}$ to include a sort $\mathbb{Z}^+$ for non-negative integers, and the predicate $deadlock_Q : S_Q \times \mathbb{Z}^+$. Intuitively, $deadlock_Q(s,t)$ indicates that a deadlock state in s is reachable in t steps from an initial state. Further, we extend $\mathcal{TS}$ to support backward reachability analysis, in addition to the forward reachability analysis afforded by the *safe_state* predicate. To this end, we add the $reachable_Q : S_Q \times Q$ predicate, with the following semantics:

$$\forall s : S_Q, q : Q.reachable(s,q) \implies$$
$$I(s,q) \vee \exists q' : S_Q, e : E.(Tr(s,q',e,q) \wedge R(s,q',e) \wedge \neg B(s,q',e) \wedge reachable(s,q'))$$

Intuitively, a state is reachable if it is initial or has a reachable predecessor. For more details, see Section 8.9.2 of the appendix.

The lemmas generated by the pattern matcher assert that there must be a time t within the hyper-period in which a violation occurs. They also limit the possible values of t based on the information gathered about the individual tasks. Specifically, the pattern matcher generates the lemma:

$$\mathfrak{P} \wedge \Phi \implies \exists t : \mathbb{Z}^+.\ deadlock(s,t) \wedge \Psi_t$$

where Ψ_t describes constraints on t that are deduced from the structure of the task threads. If

the arithmetic solver finds a solution t_0 for Ψ_t it assigns it to t, and the $\mathcal{TS}$ solver then translates it, by analyzing the task threads' possible locations in time t, into candidate reachable bad states $q_1,\ldots,q_\ell$:

$$\mathfrak{P} \wedge \Phi \wedge deadlock(s,t_0) \implies \vee_{i=1}^{\ell} reachable(s,q_i)$$

$\mathcal{TS}$ then performs backward reachability checks on candidates $q_1,\ldots,q_\ell$. If a path to an initial state is found, the system is unsafe and we are done. Otherwise, the contradiction forces the arithmetic solver to propose another solution $t = t_1$, which corresponds to additional candidate bad states. The process is repeated until the system is proven unsafe, or until all possible solutions are exhausted. Other bad states, which do not correspond to any of the proposed values of t, are guaranteed to be unreachable and are ignored.

In order to generate the constraints in Ψ_t, the pattern matcher identifies tasks *participating* in the violation: these are the threads whose requested events are part of a violating sequence. Then, it uses information about these threads, and about threads with higher priority, to put constraints on t.

We demonstrate this on a schedulable periodic program with 4 tasks: task T_1 with parameters $P_1 = 5, C_1 = 1$; T_2 with $P_2 = 6, C_2 = 1$; T_3 with $P_3 = 9, C_3 = 3$; and task T_4 with parameters $P_4 = 11, C_4 = 2$. Task 1 has the highest priority, task 2 has the second highest priority, and tasks 3 and 4 both share the lowest priority. The safety property in question is that it is impossible for task T_4 to be scheduled for three consecutive time slots. Here, direct model checking requires visiting 55000 states in the composite program.

By intersecting the violating event sequence with the events requested by each thread, the pattern matcher determines that T_4 is the only participating task. By the information extracted regarding task priorities, it deduces that tasks T_1 and T_2 supersede it. Then, it generates the Ψ_t constraint as follows. One conjunct in Ψ_t is $0 \leq t \leq 990$, as the hyper-period is $lcm(5,6,9,11) = 990$. Another conjunct is $((t \geq 3 \pmod 5) \wedge (t \geq 3 \pmod 6))$: if it did not hold, T_1 or T_2 would preempt T_4, preventing it from being scheduled 3 consecutive times. Yet another conjunct is $(t \leq 1 \pmod{11})$; it holds because in order for T_4 to be scheduled 3 consecutive times (with execution time $C_4 = 2$), a fresh period must start at time t or $t-1$. A few additional conjuncts are omitted. The complete lemma reduces the number of possible values for t from 990 to just 15, and the query as a whole entails exploring only 700 states out of 55000 reachable states in order to prove the system's safety.

8.6 Verifying Programs With Shared Arrays

Next we demonstrate the theory-aided verification of programs with shared arrays — a widespread construct in concurrent programming. In the $\mathcal{RWB}$ model, a shared m-ary ar-

ray with n cells may be implemented using n b-threads, each of size m. Each thread represents a single array cell and has a clique-like structure, where each state s_i is associated with a write event w_i and a read event r_i. Intuitively, each state s_i corresponds to a value v_i that is stored in the array cell. Whenever event w_i is triggered, the thread moves to state s_i; and whenever not in state s_i, the thread blocks r_i. Thus, other threads can request r_i in order to check if the thread is in state s_i (i.e., to check if the array cell has value v_i). See Figure 8.6 for an illustration. Note that this implementation is only needed for shared arrays; internally, threads may use any construct available in the underlying programming language.

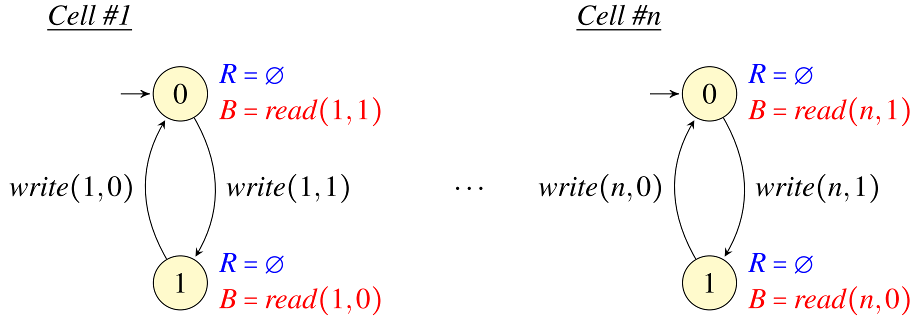


Figure 8.6: An $\mathcal{RWB}$ implementation of a binary array with n cells. Each cell is represented by a thread with two states, signifying the stored value in that cell, 0 or 1. Each thread/cell is associated with two write events, for 0 and 1; when they occur, the thread changes states to indicate the new stored value. Other threads in the program may read from a cell by requesting the two read events associated with it, one for 0 and one for 1; the read event that does not match the value in the cell will be blocked by the cell thread, and so only the "correct" read event may be triggered.

The $\mathcal{TS}$ solver has a pattern matcher that looks for threads that match this *array cell* pattern. If an array is found, the pattern matcher checks whether deadlocks are possible only in certain array configurations (e.g., when certain array cells hold certain values; an example appears later in this section). If such constraints are found, it generates a lemma that conditions the system's unsafety on the array threads reaching an unsafe configuration.

We demonstrate with an example. Observe a program with a shared array of size n and an initial state q_0. The array pattern matcher creates an array expression arr_{q_0} whose value at each index i is set to some fresh constant c_i. This expression is used to represent the value of the array in various states of the program. The matcher also creates a *target* array, arr_{target}, and asserts constraints on arr_{target} signifying the state that the array has to be in for a violation to occur. Then, it generates the lemma $\mathfrak{P} \wedge \Phi \implies (arr_{q_0} = arr_{target})$.

The bulk of the work is then performed as $\mathcal{TS}$ traverses the state space. Whenever a new state q is visited, the pattern matcher analyzes the threads (each of them separately), looking for array entries that have become fixed. This can be determined, e.g., when additional write events to a cell are never requested or are always blocked. Suppose that it is discovered that the

first cell's value has been fixed to e_0; then the lemma $\mathfrak{P} \wedge \Phi \wedge \neg \mathit{safe_state}(s,q) \implies (c_0 = e_0)$ is generated. If this is consistent with the earlier assertion $arr_{q_0} = arr_{target}$, the solver continues traversing the successors of q; otherwise, the array theory solver will raise a conflict, resulting in q's successor states not being traversed.

We now demonstrate the shared array pattern on the behavioral application for playing Tic-Tac-Toe from [86]. Recall that Tic-Tac-Toe is a game played between "X" and "O" players on a 3x3 board. Each in their turn, the players mark an empty square on the board with their respective sign. A player wins by completing a row, a column or a diagonal. If neither player errs, the game is guaranteed to end in a draw.

In [86], the authors construct a behavioral application that plays "O", where a human player plays "X". Suppose that we wish to prove that the game application upholds the property that "X never wins by taking the upper row"; verifying this property can be useful, e.g., during incremental development [86].

The theory-aided verification of this system is as follows. In the implantation of [86], the game board is in fact a ternary array — with values *empty*, O and X. By analyzing the individual threads, the $\mathcal{TS}$ solver recognizes this array and determines that a violation can occur only when the three upper row cells (say, cells 0, 1 and 2 in the array) are set to X. The $\mathcal{TS}$ solver then creates a 9-cell ternary array variable *arr*, and asserts that $\mathit{target} = \mathit{write}(\mathit{write}(\mathit{write}(\mathit{arr},0,X),1,X),2,X)$. Next, the solver writes fresh constants $c_0,\ldots,c_8$ to *arr*'s cells, and equates the result to *target*:

$$\mathfrak{P} \wedge \Phi \implies \mathit{target} = \mathit{write}(\ldots \mathit{write}(\mathit{write}(\mathit{arr},0,c_0),1,c_2)\ldots,8,c_8)$$

The Tic-Tac-Toe application has, for every board square, a thread that waits for write events and then blocks any additional writes. Thus, every triggered write event fixes an array entry, causing the $\mathcal{TS}$ solver to generate a lemma that equates the corresponding c_i constant to its final value. For instance, suppose the game so far has included moves $\mathsf{X}(0,1),\mathsf{O}(1,1),\mathsf{X}(2,2)$, and that it is now O's turn to play. We denote this current state by q_1. Now, suppose the $\mathcal{TS}$ solver explores a new state, q_2, reachable from q_1 when O marks square $\mathsf{O}(0,2)$. Thread analysis shows that starting in state q_2 the blocker thread will forever block any additional write events to square $(0,2)$, and so the $\mathcal{TS}$ solver generates the lemma: $\mathfrak{P} \wedge \Phi \wedge \neg \mathit{safe_state}(s,q_2) \implies (c_2 = O)$. This causes the array theory solver to raise a conflict, which in turn causes the underlying SAT solver to deduce that sate q_2 cannot be unsafe. Thus, backtracking is performed, and another successor of state q_1 is checked (effectively choosing another move, instead of $\mathsf{O}(0,2)$). Consequently, the successor states of q_2 need not to be explored.

8.7 Experimental Results

We evaluated our proof-of-concept tool, implemented as an extension to CVC4, by comparing it to BPMC — a symbolic model checker specifically designed for $\mathcal{RWB}$ programs [86, 7] (the tool and experiments are available online [108]). Our tool uses a portfolio approach: if the input program does not match any of the known patterns, the tool simply invokes BPMC (or any other model checker, for that matter). The decision of whether or not to invoke BPMC is made within seconds, rendering the performance of both tools effectively the same in these cases. Hence, for the remainder of this section we focus on inputs in which a pattern did apply and theory-aided model checking was indeed attempted.

We first compared the tools using a benchmark suite of over 120 hand-crafted $\mathcal{RWB}$ programs — some periodic, and some containing shared arrays. The benchmarks' sizes ranged from a few hundred to over 10 million reachable states, and contained both SAT and UNSAT instances. The results are depicted and discussed in Figures 8.7 and 8.8.

Next, we set out to test our tool's applicability to a large, real-world system by using it to verify safety properties on a web-server (implementing TCP and HTTP stacks) written in BPC [3]. We were very curious to see whether our pattern recognition mechanism would pick up any matching threads.

As it turns out, the shared array pattern proved useful in verifying this application. Per the TCP protocol, the web-server only accepts TCP *push* segments on active connections. Slightly simplified, a connection to a client is active if the client sent a *syn* segment but not a *fin* segment. This functionality is implemented using blocking: for every connection, a dedicated thread, named *EnsureActiveConnection*, blocks *push* events while the connection is inactive. This blocking is removed when a *syn* segment is received, and is restored when a *fin* segment is received. Thus, the *EnsureActiveConnection* threads were picked up as shared array cells by our tool: they each had two states, labeled *active* and *inactive*, with respective read events push and reject and write events syn and fin. Interestingly, the programmers of the web-server did not seem to have had this design pattern in mind [3].

We tested 10 safety properties on the web-server (see Figure 8.9). These properties included the proper rejection of messages on inactive connections, proper usage of allotted sequence numbers for outgoing segments, and the detection and blocking of unstable clients, who quickly and repeatedly opened and closed connections. The theory-aided approach did better on 7 of 10 instances (4 SATs and 3 UNSATs), demonstrating an average speedup of 16% over all instances. BPMC did better on 2 SAT and 1 UNSAT instances, where the property in question and the discovered patterns were disparate (e.g., properties involving proper usage of sequence numbers, that had nothing to do with the *EnsureActiveConnection* threads).

These initial results are encouraging. We conclude that (i) the theory-aided approach is

		# Instances	Avg. # States Explored			Avg. Time (milliseconds)		
			CVC4	BPMC	Change	CVC4	BPMC	Change
Periodic Programs	SAT	11	9994	9236	+8%	18791	15894	+18%
	UNSAT	50	35299	184388	-80%	10247	15041	-31%
	UNSAT†	6	59816	8195666	N/A	170673	809946	N/A
	Timeout	2						
Shared Arrays	SAT	35	24416	293525	-91%	24882	168755	-85%
	UNSAT	15	121133	511292	-76%	124911	292779	-57%
	UNSAT†	6	267000	1989666	N/A	359324	1510028	N/A
Total		111	190842	998441	-80%	178831	492469	-63%

Figure 8.7: Experiments on a benchmark suite, conducted using an X230 Lenovo laptop with 16GB memory. The suite contained SAT and UNSAT instances of periodic $\mathcal{RWB}$ programs and programs with shared arrays. The table compares our tool (*CVC4* columns) to the *BPMC* tool, measuring the average number of explored states and average solving time for each category. The *Change* columns measure the effectiveness of CVC4 in comparison to BPMC. The UNSAT† row indicates UNSAT instances on which CVC4 answered correctly but on which BPMC ran out of space (but listing the number of states it was able to explore). The *Timeout* row indicates instances on which both tools ran out of space/time. We did not encounter examples on which BPMC returned and CVC4 did not. The table reveals that for SAT queries on periodic programs, BPMC was able to outperform CVC4. This is not surprising; indeed, the pattern for periodic programs is designed to quickly show that bad states are unreachable, which is not the case for SAT instances. In all other categories, i.e. UNSAT queries on periodic programs and both types of queries on programs with shared arrays, CVC4 typically outperformed BPMC. Instances where BPMC did better were either very small (the cost of thread analysis and pattern matching exceeded the cost of the actual model checking), or instances where the property in question had nothing to do with the recognized patterns, making it impossible for our tool to trim the search space. The UNSAT† instances had too many states for BPMC to cover, but with the theory-aided approach we were able to trim the search space down to a manageable size. Finally, the *Timeout* instances were too large to handle, even with theory-aided pruning. The *Total* row sums up the instances solved by both tools, demonstrating an encouraging average speedup of 63%; these 111 instances are also the ones described in the graph in Figure 8.8.

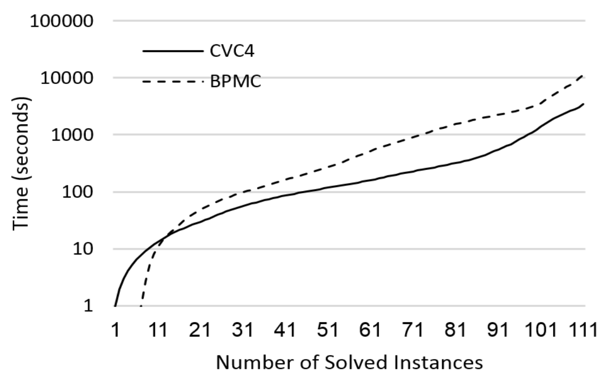


Figure 8.8: Performance of both tools, compared over the benchmark suite.

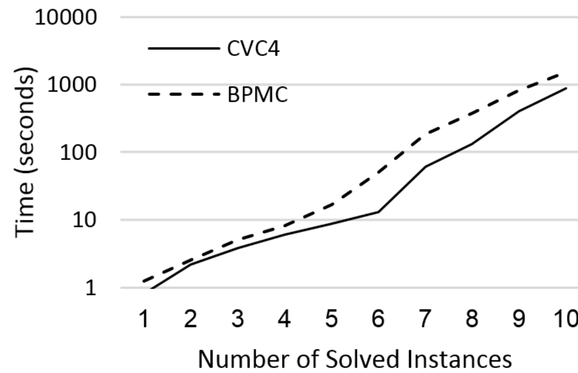


Figure 8.9: Experiments on the web-server.

viable, in the sense that the stored patterns apply to real programs, sometimes significantly reducing verification times; and (ii) that performance may be further improved by enhancing the portfolio approach; i.e., if we were able to more accurately characterize cases in which, despite matching a stored pattern, a thread does not affect the property in question, we could delegate those cases to BPMC and achieve faster running times. This is left for future work.

8.8 Related Work and Discussion

In this work, we proposed a framework for the automated compositional verification of concurrent software. Our technique was based on casting the model checking problem into the DPLL(T) framework used by the CVC4 SMT solver, and then utilizing other theory solvers to prune the search space in order to improve performance. Other theories were able to affect the search through lemmas in their respective languages that were generated by matching the input program's threads to presupplied patterns.

SMT solving has been used for various verification-related tasks such as lemma dispatching [124, 48], reachability analysis [30] and model checking concurrent programs [62, 39]. Our technique shares some of these aspects, but differs in that the state exploration is driven by an SMT solver and in that lemmas are derived using stored patterns. A related approach for circuit verification appears in [34], where the input is analyzed to find unreachable states in advance. Our framework follows a similar spirit, but extends the technique to concurrent software and utilizies a modern SMT solver.

In [147], the authors extend the Z3 solver with an automaton sort for symbolic automata over infinite alphabets. It would be interesting to combine this technique with ours, enabling it to reason about $\mathcal{RWB}$ programs with infinite event sets.

We evaluated our technique on two broad classes of $\mathcal{RWB}$ programs: periodic programs and programs with shared arrays. Specifically, we showed how the $\mathcal{TS}$ solver may leverage

CVC4's arithmetic and array theory solvers in order to expedite the model checking process. Others have explored SMT-based techniques for similar models; e.g., the validation of guessed invariants in Lustre programs [106]. We consider this as encouragement that applying SMT-based techniques to synchronous, discrete event models may prove fruitful, and intend to extend our technique to Lustre as well.

We find our initial results encouraging, and plan to continue extending our pattern database. One direction that we are presently pursuing is the addition of a new pattern matcher that leverages CVC4's *string* theory solver [115], by translating constraints imposed by certain types of input threads into regular expressions. Indeed, a prototype implementation we have created shows interesting potential.

8.9 Appendix

8.9.1 Derivation Rules for the $\mathcal{TS}$ Solver

A *derivation tree* consists of nodes containing sets of assertions. The root node contains an initial set of assertions and each non-leaf node is labeled by a derivation rule used to derive the children of the node from the node itself. The derivation rules used by the $\mathcal{TS}$ solver give rise to a sequence of derivation trees (called a *derivation*). The initial tree in the derivation contains only a single node with the initial set of assertions. Each subsequent tree in the sequence is obtained from its predecessor by the application of a derivation rule to one of the predecessor's leaves. A branch terminating with a leaf consisting of the value $\perp$ is called a *closed* branch; if all branches are closed, we say that the derivation tree is closed. A derivation culminating with a closed derivation tree indicates that the initial set of assertions is unsatisfiable. A derivation that leads to a derivation tree containing a leaf node that is not $\perp$ and to which no derivation rule can be applied indicates that the initial set of assertions is satisfiable. When such a tree is produced, the derivation terminates.

We now describe the actual derivation rules used by the theory. The first rule, used to initiate the traversal of the state space, is the *Start* rule:

$$\textsc{Start}\ \frac{\Gamma[\neg \mathit{safe}(s)]}{\Gamma, \neg \mathit{safe_state}(s, q_1)\ \ldots\ \Gamma, \neg \mathit{safe_state}(s, q_n)}\ ,$$

where Γ denotes the initial set of assertions (in particular it includes $\mathfrak{P}$ and Φ). The derivation rule contains a nondeterministic split such that each branch adds a single assertion to Γ.[2] The

[2] Splits are implemented by utilizing the SAT solver through the DPLL(T) splitting-on-demand framework.

Start rule is only applicable when the following guard conditions hold:

$$\Gamma \vDash_{TS} \forall q \in Q.\ (I(s,q) \iff (q = q_1) \vee (q = q_2) \vee \ldots \vee (q = q_n)),$$
where s is of sort S_Q,
$$\neg \mathit{safe_state}(s, q_i) \notin \Gamma \text{ for } i \in \{1 \ldots n\}.$$

Intuitively, the *Start* rule translates the fact that the system is unsafe into an assertion that one of the initial states is unsafe. The first part of the guard condition ensures that all of the potential initial states are considered as possibilities, and the second part ensures that the rule is applied only once at the beginning of the derivation.

Next comes the *Decide* rule. This rule performs a similar function to that of *Start*, and applies when the traversal of the state space is already underway:

$$\textsc{Decide}\ \frac{\Gamma[\neg \mathit{safe_state}(s,q)]}{\Gamma, \neg \mathit{safe_state}(s,q_1)\ \ldots\ \Gamma, \neg \mathit{safe_state}(s,q_n)}\ .$$

The *Decide* rule is applicable when the following conditions hold:

$$\neg \mathit{deadlock}(s,q) \notin \Gamma \tag{8.1}$$

$$\Gamma \vDash_{TS} \forall q' \in Q, e \in E.\ ((\mathrm{Tr}(s,q,e,q') \wedge R(s,q,e) \wedge \neg B(s,q,e)) \iff (q' = q_1) \vee (q' = q_2) \vee \ldots \vee (q' = q_n)) \text{ where } s \text{ is of sort } S_Q \text{ and } n \geq 1. \tag{8.2}$$

Intuitively, if a state q has not already been recorded as a non-deadlock state (condition 8.1), the *Decide* rule lets the solver derive the unsafety of one of q's successor states, as long as such successors exist (condition 8.2). Note in particular that the *Decide* rule will not apply if q is a deadlock state because condition 8.2 will fail.

For cases where *Decide* is not applicable and no deadlock state has been found, we have the *Unsat* rule:

$$\textsc{Unsat}\ \frac{\Gamma[\neg \mathit{safe_state}(s,q)]}{\bot}$$

The *Unsat* rule is applicable when there are no more states to explore on this branch (and at least one state has been explored) and no deadlock has been discovered. Specifically, the guard condition is:

$$\text{whenever } \neg \mathit{safe_state}(s,q') \in \Gamma, \text{ we also have } \neg \mathit{deadlock}(s,q') \in \Gamma.$$

In addition to the derivation rules, we use the *theory lemma* feature of the DPLL(T) framework to periodically generate a *deadlock lemma*:

$$\mathfrak{P} \wedge \Phi \implies \neg \mathit{deadlock}(s,q)$$

Of course, as a theory lemma must be valid in the theory, this lemma can only be generated if q is not a deadlock state in s, or, formally,

$$\mathfrak{P}, \Phi \vDash_{TS} \exists q' \in Q, e \in E.\ \mathrm{Tr}(s,q,e,q') \wedge R(s,q,e) \wedge \neg B(s,q,e),$$
$$\text{where } s \text{ has the sort } S_Q \text{ and } q \text{ has the sort } Q.$$

Because Γ includes $\mathfrak{P}$ and Φ, the DPLL(T) framework will ensure that whenever a deadlock lemma is added, the predicate $\neg \mathit{deadlock}(s,q)$ will be included in Γ on all future branches. For our purposes, we can view the generation of a deadlock lemma as a derivation rule which modifies a given derivation tree by adding $\neg \mathit{deadlock}(s,q)$ to *every* node of the tree (except those labeled with $\perp$).

Clearly, not every strategy for applying derivation rules and deadlock lemmas will be complete. Thus, we enforce the following strategy. A deadlock lemma for s and q is generated immediately after an invocation of *Decide* triggered on $\neg \mathit{safe_state}(s,q)$; and moreover, this is the only time a deadlock lemma is generated.

This strategy guarantees that deadlock lemmas are valid and are generated only for states whose successors have been expanded, and that each state's successors are only expanded at most once in any derivation.

Lemma 10. For each q, the *Decide* rule is applied at most once with trigger $\neg \mathit{safe_state}(s,q)$ in any derivation starting with *Start* triggered on $\neg \mathit{safe}(s)$.

Proof. Observe a derivation tree T in which the *Decide* rule has just been applied with trigger $\neg \mathit{safe_state}(s,q)$. The strategy used by the $\mathcal{TS}$ solver guarantees that in this case, the invocation of the *Decide* rule for q will be immediately followed by the generation of a deadlock lemma for q (recall that if *Decide* was applied to q, it cannot be a deadlock state). This transforms T into a new tree, T', in which the assertion $\neg \mathit{deadlock}(s,q)$ has been added to every node.

The $\mathcal{TS}$ solver now continues working on tree T'. In order for the *Decide* rule to be reapplied to q in some node of T', the assertions in that node must not contain $\neg \mathit{deadlock}(s,q)$. However, this is clearly not possible, as new nodes in T' are derived from existing nodes by the addition of terms (rules *Start*, *Decide* and the lemma generation rule). Thus the only nodes in the tree where $\neg \mathit{deadlock}(s,q)$ is not present are nodes derived using the *Unsat* rule, in which case the branch is closed and no further rules may be applied. □

We also observe that in every derivation that starts with $\mathfrak{P}, \Phi, \neg safe(s)$, the *Start* rule is applied precisely once.

Lemma 11. In every derivation starting with $\mathfrak{P}, \Phi, \neg safe(s)$, the *Start* rule is the first rule applied and it is applied precisely once.

Proof. This holds because (i) *Start* is the only rule applicable at the very first step of the derivation process; and (ii) as rules only add to the set of assertions along any branch, every leaf contains all of the assertions in its parent nodes; and (iii) because of (ii), once *Start* has been applied once, its guard rule prevents it from being applied again to any successor derivation tree. □

The described derivation rules and proof strategy also guarantee partial correctness (i.e., soundness and completeness):

Proposition 14. Let s be an input system for which the $\mathcal{TS}$ solver terminates when started with $\mathfrak{P}, \Phi, \neg safe(s)$. Then s is safe iff the final derivation tree is closed.

Proof. Suppose that s is unsafe; then let $q_0 \to q_1 \to \ldots \to q_n$ be a shortest path in s such that q_0 is an initial state, q_{i+1} is a successor of q_i for all $0 \leq i < n$, and q_n is a deadlock state.

We now claim that for each derivation tree after the first, there is a value of i with $0 \leq i \leq n$ such that the tree has a leaf node containing $\neg safe_state(s, q_i)$ and not containing $\neg safe_state(s, q_k)$ for $i < k \leq n$ and also not containing $\neg deadlock(s, q_k)$ for $i \leq k \leq n$. We prove this by induction on the derivation. It is easy to see this is true for the second derivation tree as it is generated using the *Start* rule and so does not yet contain any assertions of the form $\neg deadlock(s, q)$. Thus, the largest i such that $\neg safe_state(s, q_i)$ appears in the tree satisfies the claim.

Now, assume the claim holds for some tree T in the derivation (whose position is second or later) with value i. Let T' be its successor in the derivation. We consider each of the possible ways in which T' could have been obtained from T:

- T' cannot be derived using *Start* as (by Lemma 11) this rule can only be applied once in the sequence.
- Suppose T' is derived from T using the *Unsat* rule. It cannot be the case that the *Unsat* rule was applied to the leaf containing $\neg safe_state(s, q_i)$ as that would require $\neg deadlock(s, q_i)$ to also be present in the leaf node and our inductive hypothesis states it is not. Thus the leaf satisfying the claim in T is still present and satisfies the claim in T'.
- Suppose T' is derived using *Decide* followed by an application of the deadlock lemma (we lump these two rules together for simplicity and wlog). If the *Decide* rule uses

$\neg$*safe_state*(s, q_i) as its trigger, then T' contains a new leaf which differs from the previous leaf by the addition of $\neg$*safe_state*(s, q_{i+1}) and $\neg$*deadlock*(s, q_i). By minimality of the path, there are no new leaves with $\neg$*safe_state*(s, q_k) with $k > i+1$ and $q_{i+1} \neq q_i$. Thus, it is clear that the value $i+1$ satisfies the claim in T'. If the *Decide* rule uses $\neg$*safe_state*(s, q) as its trigger with $q \neq q_i$, there are two possibilities. The first possibility is that q contains a successor q_k with $k > i$. In this case it is easy to see that the value k satisfies the claim in T'. If q contains no such successor, then it is clear that the value i continues to satisfy the claim in T'.

This shows the claim is true for every tree in the derivation, in particular the final tree which implies that the final tree is not closed.

For the other direction, suppose towards contradiction that s is safe but that the final derivation tree is not closed. Then, there is a leaf node in the tree which is not $\perp$ and to which rules *Start*, *Decide*, and *Unsat* may not be applied. We denote this node by α.

Node α has at least one assertion of the form $\neg$*safe_state*(s, q) (generated by the *Start* rule at the beginning of the derivation). Any other terms of this form were generated by the *Decide* rule. From this, and by the guard condition for *Decide*, it follows that for every state q such that $\neg$*safe_state*(s, q) is a term in α, q is reachable in s.

We now distinguish between two cases. If there exists some q such that $\neg$*safe_state*(s, q) is in α but $\neg$*deadlock*(s, q) is not, then the *Decide* rule may be applied — because we know, by our assumption that s is safe, that q is not a deadlock state. If there is no such q, then *Unsat* may be applied. Either case contradicts our assumption that no rule may be applied in α, as needed. □

Proposition 14 tells us that when the $\mathcal{TS}$ solver terminates, it gives a correct result. However, it may not always terminate. The following lemma characterizes one case in which termination is guaranteed:

Proposition 15. For an input system s with a finite set of states, the $\mathcal{TS}$ solver terminates.

Proof. Lemma 11 ensures that in every derivation, the *Start* rule is applied only once. Further, as Lemma 10 shows, the *Decide* rule may only be applied once per state in the entire derivation. Similarly, according to the proof strategy used by the solver, the lemma generation rule may only be applied once per state, as it is only invoked after an invocation of *Decide*. Finally, the last derivation rule, *Unsat*, always reduces the number of open branches in the tree, and thus may only be applied a finite number of times once no other rules are available. From all of the above, we deduce that every derivation is of finite length. □

8.9.2 Backward Reachability

The derivation rules given so far — *Start*, *Decide* and *Unsat* — effectively perform a reachability analysis over the state space, looking for bad states. In some cases, as in the case of periodic programs (Section 8.5), it may be useful to also perform a backward reachability search, starting at bad states and checking if they are reachable. This is performed by extending the $\mathcal{TS}$ theory with the predicate $reachable_Q : S_Q \times Q$, where $reachable(s,q)$ signifies that state q is reachable in s. The semantics are extended to include (for each $Q \in \bar{Q}^+$):

$$\forall s : S_Q, q : Q.\ reachable(s,q) \implies$$
$$I(s,q) \ \vee\ \exists q' : S_Q, e : E.\ (\mathrm{Tr}(s,q',e,q) \wedge R(s,q',e) \wedge \neg B(s,q',e) \wedge reachable(s,q')).$$

Intuitively, a state is reachable if it is either initial or if it has a predecessor state that is reachable. This last condition is what makes the search "backward": we will start at bad states, and attempt to construct a legal path backwards, towards an initial state.

We will also use an alternative (but equivalent) semantics for the system safety predicate. For each $Q \in \bar{Q}^+$:

$$\forall s : S_Q.\ \neg safe(s) \iff \exists q : Q.\ (deadlock(s,q) \wedge reachable(s,q))\ ,$$

and so a system is unsafe if it has an unsafe initial state or (equivalently) a reachable deadlock state.

The derivation rules for this case are extended to include "backward" versions of the three original rules. Here, the negation of the initial state predicate plays the role that was previously played by the *deadlock* predicate, namely that of marking those states that have been visited by generating a lemma. The first derivation rule is the *BStart* rule:

$$\textsc{BStart}\ \frac{\Gamma[\neg safe(s)]}{\Gamma, reachable(s,q_1)\ \ldots\ \Gamma, reachable(s,q_n)}$$

The *BStart* rule is only applicable when the following guard conditions hold:

$$\Gamma \vDash_{TS} \forall q \in Q.\ (deadlock(s,q) \iff$$
$$(q = q_1) \vee \ldots \vee (q = q_n)), \text{where } s \text{ is of sort } S_Q$$
$$reachable(s,q_i) \notin \Gamma \text{ for } i \in \{1 \ldots n\}.$$

Intuitively, states $q_1, \ldots, q_n$ are the deadlock (bad) states of the system — the states on which

we want to perform the backward reachability analysis. Next comes the *BDecide* rule:

$$\text{BDECIDE}\ \frac{\Gamma[\mathit{reachable}(s,q)]}{\Gamma,\mathit{reachable}(s,q_1)\ \ldots\ \Gamma,\mathit{reachable}(s,q_n)}\ .$$

The *BDecide* rule is applicable when the following conditions hold:

$\Gamma \vDash_{TS} \neg I(s,q) \neg I(s,q) \notin \Gamma\Gamma \vDash_{TS} \forall q' \in Q, e \in E.\ (\mathrm{Tr}(s,q',e,q) \wedge R(s,q',e) \wedge \neg B(s,q',e)) \iff$
$(q' = q_1) \vee \ldots \vee (q' = q_n)),$ where s is of sort S_Q and $n \geq 1$.

We also have a special *BDecide2* rule which handles the special case when a non-initial state has no predecessors:

$$\text{BDECIDE2}\ \frac{\Gamma[\mathit{reachable}(s,q)]}{\Gamma}\ .$$

The *BDecide2* rule is applicable when the following conditions hold:

$\Gamma \vDash_{TS} \neg I(s,q)$
$\neg I(s,q) \notin \Gamma$
$\Gamma \vDash_{TS} \forall q' \in Q, e \in E.\ \neg(\mathrm{Tr}(s,q',e,q) \wedge R(s,q',e) \wedge \neg B(s,q',e))$ where s is of sort S_Q.

Finally, we have the *BUnsat* rule:

$$\text{BUNSAT}\ \frac{\Gamma[\mathit{reachable}(s,q)]}{\bot}$$

The *BUnsat* rule has the following guard condition:

$$\text{whenever } \mathit{reachable}(s,q') \in \Gamma, \text{ we also have } \neg I(s,q') \in \Gamma.$$

In addition to the derivation rules, we again use the *theory lemma* feature of the DPLL(T) framework to periodically generate an *initial state lemma*:

$$\mathfrak{P} \wedge \Phi \implies \neg I(s,q)$$

As before, only valid theory lemmas are allowed, so this lemma can only be generated if we know that q is not an initial state.

Similarly to the forward reachability case, the $\mathcal{TS}$ solver's strategy dictates that the initial

state lemma generation rule always be applied with trigger (s,q) immediately after an invocation of *BDecide* or *BDecide2* with trigger *reachable*(s,q). As before, this guarantees that one of these rules is applied at most once for each state q.

The backward reachability derivation rules are very similar to the forward reachability ones; it is straightforward to show that they do not change the solver's soundness and completeness, and that the solver still terminates for transition systems with finite state sets and finite event sets.

Part III

The *RWB* Model: A Software Engineering Point-of-View

Chapter 9

Distributed *RWB* Programs

In BP, an execution of the program is comprised of a series of synchronization points between the threads, each of which results in an event being triggered. The choice of the triggered event is performed by a global event selection mechanism (ESM), which, at every synchronization point, receives input from all the threads before making the choice. This high amount of coordination grants behavioral programs many of their qualities: it eliminates race conditions between the threads, allows for multi-modal, modular and incremental development, and, in general, promotes the development of comprehensible and maintainable code [90].

However, extensive synchronization has implications on system performance (see [91]). Since all threads must synchronize before the system can continue to the next synchronization point, the step from one point to another is constrained by the slowest b-thread. In parallel architectures (e.g., multi-core processors), execution resources may stand idle while the system waits for a slow b-thread to finish performing nontrivial computations or time-consuming actions and reach the next synchronization point. Similar situations can also occur in programs that run on a single processor — for instance, if a b-thread is performing lengthy input/output actions that require no processing power, but delay its synchronization. Further, in distributed architectures, repeated synchronization may be expensive or not at all possible, hindering the applicability of BP to these settings.

In [79], a new execution mechanism for behavioral programs, termed *eager execution*, was introduced. Eager execution allows relaxing the synchronization constraints between b-threads, resulting in a higher level of concurrency when executing the program. At the same time, eager execution maintains all information necessary for triggering events, and thus adheres to BP's semantics and supports its idioms.

Eager execution is made possible by automatically *analyzing* a thread prior to its execution, resulting in an approximation of the thread's behavior. With this information at hand, the eager execution mechanism can sometimes choose events for triggering without waiting for all of the threads to synchronize, thus improving the efficiency of the system's run and avoiding excessive

synchronization. In [79] the authors presented two analysis methods that lead to more eager execution: one is *static* and considers the thread as a whole, whereas the other is *dynamic* and takes into account the thread's state during the run. Both methods have been implemented and tested in *BPC*, a framework for behavioral programming in C++.

Despite its advantages, the eager execution technique still requires that each b-thread communicate with the global ESM at every synchronization point. While this constraint is significantly weaker than stepwise synchronization with all other b-threads, it may limit the applicability of the approach for designing multi-component applications in distributed architectures, in which communication is costly and time-consuming. In this chapter we show how a variant of eager execution, combined with Dead Reckoning techniques [41, 71], can be utilized to reduce these costs. We refer to this variant as *distributed eager execution*.

In order to have behavioral modules executed in a decentralized manner on different machines, we propose to distribute the ESM, so that each machine runs its own *ESM agent*. These agents serve as the ESM for their *local* threads, i.e., threads running on the local machine, but have no direct access to threads on other machines. Instead, they can communicate with other agents.

Before running the system, each agent is given the state graphs of all the threads in the system, including non-local threads. Each ESM agent then executes the program locally, using these state graphs to simulate non-local threads and predict their synchronization requests. Each agent is responsible for answering its local threads' synchronization requests, just as a central ESM would. Observe that this requires that the event selection mechanism be a deterministic function — that is, a function from $2^E - \{\varnothing\}$ to E, whose input is the set of enabled events — in order to ensure that the autonomous agents pick the same events.

In a program with deterministic threads, this form of distribution would suffice to make inter-component communication obsolete, as each ESM agent could trigger precisely the same events as the others. In the case of systems with nondeterministic threads, some communication between the distributed components is mandatory. Intuitively, this communication is used to announce the outcome of nondeterministic choices made by a thread to the other components. Specifically, all ESM agents are aware of each thread's nondeterministic forks, as they hold all the state graphs. Whenever such a nondeterministic fork is reached, the ESM agent on which that thread is actually running is responsible for disseminating the outcome of the nondeterministic choice to the remaining agents. If other agents reach this point before the outcome has been broadcasted, they must wait for it. This guarantees that the execution is consistent across all program components.

The rest of the chapter is organized as follows. In Section 9.1 we briefly describe the eager execution mechanism as proposed in [79]. Then, in Section 9.2 we illustrate with an example how that mechanism can be extended to handle distributed settings as well. In Section 9.3

we rigorously formalize the distributed execution mechanism, and in Section 9.4 we briefly discuss how the mechanism can be enhanced even further. In Section 9.5 we compare the distributed execution mechanism to an existing extension to BP, called *b-nodes*, which also provides support for distributed frameworks. We conclude with Section 9.6.

9.1 Eager Synchronization

The following notion is at the core of the eager synchronization proposed in [79]. Let $P = \{BT^1, \ldots, BT^n\}$ be a behavioral program consisting of b-threads $BT^1, \ldots, BT^n$. Assume that at some point in the execution of P, a subset $P_{\text{sync}} \subseteq P$ of the threads has reached a synchronization point, while the rest are still executing. Further, assume that the ESM has additional information about the events that the threads in $P - P_{\text{sync}}$ will request and block at the next synchronization point. If, combining the information from threads in P_{sync} with the information about threads in $P - P_{\text{sync}}$, the ESM can find an event e that will be enabled at the next synchronization point, then e can immediately be chosen for triggering.

The ESM may then pass e to the threads in P_{sync} to let them continue their execution immediately, without waiting for the remaining threads to synchronize. Once any of these other threads reaches its synchronization point, the ESM immediately passes it event e, as this event was selected for that particular synchronization point. This is accomplished by having a designated queue for each of the b-threads, of events that are waiting to be passed, and putting e in the queues corresponding to the not-yet synchronized threads. The execution mechanism described is *eager*, in the sense that it uses predetermined information to choose the next event as early as possible.

When a thread BT reaches a synchronization point, if the corresponding queue is nonempty, the ESM dequeues the next pending event e'. If BT requests or waits for e', it is passed to the thread, which then continues to execute. Otherwise, e' is ignored, and the ESM continues with the next event pending in the queue. In order to reflect the semantics of BP, from the ESM's global perspective BT is not considered synchronized as long as it has events pending in the queue. Particularly, the events that are requested or blocked by BT at this point are not considered for the selection of the next event; the ESM considers only threads that have synchronized and for which there are no pending events (so that they are halted).

Observe that the eager execution mechanism strictly adheres to the semantics of BP, as described in Section 2.2; at every synchronization point, the triggered event is indeed enabled. Consequently, the following result holds [79]:

Proposition 16. Given a behavioral program P, the sequence of events triggered by the eager execution mechanism is a valid run (under BP's semantics).

The key point, however, is that the eager mechanism makes its decisions more quickly, and thus often produces more efficient runs. For a rigorous formalization of the eager execution mechanism, see [79].

It remains to explain how the execution mechanism knows which events could be requested and blocked by threads that are yet to synchronize. In [79] the authors propose two approaches, termed *static analysis* and *dynamic analysis*.

9.1.1 Static Analysis

In this approach, the ESM is given in advance a static over-approximation of the events that a thread might block when synchronizing. Explicitly, if a thread has states $q_1, \ldots, q_n$, this over-approximation is $\bigcup_{1 \leq i \leq n} B(q_i)$, where $B(q_i)$ is the set of events blocked in state q_i. The over-approximation is static in the sense that it does not change throughout the run.

When a thread synchronizes, the ESM checks if there are events that are enabled based on the data gathered so far — namely, events that are requested and not blocked by threads in P_{sync}, and that are never blocked by the other threads, based on their over-approximations. If such an event exists, it can be triggered immediately. Otherwise, the ESM waits for more threads to synchronize. This generally results in more events becoming enabled, since the actual set of events that are blocked by a thread is always a subset of the over-approximation, and since additional requested events are revealed. As soon as enough information is gathered to deduce that an event is enabled, it is immediately triggered and passed to all synchronized threads. For threads that are yet to synchronize, the event is stored in a designated queue, to be passed to them upon reaching their synchronization point.

Observe that we only discuss over-approximating blocked events but not the approximation of requested events. The reason is that the analogous version would entail using an under-approximation of requested events; and, since threads do not generally request an event in each of their states, these under-approximations are typically empty.

9.1.2 Dynamic Analysis

In this approach, the ESM is given complete *state graphs* of the threads, which are automatically calculated before the program is executed. The labeled vertices of a state graph correspond to the thread's synchronization points and requested/blocked events, while the labeled edges correspond to the program's events (that are not blocked at that state). The graph thus provides a complete description of the thread from the ESM's point of view — that is, a complete description of the events requested and blocked by the thread, but without any calculations or input/output actions performed by the thread when not synchronized.

During runtime, the ESM keeps track of the threads' positions in the graphs, allowing it to approximate the events they will request and block at the next synchronization point — even before they actually synchronize. This method is dynamic, in the sense that the approximations for a given thread can change during the run, as different states are visited. The fundamental difference between running a thread and simulating its run using its state graph is that in the latter, no additional computations are performed, and consequently transitions can be considered immediate.

When the b-threads are deterministic, simulating a thread through its state graph yields precise predictions of its requested and blocked events at each synchronization point. In the nondeterministic model, where threads may depend on coin tosses or inputs from the environment, it may be impossible for the ESM to determine a thread's exact state until it synchronizes; however, the ESM can approximate the thread's requested and blocked events by considering all the states to which the nondeterministic transitions might send the thread. If, due to a previous transition, the thread is known to be in one of states $q_1, \ldots, q_n$, then the blocked events may be over-approximated by $\bigcup_{1 \leq i \leq n} B(q_i)$ — similarly to what is done in static analysis. Analogously, the requested events may be under-approximated by $\bigcap_{1 \leq i \leq n} R(q_i)$.

The other details are as they were in the static analysis scheme. Once an event is triggered, it is immediately sent to all synchronized threads, and is placed in queues for threads that are yet to synchronize.

9.1.3 Spanning State Graphs

The techniques discussed in Sections 9.1.1 and 9.1.2 entailed providing the ESM with information regarding the underlying transition systems of the program's threads. Providing this information manually, especially in large programs, can prove tiresome and error prone. Hence, we have developed tools for automatically spanning the state graphs of b-threads written in high-level languages — for instance, using the BPC [3] framework.

The spanning is performed for each thread of the input program individually. Each thread is run in isolation, and its state graph is iteratively explored until all its states and transitions have been found. Starting at the initial state, we check the thread's behavior in response to the triggering of each event that is not blocked by the thread in that state. After the triggering of each event, the thread arrives at a new state (synchronization point) — and, with proper book keeping, it is simple to check if the state was previously visited or not. New states are then added to a queue to be explored themselves, in an iterative BFS-like manner.

Isolating threads is performed using the CxxTest [149] tool, which is able capture and redirect function calls within programs. The thread's calls to the synchronization method `bSync` are captured, and used to determine the thread's current state; similarly, calls to the `lastEvent`

method are captured and used to fool the thread into believing that a certain event was just triggered. The strength of this method is that the entire process takes place using the original, unmodified program code. Once the state graph has been spanned, it is automatically transformed into a C++ code module and integrated into the program, to be used by the ESM.

9.2 A Distributed Implementation using Eager Execution

In order to evaluate our technique we implemented the traveling vehicles example from [91, Section 7]. The example includes several vehicles, each operating as an autonomous component traveling on pre-given cyclic route along an (x,y) grid; in each given time unit during the run, each vehicle can travel north, east, south or west. We assume that all vehicles travel at identical speeds, i.e., cover one unit of distance per time unit.

In [79] the authors implement a non-distributed version of this program. The threads of each vehicle, v_i, involve a designated set of events. In particular, threads implementing vehicle v_i do not block events that belong to vehicle v_j's threads. Thus, the eager execution mechanism allows each vehicle to operate independently of others. A code snippet for the main thread of vehicle v_i is depicted in Figure 9.1. If event selection is fair, all vehicles are constantly moving — as the ESM does not wait for vehicle v_i to finish moving and synchronize again before triggering the movement requested by another vehicle.

```
1  while ( true ) {
2    set<Event> requested;
3
4    if ( destinationIsNorth() )
5      requested.insert( #iMoveNorth );
6
7    if ( destinationIsSouth() )
8      requested.insert( #iMoveSouth );
9
10   if ( destinationIsEast() )
11     requested.insert( #iMoveEast );
12
13   if ( destinationIsWest() )
14     requested.insert( #iMoveWest );
15
16   BSYNC( requested, {}, {} );
17   adjustPositionByLastEvent();
18 }
```

Figure 9.1: The main method of each vehicle thread. The placeholder '`#i`' is replaced by the number of the vehicle; for instance, for vehicle v_5, the events are 5MoveNorth, 5MoveWest, etc. The thread requests moves in all directions that bring it closer to the destination. When the call to `bSync` returns, one of these moves was selected by the behavioral execution mechanism. The thread then updates its position (by invoking `adjustPositionByLastEvent`), and proceeds.

Eager execution allows a light-weight solution if communication between the vehicles is required — e.g., for collision prevention. Each vehicle can be accompanied by an adviser

thread that keeps track of other vehicles. Whenever its vehicle is dangerously close to another, the adviser blocks movement in the dangerous direction (for simplicity, deadlocks are ignored). As the modular design remains strict, adding the adviser threads does not impede the vehicles' ability to move independently.

Suppose now that communication to and from the vehicles is costly, and is to be minimized. In particular, it is desirable to avoid a central ESM. This could be addressed using a distributed design, where each vehicle and agent pair runs on a dedicated machine. As the threads of this example are deterministic, each vehicle can completely predict the whereabouts of the other vehicles at any point in the execution, and collisions can be averted without any inter-vehicle communication.

We now introduce a source of nondeterminism. Suppose that one of the vehicles is an antique, and often requires maintenance. Along that vehicle's route there is a garage; and whenever the vehicle passes that point of the route, it may go in for repairs. The decision of whether or not to stop for repairs is considered a nondeterministic input from the environment. This new setting prevents other vehicles from predicting the location of the malfunctioning vehicle — since each lapse it may or may not spend one time unit in repairs.

Whenever the malfunctioning vehicle passes by the garage, the ESM agents reach a nondeterministic fork in that vehicle's state graph, and suspend their execution. As soon as the malfunctioning vehicle synchronizes and reveals whether or not the vehicle stopped for repairs, its handling ESM agent disseminates the information to the other agents, allowing them to resume their execution.

Even in the nondeterministic setting, using the distributed version of the system significantly reduces the number of messages being sent between the machines. In the central ESM scenario of [79], each vehicle would have to communicate with the ESM for every single move; but in the distributed setting, only one message per round is sent from the malfunctioning vehicle to the others.

9.3 Distributed Execution Formalized

In this section we provide a rigorous definition of the distributed execution model of BP, and prove that the runs that it produces abide by the semantics of BP.

Let $P = \{BT^1, \ldots, BT^n\}$ be a (possibly nondeterministic) behavioral program, where $n \in \mathbb{N}$ and each BT^i is a distinct b-thread, and let $f : 2^E - \{\emptyset\} \to E$ be a deterministic event selection function. Suppose that the threads run on different *machines* $M_1, \ldots, M_k$. Each machine is defined as the set of thread that it runs, i.e. $\bigcup_{i=1}^{k} M_i = P$.

Each machine M_i has an ESM agent, C_i; this agent acts as the ESM for the threads of M_i, and answers their synchronization requests. Each ESM agent is supplied with the state graphs

of all threads in the system, and uses these graphs to locate nondeterministic transitions of the threads throughout the run.

The pseudocode for ESM agent C_i in charge of managing threads M_i is given below. The agent uses variables $q_1, \ldots, q_n$ to keep track of the states of all threads in the system.

Algorithm 12 Coordinator Agent C_i

```
1:  ∀i, q_i ← q_0^i
2:  LastEvent ← φ
3:  while true do
4:     Sync ← φ
5:     while |Sync| < |M_i| do
6:        Receive synchronization request from thread BT^j
7:        Mark the new state of BT^j as q'_j
8:        if LastEvent ≠ φ and |δ^j(q_j, LastEvent)| > 1 then
9:           Broadcast q'_j
10:       q_j ← q'_j
11:       Sync ← Sync ∪ BT^j
12:    for BT^ℓ ∉ M_i do
13:       if LastEvent ≠ φ then
14:          if |δ^ℓ(q_ℓ, LastEvent)| > 1 then
15:             Update q_ℓ according to broadcasts from other agents
16:          else
17:             q_ℓ ← δ^ℓ(q_ℓ, LastEvent)
18:    Enabled ← ∪_{j=1}^n (R^j(q_j)) − ∪_{j=1}^n (B^j(q_j))
19:    LastEvent ← f(Enabled)
20:    Inform threads in M_i that LastEvent was triggered
```

Note the slight abuse of notation of line 17 — where $\delta^\ell(q_\ell, \text{LastEvent})$ is not the state of the thread, but rather a set containing that state. Also, we implicitly assume that all broadcasts between the ESM agents contain the index of the synchronization point that they refer to, to prevent cases where information about synchronization point t_1 could be mistakenly used in synchronization point t_2.

Intuitively, the ESM agent waits for the threads that it manages (loop on line 5), same as in the centralized case. Whenever a thread synchronizes, the agent checks if the thread's last transition was nondeterministic (line 8). If so, the new state is broadcasted to the other agents — as they have no other way of finding out which transition was taken.

Once all the agent's threads have synchronized, it turns to consider threads that run on other machines. The key fact is that if a non-local thread is at a nondeterministic transition (line 14), the agent has to wait to receive a broadcast message (line 15) in order to determine the new state of that thread. Otherwise, it can go ahead and determine the thread's state locally (line 17).

After the synchronization requests of all threads have been determined, the next event to be triggered is selected (line 19), and then broadcasted to the agent's threads. This part is the

reason for stipulating that f be a deterministic function — in order to maintain consistency, all agents much trigger the same event on line 19.

Observe that each ESM agent uses information regarding the transition functions (lines 8 and 14) and synchronization requests (line 18) of all the threads in the system — both threads that run locally on that agent, and threads that run on other agents. This information is given prior to the run, in the form of the state graphs of all the threads in the system.

Having formally defined the operation of each agent, we can now prove the following proposition:

Proposition 17. Let $P = \{BT^1,\ldots,BT^n\}$ be a behavioral program, divided into machines $M_1,\ldots,M_k$ with ESM agents $C_1,\ldots,C_k$. Let $f : 2^E - \{\varnothing\} \to E$ be a deterministic event selection function. Then agents $C_1,\ldots,C_k$ produce a *consistent* run; that is, there exists a unique run $e_1e_2\ldots$ such that at synchronization point i, *every* ESM agent C_ℓ triggers e_i. Further, the sequence $e_1e_2\ldots$ is a valid run (under BP's semantics).

For simplicity, we prove the lemma for the case of two machines, i.e. $n = 2$; the proof can easily be extended to any $n \in \mathbb{N}$. The proof follows directly from the next proposition, which is in turn proven by induction over the index of the synchronization points of the run.

Proposition 18. For $i \in \mathbb{N}$ and $m \in \{1,2\}$, let $q^i_m(BT^\ell)$ denote the state of thread BT^ℓ at synchronization point i, from the point of view of ESM agent m. Let Q^i_m denote the system-wide state at synchronization point i from the point of view of ESM agent m; that is, $Q^i_m = \langle q^i_m(BT^1),\ldots,q^i_m(BT^n)\rangle$. Then for all $i \in \mathbb{N}$, it holds that $Q^i_1 = Q^i_2$.

Proof. Let $i = 1$, which is the first synchronization point in the program. At this point, by the initialization in line 1 in the ESM agent's code, $q^1_1(BT^\ell) = q^\ell_0$ and $q^1_2(BT^\ell) = q^\ell_0$ for all ℓ. Consequently, $Q^1_1 = Q^1_2$.

Now, suppose that $Q^i_1 = Q^i_2$ for some i. At synchronization point i, both ESM agents triggered the same event e_i. This is so because the event selection function is deterministic, and thus both agents triggered event $e_i = f(Enabled(Q^i_1)) = f(Enabled(Q^i_2))$. This event was passed to all threads of the system by their respective coordinator agents.

Observe synchronization point $i+1$ from the point of view of C_1. As soon as all threads in M_1 have synchronized, C_1 knows their states. In order to determine the states of the remaining threads (those running on machine M_2), C_1 uses their pre-supplied state graphs. For any $BT^\ell \in M_2$, agent C_1 checks whether $|\delta^\ell(q^i_1(BT^\ell))| = 1$, and if so it deduces that $q^{i+1}_1(BT^\ell) = \delta^\ell(q^i_1(BT^\ell))$. In this case, C_2 will learn the state of BT^ℓ when that thread synchronizes, and it will hold that $q^{i+1}_1(BT^\ell) = q^{i+1}_2(BT^\ell)$.

The other option is that thread BT^ℓ is performing a nondeterministic transition, i.e. $|\delta^\ell(q^i_1(BT^\ell))| > 1$. In this case, C_1 has to wait for thread BT^ℓ to synchronize and reveal its

state to C_2, after which C_2 will broadcast this state to C_1. In this case, it will also hold that $q_1^{i+1}(BT^\ell) = q_2^{i+1}(BT^\ell)$.

Further, upon receiving the synchronization request from a local thread BT^t, agent C_1 uses its stored state graphs to check whether $|\delta^t(q_1^i(BT^t))| > 1$. If so, C_1 transmits the thread's new state as learned from the synchronization request, $q_1^{i+1}(BT^t)$, to C_2 — to inform C_2 of how that nondeterministic transition was resolved.

As agent C_2 behaves symmetrically, we conclude that for all t it holds that $q_1^{i+1}(BT^t) = q_2^{i+1}(BT^t)$, and consequently that $Q_1^{i+1} = Q_2^{i+1}$. □

Proposition 17 immediately follows from Proposition 18, and from the fact that f is a deterministic function. Indeed, $Q_1^{i+1} = Q_2^{i+1}$ implies identical calculation of the set E (line 18 in both agents, and thus the same output for $f(Enabled)$. Finally, the fact that the resulting run is a legal BP follows from the definition of the set *Enabled* to be the set of enabled events at the synchronization point.

9.4 Further Relaxing the Distributed Execution Mechanism

The distributed execution mechanism described above utilizes eager execution in the sense that each machine may be able to continue its execution without waiting for slower machines — except in nondeterministic transitions. We point out that further relaxation can be achieved by applying static or dynamic analysis to threads within the scope of each coordinator agent. As in the non-distributed case, this would allow faster threads within the same machine to continue their execution without waiting for their slower counterparts.

Another possible enhancement for the distributed model above is to use approximations for nondeterministic threads on other machines that slow down execution. Suppose that controller agent C_1 of machine M_1 is waiting for thread $BT \in M_2$ to finish its nondeterministic transition in order to trigger an event. As was the case in the centralized version, if C_1 can deduce, using the state graph of BT, that its next state will be either q_1 or q_2, it can approximate its requested and blocked events with $\mathcal{R} = R(q_1) \cap R(q_2)$ and $\mathcal{B} = B(q_1) \cup B(q_2)$. This further reduces the dependency between the different machines, hopefully achieving better optimization.

9.5 Eager Execution and B-Nodes

In this chapter we presented an approach for coping with BP's synchronization requirements in the face of a distributed architecture, by using the eager execution mechanism. In this section we compare and contrast this approach with an extension to BP, called *b-nodes* [91], which advocates a different strategy for solving the same difficulties.

The b-node approach to program design is a two-layer approach, which, in essence, goes beyond a single behavioral program. Each b-node constitutes a distinct behavioral program, with its own vocabulary of internal events, within which b-threads are synchronized and BP's idioms can be used. Then, as an additional layer, external events are sent asynchronously between the b-nodes to signal the occurrence of certain internal events. This solution therefore requires an additional vocabulary of external events, to be used across b-nodes. Each external event typically corresponds to certain events that are internal to the b-nodes. Moreover, each b-node has auxiliary threads for handling asynchronous communication, as well as the translation of internal events to external inter-b-node events and vice versa.

In contrast, the eager execution mechanism allows alleviating the synchronization requirements between the b-threads of a single behavioral program with no external means. For those sets of threads within the program, called modules, that are sufficiently independent — i.e., each module handles a completely separate facet of the system — eager execution results in an execution in which distinct modules are executed independently of one another (as far as each module's "internal" events are concerned; of course, a dependency arises when one module waits for another module's events). In this solution, there is a single vocabulary of events that is used across the program, and communication between the modules is facilitated by BP's native idioms.

It turns out that the two approaches are closely related. In fact, it is straightforward to show that a program designed according to the b-node approach induces an equivalent behavioral program with a modular design [2]. Similarly, it is also possible to transform a modular design into a b-node based design: each module is transformed into a b-node, and modules that would wait for events that belong to other modules are adjusted to wait for matching external events instead. However, when performing these transformations, one must take into account that, unlike the eager execution mechanism, the external message passing mechanism is not guaranteed to preserve event order; for instance, if module M_1 signals module M_2 twice, by requesting events e_1 and e_2 in that order, the eager execution approach guarantees that event e_1 is received before e_2. This guarantee does not necessarily hold in the b-node case.

Despite these similarities, each approach offers its distinct benefits. The eager execution approach uses automated tools, and is thus simpler to use, whereas the more complicated implementation details of simultaneously executing multiple modules are hidden within the execution mechanism itself, and the user does not need to implement external mechanisms and auxiliary events. Thus, the automated approach allows the specification and direct coding of richer scenarios, without having to break the scenario at the b-node boundary.

Moreover, even when the design is not strictly modular, eager execution generally relaxes some of the synchronization that arises between threads — especially when the dynamic approach of Section 9.1.2 is used. These relaxations may yield performance improvements.

The major drawback of eager execution with respect to the b-node approach is that every thread must generally communicate with a *global* execution mechanism (or an agent thereof) for each triggered event. While this is still significantly better than synchronizing all b-threads at each step of the execution, it may limit the applicability of the approach in such cases where communication is costly or unreliable. In contrast, the b-node approach allows programmers to fine-tune their programs in the face of such constraints: the execution mechanisms are local to each b-node, and, at points where inter-node communication is needed, messages can be routed directly to the desired recipient. This produces programs that are able to run in a highly distributed fashion.

In order to enjoy the benefits of both worlds, one may combine the two approaches within a single system. One way to do this is to apply the b-node approach but to implement an eager execution mechanism within each b-node. This may yield performance improvements over the basic b-node approach. Another way entails applying the eager execution approach, and enhancing it to support distributed execution with selective message exchanges between disparate modules, as in the b-node approach. Yet another approach to the combination, aimed at automating the b-node approach, is to specify b-threads as if they are in a single b-node and then use automated tools to determine b-node boundaries (along the lines of [7]). Automated tools could then add the necessary processes for creating physically distributed b-nodes.

9.6 Conclusion and Future Work

The contribution of this chapter is in the proposed usage of the eager execution mechanism in order to create distributed behavioral programs. This approach is made possible by the realization that, by analyzing a b-thread prior to its execution, it is sometimes possible to accurately predict a valid outcome of a synchronization point without actually waiting for the thread to synchronize.

In this chapter we made no assumptions on how the coordinator chooses the next event to be triggered from among the enabled events. In practice, however, such assumptions can sometimes simplify system development. One example is the *prioritized* event selection used in [89]. We believe that our methods can be naturally adapted to such mechanisms too.

Future work can progress in several directions: studying the impact of the distributed BP mechanism in large realistic case studies, exploring ways to automatically partition large, fully behavioral systems into distributed components, developing formal methods and tools to verify (e.g., model check) distributed behavioral programs in a compositional manner, and exploring the addition of behavioral synchronization and event blocking to more mainstream actor- and agent-based platforms.

Chapter 10

Scaling-Up *RWB*: From Basic Principles to Application Architectures

10.1 Introduction

The focus of this thesis is behavioral programming and its underlying computational model. At the core of BP is the notion of programming through the specification of *scenarios*, each of which corresponds to a certain aspect of the system's behavior, not necessarily restricted to a particular component. When composed together according to a certain set of rules, the scenarios yield cohesive system behavior.

One of the main difficulties in this type of design is to ensure that the large variety of scenarios, each with their own unique characteristics and viewpoints, inter-operate correctly [98]. In particular, race conditions and unpredicted interweaving of scenarios can result in incorrect system behavior. Thus, it is desirable to define inter-scenario interfaces that are sufficiently simple to minimize these effects, but which are still powerful and expressive enough to be of practical use.

The specific structure of behavioral programs, and in particular the simple yet strict thread interaction rules, has beneficial effects: programs can be written in a "natural" way, i.e., with modules that are aligned with the specification [90, 74]; it greatly reduces the amount of (unexpected) interleaving, hopefully making race conditions more scarce; it renders the program more amenable to incremental development; and it facilitates the application of program analysis methods to behavioral systems (as we have extensively discussed in Part II). Consequently, it has been conjectured that behavioral programming is suitable for the development of large systems. However, to the best of our knowledge, this hypothesis has not yet been put to the test. Here we set out to do just that, by attempting to implement a real-world system in BP. Apart from checking the feasibility of the task, we were interested in assessing whether the aforementioned traits of BP, such as incremental development and the alignment of code with

the specification, could indeed scale-up in a large system.

For our case-study we chose to implement protocol stacks of two common and elaborate protocols, TCP and HTTP. TCP (*Transmission Control Protocol*) is a connection-oriented protocol, used mainly for the transfer of data across the internet. The principal feature of TCP is its reliability: it guarantees that data arrives without errors and in the order in which it was sent. HTTP (*Hypertext Transfer Protocol*) is a request-response protocol, aimed at allowing clients to retrieve information from remote hosts. We have implemented and combined these two protocol stacks, creating a working web-server.

Our motivation in choosing this particular project was its volume, and also our desire to test BP's effectiveness in handling the large variety of coding situations the project entails: handling timeouts, string manipulation, file access, checksum calculations, handling multiple inputs, mandatory and forbidden user behaviors, etc.

As we began constructing the system we noticed that the existing, "traditional" idioms of BP were inadequate for dealing conveniently with certain programming tasks. Some of these tasks could be performed by BP but required employing ad-hoc solutions that bypassed the built-in infrastructure, whereas others (e.g., those related to time) were downright inexpressible using traditional BP, and required incorporating external mechanisms into the program [91] — in both cases, defeating the purpose of BP's simple and intuitive interfaces.

As the development of our case-study progressed, we were able to classify the difficulties we faced, placing them in four categories. For each of these, we were able to come up with an extension idiom to BP that allowed convenient programming solutions. In defining these new idioms, we attempted to retain as much of the simplicity and intuitiveness of the original framework as possible.

The first and foremost difficulty we encountered was the issue of time: traditional BP assumes all transitions in the systems take "zero time" (as per the *synchrony hypothesis* [31]), and does not provide a mechanism for bounding the flow of time between events. The TCP protocol makes abundant use of timers and timeouts, and it was unclear to us how to implement it in BP. To overcome this difficulty we extended BP with the notion of *timeouts* which, as demonstrated later in this chapter, allowed us to express the required time constraints for our program.

The second problem we faced was the need for program-specific strategies in BP. The BP infrastructure dictates that at every synchronization point of the execution the program may have to choose between several legal execution paths, but it does not specify which should be chosen. Various schemes have been suggested in the past, which allow the user to partially prioritize between these paths, each scheme with its unique advantages and disadvantages. In the later stages of the development of our case-study, when the system contained numerous modules running in parallel, we observed that several requirements — especially those pertaining to prioritization between the modules — could be naturally enforced through a more intelli-

gent path selection method. Unfortunately, the existing mechanisms were too restrictive. We consequently extended the framework to allow programmers to supply their own path selection strategies, per program, in a way that provided sufficient flexibility for our needs.

The third type of difficulty we encountered was that of dynamic thread creation. We wanted our web-server to be able to handle different volumes of activity: that is, to allocate more resources when traffic was high and to release them when it was low. A natural approach was to employ dynamic creation and destruction of threads, which is not allowed in the traditional BP semantics. In [89, 86] a limited variant of this concept was proposed, in order to regulate external input to a behavioral program. We opted to generalize and formalize this idea, allowing it to be used anywhere within the program, and we then used it in our implementation to dynamically allocate more threads according to traffic.

Finally, the last issue we encountered was that of *parameterized inputs and events*. Previous work on scenrio-based programming and BP revolved around programs with a small bounded pool of inputs, whereas the input domain of a web-server is practically infinite. In order to implement our example, we thus generalized traditional BP to support parameterized inputs.

Having added these idioms to the BP framework, we were able to accomplish our implementation goals. We implemented our case-study in a new framework for behavioral programming in C++, termed *BPC*, which we present here, and which supports the extensions described above. The framework and code for the case-study are available online [10].

Returning to our original question of whether or not BP is suitable for the development of large systems, we believe that our case-study answers this affirmatively — provided that the above idioms, or similar ones, are made available. As for our secondary goal, namely to check whether BP's naturalness and incrementally would scale-up in a large system: while these properties are inherently difficult to quantify, the conclusions from our case-study seem to support an affirmative answer here, as well. We discuss this matter in later sections.

The remainder of the chapter is organized as follows. We begin by briefly describing the traditional means of handling input in BP in Section 10.2. In the succeeding sections, we discuss the difficulties we encountered in our case-study, one by one, and the new idioms used to overcome them. Each of the sections includes a small illustrative example and a discussion of where the difficulty occurred in the web-server application: time constraints are discussed in Section 10.3, customizable strategies in Section 10.4, dynamic thread creation in Section 10.5 and parametrized events in Section 10.6. The rigorous semantics of our extended variant of BP is available in Section 10.7.

Sections 10.8 and 10.9 are dedicated to the BPC framework and the details of the implementation of our case-study, respectively. Related work appears in Section 10.10, and we conclude in Section 10.11.

10.2 Handling Input in BP

In Section 2.2 we recapped the semantics for BP as given in, e.g., [90]; for the remainder of this chapter we refer to this semantics as "traditional BP". Recall that the synchronous nature of traditional BP dictates that the system may only progress when all threads have synchronized. Consequently, a thread that is performing some blocking *read* operation would render the rest of the system unable to process any events. This makes it difficult to design *sensor threads* — threads that wait for user input and then request events to notify other threads of this input. A naive way to design such a thread appears in Figure 10.1.

```
1 while( true ) {
2     waitForButtonClick();                    // Returns only on click
3     BSYNC( { ButtonClicked }, {}, {} );
4 }
```

Figure 10.1: Pseudocode for a naive implementation of a sensor thread. The thread runs in an infinite loop, in each iteration waiting for input via the blocking call `waitForButtonClick` and then requesting a ButtonClicked event. The `bSync` call is the synchronization API method: its first parameter (marked in blue) is the set of requested events, the second (green) is the set of waited-for events, and the third (red) is the set of blocked events. `bSync` only returns when an event that was requested or waited-for has been triggered. In particular, this enforces the convention that a thread implicitly waits for every event that it requests — though it may also wait for additional events, that it did not request. Here, the only requested event is ButtonClicked, and the other two event sets are empty. When waiting for a button click, this thread prevents the entire system from triggering any events.

In traditional BP, this difficulty can be mitigated by using the *eager execution* mechanism, which we discussed in Chapter 9. The idea is that if a thread that has not yet synchronized is known to never block an event, and that event is enabled with respect to the other threads, then it may be immediately triggered without waiting for the delayed thread. Events triggered this way are then stored in a dedicated queue, and when the delayed thread finally catches up it processes them. As a particular case, sensor threads can always be declared to block no events, allowing the rest of the system to operate normally (see Chapter 9 for more details).

We stress that the eager execution mechanism does not alter the semantics of traditional BP; rather, it only allows to more efficiently execute behavioral programs. Thus, we consider it a part as traditional BP for the remainder of this chapter, and use it in order to create sensor threads.

10.3 Support for Time Constraints

The traditional BP semantics suffices when the system in question is time oblivious — i.e., when its response to external inputs takes negligible time and there are no constraints on the instance in time in which events may be triggered. But what happens when that is not the case? Consider, for instance, a behavioral program for a railway crossing. Suppose that a sensor thread signals the approach of a train by generating a LowerGate event. Also, say the gate is to remain down for 30 seconds, after which another thread is to request a RaiseGate event. The simplest approach is to encode this thread as depicted in Figure 10.2.

```
1  while( true ) {
2      BSYNC( {}, { LowerGate }, {} );
3      sleep( 30 );
4      BSYNC( { RaiseGate }, {}, {} );
5  }
```

Figure 10.2: A thread that waits for LowerGate, sleeps for 30 seconds, and then requests a RaiseGate event.

Unfortunately, this thread pauses the execution of the entire system during its sleeping periods.

One way to tackle this difficulty is to delegate timing responsibilities to a non-behavioral component, and have the system communicate with it using *external events* [91]. However, this solution requires that the programmer goes beyond the scope of BP.

Another approach is to use the eager execution mechanism, as was the case with sensor threads. However, eager execution has limited applicability: consider, for instance, the stronger variant in which the programmer wishes to *block* the RaiseGate event for the 30 seconds following a LowerGate event, to negate any accidental requests made by other threads. This stronger requirement is difficult to accommodate using the eager synchronization mechanism, as there is no way to readily inform the blocking thread that 30 seconds have passed.

Programming tasks in which time plays a role appeared frequently in our TCP stack implementation. The TCP protocol guarantees that data arrives without errors and in the order in which it was sent. To accomplish this, the end parties acknowledge the reception of each TCP segment using a scheme of agreed-upon sequence numbers; segments that are lost or arrive corrupt are not acknowledged, and are then retransmitted. Thus, a TCP stack needs to keep track of the time passed since sending each outgoing TCP segment, and retransmit it unless an acknowledgment is received within a certain time window.

To support these requirements, we extended BP with a *timeout* idiom. This idiom allows each thread to declare, at every synchronization point, a timeout value — in addition to the requested, waited-for and blocked events. The timeout value indicates the maximal number of

seconds the thread is willing to wait, in synchronized state, for a requested or waited-for event to be triggered, after which it "withdraws" its synchronization and associated event declarations. In practice, this means that the synchronization call returns and the thread resumes. The programmer can check if the call returned due to a triggered event or because of a timeout.

Using the timeout parameter, the thread depicted in Figure 10.3 guarantees that the railway crossing system does not generate an early RaiseGate event.

```
1  while( true ) {
2      BSYNC( {}, { LowerGate }, {}, ∞ );
3      BSYNC( {}, {}, { RaiseGate }, 30 );
4  }
```

Figure 10.3: An implementation using the timeout parameter. Event RaiseGate is blocked for 30 seconds.

The thread waits for a LowerGate event, and then spends the successive 30 seconds blocking RaiseGate events. Counting these 30 seconds begins the instant the thread synchronizes, and does not depend on other threads. Since the thread neither requests nor waits for any events, the blocking is guaranteed to continue for the full 30 seconds, after which the call returns, and the thread waits for additional LowerGate events. The special ∞ symbol passed as the timeout parameter implies that the thread is willing to wait indefinitely; in fact, using this value produces the same result as the traditional synchronization interface.

The timeout idiom can also be used to implement a TCP retransmission scheme, as shown in Figure 10.4. The thread transmits the packet, and then waits for an acknowledgment for 2 seconds. If the acknowledgment is not received, the second synchronization call returns due to a timeout, and the process repeats. We describe our implemented retransmission mechanism in greater detail in Section 10.9.

```
1  do {
2      BSYNC( { SendSegment }, {}, {}, ∞ );
3      BSYNC( {}, { Acknowledgment }, {}, 2 );
4  } while( timeoutInLastSync() )
```

Figure 10.4: A retransmission scheme. The thread waits for an Acknowledgment for 2 seconds, and if it fails to arrive — retransmits the segment.

The proposed idiom appears to be natural and intuitive, and thus compatible with the rest of BP's idioms. Indeed, one of our goals was to stick to simple and intuitive idioms whenever possible. The timing idiom is also quite expressive, allowing a variety of behaviors that were previously beyond the direct scope of BP, such as "block for x seconds" (as in the railway example), "request for x seconds and then default", or just "sleep for x seconds" without delaying the system.

In our view, a call to bSync that returns due to a timeout is not considered an error — rather, it is just another possible outcome of the synchronization attempt. Thus, timeouts are not regarded as exceptions, but as a mechanism for coping with thread transitions that are not immediate: if a thread is delayed in reaching its synchronization point, other threads may react (when their timeouts expire) and change their event declarations.

In [86], the authors present a model checker for behavioral programs, capable of handling safety and liveness properties. An interesting aspect of this model checker is that it receives the property in question in the form of a b-thread: a thread that waits for unwanted event sequences and marks states as bad (safety), or a thread that waits for good event sequences and marks states as good (liveness). We observe that the addition of timeouts allows us to express, e.g., time-related safety properties. For instance, consider a system in which every e_1 event is always followed by an e_2 event, and suppose that we wish to verify that at most 2 seconds pass between every e_1 and e_2 pair. This property can be expressed by the b-thread depicted in Figure 10.5.

```
1 while( true ) {
2     BSYNC( {}, { e1 }, {}, ∞ );
3     BSYNC( {}, { e2 }, {}, 2 );
4     if( timeoutInLastSync() )
5         BSYNC( { Error }, {}, {}, ∞ );
6 }
```

Figure 10.5: A thread that requests an Error event if event e_2 is not triggered within 2 seconds of event e_1.

This sort of constraint can be useful for model checking (by extending the tool of [86] to support timeouts); and it can also give rise to a lookahead mechanism that influences the choice of triggered events so as to satisfy the constraint, similarly to the smart play-out mechanism of [84, 85]. Both directions are left for future work.

Technically, thread timeouts are triggered by the event selection mechanism, similarly to the way regular events are handled. Further details appear in Sections 10.8.2 and 10.7.

10.4 Customizable Event Selection

BP's traditional semantics dictates that in every synchronization cycle one event that is requested and not blocked is triggered. However, if there is more than one viable event for triggering, it is unspecified which of those will be selected.

In practice, however, it is often useful to have events selected using a certain strategy. For instance, consider the following system that manages a smart door lock. When a person approaches, he/she must place the appropriate identification card on a reader, and a behavioral program decides whether or not to let the person through. A simple design for the program is

to have a dedicated thread handle the lock; and whenever an id card is scanned, have it request an Open event, which is translated by an actuator thread to opening the actual lock.

Next, suppose some people should be denied passage — e.g., if they do not appear in the white list, if they appear in the black list, or if their access card has expired. We assume that the Open event has an *id* parameter, that identifies the person, and that other parts of the system may block Open events for certain *id*s. However, as the lock thread does not know in advance which events are blocked, it cannot just request the Open event, or the system could get stuck if that event is blocked. One possible solution appears in Figure 10.6: the thread requests, along with an Open event, also an Idle event. If the open event is blocked, the idle event is processed, and the thread can continue processing future requests.

```
1  while( true ) {                          // Door thread
2      BSYNC( {}, { Request }, {}, ∞ );
3      bSync( { Open( lastEvent().id ), Idle }, {}, {}, ∞ );
4  }
5
6  while( true ) {                          // Blocker Thread
7      bSync( {}, {}, { Open | Open.id is black listed }, ∞ );
8  }
```

Figure 10.6: A sketch of the lock program. The door thread waits for requests, and tries to grant them. Other threads may block the Open(id) event for certain values of the id parameter. As the door thread has no way of knowing whether a specific event is blocked or not, it also requests an Idle event, to allow itself to ignore the request and move on to process future requests.

In order for this scheme to work properly and not deny entrance to authorized ids, we need to be certain that any Open event will take precedence over the Idle event. Thus, we need a way to enact a certain strategy for how enabled events are to be selected for triggering.

On several occasions we encountered similar, though more complex, situations in our case-study. In one case, during tests with multiple simultaneous connections, we observed that some clients would get starved. Hence, we wanted to enforce the requirement that higher priority be given to requests from starving connections. In another case, we wanted to ensure that segment-sending requests always received a higher priority than connection-termination requests, so that segments were never sent on closed connections.

Previous work discussed several event selection strategies, such as thread-based priority, round robin, random and arbitrary selection [89]. The BPJ framework, for instance, uses the thread-based priority scheme. Through our case-study and the specific requirements that it entailed regarding event selection, we came to recognize that no one strategy fitted all programs, and that it was useful to allow programmers to supply their own program-specific strategy.

Consequently, we extend BP's event selection in the following way. Let Γ denote the set of possible system configurations. In its most general form, an event selection strategy is a

function $f_{es} : \Gamma^* \times \Gamma \rightarrow E$, which takes as input the history of previous system configurations and the current configuration, and chooses an event for triggering from among the enabled events. The selection function is supplied by the programmer and is considered part of the behavioral program rather than of the BP framework. Apart from subsuming the above mentioned mechanisms, this approach also allows event selection strategies that change over time, such as learning [68] or look-ahead algorithms.

Technically, the BPC framework allows the programmer to provide a callback object to manage event selection. In particular, modifying the selection strategy does not entail recompiling the framework. For more details on the specific strategy used in our case-study and its implementation, see Section 10.9.

10.5 Dynamic Thread Creation

A reactive application may, throughout the course of its run, have to deal with varying volumes of activity. It is desirable to have applications that adjust — that is, dedicate more computational resources — when activity is high, and free them when they are no longer needed. This goal may be difficult to achieve with traditional BP.

Consider, for instance, a mail client application, which takes as input an email address and the body of a message and then sends it. Further suppose that the application waits for an acknowledgment message for every email sent. It must thus keep track of previous mails that have yet to be acknowledged.

One possible design for such an application is to direct all requests and acknowledgments to a single thread, which can then keep track of traffic, using an internal database. A cleaner solution, however, is to have multiple threads, each in charge of sending a single message and tracking its acknowledgment. The resulting threads are simpler and do not require a database, and are thus less prone to error.

The question then arises of how many of these threads we should instantiate. Statically determining this number would raise the risk of not having enough resources when many requests are performed simultaneously, and the risk of wasting computational resources when traffic is low.

To resolve this issue, we propose to allow threads to dynamically spawn other threads, and similarly, to allow threads to terminate during execution. That way, new threads can be instantiated when needed, and can be terminated when they are no longer needed, freeing system resources. An implementation of the above program using dynamic thread spawning appears in Figure 10.7.

We faced similar situations in our case-study. In particular, the TCP message acknowledgment scheme is very similar to the above example, and indeed we implemented it by spawning

```
1  while( true ) {                    // Dispatcher thread
2      BSYNC( {}, { Mail }, {}, ∞ );
3      new Sender( lastEvent().address, lastEvent().text );
4  }
5
6  do {                                // Sender thread
7      BSYNC( { Send( address, text ) }, {}, {}, ∞ );
8      BSYNC( {}, { Ack( address ) }, {}, 10 );
9  } while( timeoutInLastSync() );
```

Figure 10.7: The *Dispatcher* thread waits for incoming mail requests. For every such request, it dynamically creates a new *Sender* thread and passes to it as parameters the address and text fields of the request. Each *Sender* instance deals with just one mail, which was passed to it during construction. Immediately upon its instantiation, the thread sends the mail and awaits an acknowledgment. If no such acknowledgment is received within 10 seconds, the mail is resent. As soon as the acknowledgment has been received, the thread terminates.

dedicated threads that wait for message acknowledgment. Further, each active connection in the protocol stack requires some bookkeeping (e.g., the state of the connection, last received incoming sequence number, and last used outgoing sequence number), and we explored implementation variants, in which these bookkeeping threads were spawned per connection, in order to improve efficiency. More details appear in Section 10.9.

The reader may notice that, assuming that the address and text parameters in Figure 10.7 are unbounded, there are infinitely many versions of thread *Sender* that may be created throughout the run. Indeed, the traditional definition of behavioral programs as a set of threads that exist through the program's run no longer applies in the face of dynamic thread creation. Instead, we associate a behavioral program with a set of *thread templates* — copies of which may be instantiated at different times throughout the run. See Section 10.7 for a rigorous definition.

We point out that the concept of dynamic thread creation was already introduced in [89], where the authors used dynamically created *sensor threads*. These threads could only be spawned by non-behavioral components, and were only used to signal user input. In contrast, we allow the dynamic creation of general threads by other threads throughout the program (and use the eager synchronization mechanism to manage system input).

Dynamic thread creation is a useful feature, but it also has its tolls: in our BPC implementation, each behavioral thread is presently implemented as a POSIX thread, and so the creation of a large number of thread incurs a overhead. We plan to mitigate this problem by implementing of a more efficient, lightweight threading mechanism, similar to the one used, e.g., in Erlang [22].

10.6 Parameterized Input

In this section we examine another issue that may arise in applying the BP principles to a large system: the need to handle a pool of (practically) infinitely many possible inputs. This requirement is quite common; in particular, it arose in our case-study, where the system had to handle incoming TCP segments, i.e., byte sequences of unknown length.

For illustration, consider a simple program that takes as input a number x and checks whether it is a multiple of 3 and also ends with the digit 5. One approach to writing such a program would be to have a sensor thread wait for inputs, and then broadcast them to the rest of the system. One checker thread would then check whether x is divisible by 3, and another would check whether x ends with 5. Using event blocking, the two checker threads could then reach a combined decision on the final answer.

The input parameter x is unbounded, and as the BP framework dictates that threads only exchange information through the synchronization mechanism, this implies that the event set E of the program must be infinite. A convenient way to facilitate handling infinite event sets is to extend the definitions of traditional BP, and allow events with unbounded parameters. This is a generalization of an approach that appeared in examples described in [86]; there, the authors used events with bounded parameters to facilitate waiting-for or blocking finite sets of events. Pseudocode for the program described above, using parameterized events, appears in Figure 10.8.

Using this idiom in our case-study, we employed a sensor thread to read incoming segments, and for each segment to request a TcpSegmentReceived event — with the segment as its parameter. The segment could then be processed by other threads. Other parts of the system also made use of parameterized events; for instance, events associated with incoming HTTP requests carried ip and port parameters, indicating the address to which a response to the request needed to be sent. For additional details, see Section 10.9.

We observe that in many cases, dealing with unbounded parameterized events calls for threads with infinitely many states — states that may depend on the parameters. For instance, consider the sensor thread in Figure 10.8. A "state" of the thread is a synchronization point with fixed requested, waited-for and blocked events. For any input x, the thread requests different events; and hence, it must have as many states as there are x's. Therefore, this extension calls for allowing infinite state sets in the rigorous semantics of BP; see Section 10.7.

Finally, once we allow unbounded parameterized events we should consider that threads may wish to wait-for or block infinitely many events (as in the example of Figure 10.8). More complex cases include waiting-for or blocking events depending on their parameters. In BPC, we follow the example of the BPJ framework [89] for BP in Java, and allow parameter-dependent blocking or waiting-for events via *event predicates* — functions that take events and

```
1  while( true ) {                          // Sensor thread
2      int x = readInput();
3      BSYNC( { Check( x ) }, {}, {}, ∞ );
4  }
5
6  while( true ) {                          // First checker thread
7      BSYNC( {}, { Check }, {}, ∞ );
8      if( ( lastEvent().x % 3 ) == 0 )
9        BSYNC( { Good }, { Bad }, { Check }, ∞ );
10     else
11       BSYNC( { Bad }, {}, { Check, Good }, ∞ );
12 }
13
14 while( true ) {                          // Second checker thread
15     BSYNC( {}, { Check }, {}, ∞ );
16     if( ( lastEvent().x % 10 ) == 5 )
17       BSYNC( { Good }, { Bad }, { Check }, ∞ );
18     else
19       BSYNC( { Bad }, {}, { Check, Good }, ∞ );
20 }
```

Figure 10.8: A program that takes as input a number x, and decides whether it is a multiple of 3 that ends with 5. The *Sensor* thread waits for exterior inputs, and translates them into a parameterized event Check. This event is waited for by the two checker threads, each of which checks one of the two conditions. Both threads then proceed symmetrically: if the condition holds, they request event Good; otherwise, they request event Bad and block event Good. Thus, event Good is triggered if and only if both conditions hold for x. If either thread discovers that its respective condition does not hold, event Good becomes blocked and cannot be triggered, resulting in the triggering of Bad. Observe that when handling a previous request both threads block new Check events, to delay new inputs until they can be processed.

answer *true* or *false*. When a thread uses a predicate to indicate its waited-for events, it is notified of a triggered event if that event causes the predicate to evaluate to true, and similarly, if it uses a predicate to indicate its blocked events, an event can be triggered only if the predicate returns *false* for that event. As in BPJ, we require that the requested events be explicitly declared, so that set must be finite. This is required for the event selection process in the ESM.

10.7 Formal Semantics

Having informally described the extensions we propose for the traditional BP framework, this section is dedicated to their rigorous formulation. The definitions are based on, and are similar to, those described in Section 2.2, with alterations to accommodate parameterized events, customizable event selection and dynamic thread creation. The more extensive changes are meant to support the notion of time; in particular, we no longer assume that threads transitions occur instantly, and introduce special “synchronization transitions” and “timeout transitions”.

10.7.1 Event Sets

As discussed in Section 10.6, we want our new semantics to support unbounded parameterized events. For simplicity, we assume that the domains of these parameters are enumerable (say, integers or strings), and so the set of all possible parameterized events E is also enumerable. We assume that this set does not contain the two special symbols: $\bot$, denoting a timeout, and $\top$, denoting thread synchronization.

10.7.2 Behavior Threads

Let E be an event set and let $BT = \{BT^1, BT^2, \ldots\}$ denote a (possibly infinite) set of threads. These threads can be thought of as *templates*, instances of which are spawned as the program runs. In particular, multiple instances of the same thread may exist simultaneously. Each thread is formalized by the tuple $BT^i = \langle Q^i, q_0^i, \delta^i, \xi^i, R^i, B^i, T^i \rangle$, where Q^i is the (possibly infinite) set of states, q_0^i is an initial state, and $R, B : Q^i \to 2^E$ are functions that map states to requested and blocked states (respectively), as before. As discussed in Section 10.6, the range of B^i may contain infinite sets, but the range of R^i may only contain finite sets.

$T^i : Q^i \to (0, \infty)$ is a timeout function, assigning each state a positive timeout value. This value is not an absolute time, but rather the amount of (say, seconds) the thread is willing to spend in that state before a timeout should occur.

The transition relation $\delta^i \subseteq Q^i \times (E \cup \{\bot\}) \times Q^i$ is used to map states and events to new states. We stipulate that for every state q for which $T^i(q) < \infty$ there is an edge $\langle q, \bot, \tilde{q} \rangle \in \delta^i$. In other words, if a thread declares a timeout, a timeout would cause it to transition.

Finally, $\xi^i : \delta^i \to BT \times \mathbb{N}$ is a function that maps transitions to thread instances that should be spawned when they are traversed. Each such thread is paired with the number of instances that should be spawned. We sometimes abuse notation, and consider ξ^i as mapping states to multisets.

Observe that the above definition does not support thread *termination*; indeed, for simplicity, we assume that when a thread terminates it goes into a special "shutdown" state, in which it does not request, wait-for or block any events, and sets its timeout value to ∞. This is the semantic equivalent of thread termination, as the thread can no longer affect the execution; in practice, the thread can safely be discarded.

10.7.3 Configurations

A *thread configuration* c is given by the tuple

$$c = \langle index, sync, state, time \rangle,$$

where:

- $index$ is an integer, denoting that the thread in question is an instance of thread template BT^{index}.
- $sync$ is a boolean variable, indicating whether the thread is currently synchronized or not.
- $state \in Q^{index}$ indicates the state the thread is in if $sync = true$, or the state the thread is expected to reach in its next synchronization if $sync = false$.
- $time$ is a positive real, indicating the instant in time when the thread last synchronized. This field is only meaningful if $sync = true$.

Two thread configurations are equal if and only if all of their meaningful fields are equal. A *system configuration* γ is a finite tuple $\gamma = \langle c^1, \ldots, c^k, t \rangle$, where $k \geq 1$ and each $c^i = \langle index^i, sync^i, state^i, time^i \rangle$ is a thread configuration, and t is a positive real, indicating the current time. The system configuration indicates which thread instances are currently active, and thus indicates the global configuration of the system.

A system configuration γ is called *initial* if and only if:

$$\forall_{1 \leq i \leq k}, \left(sync^i = false \wedge state^i = q_0^{index^i} \right) \bigwedge t = 0$$

That is, all currently running thread instances are unsynchronized, are expected to arrive at their initial states in their next synchronization, and the system time is 0. This is the state of the system at the beginning of the execution.

Given two configurations $\gamma = \langle c^1, c^2, \ldots, c^k, t \rangle$ and $\tilde{\gamma} = \langle \tilde{c}^1, \tilde{c}^2, \ldots, \tilde{c}^{k'}, \tilde{t} \rangle$ and an event $e \in E \cup \{\bot, \top\}$, we say that $\tilde{\gamma}$ is a successor of γ with respect to e, denoted $\gamma \xrightarrow{e} \tilde{\gamma}$, if the following conditions apply:

1. Existing threads are preserved: $k' \geq k$ and

$$\forall_{1 \leq i \leq k}, \ index^i = \widetilde{index^i}.$$

2. Time moves forward: $\tilde{t} \geq t$. We assume event selection can be resolved in zero time, and allow $\tilde{t} = t$.
3. If $e = \top$, then $\gamma \xrightarrow{e} \tilde{\gamma}$ is a valid *synchronization transition*; if $e = \bot$, then it is a *timeout transition*; and otherwise, it is an *event selection transition*. These terms are defined next.

Synchronization Transitions

These are transitions for which $e = \top$; they correspond to a single thread synchronizing. A transition is a valid synchronization transition if the following conditions hold:

- Synchronization transitions cannot spawn new threads, i.e. $k = k'$.
- At least one thread is unsynchronized, i.e. $\exists_{1 \le i \le k}$ such that $sync^i = false$.
- No synchronized threads have timed-out:

$$\forall_{1 \le j \le k},\ \left(sync^j = true \implies T^j(state^j) + time^j < \tilde{t}\right)$$

- All threads except the thread that timed-out remain in the same configuration: $\forall_{1 \le j \le k},\ (j \ne i \implies c^j = \tilde{c}^j)$.
- The thread that synchronized is updated to indicate that it has arrived at its expected state in that instant: $\widetilde{sync^i} = true \ \wedge\ \widetilde{state}^i = state^i \ \wedge\ \widetilde{time^i} = \tilde{t}$.

Event Selection Transitions

These are transitions with "real" events, i.e. $e \in E$. They are allowed only when the following conditions hold:

- All threads are synchronized, i.e. $\forall_{1 \le i \le k},\ sync^i = true$.
- The transition must occur immediately upon the last synchronization event: $\tilde{t} = \max_{1 \le i \le k}\{time^i\}$.
- Event e must be enabled: $e \in \bigcup_{i=1}^{n} R^{index^i}(state^i) - \bigcup_{i=1}^{n} B^{index^i}(state^i)$.
- As a result of the event being triggered, all threads that have transitions for the event become unsynchronized, and their new expected states (when next they synchronize) are determined according to their transition rules:

$$\forall_{1 \le i \le k},\ (\delta^i(state^i, e) \ne \varnothing \implies \widetilde{sync^i} = false \ \wedge\ \widetilde{state^i} \in \delta^i(state^i, e))$$

- Threads that did not have a transition to traverse retain their configurations:

$$\forall_{1 \le i \le k},\ \left(\delta^i(state^i, e) = \varnothing \implies c^i = \tilde{c}^i\right)$$

- New threads may be spawned as a result of the transition. These threads correspond to thread configurations $\tilde{c}^{k+1}, \ldots, \tilde{c}^{k'}$. These threads must be precisely those threads that the existing threads spawn, i.e.

$$\{\widetilde{index}^{k+1}, \ldots, \widetilde{index}^{k'}\} = \bigcup_{1 \le i \le k \mid \delta^i(state^i, e) \ne \varnothing} \xi^{index_i}(\langle state^i, e, \widetilde{state}^i \rangle)$$

where both hands of the equation are interpreted as multisets.

- New threads are spawned unsynchronized, and are expected to reach their initial states:

$$\forall_{k < i \le k'}, \left(\widetilde{sync}^i = false \ \wedge \ \widetilde{state}^i = q_0^{\widetilde{index}^i} \right).$$

Timeout Transitions

These are transitions with $e = \bot$. They occur when a previously synchronized thread times out. Formally, they are allowed if and only if the following holds:

- Timeout transitions are only allowed when no *event selection transitions* are enabled; that is, if there exists at least one unsynchronized thread or if all threads are synchronized but there are no enabled events. Formally, denoting by $Enabled$ the set of enabled events

$$Enabled = \bigcup_{i=1}^{k} R^{index^i}(state^i) - \bigcup_{i=1}^{k} B^{index^i}(state^i),$$

  we stipulate that $(\exists_{1 \le i \le k},\ sync^i = false) \vee (Enabled = \varnothing)$.

- One thread has to have timed out, i.e

$$\exists_{1 \le i \le k}, \left(sync^i = true \ \wedge \ T^i(state^i) + time^i = \tilde{t} \right)$$

- This thread becomes unsynchronized and traverses a timeout transition:

$$\widetilde{sync}^i = false \ \wedge \ \widetilde{state}^i \in \delta^i(state^i, \bot)$$

- All other threads remain in the same configurations:

$$\forall_{1 \le j \le k} \left(j \ne i \implies c^j = \tilde{c}^j \right)$$

- The transition may spawn new threads:

$$\{\widetilde{index}^{k+1}, \ldots, \widetilde{index}^{k'}\} = \xi^{index^i}(\langle state^i, \bot, \widetilde{state}^i \rangle)$$

- New threads are spawned unsynchronized, and are expected to reach their initial states:

$$\forall_{k<j\leq k'},\ \left(\widetilde{sync^j} = false \wedge \widetilde{state^j} = q_0^{\widetilde{index^j}}\right).$$

Thread Termination

For simplicity, the concept of thread termination is not included in the semantics. Instead, we assume that terminated threads enter a dormant state, in which they request, wait-for and block nothing, and do not specify a timeout. This is the equivalent of removing the thread from the pool of active threads. Naturally, in practice it is better to let threads terminate and free the resources they were allocated. Indeed, this is the case in BPC.

System Vs. Environment

From the ESM's point of view, synchronization transitions can be seen as managed by the environment; the ESM has no control on when threads will synchronize. The other two kinds — event selection transitions and timeout transitions — are triggered by the ESM, and it has no flexibility in selecting and scheduling them. As previously mentioned, it would be interesting to extend this work to allow the ESM flexibility in, say, purposely delaying event selection transitions, in order to achieve some goal, along the outline of the smart play-out mechanism of [84, 85].

10.7.4 Behavioral Programs

A behavioral program P consists of a (possibly infinite) event set E, a (possible infinite) thread template set $BT = \{BT^1, BT^2, \ldots\}$ an initial system configuration γ^0, and a event selection strategy f_{es} (as defined in Section 10.4).

An execution ε of P is a sequence $\varepsilon = \gamma^0 \xrightarrow{e_0} \gamma^1 \xrightarrow{e_1} \ldots$ of successive configurations, where $e_i \in E \cup \{\bot, \top\}$ for all i. For every $e_i \notin \{\top, \bot\}$, we require that $f_{es}(\gamma_0, \gamma_1, \ldots, \gamma_i) = e_i$, that is that the triggering of "real" events is performed according to the strategy. The execution may either be infinite, or finite if it ends in a *terminal* configuration — a configuration with no successors. Specifically, a terminal configuration is one in which all threads have synchronized, there are no enabled events, and all threads have set their timeout values to ∞.

The *run* that corresponds to execution ε is a (possibly infinite) sequence of event-and-time pairs $\langle\langle e_{i_0}, t_{i_0}\rangle, \langle e_{i_1}, t_{i_1}\rangle, \ldots\rangle$ that correspond to just the *event selection transitions* of ε. The run indicates the "real" events that were triggered, and the time of their triggering. Timeout and synchronization transitions are considered internal, and do not appear in the run. The language of a behavioral program P, denoted $\mathfrak{L}(P)$, is the set of runs of all valid executions of the system.

Note. The above semantics can be extended to support sensor threads that do not delay the system as they wait for input. Intuitively, this is performed by relaxing the prerequisites of *event selection transitions*, to no longer require that the sensor threads be synchronized. Extending the formalism to fully support the eager execution mechanism discussed in Chapter 9 in the presence of timeouts is left for future work.

10.8 The BPC Framework

In this section we present the BPC framework for behavioral programming in C++, which implements the extensions discussed in previous sections. It is available online at [9]. The framework is designed to allow the user to conveniently define and write behavior threads while using the full power of the C++ programming language. The synchronization and coordination mechanism is implemented as part of the framework, and is concealed from the user.

10.8.1 User Interface

Behavior threads are implemented as classes that inherit from the *BThread* class. They are customized to carry out particular behavior by overriding the `entryPoint` method, which the framework invokes when the thread starts. The interface provided by the parent class includes the `bSync` method to perform thread synchronization and the `lastEvent` method to retrieve the result of the last synchronization point — be it an event or a timeout. The `bSync` method pauses the thread until a requested or waited-for event is triggered, or until a timeout occurs.

Events in the system are instances of class *Event*. All events have a *type* (an integer), and additional parameters can be added by supplying classes that inherit from *Event*.

In order to run the application, the user instantiates the initial threads inside the `main` method of the program and calls a special `start` method provided by the framework.

Parts of the application that corresponds to the example in Figure 2.2 appear in Figure 10.9. Additional features of BPC, such as dynamic thread creation and customized event selection strategies, appear as parts of the case-study, described in Section 10.9. In later code snippets we sometimes omit parts of the C++ syntax and focus on the body of the threads.

In order to facilitate the migration of existing behavioral code, BPC supports programs where the extension idioms that we propose in this work are not used. For instance, synchronization calls may contain just the first 3 parameters, ignoring the timeout parameter; the effect is the same as passing the NO_TIMEOUT value, which has the same semantics as a synchronization call in traditional BP. If no customized event selection strategy is defined, BPC uses a default, arbitrary selection scheme. Naturally, events without parameters and programs with just statically created threads are also allowed.

```
1  enum { WaterLow, AddHot, AddCold };
2
3  class WhenLowAddHot : public BThread {
4     void entryPoint() {
5        while( true ) {
6        set<Event> requested;
7        set<Event> waitedFor = { WaterLow };
8        set<Event> blocked;
9
10       BSYNC( requested, waitedFor, blocked, NO_TIMEOUT );
11
12       waitedFor.clear();
13       requested = { AddHot };
14       for( unsigned i = 0; i < 3; ++i )
15         BSYNC( requested, waitedFor, blocked, NO_TIMEOUT );
16     }
17   }
18 };
```

Figure 10.9: The events in the program have a *type* field with possible values WaterLow, AddHot and AddCold, and no parameters. The *WhenLowAddHot* class inherits from *BThread*, with the `entryPoint` method customized to carry out the specific thread behavior. The thread runs in an infinite loop, and in each iteration it waits-for event WaterLow and then requests event AddHot three times. The `bSync` method takes three event vectors and a timeout parameter (here, set to the special value NO_TIMEOUT).

10.8.2 The Underlying Mechanism

Communication between threads and the event selection mechanism is performed using standard client-server sockets. Throughout the run, the ESM maintains a server socket which awaits new threads that might connect. For each currently active thread, the ESM maintains an active socket connection on which synchronization data and triggered event information are exchanged. Whenever a thread synchronizes with the ESM, the latter checks if the thread declared a timeout at this synchronization point; if so, it sets a timer to expire accordingly. If an event that the thread requested or waited-for is triggered before the timeout expires, the timer is reset. The pseudocode for the ESM appears in Algorithm 13.

Line 13 of the algorithm does not specify which event e to choose in case there are multiple enabled events. The default option in BPC is arbitrary event selection. If the programmer has customized the event selection mechanism, he/she has provided an object that can take the list of enabled events and return the next choice, in which case that object is then invoked. The object may also store information from previous iterations and use it in the present selection iteration. An example appears in Section 10.9.

Another variant of BP that may be useful in a distributed setting includes a distributed version ESM (see Chapter 9). In this variant, which is also supported in BPC, the threads are partitioned into sets — each of which is managed by a different *ESM agent*. The agents exchange information among themselves when needed. Distributing the ESM can be useful, e.g., when threads run on multiple machines, and communication between these machines is

Algorithm 13 Event Selection Mechanism

```
1:  ActiveThreads ← ∅
2:  Synchronized ← ∅
3:  while true do
4:      wait for new threads, synchronizations and timeouts
5:      if  new thread bt connected  then
6:          ActiveThreads ← ActiveThreads ∪ {bt}
7:      else if timeout for thread bt expired then
8:          Synchronized ← Synchronized − {bt}
9:          inform bt of a timeout
10:     else if thread bt synchronized then
11:         set the timeout timer for bt
12:         if  ActiveThreads = Synchronized then
13:             if exists event e enabled for triggering then
14:                 for  every thread bt′ that requested/waited-for e  do
15:                     send e to bt′
16:                     Synchronized ← Synchronized − {bt′}
17:                     reset timeout timer for bt′
```

slow or costly.

10.9 Case-Study: a Web-Server

In this section we survey the architecture of our web-server case-study, dwelling in particular on the implementation of the examples discussed in Sections 10.3-10.5. Most of the threads and inter-thread interactions described in this section are displayed in Figure 10.10. Apart from giving the technical details, throughout this section we also try to convey to the reader a sense of the interaction between the behavioral code and native C++ code in our implementation, and also of the incremental development process of a behavioral application.

10.9.1 The Implementation's Layout

Our application consists of two distinct sets of threads, one for the TCP layer and one for the HTTP layer. The two layers interact with each other via behavioral events, and each also has an additional source of input: the TCP layer reads TCP segments off a "raw socket", and the HTTP layer reads files from a given directory. These additional inputs are obtained by threads containing non-trivial native C++ code, and are then translated into behavioral events in order to be passed to other threads.

Internally, each layer is designed using a *dispatcher* architecture: a dispatcher thread handles each incoming segment, classifies it according to its attributes, and then passes it to specific *handler* threads via behavioral events. These handler threads can then request additional events

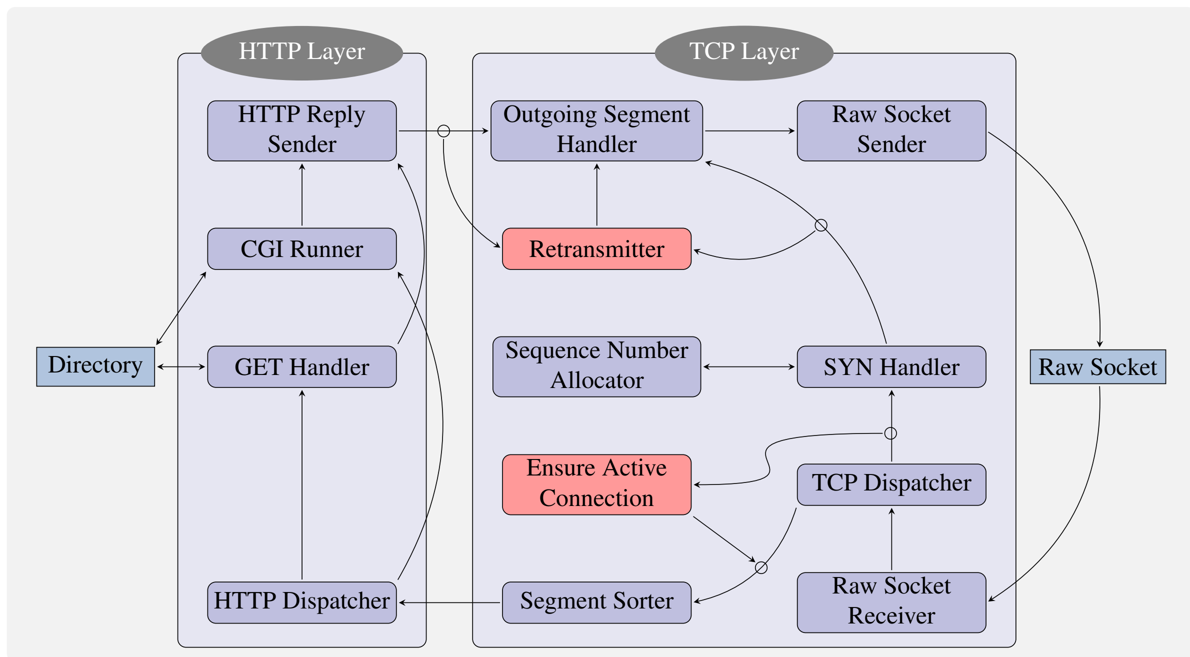


Figure 10.10: An overview of the web-server's architecture. For clarity, many details have been omitted; the full code is available online [10]. Rounded rectangles represent threads, and edges represent the logical interactions described throughout Section 10.9. The threads are partitioned into a TCP layer and a HTTP layer; the layers interact with each other and with additional sources of input ("Raw Socket" and "Directory"). Threads marked in orange were *incrementally* added on top of an already working basis: (1) the *Retransmitter* mechanism that waits for outgoing segments and re-feeds them to *Outgoing Segment Handler* if they are not acknowledged, and (2) the *Ensure Active Connection* mechanism that waits for connection activation/termination events, and blocks HTTP segments on inactive connections from reaching *Segment Sorter*.

in order to issue a reply and/or update other threads of the contents of the segment. Handlers typically perform local computation using native C++ code — e.g., calculating TCP checksums or reading files from the directory. Incoming TCP segments containing HTTP requests are passed between the layers, and the same happens to HTTP replies on their way to the client.

To exemplify the server's operation we describe in more detail the handling of TCP connection establishment requests (SYN segments). Initially, the *RawSocketReceiver* sensor thread reads the incoming segment from the socket. When it is received, the thread requests a TcpSegmentReceived event — with the segment as its parameter — in order to pass the segment to the *TcpDispatcher* thread.

The *TcpDispatcher* uses native C++ code to classify incoming TCP segments according to their attributes, and requests additional events accordingly. SYN requests, for example, are identified by reading the SYN flag from the TCP header of the segment. The thread then requests a TcpSynRequest event in order to notify *TcpSynHandler* — the specific handler for SYN requests.

The *TcpSynHandler* thread responds to each request by generating a TcpOutgoingSegment event, with a SYN-ACK segment as its parameter. Finally, this event then gets translated into a TcpSendSegment event — handled by the *RawSocketSender* sender thread, which actually sends the segment to the client.

We note that in order to construct the SYN-ACK segment, the *TcpSynHandler* thread must first acquire a fresh sequence number. This is performed by sending a request to, and receiving a response from, the *SequenceNumberAllocator* thread — the thread in charge of managing the sequence numbers of every TCP connection. *SequenceNumberAllocator* may handle simultaneous requests for sequence numbers (for the same connection) from multiple threads, and here the BP event selection mechanism guarantees that each outgoing segment has a fresh sequence number: *SequenceNumberAllocator* handles requests (represented by events) sequentially, and thus race conditions are avoided.

Apart from dispatcher and handler threads, additional "standalone" threads exist in the system: for instance, the requirement that TCP segments be sent only on active connections is enforced by the *TcpEnsureActiveConnection* thread. This thread uses blocking to ensure that a TcpSynRequest is triggered before other TCP events — such as those signalling PUSH or ACK segments — are triggered. Likewise, once a FIN segment is triggered, the thread blocks any additional TCP events for that connection.

Segment Reordering

TCP segments that contain data for the HTTP layer are not guaranteed to arrive in the order in which they were sent. Hence, the TCP layer needs to reorder them before passing them on.

During data transfer, TCP segments with data for the HTTP layer cause the triggering of DataToHttp events. Each of these events carries the received segment's sequence number as a parameter. The TCP stack knows the expected sequence number of the next data segment: the initial sequence number is stated by the client at the time of connection establishment, and is subsequently incremented for each segment. Whenever an incoming sequence number is greater than the one expected, the stack realizes that a segment is missing; and when this segment later arrives, reordering takes place. Pseudocode for the *SegmentSorter* thread, which is in charge of this reordering, appears in Figure 10.11.

Segment Retransmission

As mentioned in Section 10.3, when sending out a segment, the TCP stack must wait for an acknowledgment message — and if one does not arrive, the segment needs to be resent. There exist several sophisticated retransmission policies, aimed at reducing traffic congestion, which have adjustable retransmission periods. For our case-study we opted for the simplest scheme —

```
1  while( true ) {                    // SegmentSorter thread
2    set<Event> requested;
3    set<Event> waitedFor = { TcpSynRequest, DataToHttp };
4    set<Event> blocked;
5
6    BSYNC( requested, waitedFor, blocked, NO_TIMEOUT );
7
8    if( lastEvent().type() == TcpSynRequest )
9      storeSeqNumber( lastEvent().seqNumber() );
10   else if( lastEvent().seqNumber() != expectedNumber() )
11       storeData( lastEvent().data() );
12   else
13     sendReorderedSegments( lastEvent().data() );
14 };
```

Figure 10.11: The *SegmentSorter* thread waits for TcpSynRequest and DataToHttp events. When a TcpSynRequest event occurs, the thread extracts the sequence number for later use. When a data segment is received, its sequence number is compared to the expected number. If it does not match, the segment is stored. If it does match, the segment is passed on to the HTTP layer, along with any consecutive segments previously stored, and the expected sequence number is updated.

retransmission after a fixed waiting period. Implementing additional schemes is left for future work.

In our implementation, segments leaving the TCP stack on their way to be sent to the client via the raw socket always pass as TcpOutgoingSegment events. Our retransmission mechanism waits for these events and stores the outgoing segments. Then, if they are not acknowledged withing a fixed period of time, it retransmits them — until an acknowledgment is received. Pseudocode appears in Figure 10.12[1]

Customized Event Selection

As mentioned in Section 10.4, we occasionally found customizing the event selection strategy a straightforward method in order to enforce certain requirements. For example, in one case we wanted to ensure that all outgoing segments finish sending prior to sending the segment indicating the connection being closed (a FIN segment) — a property that was not trivially upheld by the TCP stack. Another example was giving priority to starving connections, namely connections whose events have not been triggered in a while, in order to avoid retransmission of segments and the congestion incurred by it.

Pseudocode for a customized event selection function that addresses these two issues appears in Figure 10.13.

[1]The depicted solution spawns a thread for each outgoing segment, which incurs overhead (as discussed in Section 10.5). In practice, we found that it was more efficient to spawn one *Retransmitter* thread per connection, and have it handle all of that connection's segments. Nevertheless, we feel Figure 10.12 better illustrates the principles described in Section 10.5.

10.9.2 Features and Evaluation

Our implemented TCP stack supports connection establishment and termination, data sending and acknowledgments, keep-alive messages, segment reordering and segment retransmission. Simultaneous connections are also supported, whereas dealing with flow and congestion control is still work in progress. The HTTP stack supports GET requests, error and redirection messages, and the execution of CGI scripts. The project contains over 20k lines of code, and is available online [10].

We constructed our case-study as a proof of concept, and, in our experiments, it provided "smooth" surfing of websites. Nonetheless, it cannot presently compete with industrial web-servers performance-wise, e.g. in throughput rate. We thus focused our evaluation on proper adherence to the TCP/HTTP protocols.

We tested the system with two widespread browsers, Firefox and Google Chrome. In both of these, the server properly displayed non-trivial sites, with both static and dynamic (e.g., PHP, CGI) pages. In particular, we ran a copy of the BP website [144] on the behaviorally-programmed server.

To test the more advanced features of the server, such as segment retransmission and segment reordering, we conducted tests in which the client connected to the server through a *proxy* — a third piece of software, which we controlled. We then simulated unreliable networks by having the proxy delay or drop segments, or deliver them out of order. All webpages were nevertheless properly displayed.

Finally, for stress testing we ran ten clients that were simultaneously trying to upload a 10 megabyte file each to the server. The proxy was set to maximal interference — that is, not a single segment was delivered to the server in the correct order. All files were successfully received and reassembled by the server.

10.9.3 Discussion: Incremental Development

An important feature previously attributed to scenario-based programming (and hence BP too) is that it facilitates incremental development [90, 89]. One of our goals was to check whether this would still hold for large programs. While this property is difficult to quantify and may depend on coding habits, the experience we gained when developing our case-study indicates an affirmative answer.

We built our web-server iteratively, repeatedly translating parts of the specification into threads, and with only a vague "big picture" in mind. For instance, we first programmed the connection establishment and data exchange parts of the TCP stack, but without considering segment retransmission or ignoring segments on closed connections. Later, we found that adding these features incurred no changes to existing code (see also Figure 10.10). Naturally,

in some cases — e.g., adding the reordering of TCP segments — some code was changed, but the changes were usually local and contained.

One could argue that the incremental development of our case-study was made possible because of the *dispatcher-handler* design pattern that we used. This was indeed partially the case, but we feel that two remarks are in order: (i) BP promoted the use of a dispatcher-handler pattern in the first place; and (ii) in some cases, as in the case of segment retransmission, incremental development was made possible because communication between threads was performed strictly through the triggering of events. Hence, we could easily "hook" onto these events when needed.

10.10 Related Work

Our proposed extensions to BP are common programming idioms, and exist, in various forms, in numerous high-level languages. Thus, comparisons in this section focus mainly on popular programming formalisms that, similarly to BP, are geared towards discrete event systems.

The principal extension to BP that we proposed is the timeout idiom. This is a step in moving away from the *synchrony hypothesis*, according to which local computation takes negligible time. The synchrony hypothesis is used in popular languages for programming event-driven reactive systems, such as Esterel [31], Lustre [77] and Signal [112], and also in the non-object oriented version of Statecharts [92]. Formally allowing non-negligible computation time broadens the scope of problems to which BP can be applied.

Other parallel programming languages support various idioms for manipulating time. *UML Sequence Diagrams* [11] support constructs that impose a required delay between the send and receive events of a message [37]. *Message Sequence Charts* (*MSC*s) support timers [102, 16] that can be set, reset and checked for timeouts, and delay intervals [16, 126] to specify maximal or minimal delay between actions. We have demonstrated that similar constraints can be applied using our timeout mechanism. Of particular interest is the *live sequence charts* (*LSC*s) language [57, 87], the precursor to and main motivation for behavioral programming. In this context, the ongoing evolution of BP is, perhaps unsurprisingly, similar to the one LSCs underwent with the addition of time-aware charts [88], allowing one to bound the flow of time between consecutive events. Similar concepts appear also in component based programming languages, such as *BIP* [29]. In BIP, components may contain *timed variables*, and may use them transition guards. These variables are globally incremented when no higher priority action is enabled.

The second extension that we proposed was customizable event selection strategies. Scenario-based programs typically have to choose between several enabled events, and several selection strategies have been previously proposed. Among these are arbitrary selection,

look-ahead algorithms (*smart play-out*) [84, 85] and planning-based approaches [93]. Selection can also be interactive, which is useful, for instance, in the context of debugging unrealizable behavioral specifications [122]. By allowing general user-specified event selection strategies, these approaches could be more readily integrated into BP.

Another extension that proposed was to consider threads as templates of which multiple copies may be instantiated. This technique bears resemblance to the situation in LSCs, where multiple instances of the same chart may run simultaneously. There, additional copies are spawned based on preconditions defined in each chart, instead of actively by other threads as in our case, although the two approaches seem equivalent. The direct spawning of modules by other modules is also supported in Esterel and Signal.

Finally, we proposed to extend BP with parameterized events. Parameterized message passing between modules is quite fundamental in concurrent programming. It exists, e.g., in Esterel, UML sequence diagrams, LSCs and, for bounded parameters, also in earlier work on BP [86].

10.11 Conclusion and Future Work

In this work we set out to study the applicability of the BP paradigm to real-world systems. Through our work on a large case-study, we were able to identify several common programming tasks for which BP's traditional idioms provide only partial solutions, and proposed additional idioms to overcome these difficulties while — trying to maintain BP's simple and intuitive interfaces. The new idioms include time-aware threads, a customizable event selection mechanism, the dynamic creation of threads, and parameterized events. By integrating these idioms into our development environment we were able to complete our implementation, thus providing what we feel is significant evidence that BP does indeed scale-up to real-world problems.

In choosing our proposed extension to BP, we took care to only add idioms that allowed us to accomplish programming tasks that were previously beyond the scope, or at least very difficult to accomplish, in BP. Thus, we hope we were able to avoid clutter, and retain most of the simplicity that characterizes the traditional BP framework.

Our proposed extensions were driven by the needs that arose during the development of our specific case-study; and hence it is possible that, through the development of additional behavioral projects, BP may need to be extended with additional idioms. However, due to the large variety of programming tasks entailed by the web-server project (e.g., handling timeouts, string manipulation, file access, checksum calculations, etc), we believe that our proposed extensions are robust, and could prove sufficient for a variety of programming tasks. We regard our extensions, and also future extensions to BP, a part of the typical evolution of programming languages.

In the future, we plan to enhance our case-study by adding features like flow control, congestion control, selective acknowledgments and smart retransmission schemes to our protocol stacks. These extra features may reveal additional idioms worth adding to BP. Further, we plan to work on improving the efficiency of our case-study, in order to gain a better understanding of the overhead the BP infrastructure might incur in large systems.

```
class Retransmitter : public BThread {
  void entryPoint() {
    set<Event> requested;
    set<Event> waitedFor = { TcpOutgoingSegment };
    set<Event> blocked;

    while( true ) {
      BSYNC( requested, waitedFor, blocked, NO_TIMEOUT );
      new PeriodicSender( lastEvent() );
    }
  }
};

class PeriodicSender : public BThread {
  PeriodicSender( Event TcpOutgoingSegment ) {
    storedSegment = TcpOutgoingSegment;
  }

  void entryPoint() {
    bool done = false;

    while( !done ) {
      set<Event> requested;
      set<Event> waitedFor = { ackForStoredSegment() };
      set<Event> blocked;

      BSYNC( requested, waitedFor, blocked, 2 );
      if( timeoutOnlastSync() ) {
        waitedFor.clear();
        requested = { storedSegment };
        BSYNC( requested, waitedFor, blocked, NO_TIMEOUT );
      }
      else {
        done = true;
      }
    }
  }
};
```

Figure 10.12: Pseudocode for the segment retransmission mechanism. The *Retransmitter* thread waits-for TcpOutgoingSegment events — events that indicate a TCP segment about to be sent — and for each such event it spawns an instance of the *PeriodicSender* thread. The *PeriodicSender* instance receives through its constructor the segment that it is supposed to monitor. It then waits for an acknowledgment of that segment for 2 seconds. If an acknowledgment message fails to arrive, the thread retransmits the segment, and the process repeats. When an acknowledgment is received, the thread terminates. Note that the `ackForStoredSegment` method (code omitted) is a *predicate* — it evaluates to true only for TcpAckReceived events with the proper acknowledgment information. Also, a bookkeeping mechanisms (also omitted) is required to prevent the creation of additional *PeriodicSender* threads for a segment that is being retransmitted.

```
Event choose( set<Event> enabledEvents ) {
   set<Event> candidates = eventsOfStarvedConnection( enabledEvents );

   if( candidates.has( SendTcpFin ) && candidates.hasOtherThan( SendTcpFin ) )
      return candidates.otherThan( SendTcpFin );

   else return *candidates.begin();
};
```

Figure 10.13: Pseudocode for the customized event selection strategy. At every synchronization point, this function is invoked with the set of enabled events, of which it must select one for triggering. Information from previous iterations may be stored. Our specific implementation gives precedence to previously "starved" connections: that is, it favors the connection that has waited the longest for an event to be triggered. This part is abstracted away in the method `eventsOfStarvedConnection`. Once a connection is selected, its associated events are the candidates for triggering; among these, we prefer events that are not SendTcpFin, so that pending data transmission requests are addressed before the connection is closed. Otherwise, an arbitrary event is selected.

Chapter 11

An Initial Wise Development Framework for *RWB*

11.1 Introduction

In this chapter we seek to bring together several of our aforementioned results, by providing an interactive and proactive framework for developing $\mathcal{RWB}$ programs — which utilizes our the various analysis tools that we have discussed in Part II.

The development of large reactive software systems is an expensive and error-prone undertaking. Deliverables will often fail, resulting in unintended software behavior, exceeded budgets and breached time schedules. One of the key reasons for this difficulty is the growing complexity of many kinds of reactive systems, which increasingly prevents the human mind from managing a comprehensive picture of all their relevant elements and behaviors. Moreover, of course, the state-explosion problem typically prevents us from exhaustively analyzing all possible behaviors. While major advances in modeling tools and methodologies have greatly improved our ability to develop reactive systems by allowing us to reason on abstract models thereof, specific solutions are quickly reaching their limits, and resolving the great difficulties in developing reliable reactive systems remains a major, and critical, moving target.

Over the years it has been proposed, in various contexts, e.g., [138, 137, 43, 82], that a possible strategy for mitigating these difficulties could lay in changing the role of the computer in the development process. Instead of having the computer serve as a tool, used only to analyze or check specific aspects of the code as instructed by the developer, one could seek to actually transform it into a member of the development team — a proactive participant, analyzing the entire system and making informed observations and suggestions. This way, the idea goes, the computer's superior capabilities of handling large amounts of code could be manifested. Combined with human insight and understanding of the system's goals, this synergy could produce more reliable and error-free systems.

In this chapter we follow this spirit, and present a methodology and an interactive framework for the modeling and development of complex reactive systems, in which the computer plays a proactive role. Following the terminology of [82], and constituting a very modest initial effort along the lines of the Wise Computing vision outlined there, we term this framework a *wise* framework. Intuitively, a truly *wise* framework should provide the developer with an interactive companion for all phases of system development, "understand" the system, draw attention to potential errors and suggest improvements and generalizations; and this should be done via two-way communication with the developer, which will be very high-level, using natural (perhaps natural-language-based) interfaces. The framework presented here is but a first step in that direction, and focuses solely on providing an interactive development assistant capable of discovering interesting properties and drawing attention to potential bugs; still, it can already handle non-trivial programs, as we later demonstrate through a case-study.

Various parts of this approach have been implemented by a variety of researchers in other forms, as described in Section 11.5. A main novel aspect of our approach, however, is in the coupling of the notion of a proactive and interactive framework with the BP scenario-based programming language. The BP formalism makes it possible for our interactive development framework to repeatedly and quickly construct abstract executable models of the program, and then analyze them in order to reach meaningful conclusions. It is now widely accepted that a key aspect in the viability of analysis tools and environments is that they are sufficiently lightweight to be integrated into the developer's workflow without significantly slowing it down [139, 56]. We attempt to achieve this by leveraging scenario-based modeling. As demonstrated in later sections, the proactiveness of our approach and its tight integration into the development cycle can lead to early detection of bugs during development, when they are still relatively easy and cheap to fix.

The rest of this chapter is organized as follows. In Section 11.2 we introduce our development framework by means of a simple example. In Section 11.3 we discuss the various components of the framework in more detail, and in Section 11.4 we describe a case-study that we conducted. Related work appears in Section 11.5, and we conclude in Section 11.6.

11.2 A Simple Example

In this section we attempt to convey to the reader, intuitively, the sense of working in a wise development framework from a developer's point of view. Thus, we focus almost exclusively on the user experience, and defer more details about the inner workings of the framework itself to Section 11.3.

We demonstrate the framework's operation through the incremental modeling of a small, illustrative system. Suppose we are developing behavioral code for a safe that has three levers

and an "open door" button. The specification given to us indicates that in order to open the door, a user needs to correctly configure the three levers and then click the button. Clicking the button when the levers are not correctly configured should not open the door. We refer to the three levers as levers A, B and C; and each lever has three possible positions, denoted as *one*, *two* and *three*. We denote the configuration of the levers as a tuple: for instance, configuration $\langle 1,3,2 \rangle$ indicates that lever A is in position one, lever B is in position three, and lever C is in position two. The initial configuration is $\langle 1,1,1 \rangle$, and the correct configuration for opening the door is $\langle 2,3,2 \rangle$. The user can request the triggering of events of the form SetXToY, indicating that lever X is set to position Y, and also of ClickButton events. The system may request an OpenDoor event, as well as any internal event needed for the implementation.

We now describe the incremental modeling of this system in BPC, accompanied by the wise framework. We start by modeling the three levers. This is done by creating, for each lever, a scenario object that waits for events signaling that the position of that lever has changed, and storing the current position. The code appears and is explained in Figure 11.1.

```
1  Event position = SetXToOne;
2
3  while( true ) {
4    set<Event> requested = {};
5    set<Event> waitedFor = { SetXToOne, SetXToTwo, SetXToThree };
6    set<Event> blocked;
7
8    switch( position ) {
9    case SetToOne:
10     blocked = { LeverXInTwo, LeverXInThree };
11   case SetToTwo:
12     blocked = { LeverXInOne, LeverXInThree };
13   case SetToThree:
14     blocked = { LeverXInOne, LeverXInTwo };
15   }
16
17   BSYNC( requested, waitedFor, blocked );
18   position = lastEvent();
19 }
```

Figure 11.1: BPC code for a scenario object called *LeverX*, representing the behavior of a single lever X (X represents A, B or C). Line 17 contains the *BSYNC* synchronization call, where the object synchronizes with all other objects and declares its requested, waited-for and blocked events. The lever object never requests any events, and continuously waits for events signifying that the lever has changed its physical position — events SetXToOne, SetXToTwo, and SetXToThree. When one of these is triggered, line 17 returns, and the object updates its internal state in line 18. Note also events LeverXInOne, LeverXInTwo and LeverXInThree, which represent other scenarios querying the physical position of lever X. The lever object constantly blocks those events that correspond to all "wrong" physical positions. Thus, if another object requests all three events, then only one event — the one corresponding to the actual lever's position — will be triggered. An example appears in Figure 11.2.

After modeling the three lever objects, we get the first input from the wise development

framework:

> **Warning:** *Objects LeverA, LeverB and LeverC constitute a ternary shared array. However, they are not used. Consider removing them.*

We should emphasize that the wise development framework is oblivious to the specifics of our program, i.e., it has no concept of levers. It did, however, recognize a pattern in our system model: that the three lever objects actually operate like a "shared array". Here, the term shared array means that other objects can "write" to it (i.e., by requesting SetXToY events), or "read" from it (by requesting LeverXInY events). This is an interesting insight about the implementation, which we did not even have in mind, but which the development framework will utilize later on. As for the comment that the levers are currently unused, this makes sense — as we have not written additional code yet.

Next, we add a scenario that allows the user, through a simple interface, to request the triggering of SetXToY events, and also the ClickButton event (code omitted). When we recompile the code, the development framework prompts us that now the shared array is written to but is never read from, and can still be removed. Then, we add the *ButtonPressed* scenario (Figure 11.2) that handles the pressing of the button — it queries the lever configuration, and if it is $\langle 2,3,2\rangle$ it requests an OpenDoor event.

However, as the caption explains, the code in Figure 11.2 is actually erroneous: we copied and pasted the code checking lever B but did not correctly modify it to check lever C. The wise development framework now produces the following message:

> **Warning:** *Scenario ButtonPressed has an unreachable synchronization point in line 20. Suggesting an optimization. Also, the state of LeverC is never read.*

This message immediately points us to the error in the model, giving us enough information to quickly realize what has happened. The optimization proposed by the framework (not shown), in which the unreachable state is removed, is actually a graphical representation using the Goal visualization tool [145].

We stress that the realization that line 20 is unreachable is not trivial, as it is not a property that is local to the *ButtonPressed* object. In particular, it cannot be deduced by inspecting the *ButtonPressed* object in isolation, and thus it is very different from deducing, say, that in $if(false)(foo())$ the function $foo()$ can never be called. Rather, this property stems from the joint behavior of *ButtonPressed* and *LeverB*, where *ButtonPressed* expects *LeverB* to be in two different states simultaneously, which cannot occur.

And so, we correct the error in line 16 of *ButtonPressed*. Now the warnings from the development framework disappear, and instead we receive the following information:

> **Information:** *Event OpenDoor appears to only be triggered after event LeverCInTwo.*

```
while( true ) {
  BSYNC( {}, { ClickButton }, {} );

  Set<Event> queryA = { LeverAInOne, LeverAInTwo, LeverAInThree };
  Set<Event> queryB = { LeverBInOne, LeverBInTwo, LeverBInThree };
  Set<Event> queryC = { LeverCInOne, LeverCInTwo, LeverCInThree };

  BSYNC( queryA, {}, {} );
  if( lastEvent() != LeverAInTwo )
    continue;

  BSYNC( queryB, {}, {} );
  if( lastEvent() != LeverBInThree )
    continue;

  BSYNC( queryB, {}, {} );
  if( lastEvent() != LeverBInTwo )
    continue;

  BSYNC( { OpenDoor }, {}, {} );
}
```

Figure 11.2: The *ButtonPressed* scenario, which waits for a ClickButton event, queries the configuration of the three levers (lines 8, 12 and 16), and if they are correctly set requests an OpenDoor event (line 20). Querying the position of lever X is performed by simultaneously requesting events LeverXInOne, LeverXInTwo and LeverXInThree. Only the "correct" event, i.e. the event that corresponds to lever X's current position, will be triggered, because the other two events will be blocked by LeverX's scenario object. Observe that this scenario has a bug: in line 16, instead of checking whether lever C is in position two, we mistakenly check if lever B is in position two. When this line in the code (line 16) is reached we already know that lever B is in position three (line 12), and so line 20 can never be reached until this bug is fixed.

And then, a few seconds later:

> **Information:** *Event OpenDoor appears to only be triggered when the shared array is in configuration LeverAInTwo, LeverBInThree, LeverCInTwo.*

Here, the development framework was able to deduce — without any information regarding the specific system being modeled — that configuration $\langle 2,3,2 \rangle$ is of special importance in the triggering of OpenDoor events! This does not indicate a potential error that the development framework found, as in the previous cases shown, but rather an *emergent property* that the framework was able to deduce — completely on its own — and which may be of interest to the developer. Such emergent properties can serve to either draw attention to bugs or reassure the developer that the model functions as intended, which was the case here. Details about how this conclusion was reached are presented in the next section. A video demonstrating the examples described in this section is available online [81].

11.3 Explaining the Framework: The Three "Sisters"

We now describe in some detail the inner workings of our wise development framework and the various components from which it is comprised. Although this framework is but a first step towards the ultimate goal described in [138, 137, 43, 82], it utilizes some powerful techniques, and building it was far from trivial. An up-to-date version of the tool, as well as video clips demonstrating its main principles, can be found online [81].

As mentioned earlier, our wise development framework is designed to accompany the development of behavioral models, as defined in Section 2.2, and in particular behavioral programs written in C++ using the BPC package [3]. The framework involves three new logical components, over and above the BPC package itself, and apart from the additional external tools we invoke, such as a model checker and an SMT solver (see Figure 11.3). We call these components *the three sisters*: *Athena*, *Regina* and *Livia*.

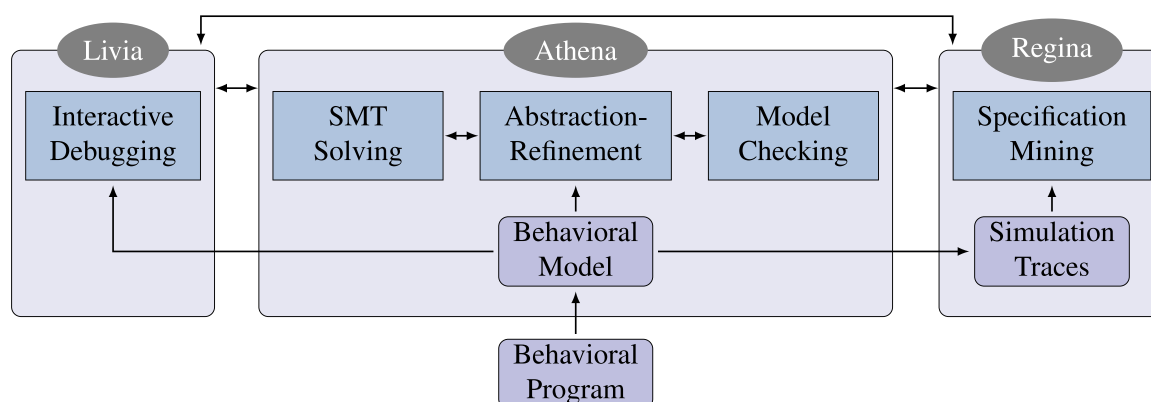


Figure 11.3: A high-level overview of the three sisters. The developer provides a behavioral program, from which Athena extracts a behavioral model. She then analyzes this model using abstraction-refinement, model checking and SMT solving. Athena also shares the behavioral model with her sisters: with Regina for the purpose of specification mining, and with Livia for interactive debugging. The three sisters also exchange information with each other — for instance, Regina may ask Athena to attempt to formally prove an emergent property that she mined.

Intuitively, each sister handles a different set of services provided by the wise development environment. *Athena*, the wise one, works proactively during development, in an off-line fashion. Her purview is the usage of formal tools to analyze scenario objects and produce logically accurate conclusions about them, which are valid for all runs. For instance, in the example discussed in Section 11.2, the conclusion that a certain scenario state could never be reached was derived by Athena, using model checking.

Regina, more regal than her sisters, also works off-line, but her purview includes semi-formal methods: using abstract models of the system, she runs multiple simulations, collecting statistical information as she goes. In what is a form of specification mining she then attempts

to reach interesting conclusions, to be presented to the modeler. Her conclusions may not be valid for all runs, but they have the advantage of reflecting numerous executions, and can thus provide valuable insights about what will happen in typical runs. Again recalling the example in Section 11.2, the discovery that OpenDoor events were related to lever configuration $\langle 2,3,2 \rangle$ was made by Regina, as a result of running multiple simulations of the system.

The last sister, *Livia*, who was not demonstrated in Section 11.2, complements the other components by providing on-line support for the developer, for debugging purposes. She can monitor the system as it runs, and help the developer recognize and comprehend unexpected behavior — also by sometimes running local simulations and tests, and by using an abstract model of the system.

The three sisters also cooperate: for instance, emergent properties recognized by Regina can be passed to Athena for formal verification, and Livia may use Athena's formal analysis tools for local analysis at runtime. Together, the three sisters are meant to accompany the programmer during development time and provide the various features which together constitute the initial wise development framework.

We now delve deeper into the technical aspects of the framework. The offline components Athena and Regina continuously run as background processes at development time. After each successful compilation of the code, these two sisters receive a fresh snapshot of the program and begin to analyze it. Next, we discuss the main steps in their analysis process, repeated after each compilation.

The first step is a key one, and is performed by Athena: she constructs an abstract, executable behavioral model of the program, to be used by all three sisters, in all their further analysis operations. Intuitively, Athena extracts from the program — given as C++ code — the underlying scenario objects, as described in Section 2.2. This technique, discussed in Chapter 9, leverages the fact that concurrent scenarios communicate only through the strict BP synchronization mechanism. Athena thus runs each scenario individually in a "sandbox", while mimicking the program's event selection mechanism, exploring the scenario's states and constructing its underlying scenario object. The resulting abstract model of the program thus completely and correctly describes all inter-scenario communication, while the rest of the information (internal scenario actions) is abstracted away, allowing the development framework to handle larger programs. Athena then shares this abstract behavioral model with Regina for the purpose of running simulations, and with Livia for the purpose of online analysis.

The next phase is also performed by Athena, and it involves partitioning the program's scenarios into logical modules according to their functionality. This clustering phase is needed to increase the tool's scalability: when trying later to check a property ϕ that does not involve program module A, the sisters will attempt to abstract away module A — reducing the total number of states to explore. We have set things up so that information regarding the scenario

grouping into modules is not provided by the programmer; rather, Athena uses a clustering algorithm (Chapter 5) to determine scenarios' correlations to events, and then groups them accordingly.

For the purpose of identifying logical modules, Athena also compares the extracted behavioral model to a predefined meta-model with known/common parallelism constructs (e.g., semaphores, shared arrays, sensors and actuators; see Chapter 8) which we have built into our tool. If it is discovered that certain scenario objects are instantiations of meta-objects that are logically connected (e.g., one scenario implements a semaphore and another scenario waits on that semaphore), they may also be grouped together into the same logical module. Recalling the example of Section 11.2, it was Athena who realized, by comparing the input model to her stored meta-model, that the lever scenarios constituted a shared ternary array.

The next step employs specification mining techniques, and is performed by Regina. She attempts to determine, by running multiple simulations on the behavioral model of the program (which was provided by Athena), a list of possible properties of the system. These are discovered by analyzing simulation traces and looking for patterns: events that always (or never) appear together, events that cause other events to occur, producer-consumer patterns, etc. Such abilities can be seen as a form of mining traces for scenario-based specifications (see, e.g., [118]). The generated properties are not guaranteed to be valid, and need to be checked — either formally, by Athena (e.g., by model checking), or statistically, by Regina (e.g., by running even more simulations of the system). If and when proven correct, and assuming that they are relevant, these emergent properties can serve as part of the official certification that the system performs as intended (one example appeared at the end of Section 11.2). However, even when the sisters guess "incorrectly", i.e., come up with properties that are later shown not to hold, this can still be quite useful, often drawing the developer's attention to bugs.

Once Regina has obtained a list of candidate properties, the next step is to attempt to prove or disprove each of them. In our experience with the tool, for a large system this list tends to contain dozens of properties, and so it is typically infeasible to model check each and every one of them and present the conclusions quickly. We mitigate this difficulty in several ways: (i) We attempt to reduce redundancy. Thus, if we have identified a class of similar emergent properties, we may start by checking just one of them and assign the remaining properties a lower priority. (ii) We employ a prioritization heuristic, aimed at checking first those properties that are likely to be more interesting to the user. For instance, if a semaphore-like construct was identified, we will prioritize the checking of a property that states that in some cases mutual exclusion may be incorrectly implemented, as this is considered a safety critical property, which may be more interesting to the user. (iii) We present any conclusion to the user as soon as it is reached, while the sisters continue to check additional properties. (iv) We leave room for manual configuration of the framework; i.e. the developers can manually prioritize the testing of certain properties if

they so desire.

This prioritization scheme yields an ordered list of properties, which are then checked in sequence. However, this list is still typically quite long, and it is desirable to dispatch properties as soon as possible, so that the results will appear in time to be relevant. To this end, we build upon a large body of existing work regarding the formal analysis of scenario-based models, including, e.g., abstraction-refinement techniques (Chapter 5), program instrumentation techniques (Chapter 4) and SMT-based (*theory-aided*) techniques (Chapters 7 and 8). Indeed, this is the main reason why we chose to implement a wise framework in the context of the scenario-based paradigm: it is sufficiently expressive for real-world systems (Chapter 10), but on the other hand is amenable to, and even facilitates, program analysis [83]. Since the ability to quickly and repeatedly analyze behavioral models is critical to our approach, this seemed like a natural fit.

From this point on, Athena and Regina will attempt to discharge as many properties as possible, and present the results to the user. By default, Athena will attempt to model check the properties in sequence, using the abstraction-refinement based model checking for scenario-based programs that we discussed in Chapter 5. Alternatively, the user may configure the framework to use other tools: explicit model checking or an SMT-based approach (also performed by Athena), or have Regina perform statistical checking. Here, statistical checking entails Regina running many simulations under various environment assumptions (fair/unfair environment, starvation, round-robin triggering of events, etc.), and repeatedly checking the property at hand. This technique is not guaranteed to be sound, of course, but it can yield interesting conclusions nonetheless, and it affords a level of assurance of the property holding, which may suffice for ones that are not safety-critical. We are currently in the process of implementing an adaptive mechanism that would attempt to run the various techniques in Athena's arsenal with a timeout value, abandoning a technique if it does not prove useful for a specific input.

The final phase of the sisters' analysis cycle is showing the user the properties that were proved or disproved. In some cases, the mined properties are irrelevant, and the user may discard them. In other cases, desirable properties are shown to hold, and the user is then reassured that the program is working as intended. The remaining cases can either be undesired properties that do hold, or "classical" bugs, where a property that the user assumed to hold is proven by Athena to be violated. In the latter case, the user can interact with the development framework, and ask for, (i) a trace log showing how the property was violated; (ii) a suggestion for a fix, in the form of a scenario that is to be added the model (along the lines of Chapter 4); or (iii) the addition of a monitor scenario, to alert the user when the property is violated at run-time (usually used for debugging purposes).

Apart from the analysis flow just described, Athena also supports some forms of automatic

optimization — e.g., identifying parts of the code that may never be reached and suggesting how to remove them, as we saw in Section 11.2.

So far we have dealt with the framework's *offline* capabilities, performed by Athena and Regina — that is, analysis performed during development, usually after compilation, but without running the actual system. In contrast, the *online* sister Livia participates in debugging the system as it runs. She connects to the system and monitors it by "pretending" to be a scenario object in the behavioral program, which constantly waits for every one of the program's events. Livia also has at her disposal the abstract model of the program produced by Athena in the first step of the analysis, and she uses it — along with the sequence of events triggered so far — to keep track of the internal states of every object in the system.

Livia's main capability is to launch bounded model checking from a given state, checking for properties at run time. For instance, the user debugging the program might believe that a corner case has been arrived at, from which the initial state can never be reached, and can have Livia investigate this. Livia will attempt to verify the property using bounded model checking. This sort of operation will typically be initiated manually by the user, but Livia also attempts to recognize problematic cases on her own — for instance, when certain objects in the system have become deadlocked or simply have not changed states in a while — and asks the user whether she should investigate. As previously mentioned, whenever a more thorough analysis is requested Livia can also pass queries over to Athena and Regina.

11.4 A Case-Study: A Cache Coherence Protocol

In order to evaluate the applicability of our wise development framework to larger systems, we used it to develop a *cache coherence* protocol. Such protocols are designed to ensure consistent shared memory access in a set of distributed processors. In order to minimize the number of read operations on the actual memory, processors cache the results of previous reads. Consistency then means that cached values stored throughout the system need to be invalidated when a processor writes a new value to the actual memory. The motivation for choosing this particular example was that cache coherence protocols are notoriously susceptible to subtle, concurrency-related bugs, making them a prime candidate to benefit from a wise development environment. The specific protocol that we implemented is a variant of the well-studied Futurebus protocol [50].

An important question that we attempted to address through the case-study was whether the notion at the core of our approach — namely, developing a non-trivial system *together* with the aid of a proactive framework — is convenient and/or useful. While this issue is highly subjective, we can report that in the systems we modeled the sisters' aid proved valuable. In particular, they typically displayed their insights about the program in a timely manner,

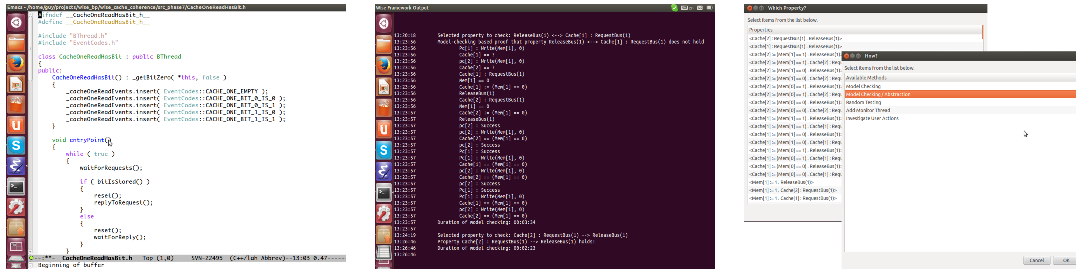


Figure 11.4: Screenshots of our wise development framework, taken during the cache coherence case-study. The left window depicts a standard editor, in which the code of the program is being written. The analysis tools are running in the background, and with every successful compilation of the code they automatically receive a fresh snapshot and analyze it. The middle window shows output from the analysis — in this case, emergent properties that were examined. One property was proved correct and another was shown not to hold (a counter-example is provided). Most of the time we had these two windows open on separate screens. The right-hand side window shows a simple GUI that we used to instruct the tools to focus on certain properties and the means they should use: explicit or abstraction based model checking, statistical testing, creating a monitor thread, etc. We used this option when additional clarification regarding certain emergent properties was required.

with results starting to flow in seconds after each compilation; and although sometimes the insights proved irrelevant, in several cases they pointed out concurrency-related bugs that we had overlooked, and which we then repaired. In other cases, the framework's conclusions served to confirm that the model was "working as intended", which was particularly reassuring, for example, after adding a new feature.

Another goal that we had was to identify a basic methodology for how modeling or programming should be conducted in such an environment. A setup that we found convenient is depicted in Figure 11.4. As for the flow of the process, we found it useful to have a quick glance at the framework's logs after each compilation to check for any critical mistakes, and to look more thoroughly at the logs after making significant changes to the code base.

We now show two examples of the usage of the wise development framework during our case-study. A more complete set of examples, as well as the entire code base, is available online [81]. In order to properly illustrate the tool's usage during development, we took snapshots of our code at significant milestones, along with the conclusions that the wise framework was able to draw from it — these are also available online. Finally, we also provide there a video clip that features the development framework in action.

Figure 11.5 depicts a list of emergent properties that the development framework produced at one point during development. Recall that unless given specific instructions by the developer, the tool begins to check these properties, one by one; the figure shows a list of properties that have been checked at that point, indicating which of them hold and which do not. The tool mines for various types of properties, two of which are depicted in the figure: implications,

```
Checking emergent properties:

ReleaseBus(1) <--> Cache[2] : RequestBus(1)
   [fails]
Cache[2] : RequestBus(1) --> ReleaseBus(1)
   [holds]
ReleaseBus(1) <--> Cache[1] : RequestBus(1)
   [fails]
Cache[1] : RequestBus(1) --> ReleaseBus(1)
   [holds]
Cache[2] := (Mem[1] == 1) --> ReleaseBus(1)
   [holds]
ReleaseBus(1) <--> Pc[1] : Success
   [fails]
Cache[2] : RequestBus(1) <--> Pc[2] : Success
   [fails]
...
```

Figure 11.5: A list of emergent properties produced and checked by the wise development framework. The tool typically does not finish checking everything on the list, and so information is displayed as soon as it is available. A counter-example is available for properties that fail to hold.

denoted $a \rightarrow b$, i.e., whenever event a occurs b also occurs a short time earlier or later, and equivalences, denoted $a \leftrightarrow b$, i.e., the implication holds in both directions.

Figure 11.6 depicts an example for which Athena's abstraction-based model checking proved especially handy, allowing her to quickly cover more properties. There, the emergent property being verified was that "cache 3 cannot acquire bus 2 repeatedly without first releasing it" — a property that describes mutual exclusion in the bus ownership. This property is an instantiation of the general pattern "consecutive a events must have b events between them". At the time this property was mined and tested, directly model checking it entailed exploring 972233 reachable states and took over 27 minutes. By using abstraction-refinement techniques, Athena was able to abstract away irrelevant parts of the code (namely code modules that only pertained to other buses). In this way, verifying the property entailed exploring just 21000 reachable states, and took less than 31 seconds. The key observation here is that this is by no means merely a standard direct usage of abstraction-refinement. The entire process — finding the emergent property, figuring out which modules are not likely to affect it so that they can be abstracted away, and then model checking the property on the abstract model — were all handled proactively and automatically by the framework. And clearly, such speedups allow the framework to cover more properties, and present them to the programmer in a timely manner.

11.5 Related Work

Work related to subject of this chapter can be viewed in two perspectives. One is over the individual capabilities of the three sisters, that is, mainly, discovering and proposing candidate

```
Checking emergent property:
  Consecutive Cache[3] : RequestBus(2) events
  must have ReleaseBus(2) events between them

Attempting abstraction-based model checking
Abstracting module 1:
  CacheOneUpdate, CacheTwoUpdate,
  CacheTwo, CacheOne,
  CacheTwoReadFetchBit, CacheOneReadFetchBit,
  CacheTwoReadHasBit, CacheTwoWriteFetchBit,
  CacheTwoWriteHasBit, PcTwoRead, PcTwoWrite,
  CacheOneReadHasBit, CacheOneWriteHasBit,
  PcOneRead, PcOneWrite, CacheOneWriteFetchBit

Abstracting module 2:
  CacheTwoInvalidate

Abstracting module 3:
  CacheOneInvalidate

Conclusion: property [holds]
```

Figure 11.6: Extracts from the logs of the wise development framework, illustrating the autonomous verification of an emergent property that has been identified. The three code modules depicted (each a set of scenario objects) are irrelevant to the property at hand, and are automatically abstracted. Other modules in the program, those that are relevant to the property at hand, are not abstracted. The property is then verified for the resulting over-approximation — leading to improved performance.

emergent properties, and then verifying or refuting these properties. The other perspective is that of the overall view of a wise development environment that accompanies the developer and automatically and proactively carries out these tasks and others, such as requirements analysis, test generation, synthesis, and more.

From the first perspective, there is a vast amount of pertinent research, and we focus here on only a few of the relevant papers. The actions performed by Regina, i.e. the dynamic discovery of candidate properties and invariants from program execution logs, is a form of *specification mining* [20]. This topic has been studied in the context of scenario-based specification in, e.g., [42, 117], and Regina uses similar techniques. For instance, she looks for emergent properties that have the *trigger and effect* structure of [117]. However, a key aspect in Regina's operation is the need to conclude the mining phase as quickly as possible, so that she can be seamlessly integrated into the development cycle. This is achieved by employing prioritization heuristics, and putting limits on the number of traces (and lengths thereof) that Regina considers. In the future we intend to enhance Regina with a mechanism similar to the one discussed in [55], where statistical criteria are used to determine when "enough" traces have been considered, hopefully boosting her performance even further.

Checking whether properties mined from traces indeed hold for the model in general brings us to the broad field of program and model verification. Many powerful and well known tools exist, such as Spin, Slam, Blast, Uppaal, Java Pathfinder, Astrée, ESC/Java and others, and

they utilize many forms of explicit and symbolic model checking, static analysis, deductive reasoning, and SAT and SMT solving (see [15] for a brief survey of the application of such methods in practice). In our framework these tasks are handled by Athena, and she uses tools specifically optimized for behavioral models [86, 7, 8].

As to the second perspective, successful attempts at automatic property discovery and subsequent verification appear, e.g., in [131, 151]. There, the Daikon tool is used to dynamically detect candidate program invariants which are then used to either annotate or instrument the program. In [131] these guide ESC/Java in verifying the properties, and in [151] they help guide symbolic execution in the discovery of additional or refined invariants. The motivation and approach of Daikon are very close to ours, but we aim at constructing a fully integrated, proactive and interactive environment, built upon the highly incremental paradigm of behavioral modeling.

Providing an interactive analysis framework that is tightly integrated into the development cycle/environment has become quite widespread in the industry over recent years. Some noticeable examples are Google's *Tricorder* [139], Facebook's *Infer* [56] and VMWare's *Review Bot* [24] tools. These tools use static analysis to automate the checking for violations of coding standards and for common defect patterns. Lessons learned from these projects indicate that, in order to be successfully accepted by programmers, an integrated analysis framework should have the following properties: (i) it needs to seamlessly integrate into the workflow of developers; (ii) it must produce results quickly; and (iii) it has to perform its analysis in a modular manner, so that it can scale reasonably well to large projects. The design of our framework is indeed aimed at achieving these properties. In particular, for the modular analysis part, Athena attempts to leverage the special properties of scenario-based models and reason about individual objects. In [4], it is shown that objects in behavioral models often have very small state spaces; and this allows Athena to effectively compare these objects to her stored meta-model and identify object patterns that can later be used for analysis.

11.6 Conclusion

In this chapter we contribute to the effort of simplifying and accelerating development of robust reactive systems, by proposing a development framework along the lines raised in e.g., [43, 82]. In a nutshell, the idea is to start with a modeling/programming formalism that is expressive, modular and relatively simple, and integrate quick, continuous, and easy-to-use analysis into the development process. This entails extending and adjusting existing analysis techniques in order to render them more interactive and proactive.

Our development framework is currently comprised of three main elements: specification mining and initial semi-formal analysis for generating candidate system properties, abstraction-

assisted formal analysis for verification of detected properties, and run-time debugging. When integrated into the development cycle, these elements can often draw developers' attention to subtle bugs that could otherwise be missed. We carried out initial evaluation of the framework by iteratively developing a cache coherence protocol, and saw that it was successful in discovering and reporting bugs.

In the future we plan to carry out a more extensive, empirical comparison between our development framework and related tools, such as Tricorder [139] and Infer [56]. We also plan to enhance Regina's specification-mining capabilities with learning techniques [20], allowing her to learn over time which emergent properties are most valuable to programmers and should be checked first.

While our work so far is but an early step towards the vision of the computer acting as a wise, fully-fledged proactive member of the development team, we hope that it contributes to demonstrating both the viability and the potential value of this direction.

Part IV

Conclusion

Chapter 12

Discussion and Next Steps

12.1 Discussion

Due to the pervasiveness of software in today's world, the reliable design and implementation of reactive systems is a highly important task. Unfortunately, the inherent intricacy of complex concurrent software makes this task very difficult to accomplish manually. A great deal of effort has been put into devising automated tools to mitigate this difficulty by assisting in software development and verification — but so far, these have proven insufficient.

In this thesis we attempted to point out a connection, which, we feel, could help improve the scalability of formal analysis techniques, making them more suited to real-world systems. Specifically, we began to characterize the deep connection between the programming model, or language, that the developers choose to use, and the difficulty of analyzing the constructed software. Our goal was to emphasize that, when choosing a programming model for the development of a project, it is not enough to pick just the model or language that is most expressive, powerful or popular; rather, we must take into account that this choice can have an important effect on the developed software's reliability. Thus, one must take into account the analysis tasks that would have to be performed later on, and attempt to pick a programming model that will facilitate them.

Our studies indicated a certain trade-off between a model's likely appeal to programmers and its ease of analysis. Specifically, programmers are likely to prefer stronger models, as they afford a greater degree of freedom — but this freedom can be a hindrance to analysis. A balance does appear to exist, but the choice must be made intelligently, and a great deal of research is still required in order to fully characterize the costs and benefits of various programming idioms. Still, we hope that the results presented here will serve as a good basis for this endeavor.

In this thesis we focused on the behavioral programming approach, and its underlying $\mathcal{RWB}$ model. While the ideas behind our research are compatible with a verity of additional programming languages, this model seemed like a good place to start and address the afore-

mentioned trade-off — due to BP's relative simplicity on the one hand, and its appeal and intuitively to programmers on the other. Also, the $\mathcal{RWB}$ idioms appear, sometimes in related form, in a variety of programming models. Thus, we hope, some of our conclusions could be adapted and carried over to those models as well.

Part II of our thesis was dedicated to studying the benefits of the $\mathcal{RWB}$ model and its constituent individual idioms from an analysis perspective. We were able to demonstrate the following connections:

- **Program Repair.** In Chapter 4 we introduced a program repair technique that is automatic and non-intrusive, i.e. is performed strictly by adding code to existing software. Non-intrusiveness is a desirable property when attempting to fix legacy code, whose authors may be unavailable. The blocking idiom of the $\mathcal{RWB}$ model proved to be a key factor in the repair process: in the case of safety violations it allowed us to block harmful event sequences, and in the case of liveness violations it allowed us to delicately inject fairness when the system's execution was going in the wrong direction. We feel that this use-case demonstrates our approach nicely: if software being developed is likely to require repeated repairs over a long period of time, including the blocking idiom in the computational model may be advisable.

- **Abstraction.** In Chapter 5 we went a step further, and showed how the properties of the $\mathcal{RWB}$ model could be leveraged in order to greatly increase the scalability of the repair algorithm from Chapter 4. Specifically, we showed how $\mathcal{RWB}$ threads can be easily analyzed and grouped together into modules — the irrelevant of which can then be abstracted away, allowing for a more effective repair procedure. This result served to show that simple programming models, such as $\mathcal{RWB}$ and variants thereof, can create a powerful synergy with existing advanced analysis techniques. Thus, our research direction can be regarded as orthogonal to traditional efforts on improving the scalability of analysis tools.

- **Succinctness.** In Chapter 6 we took a more rigorous view, and attempted to quantify mathematically the advantages that the $\mathcal{RWB}$ idioms seemed to afford. Specifically, we were able to characterize cases in which including each of these idioms in the programming model could exponentially reduce the size of the resulting program. This exponential decrease in size can lead to a dramatic increase in the performance of analysis tools, such as compositional model checkers.

- **Compositional Verification.** Having shown the potential that models such as $\mathcal{RWB}$ possess with regard to compositional verification, we then proceeded to demonstrate this potential empirically. We developed two techniques, one semi-automatic (Chapter 7)

and one fully automatic (Chapter 8), in which SMT solvers were used in proving system correctness in a compositional fashion. Specifically, we utilized the fact that the $\mathcal{RWB}$ threads in our input programs were very succinct in order to check and formulate their individual properties.

In our opinion, Part II adequately demonstrates the benevolent effect that concurrency idioms can have on formal analysis. This addresses the first part of the trade-off that we were studying. We then continued, in Part III, to demonstrate that such simple models are also sufficiently powerful and appealing from a software engineering point of view:

- **Distributed *RWB* Programs.** The tight synchronization of threads in the $\mathcal{RWB}$ model was a key aspect in facilitating the analysis tasks that we studied in the previous chapters. Thus, it was important to show that this high degree of synchronization does not come at the expense of the model's applicability to distributed frameworks. In Chapter 9 we presented a mechanism for running $\mathcal{RWB}$ programs in a distributed manner, while preserving their semantics. Thus, an $\mathcal{RWB}$ program can be analyzed as if it were synchronous, and the results would still apply if it is later run in a distributed manner.

- **Scaling-Up *RWB*.** In Chapter 10 we set out to check whether the $\mathcal{RWB}$ model was indeed sufficient for programming large software systems. To address this we implemented a web-server using the $\mathcal{RWB}$ idioms. To the best of our knowledge, this is the largest piece of software developed with the $\mathcal{RWB}$ framework heretofore; and our experience indicated that, with a few small extensions, $\mathcal{RWB}$ was indeed up to the task. This empirical result was important in establishing the fact that models that satisfy both sides of the trade-off that we were studying do exist.

- **Wise Computing and *RWB* Programs.** In Chapter 11 we attempted to bring both parts of our thesis together, and implement a framework in which a programmer could effectively tap the analysis advantages that $\mathcal{RWB}$ provides. Specifically, we followed the spirit of the Wise Computing vision [82], and created a basic *wise* framework — i.e., a framework that works with the developer interactively and proactively in attempting to create reliable software. To this end we harnessed most of the analysis tools developed in Part II and integrated them into a single development environment.

12.2 Next Steps

We feel that our results serve as strong indication of the validity and usefulness of the research direction we have taken. However, a great deal of work remains to be done in order to truly

understand the subtle connections between programming models and program analysis. In general, we propose (and hope to eventually pursue ourselves...) the following directions:

1. **Going beyond *RWB*.** We feel that our choice of $\mathcal{RWB}$ as an initial model to study was justified. However, having gained some initial insight into the nature of the problem, it now seems advisable to expand the work beyond these initial idioms. Other idioms that come to mind as worthwhile to focus on are various synchronization and voting idioms, pairwise or n-wise message passing and hierarchical components.

2. **Applying the technique to larger case-studies.** In Chapter 10 we created a large web-server application using $\mathcal{RWB}$, which was used to evaluate some of our analysis techniques (Chapters 5 and 8). This process proved highly useful and, given more time, we would carry out similar evaluations with even larger pieces of software.

3. **The Wise Computing effort.** The Wise Computing vision [82] has, of course, many merits in its own right; however, we consider it also as a means for making advanced analysis tools, such as those discussed in Part II, easily accessible to programmers. In general, we believe that the selected programming model will have a similar effect on the wise computing environment as it does on other analysis algorithms, and so the two approaches — our approach and that of wise computing — could prove mutually beneficial.

List of Abbreviations

- RWB: the request-wait-block model
- BP: behavioral programming
- B-thread: behavior thread
- ESM: event selection mechanism
- BPJ: a behavioral programming framework in Java
- BPC: a behavioral programming framework in C++
- SMT: satisfiability modulo theories
- TCP: transition control protocol
- HTTP: hypertext transfer protocol
- CEGAR: counter-example guided abstraction refinement
- POSIX: portable operating system interface
- B-node: behavioral node

Statement Regarding Collaboration

I hereby state that the work presented in this thesis summarizes my independent research throughout my PhD. Most of this work was conducted by me and supervised by Prof. Harel. In addition, for the work discussed in Chapters 4, 6, 8 and 11 I benefitted from additional guidance by Dr. Assaf Marron and Dr. Rami Marelly from Prof. Harel's research group, Dr. Gera Weiss from Ben Gurion University, and Prof. Clark Barrett from New York University. The work described in Chapter 7 was conducted in close collaboration with Dr. Marron and Dr. Weiss, and my contributions were to the formalization of the approach described therein and also to its evaluation. The work described in Chapter 9 appeared in a joint paper with several co-authors; however, the parts of that work described in this thesis were conducted primarily by myself.

List of Publications Included in the Thesis

[1] D. Harel, A. Kantor, G. Katz, A. Marron, L. Mizrahi, and G. Weiss. On Composing and Proving the Correctness of Reactive Behavior. In *Proc. 13th Int. Conf. on Embedded Software (EMSOFT)*, pages 1–10, 2013.

[2] D. Harel, A. Kantor, G. Katz, A. Marron, G. Weiss, and G. Wiener. Towards Behavioral Programming in Distributed Architectures. *Science of Computer Programming*, 98(2):233–267, 2015.

[3] D. Harel and G. Katz. Scaling-Up Behavioral Programming: Steps from Basic Principles to Application Architectures. In *Proc. 4th Int. Workshop on Programming based on Actors, Agents, and Decentralized Control (AGERE!)*, pages 95–108, 2014.

[4] D. Harel, G. Katz, R. Lampert, A. Marron, and G. Weiss. On the Succinctness of Idioms for Concurrent Programming. In *Proc. 26th Int. Conf. on Concurrency Theory (CONCUR)*, pages 85–99, 2015.

[5] D. Harel, G. Katz, R. Marelly, and A. Marron. An Initial Wise Development Environment for Behavioral Models. In *Proc. 4th Int. Conf. on Model-Driven Engineering and Software Development (MODELSWARD)*, 2016. To appear.

[6] D. Harel, G. Katz, A. Marron, and G. Weiss. Non-Intrusive Repair of Safety and Liveness Violations in Reactive Programs. *Transactions on Computational Collective Intelligence (TCCI)*, 16:1–33, 2014.

[7] G. Katz. On Module-Based Abstraction and Repair of Behavioral Programs. In *Proc. 19th Int. Conf. on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, pages 518–535, 2013.

[8] G. Katz, C. Barrett, and D. Harel. Theory-Aided Model Checking of Concurrent Transition Systems. In *Proc. 15th Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 81–88, 2015.

References

[9] BPC: Behavioral Programming in $C++$. `http://www.wisdom.weizmann.ac.il/~bprogram/bpc/`.

[10] Case-study: a BPC web-server. `http://www.wisdom.weizmann.ac.il/~bprogram/bpc/webserver.zip`.

[11] The UML Standard. `http://www.uml.org/`.

[12] G. Alexandron, M. Armoni, M. Gordon, and D. Harel. Scenario-Based Programming: Reducing the Cognitive Load, Fostering Abstract Thinking. In *Proc. 36th Int. Conf. on Software Engineering (ICSE)*, pages 311–320, 2014.

[13] R. Alur, R. Brayton, T. Henzinger, S. Qadeer, and S. Rajamani. Partial-Order Reduction in Symbolic State Space Exploration. In *Proc. 9th. Int. Conf. on Computer Aided Verification (CAV)*, pages 340–351, 1997.

[14] R. Alur and D. L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126:183–235, 1994.

[15] R. Alur, T. A. Henzinger, and M. Y. Vardi. Theory in Practice for System Design and Verification. *ACM Siglog News*, 2(1):46–51, 2015.

[16] R. Alur, G. Holzmann, and D. Peled. An Analyzer for Message Sequence Charts. *Software Concepts and Tools*, 17(2):70–77, 1996.

[17] R. Alur, P. Madhusudan, and W. Nam. Symbolic compositional verification by learning assumptions. In *Proc. 17th Int. Conf. on Computer Aided Verification (CAV)*, pages 548–562, 2005.

[18] B. Aminof, T. Ball, and O. Kupferman. Reasoning about Systems with Transition Fairness. In *Proc. 11th Int. Conf. on Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, pages 194–208, 2004.

[19] N. Amla and K. L. McMillan. Combining Abstraction Refinement and SAT-Based Model Checking. In *Proc. 13th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 405–419, 2007.

[20] G. Ammons, R. Bodik, and J. Larus. Mining Specifications. *ACM Sigplan Notices*, 37(1):4–16, 2002.

[21] A. Arcuri and X. Yao. A Novel Co-evolutionary Approach to Automatic Software Bug Fixing. In *Proc. 10th Congress on Evolutionary Computation (CEC)*, pages 162–168, 2008.

[22] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in ERLANG*. Prentice Hall, 1993.

[23] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.

[24] V. Balachandran. Reducing Human Effort and Improving Quality in Peer Code Reviews using Automatic Static Analysis and Reviewer Recommendation. In *Proc. 35th Int. Conf. on Software Engineering (ICSE)*, pages 931–940, 2013.

[25] T. Ball and S. K. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. In *Proc. 8th Int. Workshop on Model Checking of Software (SPIN)*, pages 103–122, 2001.

[26] C. Barret, C. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *Proc. 23rd Int. Conf. on Computer Aided Verification (CAV)*, pages 171–177, 2011.

[27] C. Barrett, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Splitting On Demand in SAT Modulo Theories. In *Proc. 13th Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, pages 512–526, 2006.

[28] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli. Satisfiability Modulo Theories. *Handbook of Satisfiability*, 185:825–885, 2009.

[29] A. Basu, M. Bozga, and J. Sifakis. Modeling Heterogeneous Real-time Systems in BIP. In *Proc. 4th IEEE Int. Conf. on Software Engineering and Formal Methods (SEFM)*, pages 3–12, 2006.

[30] J. Berdine, N. Bjørner, S. Ishtiaq, J. E. Kriener, and C. M. Wintersteiger. Resourceful Reachability as HORN-LA. In *Proc. 19th Int. Conf. on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, pages 137–146, 2013.

[31] G. Berry and G. Gonthier. The Esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[32] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. Bounded Model Checking. *Advances in Computers*, 58:117–148, 2003.

[33] J. Birget. Two-Way Automata and Length-Preserving Homomorphisms. *Mathematical Systems Theory*, 29(3):191–226, 1996.

[34] P. Bjesse and K. Claessen. SAT-Based Verification without State Space Traversal. In *Proc. 3rd Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 372–389, 2000.

[35] S. Bliudze and J. Sifakis. A Notion of Glue Expressiveness for Component-Based Systems. In *Proc. 19th Int. Conf. on Concurrency Theory (CONCUR)*, pages 508–522, 2008.

[36] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Saar. Synthesis of Reactive(1) Designs. *Journal of Computer and System Sciences*, 78(3):911–938, 2012.

[37] G. Booch, J. Rumbaugh, and I. Jacobson. Unified Modeling Language for Object-Oriented Development (Version 0.91 Addendum). RATIONAL Software Corporation, 1996.

[38] R. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. on Computers*, 100(8):677–691, 1986.

[39] R. Bryant, S. Lahiri, and S. Seshia. Modeling and Verifying Systems using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions. *Proc. 14th Int. Conf. on Computer Aided Verification (CAV)*, pages 78–92, 2002.

[40] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proc. 5th Symp. on Logic in Computer Science (LICS)*, pages 428–439, 1990.

[41] W. Cai, F. Lee, and L. Chen. An Auto-Adaptive Dead Reckoning Algorithm for Distributed Interactive Simulation. In *Proc. 13th IEEE. Workshop on Parallel and Distributed Simulation (PADS)*, pages 82–89, 1999.

[42] F. Cantal de Sousa, N. Mendonca, S. Uchitel, and J. Kramer. Detecting Implied Scenarios from Execution Traces. In *Proc. 14th Working Conf. on Reverse Engineering (WCRE)*, pages 50–59, 2007.

[43] V. Cerf. A Long Way to Have Come and Still to Go. *Communications of the ACM*, 1(58):7–7, 2014.

[44] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular Verification of Software Components in C. *IEEE Transactions on Software Engineering*, pages 385–395, 2004.

[45] S. Chaki, A. Gurfinkel, S. Kong, and O. Strichman. Compositional Sequentialization of Periodic Programs. In *Proc. 14th Int. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI)*, pages 536–554, 2013.

[46] A. Chandra, D. Kozen, and L. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, 1981.

[47] K. Chatterjee, T. A. Henzinger, and B. Jobstmann. Environment Assumptions for Synthesis. In *19th Int. Conf. on Concurrency Theory (CONCUR)*, pages 147–161, 2008.

[48] A. Cimatti and A. Griggio. Software Model Checking via IC3. *Proc. 24th Int. Conf. on Computer Aided Verification (CAV)*, pages 277–293, 2012.

[49] E. Clarke, A. Groce, J. Ouaknine, O. Strichman, and K. Yorav. Efficient Verification of Sequential and Concurrent C Programs. *Formal Methods in System Design*, 25(2-3):129–166, 2004.

[50] E. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. Long, K. McMillan, and L. Ness. Verification of the Futurebus+ Cache Coherence Protocol. *Formal Methods in System Design*, 6(2):217–232, 1995.

[51] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *Proc. 12th Int. Conf. on Computer Aided Verification (CAV)*, pages 154–169, 2000.

[52] E. Clarke, O. Grumberg, and D. E. Long. Model Checking and Abstraction. In *Proc. 19th. Symp. on Principles of Programming Languages (POPL)*, pages 343–354, 1992.

[53] E. Clarke, D. Long, and K. McMillan. Compositional Model Checking. In *Proc. 4th IEEE Symp. on Logic in Computer Science (LICS)*, pages 353–362, 1989.

[54] J. Cobleigh, G. Avrunin, and L. Clarke. Breaking Up is Hard to do: an Investigation of Decomposition for Assume-Guarantee Reasoning. In *Proc. Int. Symp. on Software Testing and Analysis (ISSTA)*, pages 97–108, 2006.

[55] H. Cohen and S. Maoz. Have We Seen Enough Traces? In *Proc. 30rd Int. Conf. on Automated Software Engineering (ASE)*, 2015. To appear.

[56] C. Cristiano, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez. Moving Fast with Software Verification. In *Proc. 7th. Int. Conf. on NASA Formal Methods (NFM)*, pages 3–11, 2015.

[57] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *J. on Formal Methods in System Design*, 19(1):45–80, 2001.

[58] L. De Alfaro and T. Henzinger. Interface automata. *Software Engineering Notes*, 26(5):109–120, 2001.

[59] L. De Alfaro and P. Roy. Solving Games via Three-Valued Abstraction Refinement. In *Proc. 18th Int. Conf. on Concurrency Theory (CONCUR)*, pages 74–89, 2007.

[60] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proc. 14th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, 2008.

[61] L. De Moura and N. Bjørner. Satisfiability Modulo Theories: Introduction and Applications. *Communications of the ACM*, 54(9):69–77, 2011.

[62] L. De Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari. SAL 2. *Proc. 16th Int. Conf. on Computer Aided Verification (CAV)*, pages 496–500, 2004.

[63] E. Dijkstra. Hierarchical Ordering of Sequential Processes. *Acta Inf.*, 1:115–138, 1971.

[64] C. Disenfeld and S. Katz. A Closer Look at Aspect Interference and Cooperation. In *Proc. 11th Int. Conf. on Aspect-Oriented Software Development (AOSD)*, pages 107–118, 2012.

[65] R. Dong, J. Faber, Z. Liu, J. Srba, N. Zhan, and J. Zhu. Unblockable Compositions of Software Components. In *Proc. 15th Symp. on Component Based Software Engineering (CBSE)*, pages 103–108, 2012.

[66] D. Drusinsky and D. Harel. On the Power of Bounded Concurrency I: Finite Automata. *Journal of the ACM*, 41:517–539, 1994.

[67] M. Dwyer, J. Hatcliff, R. Robby, C. Pasareanu, and W. Visser. Formal Software Analysis Emerging Trends in Software Model Checking. In *Proc. 29th Int. Conf. on Software Engineering (ICSE)*, pages 120–136, 2007.

[68] N. Eitan and D. Harel. Adaptive Behavioral Programming. In *23rd Int. Conf. on Tools with Artificial Intelligence (ICTAI)*, pages 685–692, 2011.

[69] P. Eugster, P. Felber, R. Guerraoui, and A. Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[70] C. Flanagan and S. Qadeer. Thread-Modular Model Checking. In *Model Checking Software*, pages 213–224. Springer, 2003.

[71] R. Fujimoto. Parallel and Distributed Simulation. In *Proc. Winter Simulation Conf. (WSC)*, pages 118–125, 1995.

[72] S. Ghilardi and S. Ranise. MCMT: A Model Checker Modulo Theories. In *Proc. 5th Int. Joint Conf. on Automated Reasoning (IJCAR)*, pages 22–29, 2012.

[73] D. Giannakopoulou, C. S. Pasareanu, and J. M. Cobleigh. Assume-Guarantee Verification of Source Code with Design-Level Assumptions. In *Proc. 26th Int. Conf. on Software Engineering (ICSE)*, pages 211–220, 2004.

[74] M. Gordon, A. Marron, and O. Meerbaum-Salant. Spaghetti for the Main Course? Observations on the Naturalness of Scenario-Based Programming. In *Proc. 17th Conf. on Innovation and Technology in Computer Science Education (ITICSE)*, pages 198–203, 2012.

[75] A. Griesmayer, S. Staber, and R. Bloem. Automated Fault Localization for C Programs. In *Proc. 18th Int. Conf. on Computer Aided Verification (CAV)*, pages 82–99, 2006.

[76] O. Grumberg and D. Long. Model Checking and Modular Verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, 1994.

[77] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The Synchronous Data-Flow Programming Language LUSTRE. *Proc. of the IEEE*, 79(9):1305–1320, 1991.

[78] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[79] D. Harel, A. Kantor, and G. Katz. Relaxing Synchronization Constraints in Behavioral Programs. In *Proc. 19th Int. Conf. on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, pages 355–372, 2013.

[80] D. Harel, A. Kantor, G. Katz, A. Marron, L. Mizrahi, and G. Weiss. On Composing and Proving the Correctness of Reactive Behavior: Supplementary Material. `http://www.wisdom.weizmann.ac.il/~bprogram/emsoft13/`.

[81] D. Harel, G. Katz, R. Marelly, and A. Marron. An Initial Wise Development Environment for Behavioral Models: Supplementary Material. `https://sites.google.com/site/guykatzhomepage/wc/`.

[82] D. Harel, G. Katz, R. Marelly, and A. Marron. Wise Computing: Towards Endowing System Development with True Wisdom, 2015. Technical Report. `http://arxiv.org/abs/1501.05924`.

[83] D. Harel, G. Katz, A. Marron, and G. Weiss. The Effect of Concurrent Programming Idioms on Verification. In *Proc. 3rd Int. Conf. on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 363–369, 2015.

[84] D. Harel, H. Kugler, R. Marelly, and A. Pnueli. Smart Play-Out of Behavioral Requirements. In *Proc. 4th Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 378–398, 2002.

[85] D. Harel, H. Kugler, and A. Pnueli. Smart Play-Out Extended: Time and Forbidden Elements. In *Proc. 4th Int. Conf. on Quality Software (QSIC)*, pages 2–10, 2004.

[86] D. Harel, R. Lampert, A. Marron, and G. Weiss. Model-Checking Behavioral Programs. In *Proc. 11th Int. Conf. on Embedded Software (EMSOFT)*, pages 279–288, 2011.

[87] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer, 2003.

[88] D. Harel and R. Marelly. Playing with Time: On the Specification and Execution of Time-Enriched LSCs. In *Proc. 10th Int. Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2006.

[89] D. Harel, A. Marron, and G. Weiss. Programming Coordinated Scenarios in Java. In *Proc. 24th European Conf. on Object-Oriented Programming (ECOOP)*, pages 250–274, 2010.

[90] D. Harel, A. Marron, and G. Weiss. Behavioral Programming. *Communications of the ACM*, 55(7):90–100, 2012.

[91] D. Harel, A. Marron, G. Weiss, and G. Wiener. Behavioral Programming, Decentralized Control, and Multiple Time Scales. In *Proc. 1st SPLASH Workshop on Programming Systems, Languages, and Applications based on Agents, Actors, and Decentralized Control (AGERE!)*, pages 171–182, 2011.

[92] D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts: The STATEMATE Approach*. McGraw-Hill, 1998.

[93] D. Harel and I. Segall. Planned and Traversable Play-Out: A Flexible Method for Executing Scenario-Based Programs. In *Proc. 13th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 485–499, 2007.

[94] J. Hatcliff, G. Leavens, K. Leino, P. Müller, and M. Parkinson. Behavioral Interface Specification Languages. *Computing Surveys (CSUR)*, 44(3):16, 2012.

[95] T. Henzinger, R. Jhala, and R. Majumdar. Counterexample-Guided Control. In *Proc. 30th Int. Conf. on Automata, Languages and Programming (ICALP)*, pages 886–902, 2003.

[96] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software Verification with BLAST. In *Proc. 10th Int. Workshop on Model Checking of Software (SPIN)*, pages 235–339, 2003.

[97] T. Henzinger, S. Qadeer, and S. Rajamani. You Assume, We Guarantee: Methodology and Case Studies. In *Proc. 10th Int. Conf. on Computer Aided Verification (CAV)*, pages 440–451, 1998.

[98] T. Henzinger and J. Sifakis. The Embedded Systems Design Challenge. In *Proc. 14th Int. Symp. on Formal Methods (FM)*, pages 1–15, 2006.

[99] T. Hirst and D. Harel. On the Power of Bounded Concurrency II: Pushdown Automata. *Journal of the ACM*, 41:540–559, 1994.

[100] M. Holzer and M. Kutrib. Descriptional and Computational Complexity of Finite Automata - a Survey. *Information and Computation*, 209(3):456–470, 2011.

[101] J. Hromkovič and G. Schnitger. Lower Bounds on the Size of Sweeping Automata. *Journal of Automata, Languages and Combinatorics*, 14(1):23–13, 2009.

[102] ITU. ITU-TS Recommendation Z.120: Message Sequence Chart (MSC), 1996.

[103] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit. Automated Atomicity-Violation Fixing. In *Proc. 32th Conf. on Programming Language Designand Implementation (PLDI)*, pages 389–400, 2011.

[104] B. Jobstmann, A. Griesmayer, and R. Bloem. Program Repair as a Game. In *Proc. 17th Int. Conf. on Computer Aided Verification (CAV)*, pages 226–238, 2005.

[105] C. Jones. Tentative Steps toward a Development Method for Interfering Programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.

[106] T. Kahsai, Y. Ge, and C. Tinelli. Instantiation-Based Invariant Discovery. In *Proc. 3rd. Int. Conf. on NASA Formal Methods (NFM)*, pages 192–206, 2011.

[107] G. Katz. On Module-Based Abstraction and Repair of Behavioral Programs: Supplementary Material. `http://www.wisdom.weizmann.ac.il/~bprogram/bpc/module_based_abstraction/`.

[108] G. Katz, C. Barrett, and D. Harel. Theory-Aided Model Checking of Concurrent Transition Systems: Supplementary Material. `https://sites.google.com/site/guykatzhomepage/fmcad15/`.

[109] R. Könighofer and R. Bloem. Repair with On-The-Fly Program Analysis. In *Proc. 8th Haifa Verification Conference (HVC)*, pages 56–71, 2012.

[110] O. Kupferman, A. Ta-Shma, and M. Vardi. *Counting With Automata*, 1999. Technical Report.

[111] C. Le Goues, S. Forrest, and W. Weimer. Current Challenges in Automatic Software Repair. *Software Quality Journal*, 21(3):421–443, 2013.

[112] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming Real-Time Applications with Signal. *Proceedings of the IEEE*, 79(9):1321–1336, 1991.

[113] E. Leiss. Succinct Representation of Regular Languages by Boolean Automata. *Theoretical Computer Science*, 13:323–330, 1981.

[114] J. Leung and M. Merrill. A Note on Preemptive Scheduling of Periodic, Real-Time Tasks. *Information Processing Letters*, 11(3):115–118, 1980.

[115] T. Liang, A. Reynolds, C. Tinelli, C. Barrett, and M. Deters. A DPLL(T) Theory Solver for a Theory of Strings and Regular Expressions. *Proc. 26th Int. Conf. on Computer Aided Verification (CAV)*, pages 646–662, 2014.

[116] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, 1973.

[117] D. Lo and S. Maoz. Mining Scenario-based Triggers and Effects. In *Proc. 23rd Int. Conf. on Automated Software Engineering (ASE)*, pages 109–118, 2008.

[118] D. Lo, S. Maoz, and S.-C. Khoo. Mining Modal Scenario-Based Specifications from Execution Traces of Reactive Systems. In *Proc. 22nd Int. Conf. on Automated Software Engineering (ASE)*, pages 465–468, 2007.

[119] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from Mistakes: a Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proc. 13th. Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 329–339, 2008.

[120] N. Lynch and M. Tuttle. Hierarchical Correctness Proofs for Distributed Algorithms. In *Proc. 6th Symp. on Principles of Distributed Computing*, pages 137–151, 1987.

[121] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive Systems: Specification*. Springer-Verlag, 1992.

[122] S. Maoz and Y. Sa'ar. Counter Play-Out: Executing Unrealizable Scenario-Based Specifications. In *Proc. 35th Int. Conf. on Software Engineering (ICSE)*, pages 242–251, 2013.

[123] K. McMillan. A Methodology for Hardware Verification using Compositional Model Checking. *Science of Computer Programming*, 37(1):279–309, 2000.

[124] K. McMillan. Lazy Annotation Revisited. In *Proc. 26th Int. Conf. on Computer Aided Verification (CAV)*, pages 243–259, 2014.

[125] K. L. McMillan and L. D. Zuck. Abstract Counterexamples for Non-disjunctive Abstractions. In *Proc. 3rd Int. Workshop on Reachability Problems (RP)*, pages 176–188, 2009.

[126] N. Meng-Siew. Reasoning with Timing Constraints in Message Sequence Charts. Master's thesis, University of Stirling, 1993.

[127] A. Meyer and M. Fischer. Economy of Description by Automata, Grammars, and Formal Systems. In *Proc. 12th Sym. on Switching and Automata Theory (SWAT)*, pages 188–191, 1971.

[128] J. Misra and K. Chandy. Proofs of Networks of Processes. *IEEE Trans. on Software Eng.*, SE-7(4):417–426, 1981.

[129] J. Musa. *Software Reliability Engineered Testing*. McGraw-Hill, 1998.

[130] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: From an Abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006.

[131] J. Nimmer and M. Ernst. Static Verification of Dynamically Detected Program Invariants: Integrating Daikon and ESC/Java. *Electronic Notes in Theoretical Computer Science*, 55(2):255–276, 2001.

[132] D. Parnas. On the Criteria to be used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[133] A. Pnueli. In Transition from Global to Modular Temporal Reasoning about Programs. In *Logics and Models of Concurrent Systems*, pages 123–144. Springer-Verlag, 1985.

[134] A. Pnueli and R. Rosner. On the Synthesis of a Reactive Module. In *Proc. 16th Symp. on Principles of Programming Languages (POPL)*, pages 179–190, 1989.

[135] M. Rabin and D. Scott. Finite Automata and Their Decision Problem. *IBM Journal of Research and Development*, 3(2):114–125, 1959.

[136] P. Ramadge and W. Wonham. Supervisory Control of a Class of Discrete Event Processes. *SIAM J. on Control and Optimization*, 25(1):206–230, 1987.

[137] H. Reubenstein and R. Waters. The Requirements Apprentice: Automated Assistance for Requirements Acquisition. *IEEE Transactions on Software Engineering*, 17(3):226–240, 1991.

[138] C. Rich and R. Waters. The Programmer's Apprentice: A Research Overview. *Computer*, 21(11):10–25, 1988.

[139] C. Sadowski, J. van Gogh, C. Jaspan, E. Söederberg, and C. Winter. Tricorder: Building a Program Analysis Ecosystem. In *Proc. 37th Int. Conf. on Software Engineering (ICSE)*, 2015.

[140] S. Safra and M. Vardi. On ω-Automata and Temporal Logic. In *Proc. 21st Sym. on Theory of Computing (STOC)*, pages 127–137, 1989.

[141] W. Sakoda and M. Sipser. Nondeterminism and the Size of Two Way Finite Automata. In *Proc. 10th Sym. on Theory of Computing (STOC)*, pages 275–286, 1978.

[142] S. Staber, B. Jobstmann, and R. Bloem. Diagnosis is Repair. In *Proc. 16th Int. Workshop on Principles of Diagnosis (DX)*, pages 169–174, 2005.

[143] S. Staber, B. Jobstmann, and R. Bloem. Finding and Fixing Faults. *Correct Hardware Design and Verification Methods*, 3275:35–49, 2005.

[144] The Behavioral Programming Webpage. `http://www.b-prog.org/`.

[145] Y. Tsay, Y. Chen, M. Tsai, K. Wu, and W. Chan. GOAL: A Graphical Tool for Manipulating Büchi Automata and Temporal Formulae. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 466–471. Springer, 2007.

[146] A. Valmari. The State Explosion Problem. In *Lectures on Petri Nets I: Basic models*, pages 429–528. Springer, 1998.

[147] M. Veanes and N. Bjørner. Symbolic Automata: The Toolkit. In *Proc. 18th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 472–477, 2012.

[148] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model Checking Programs. *Automated Software Engineering*, 10(2):203–232, 2003.

[149] E. Volk. CxxTest: A Unit Testing Framework for $C++$. http://cxxtest.com/.

[150] W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen. Automatic Program Repair with Evolutionary Computation. *Communications of the ACM*, 53:109–116, 2010.

[151] L. Zhang, G. Yang, N. Rungta, S. Person, and S. Khurshid. Feedback-Driven Dynamic Invariant Discovery. In *Proc. Int. Symp. on Software Testing and Analysis (ISSTA)*, pages 362–372, 2014.