Modeling Systems from Logs of their Behavior

Ivan Beschastnikh

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2013

Reading Committee:

Michael D. Ernst, Chair

Arvind Krishnamurthy, Chair

Thomas E. Anderson

Program Authorized to Offer Degree:
Computer Science and Engineering

University of Washington

Abstract

Modeling Systems from Logs of their Behavior

Ivan Beschastnikh

Co-Chairs of the Supervisory Committee:

Associate Professor Michael D. Ernst
Computer Science and Engineering

Associate Professor Arvind Krishnamurthy
Computer Science and Engineering

Billions of people rely on correct and efficient execution of large systems, such as the distributed systems that power Google and Facebook. Yet these systems are complex and challenging to build and understand. Logging of important events is known to be invaluable for debugging and diagnosing problems in such systems. Unfortunately, many execution logs are inscrutable in their raw form. For example, a production Google system may generate a billion-line log file in a single day.

This dissertation addresses the challenges that developers and operators face in debugging and reasoning about large systems. Specifically, it presents runtime analysis techniques and corresponding tools to help developers make sense of large systems by leveraging the extensive logs generated by the systems.

This dissertation presents three log-analysis tools to infer concise and precise *models* from execution logs of sequential and distributed systems. Three features distinguish these tools and make them simple to use and applicable to a variety of systems and tasks. (1) These tools process the logs most systems already produce and require developers only to specify a set of regular expressions for parsing the logs. (2) These tools do not depend on source code or other implementation-level details of the systems they model, and they do not constrain the semantics of the events in the log. Finally, (3) these tools are designed to scale to large logs.

The first tool, *Synoptic*, infers a finite state machine model of a sequential system from a log of system behavior. Synoptic has two unique features. First, the model it produces satisfies three kinds of temporal invariants mined from the logs, improving accuracy over related approaches. Second, Synoptic uses both refinement and coarsening to explore the space of models. This dual approach improves model efficiency and precision, compared to an approach that uses just one of these options.

The second tool, *Dynoptic*, infers a communicating finite state machine (CFSM) model of a networked system. This model represents each process independently as a finite state machine, and the process machines are augmented with communication events that allow the processes to coordinate over FIFO queues.

Finally, *InvariMint* generalizes the insights from Synoptic and Dynoptic into an approach for declaratively specifying *model inference algorithms*. Existing model inference algorithms are difficult to understand, extend, and compare. InvariMint simplifies each of these tasks.

The dissertation formally defines the model inference techniques underlying all three tools and proves important properties of each of the approaches. Empirical experiments show that developers find the inferred models useful for identifying bugs, confirming previously known bugs, and increasing their confidence in their implementations. By making these tools publicly available, this dissertation helps to bridge the gap between a systems development culture of logging for debugging with advanced techniques from the formal methods and software engineering research communities.

Table of Contents

Page

List of Figures . . . iii

Chapter 1: Introduction . . . 1
1.1 Software models . . . 2
1.2 Informal modeling with mental models . . . 3
1.3 Model inference . . . 5
1.4 Contributions . . . 6

Chapter 2: Synoptic: inferring models of sequential systems . . . 10
2.1 Modeling sequential systems . . . 11
2.2 Overview of approach . . . 13
2.3 Formal description . . . 22
2.4 Tool implementation . . . 29
2.5 Experience with Synoptic . . . 34
2.6 Discussion . . . 43
2.7 Summary . . . 45

Chapter 3: Dynoptic: inferring models of networked systems . . . 46
3.1 Partial order in networked systems . . . 47
3.2 Networked systems as communicating finite state machines . . . 51
3.3 Overview of approach . . . 53
3.4 Formalizing a log of events . . . 58
3.5 Log parsing . . . 61
3.6 Invariant mining . . . 61
3.7 Deriving a concrete FSM . . . 68
3.8 Validating the mined invariants . . . 71
3.9 Abstracting a concrete FSM . . . 74
3.10 Model-checking an AFSM . . . 75
3.11 Refining an AFSM to satisfy invariants . . . 81
3.12 Formal evaluation . . . 83
3.13 Experimental evaluation . . . 85
3.14 Discussion . . . 95
3.15 Summary . . . 96

Chapter 4: InvariMint: model inference through declarative specification 98
4.1 Problems with existing model inference algorithms 98
4.2 Overview of approach 100
4.3 The InvariMint approach 101
4.4 Expressing kTails with InvariMint 109
4.5 Expressing Synoptic with InvariMint 117
4.6 Discussion 124
4.7 Summary 126

Chapter 5: Related work 127
5.1 Model inference and formal methods 127
5.2 Log analysis and visualization 134
5.3 Comprehension and debugging of systems 136

Chapter 6: Conclusion 139

Bibliography 140

List of Figures

Figure Number Page

1.1 An example mental model. 3
1.2 Two logs that are difficult to compare and interpret. 5
1.3 A model inferred by Synoptic. 7
1.4 The most abstract model of a software system. 7
1.5 **(a)** Overview of Synoptic and Dynoptic. **(b)** InvariMint overview. 8

2.1 An example of Synoptic inputs along with the returned model. 12
2.2 The parsed trace graph corresponding to the log in Figure 2.1. 13
2.3 Synoptic process flow chart. This is an elaboration of the more abstract process description in Figure 1.5(a) 15
2.4 The BisimH algorithm. 16
2.5 Initial model corresponding to the trace graph in Figure 2.2. 17
2.6 A path through the initial model in Figure 2.5 that violates a mined invariant. . . . 19
2.7 A refinement of the model in Figure 2.5. 21
2.8 Event invariants mined by Synoptic. 22
2.9 BisimH as a model-space exploration algorithm. 30
2.10 Screenshot of the inputs page in a Synoptic web service. 31
2.11 Screenshot of the Synoptic desktop application. 32
2.12 Screenshot of the model page in a Synoptic web service. 33
2.13 Screenshot of path exploration in a Synoptic web service. 34
2.14 Screenshot of the invariants page in the Synoptic web service. 35
2.15 Synoptic-generated model for a Peterson algorithm log. 36
2.16 Synoptic-generated model for a reverse traceroute log. 37
2.17 Synoptic models generated by a group of students in the distributed systems class. . 39

3.1 An example that illustrates the ambiguity of a total order in a distributed setting. . . 47
3.2 An example of vector clocks, time-space diagrams, and mined partially ordered invariants. 49
3.3 An example mental model of a distributed system. 51
3.4 A simple CFSM model example. 53
3.5 Example Dynoptic inputs and outputs. 54
3.6 Dynoptic process flow chart. This is an elaboration of the more abstract process description in Figure 1.5(a). 57
3.7 A summary of the Dynoptic refinement process. 58
3.8 The co-occurrence counting algorithm v1. 65

3.9 An alternative version of the co-occurrence counting algorithm. 66
3.10 Example execution DAGs and anonymous-state traces parsed by Dynoptic. 72
3.11 The concrete FSM for the execution DAGs in Figure 3.10(a). 73
3.12 ValidateInvariants routine pseudocode. 74
3.13 AFSMtoCFSM routine pseudocode. 76
3.14 An abstract example illustrating partition refinement. 76
3.15 Transforming an edge in a CFSM to track a_i. 77
3.16 TranslatePath routine pseudocode. 82
3.17 Refine routine pseudocode. 83
3.18 The complete Dynoptic algorithm. 84
3.19 Invariant mining time for the transitive closure and the co-occurrence counting algorithms. 86
3.20 Dynoptic-generated TCP server and TCP client models. 87
3.21 Dynoptic-generated model of the replication algorithm in the Voldemort system. . . 91

4.1 An overview of the InvariMint approach. 102
4.2 Example SimpleAlg model inference algorithm inputs and outputs. 103
4.3 Procedural pseudocode of the SimpleAlg algorithm. 104
4.4 An InvariMint specification of SimpleAlg. 105
4.5 Property instances mined by InvariMint for the SimpleAlg example. 106
4.6 The generic property miner algorithm. 107
4.7 The generic property composition algorithm. 108
4.8 Procedural pseudocode of the kTails algorithm. 109
4.9 kTails InvariMint property types for $k = 1$ and $k = 2$. 110
4.10 kTails InvariMint property types for $k = i$. 112
4.11 Runtimes of the procedural and declarative kTails versions for varying log sizes. . . 116
4.12 Runtimes of the procedural and declarative kTails versions for varying number of property instances. 118
4.13 Three of the four property types used by InvariMint to model the Synoptic algorithm. 120
4.14 Model language relationships for InvariMint Synoptic and Synoptic. 121
4.15 Runtimes of the procedural and declarative Synoptic versions for varying log sizes. 123
4.16 Runtimes of the procedural and declarative Synoptic versions for varying number of property instances. 124

Acknowledgments

I have been fortunate to work with an amazing group of mentors. My trio of advisors — Tom Anderson, Michael Ernst, and Arvind Krishnamurthy — have encouraged me to develop my own research agenda, even as I became interested in unconventional and multi-disciplinary topics. They have also kept me focused on my goals and made certain that I never needed to worry about funding. Their wisdom and a passion for research have been a continuous inspiration to me.

Yuriy Brun and Justin Cappos, as CSE postdocs, offered me considerable support. Yuriy was instrumental in engaging me with the software engineering community, and Justin inspired me to work with more undergraduate students. Their energy, fresh perspective, and continued support have played an important role in my career.

Krishna Gummadi and Lidong Zhou hosted me for research internships at MPI-SWS in Germany and Microsoft Research in China. I thank them for taking a chance on me as a young PhD student and allowing me to participate as a full member of their research groups and explore a variety of projects.

Over the years, I have worked on a number of other projects that are not included in this dissertation. All of these projects were undertaken in collaboration with other researchers — Lisa Glendenning, Michael Piatek, Tomas Isdal, Jochen Wuttke, Travis Kriplean and David McDonald. Close collaborators beyond UW include Amre Shakimov, Sigurd Schneider, Fan Yang, and Stanislav Shalunov. The perspectives and humor of all of these people have made my graduate experience memorable, positive, and fun.

I am also indebted to numerous students in the networks, systems, and HCI labs who have offered comments on dozens of my practice presentations, read early paper drafts, and have continually offered their support.

I have worked with a long list of masters and undergraduate students at CSE. All of these students worked on topics related to my research, and many of them have worked on projects described in this thesis: Jenny Abrahamson, Katherine Baker, Vjekoslav Brajkovic, Andrew Davies, Maxwell Forbes, Yoong Woo Kim, Linda Le, Allison Obourn, Tyler Oshiro, Sean Ren, Alper Sarikaya, Michael Sloan, Kevin Thai, Timothy Vega, Zachary Stein, Roykrong Sukkerd, Patrick Williams. I am grateful to all of them for their patience and feedback as I developed my mentoring and advising skills.

I am thankful to the close bunch of fellow students with whom I shared the CSE offices 314 and 352 — Travis Kriplean, Sandra Fan, Dan Halperin, Lisa Glendenning, Chloe Kiddon, and Adrian Sampson. My day-to-day graduate experience was always fun because of you.

David Notkin became a key mentor for me and my cheerleader, especially as I started interviewing for jobs over the last year. Rest in peace, David. Evangelos Theodorou and David Choffnes provided key advice and moral support with the job search process. High-impact coffee conversations with Evangelos are a hallmark memory of my final year at CSE.

My graduate work has been generously supported by the National Science Foundation, DARPA, and Google. I would also like to acknowledge dozens of anonymous reviewers who have spent their valuable time to reflect on my work and to offer valuable external advice on all of the ideas in this dissertation.

Looking further back, four key mentors from the University of Chicago and Stuyvesant H.S. encouraged and inspired me to study Computer Science and attend graduate school – Pete Beckman, Matei Ripeanu, Ian Foster, and Mike Zamansky. Pete was especially instrumental in getting me started with CS research.

Finally, I would like to express gratitude to my loving family and the Chen family, who have been patient with the intensity of my graduate studies, and always ready to help me in any way they could. In particular, my partner Julianna Chen has been the cornerstone of the last decade of my life and I thank her for the unrelenting love and daily support she continues to bestow upon me.

Dedication

To my parents — Сергей and Елена.

Chapter 1

Introduction

When a system behaves in an unexpected way or when a developer makes changes to code written by someone else, the developer faces the challenging task of understanding the system's behavior. A common way of gaining insight into system behavior is to inspect execution logs. Logging system behavior is one of the most ubiquitous, simple, and effective debugging tools. Developers instrument key locations in the code to gain insight into the state of a process, the execution sequence, and the presence or absence of certain events. Production systems at companies like Google are instrumented to generate *billions* of log events each day, indicating that logging is recognized as an important activity. The logs at Google are stored for weeks to help diagnose errant future behaviors [128].

Unfortunately, the collected logs are typically analyzed by hand or with ad hoc tools. This manual inspection of logs is an arduous process as the size and complexity of logs often exceed a human's ability to navigate and make sense of the captured data.

This dissertation presents three tools that infer concise and accurate models of system behavior from logs. The inferred models aim to help developers in three ways: (1) help find bugs, (2) increase developers' confidence in the absence of certain bugs, and (3) improve developers' understanding of their systems. Besides helping developers with system understanding, these *model inference* tools can be used in numerous other contexts. For example, they can be used to detect execution anomalies by comparing models of the system from day to day.

This chapter discusses why models are useful, how developers build and use models, and how automated model inference can help with this process. This is followed by an overview of the approaches to model inference presented in this dissertation by presenting specific challenges to effective model inference and a brief summary of the corresponding solutions and contributions detailed in the dissertation.

1.1 Software models

Models are abstractions that provide a point of reference from which one can consider a problem, work on solutions, and gain a broader understanding of the artifact or process that a model describes. In the natural sciences model generation and validation is the principle goal of the scientific enterprise. The focus of this dissertation are models of *computer systems*, which capture something about how a man-made system operates. As such, these software models are more similar to models used by engineering disciplines, like civil engineering, than to models used by physicists.

Software models vary in the details that they capture. Often, information captured by a model depends on the intended task. For example, one model may be well-suited to answer "How large of a computer cluster should I purchase?" but ill-suited to answer "What is the consistency of data maintained by the system." In addition, models vary in their level of abstraction — a model can describe the execution of the system at a very low level, or it can be more of an architectural representation.

Software models have numerous uses. They can capture intent (e.g., specification), facilitate reasoning about important or hidden aspects of a system, guide implementation efforts, and more. Unfortunately, creating an accurate and useful software model has been a predominantly manual process — creating a complete model of a system is often as challenging as building the system itself.

Because software modeling is difficult and expensive, modern software development practices have emphasized techniques like rapid prototyping. These techniques depend on the fact that software (unlike most other engineering projects, like bridges) is malleable and can be readily changed[1]. If software is evolved, rather than modeled ahead of time and constructed to exact specifications, then modeling can be partially or completely skipped. For example, with rapid prototyping a model of the system would need to be updated to evolve along with the system. Maintaining consistency between the model and the system requires extra effort. In this style of software engineering, the benefits of software modeling are often eclipsed by its costs.

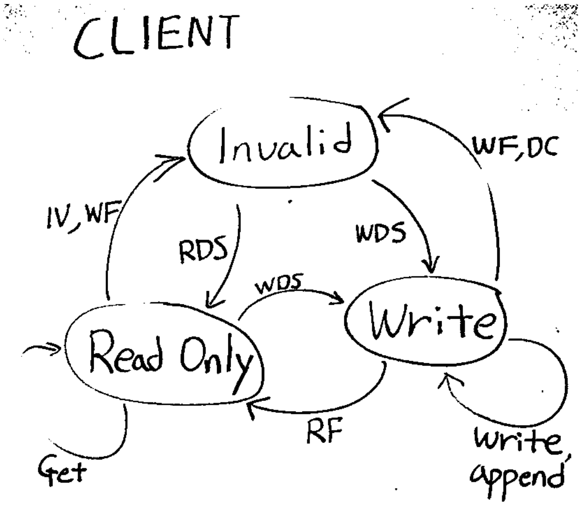


Figure 1.1: An example mental model that depicts client behavior drawn by an undergraduate student for a systems course.

1.2 *Informal modeling with mental models*

While some recent software engineering practices de-emphasize formal modeling, they cannot change the fact that developers often reason about systems abstractly. The absence of a coherent, agreed to, and documented model of a system does not eliminate these requisite mental processes. Quite the opposite — *mental models* must fill the modeling void to provide developers with a means of abstract reasoning about their implementations.

Developers utilize their mental models in many ways. For example, some developers write down a model in some informal manner to facilitate thinking, discussion, and sharing. Figure 1.1 shows an example mental model drawn by an undergraduate student. This model describes a client's behavior in a system that they were working on for class.

A graphically represented model is more concrete than a mental image, but remains disassociated from the implementation. The developer must manually make the link between the drawing and code

[1] The cost of software changes is an active field of research. Software may be malleable, but changing it incurs costs that are often little understood[28].

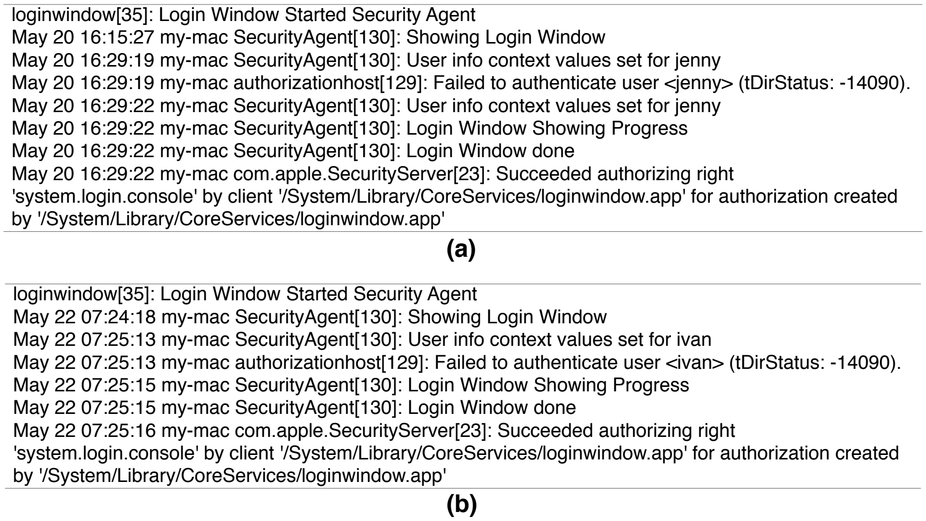
representations of a system's state. A more important issue is that drawn models capture intent, or what the developer expects the system to be; rather than the system's actual implementation behavior.

Logging and mental models. A complementary approach to mental models is logging and log inspection. Examining a runtime log that captures a system's behavior is an effective and popular means of gaining insight into what a system actually does. A log can be generated by instrumenting a system (or by using existing instrumentation) and recording system activity as the system executes. The captured log can then be inspected to find anomalies, verify correctness, debug performance, and for other tasks. Many of these tasks require the developer to reason about low-level system behavior recorded in the log and to match this behavior with their mental model of how the system is supposed to behave.

Unfortunately, developers find it difficult to inspect and reason about logged information as logs record low-level behaviors. For an example of this consider Figure 1.2, which lists two log snippets based on a real log of security-related events from an OS X system. Each snippet represents a sequence of login attempts that result in authorization. One of the two snippets contains a bug, but it is difficult to tell which one.

One major goal of the techniques and tools described in this dissertation is to lower the mental modeling effort that a developer must expend to make sense of a system. The tool described here process logs of systems' *behaviors* to help developers understand the system implementations.

A model of the implementation (as opposed to a low-level log of behavior) that closely matches the user's mental model can help the user utilize their mental model of the system more effectively. The tools presented in this dissertation *infer* this kind of system model from logged behaviors and present the inferred model to the user (i.e., the developer) for inspection. For example, a longer log consisting of login attempts like the ones listed in Figure 1.2 can be processed to derive the finite state machine model in Figure 1.3. This model captures the essential information necessary to understand basic temporal relationships between the logged events and can be used for various tasks by the developer, supporting their mental modeling activities. For example, the derived model makes it easier to notice the aforementioned bug: a failed authentication attempt sometimes results in an authorized login (Figure 1.2(b)). After addressing the bug, the developer can also gain confidence in that the bug has been successfully fixed by inspecting the model generated from a log of a system



```
loginwindow[35]: Login Window Started Security Agent
May 20 16:15:27 my-mac SecurityAgent[130]: Showing Login Window
May 20 16:29:19 my-mac SecurityAgent[130]: User info context values set for jenny
May 20 16:29:19 my-mac authorizationhost[129]: Failed to authenticate user <jenny> (tDirStatus: -14090).
May 20 16:29:22 my-mac SecurityAgent[130]: User info context values set for jenny
May 20 16:29:22 my-mac SecurityAgent[130]: Login Window Showing Progress
May 20 16:29:22 my-mac SecurityAgent[130]: Login Window done
May 20 16:29:22 my-mac com.apple.SecurityServer[23]: Succeeded authorizing right
'system.login.console' by client '/System/Library/CoreServices/loginwindow.app' for authorization created
by '/System/Library/CoreServices/loginwindow.app'
```

(a)

```
loginwindow[35]: Login Window Started Security Agent
May 22 07:24:18 my-mac SecurityAgent[130]: Showing Login Window
May 22 07:25:13 my-mac SecurityAgent[130]: User info context values set for ivan
May 22 07:25:13 my-mac authorizationhost[129]: Failed to authenticate user <ivan> (tDirStatus: -14090).
May 22 07:25:15 my-mac SecurityAgent[130]: Login Window Showing Progress
May 22 07:25:15 my-mac SecurityAgent[130]: Login Window done
May 22 07:25:16 my-mac com.apple.SecurityServer[23]: Succeeded authorizing right
'system.login.console' by client '/System/Library/CoreServices/loginwindow.app' for authorization created
by '/System/Library/CoreServices/loginwindow.app'
```

(b)

Figure 1.2: Two log snippets based on the `/var/log/secure.log` file found in OS X 10.6.8. Each snippet represents a sequence of login attempts resulting in authorization. One of the snippets contains a security bug, but it is difficult to tell which one. Synoptic, one of the tools described in this dissertation, facilitates the task of understanding the contents of a log by generating a model that describes it (Figure 1.3).

with the bug fix.

To provide the developer with an accurate and concise model of the system the tools in the dissertation rely on a procedure called *model inference*, which is described next.

1.3 Model inference

Model inference is the process of automatically, or semi-automatically, generating a model of a system. The most abstract model of a system, shown in Figure 1.4, is a process that takes an input, computes on the input, and produces an output. The goal of model inference is to elucidate the `System` box in Figure 1.4 by producing a model that is less abstract.

There are two general ways to infer a model of a system — statically and dynamically. Static model inference relies on having a detailed specification of the system; for this, a pragmatic choice is code and associated resources that make up the implementation. Dynamic model inference relies on observations of the implementation as it executes.

Dynamic model inference is unconstrained by many implementation details, such as the choice of programming language. However, such techniques are constrained by the input set of observations that can be made about the system. Static model inference, on the other hand, requires a deeper integration with the specification of the implementation, but can more completely reason about a broad space of system behaviors. In the remainder of this dissertation, “model inference” will refer to dynamic model inference.

A model inference procedure typically requires two inputs:

1. *System knowledge* or observations of the system. This is the ground truth about the system and is the principle factor underlying the quality of the inferred model. System knowledge may take many forms, but the most common form is behavioral. Behavioral observations can be gathered at runtime by recording system behavior as it occurs. The techniques in this dissertation rely on these kinds of runtime observations for system knowledge.

2. *Generalization assumptions* capture assumptions about systems in general. Generalization assumptions allow the inference process to generalize from some particular bit of knowledge about a system to infer unobserved (or unknown) behavior of the system.

A model inference algorithm is robust if it produces better models with more knowledge about the system, and requires few generalization assumptions. That is, a robust model inference algorithm should asymptotically approach the true model as the input system knowledge grows to include all aspects of the system. And, a robust model inference algorithm must be broadly applicable and therefore make as few generalization assumptions as possible.

1.4 Contributions

This dissertation explores the possibility and benefits of automated model inference in the context of logs generated by complex systems. More specifically, this work supports the following claim:

> *It is possible to mine execution logs generated by large systems with minimal input from the user to generate models of system behavior that are useful for improving developers’ understanding of their systems.*

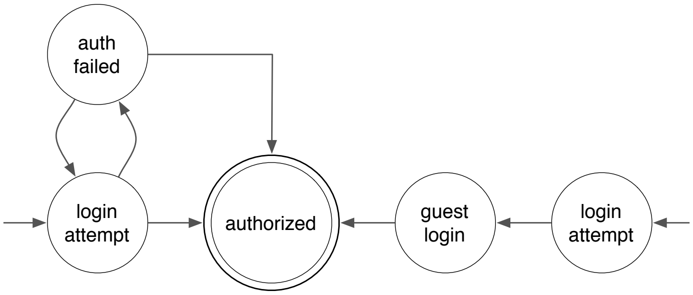


Figure 1.3: A finite state machine model inferred by Synoptic from a log that includes the traces in Figure 1.2.

Input
System
Output

Figure 1.4: The most abstract model of a software system.

The primary contribution of this work are three tools — Synoptic, Dynoptic, and InvariMint — along with both theoretical and empirical evaluations of these tools[2].

Synoptic: inferring models of sequential systems. Synoptic infers a finite state machine model of a sequential system from a log of system behavior. Synoptic differs from prior model-generation tools by its versatility and by imposing few requirements on the developer. To use the tool, developers do not need to specify their systems as part of the design, identify properties for the tool to verify, or modify their code. Instead, Synoptic mines three kinds of temporal properties, or *invariants*, from existing logs and uses these to generate a concise model satisfying the invariants. A developer only needs to provide a set of regular expressions to parse events from the logs. Using this approach, Synoptic (1) does not restrict developers to a particular log format and (2) allows developers to specify the events to include in the model.

In our evaluation, Synoptic generated models on logs up to 900,000 events that represent over

[2] All tools are released as open source and available for download at http://synoptic.googlecode.com

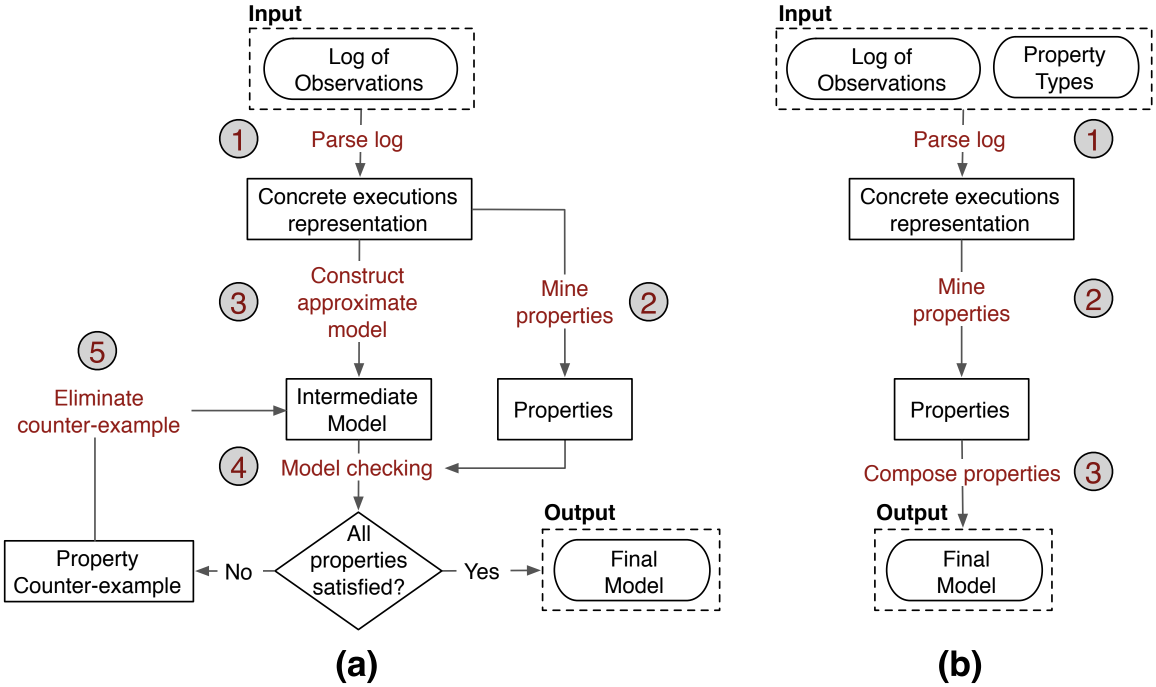


Figure 1.5: **(a)** Overview of Synoptic and Dynoptic. **(b)** InvariMint overview.

28,000 unique system executions. Most developers in our studies found the generated models helpful in understanding their systems. Synoptic models increased developer confidence in the correctness of their implementations, helped identify previously unknown bugs, and confirmed the existence of known bugs. Chapter 2 presents Synoptic, which was previously described in [114, 22, 18].

Dynoptic: inferring models of networked systems. Dynoptic applies the core ideas in Synoptic to infer models of networked, or distributed, systems. Dynoptic infers a communicating finite state machine (CFSM) model from a partially ordered log generated by the system. This model represents each process independently as a finite state machine. The process machines are augmented with communication events that allow the processes to coordinate over FIFO queues.

Building on Synoptic, Dynoptic also uses temporal invariants to improve the CFSM model accuracy, and also imposes minimal requirements on inputs from the developer.

To evaluate Dynoptic we applied it to logs of three different systems — the stop-and-wait protocol, opening/closing handshakes of TCP, and the replication strategy in the Voldemort [125] distributed hash table. We also carried out a user study to evaluate the efficacy of CFSM models in finding bugs. Our evaluation found that Dynoptic produces models that are accurate and useful in finding implementation bugs. Chapter 3 presents Dynoptic, which was previously described in [21].

Figure 1.5(a) summarizes the approach underlying both Synoptic and Dynoptic. Both tools maintain their unique representations for traces parsed from the input log (step ① in Figure 1.5(a)). Both tools then mine a set of properties (step ②), which are used by a counter-example guided abstraction refinement [32] process in steps ④ and ⑤ to improve the model of logged observations. The Synoptic- and Dynoptic-specific versions of Figure 1.5(a) are presented in Figure 2.3 and Figure 3.6, respectively.

InvariMint: model inference through declarative specification. Existing model inference algorithms are challenging to understand, extend, and compare. InvariMint simplifies each of these tasks by generalizing the insights in Synoptic into an approach for declaratively specifying *model inference algorithms* that infer finite state machine models of systems. InvariMint enables specification of algorithms in terms of the types of *properties* they enforce in the inferred models.

To evaluate InvariMint, we applied it to two previously-published algorithms. First, we used InvariMint to declaratively and exactly specify the well-known kTails [23] algorithm. Second, we used InvariMint to approximate Synoptic [22]. Chapter 4 presents InvariMint, which was previously described in [19, 20].

Figure 1.5(b), illustrates the InvariMint approach, which is centered around the property types input. These types direct the mining step (step ② in Figure 1.5(b)) to derive property instances, which are then composed (step ③) into the final model. Figure 4.1 presents a more detailed InvariMint overview.

The three chapters described above cover the design, implementation, and evaluation of Synoptic, Dynoptic, and InvariMint. Chapter 5 places the work in this dissertation in the context of prior academic work, and Chapter 6 concludes.

Chapter 2

Synoptic: inferring models of sequential systems

Synoptic is a tool that helps developers by inferring a concise and accurate system model. Unlike most related work, Synoptic does not require developer-written scenarios, specifications, negative execution examples, or other complex user input. Synoptic processes the logs most systems already produce and requires developers only to specify a set of regular expressions for parsing the logs.

Synoptic has two unique features. First, the model it produces satisfies three kinds of temporal invariants mined from the logs, improving accuracy over related approaches. Second, Synoptic uses refinement and coarsening to explore the space of models. This improves model efficiency and precision, compared to using just one approach.

To see why Synoptic is useful consider Figure 2.1, which shows a web server log for a shopping cart application. Using the two listed regular expressions, Synoptic parses the log into three traces, one for each of the three user IP addresses accessing the server (Figure 2.2). Synoptic then mines temporal invariants that hold in those traces and uses the invariants to infer a model of the system (bottom of Figure 2.1). The model clearly illustrates a bug that would be difficult to find by examining the log directly: applying an invalid coupon allows the user to reduce the price. Not only can Synoptic help a developer find this bug, it can also increase the developer's confidence that the bug has been successfully removed. For example, the developer can run Synoptic on logs generated by a new version of the system and compare the new model with the prior model.

We evaluated Synoptic both theoretically and experimentally. Section 2.3 formally proves that Synoptic produces a model that satisfies all the true temporal invariants mined from the log and none of the invariants that are not satisfied by the log. Further, we argue that Synoptic's exploration of the model space is efficient and produces concise models.

We detail Synoptic tool prototype in Section 2.4. In our experimental evaluation, Synoptic generated models on logs up to 900,000 events that represent over 28,000 unique system executions. Most developers in our studies found the generated models helpful in understanding their systems.

Synoptic models increased developer confidence in the correctness of their implementations, helped identify previously unknown bugs, and confirmed the existence of known bugs.

Additionally, to demonstrate Synoptic's ability to produce useful representations in practice, we evaluated it in two user experience studies (Section 2.5). We first report on a study with a developer working on reverse traceroute [14], a distributed system that determines the likely reverse Internet route between two hosts. Reverse traceroute has been in deployment for over 7 months, has handled a total of 3.6 million requests to date, and has been recently internally deployed by a large, popular, and ubiquitous Internet company. Second, we report on the experiences of 45 undergraduate students who used Synoptic in a distributed systems course. The students applied Synoptic to logs generated by their implementations of a distributed version of a cache coherence protocol [84].

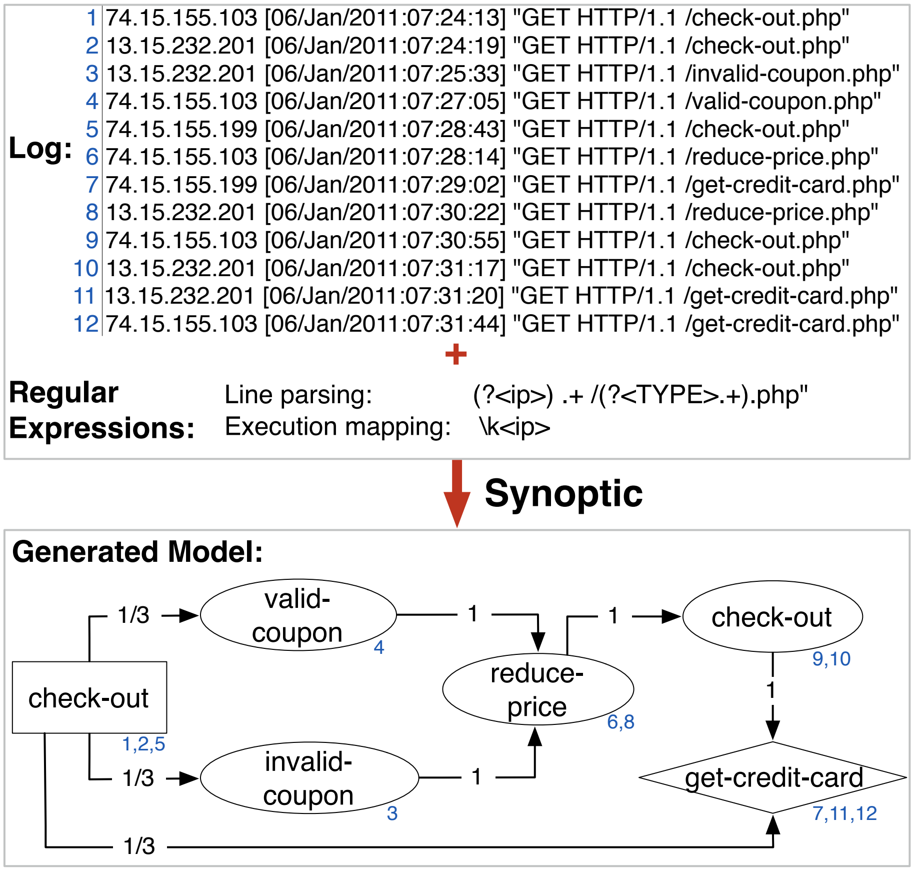
Next, we motivate and explain how Synoptic works by describing BisimH, the central algorithm that Synoptic uses.

2.1 Modeling sequential systems

Synoptic addresses the problem of finding a compact representation that summarizes a sequence of events logged by a software system. The notion of "event" depends on the system — events may be sent and received messages, local procedure invocations, debug output, or a combination of all of these. Synoptic uses a relational model, which is equivalent to a finite state machine model with anonymous states.

State-based model. Developers often structure node logic as a finite state machine (FSM) in which nodes represent system states. Although the FSM model is widely used, it is overly complex in our context because it forces an inference algorithm to reason about system state. Events that appear in the log may indicate that the system is in a particular state, but we cannot assume that the log contains explicit state information. Reasoning about states detracts from the ultimate purpose of finding a model that captures sequences of logged events.

Relational model. The relational model (e.g., bottom of Figure 2.1) captures possibly multiple relations between log events. For example, events may be related through time (the default relation in Synoptic), physically (by co-occurring at the same node), or in other user-defined ways. We visualize the relational model as a graph in which each vertex represents a set of event instances. A directed


Log:

```
 1 74.15.155.103 [06/Jan/2011:07:24:13] "GET HTTP/1.1 /check-out.php"
 2 13.15.232.201 [06/Jan/2011:07:24:19] "GET HTTP/1.1 /check-out.php"
 3 13.15.232.201 [06/Jan/2011:07:25:33] "GET HTTP/1.1 /invalid-coupon.php"
 4 74.15.155.103 [06/Jan/2011:07:27:05] "GET HTTP/1.1 /valid-coupon.php"
 5 74.15.155.199 [06/Jan/2011:07:28:43] "GET HTTP/1.1 /check-out.php"
 6 74.15.155.103 [06/Jan/2011:07:28:14] "GET HTTP/1.1 /reduce-price.php"
 7 74.15.155.199 [06/Jan/2011:07:29:02] "GET HTTP/1.1 /get-credit-card.php"
 8 13.15.232.201 [06/Jan/2011:07:30:22] "GET HTTP/1.1 /reduce-price.php"
 9 74.15.155.103 [06/Jan/2011:07:30:55] "GET HTTP/1.1 /check-out.php"
10 13.15.232.201 [06/Jan/2011:07:31:17] "GET HTTP/1.1 /check-out.php"
11 13.15.232.201 [06/Jan/2011:07:31:20] "GET HTTP/1.1 /get-credit-card.php"
12 74.15.155.103 [06/Jan/2011:07:31:44] "GET HTTP/1.1 /get-credit-card.php"
```

+

Regular Expressions:

```
Line parsing:        (?<ip>) .+ /(?<TYPE>.+).php"
Execution mapping:   \k<ip>
```

Figure 2.1: (Top) A log with line numbers for an online shopping cart, and two complete regular expressions for processing this log with Synoptic. (Bottom) The generated Synoptic model. In the model, rectangular/diamond/oval nodes indicate initial/terminal/intermediate nodes. Edge labels indicate transition probabilities. The subscript to the right of each node lists the log line numbers of events that are associated with the node. This application contains a bug that is easily noticed in the generated model: processing an invalid coupon incorrectly reduces the shopping cart's total price.

edge between two vertices indicates that the associated event instances are related.

We have found the relational model to be the most appropriate for capturing sequences of events. This model resembles a modal transition system, which is a natural fit for reasoning about temporal invariants between events. And, unlike a state-based model, a relational model makes minimal

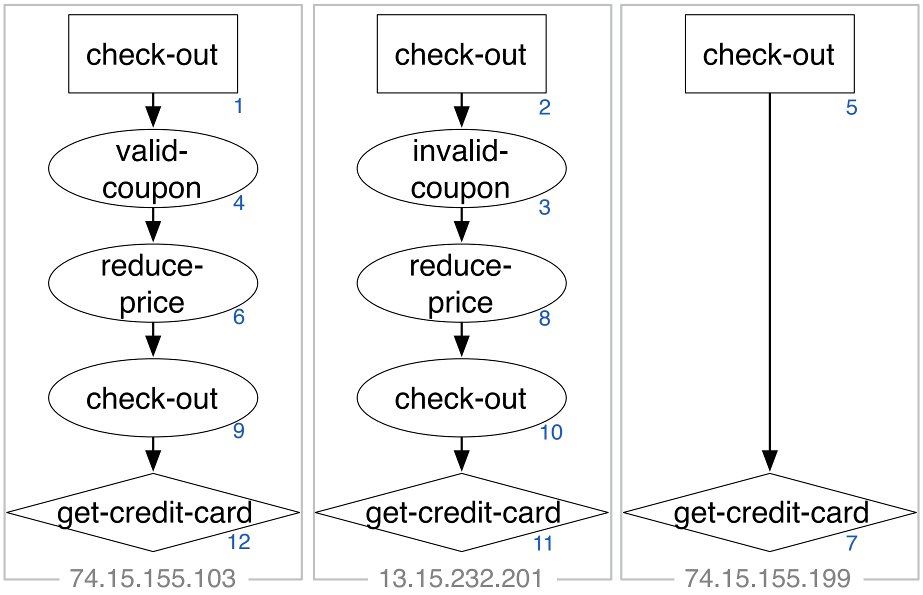


Figure 2.2: Trace graph parsed from the log in Figure 2.1. Each execution corresponds to an IP address that accessed the web application. The subscript to the right of each node lists the line number of the log line from which the event instance was extracted.

assumptions about the underlying process that produced the logged events. We also successfully experimented with mappings between the relational and state-based models, that make it possible to automatically convert representations between the two model types.

2.2 Overview of approach

Figure 2.3 overviews how Synoptic works. At its core Synoptic uses a hybrid refinement and coarsening algorithm called BisimH. The rest of this section explains the algorithm in detail by walking through its pseudo-code listed in Figure 2.4, and by illustrating how Synoptic would process the log in Figure 2.1.

2.2.1 Log parsing

Synoptic constructs a system model from a set of observed system execution traces. It takes as input a log file containing the execution traces, and a set of user-defined regular expressions. Synoptic uses the regular expressions [3] to parse the log file and extract from some of the log lines an *event instance*: a triplet containing: (1) a trace identifier, (2) a timestamp, and (3) an *event type*. Trace identifiers are used to group together event instances from the same *trace*. Synoptic requires that the event instances in a trace be totally ordered using their timestamps. Therefore, no two event instances in a trace may have identical timestamps. An event type can be an arbitrary string, and is usually defined by the developer as something that conveys important information about the system. For example, Section 2.5 presents two Synoptic-generated models in which an event type represents (1) an executed method's name, and (2) the state of a node in a distributed system.

A trace can be considered to be a linear graph — each vertex is an event instance, and the edges represent the total ordering. We term the union of such graphs a *trace graph*. The trace graph is built from the log using the provided regular expressions (line 2 in Figure 2.4).

Recall the shopping cart application. Figure 2.1 shows the log and the two complete regular expressions that Synoptic uses to parse the log into three traces, one per unique IP in the log; the php script names denote the trace event types. Figure 2.2 shows the three traces parsed from the log. For example, the trace corresponding to the IP 74.15.155.103 is ⟨0, check-out⟩, ⟨1, valid-coupon⟩, ⟨2, reduce-price⟩, ⟨3, check-out⟩, ⟨4, get-credit-card⟩. Here, the integer timestamp is derived implicitly from the order of lines in the log.

2.2.2 Mining invariants from the trace graph

To guide model generation, Synoptic mines three kinds of temporal invariants relating *event types* from the trace graph (line 3 in Figure 2.4):

$a \rightarrow b$: An event type a is **always followed by** an event type b.

$a \not\rightarrow b$: An event type a is **never followed by** an event type b.

$a \leftarrow b$: An event type a **always precedes** an event type b.

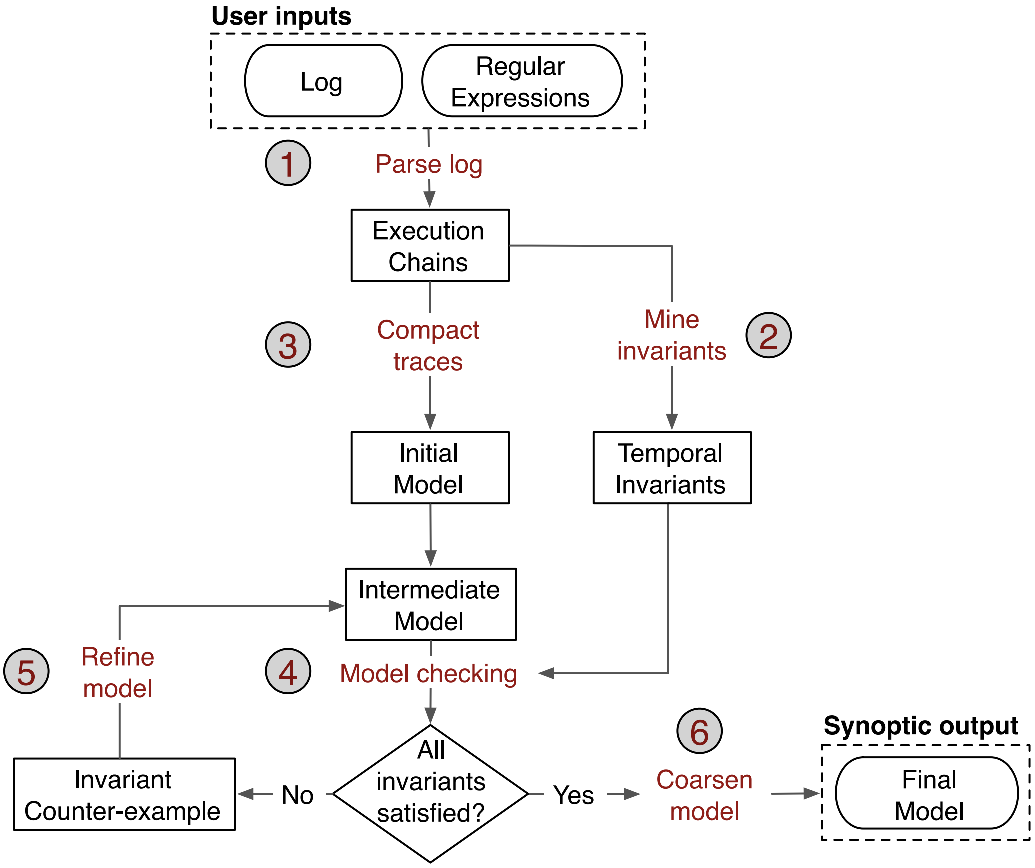


Figure 2.3: Synoptic process flow chart. This is an elaboration of the more abstract process description in Figure 1.5(a)

The missing symmetrical invariant *Never Precedes*, defined as *a* Never Precedes *b* iff *b* can be generated only when no *a* was yet generated, is equivalent to the *Never Followed by* invariant.

We term these relations "invariants" because they succinctly capture temporal event type relationships that must hold true over all the input traces. The trace graph in Figure 2.2 yields 27 such invariants. Two examples are *reduce-price* $\not\to$ *valid-coupon*, and *invalid-coupon* $\to$ *check-out*. Section 2.2.5 justifies our use of these particular invariant types, and Section 2.3.2 explains how these invariants are mined. Next, we introduce Synoptic models.

```
1  Input: log $L$, regular expressions $RegExps$
2  let traceGraph = extract($L$, $RegExps$)
3  let $I$ = mineInvariants(traceGraph)
4  let $(V,E)$ = partition(traceGraph)
5  while $(V,E)$ does not satisfy invariants $I$
6    // p: event $\rightarrow$ boolean, $\pi$: partition that will be split
7    let $(p, \pi)$ = selectSplit($(V,E)$, $I$)
8    let $\pi_1$ = $\{\text{event} \in \pi \mid p(\text{event})\}$
9    let $\pi_2$ = $\{\text{event} \in \pi \mid \neg p(\text{event})\}$
10   $V$ := $(V - \{\pi\}) \cup \{\pi_1, \pi_2\}$
11   $E$ := $\{(\pi_3, \pi_4, r) \in V \times V \times R \mid \exists\ \text{event}_1 \in \pi_3, \exists\ \text{event}_2 \in \pi_4$
12         $:\ \text{event}_1\, r\, \text{event}_2 \in \text{traceGraph}\}$
13 end while
14 $(V,E)$ := kTail($(V,E)$, 0, $I$)
15 Output: $(V,E)$
```

Figure 2.4: The BisimH algorithm. Section 2.2 describes the `extract`, `mineInvariants`, `partition`, `selectSplit`, and `kTail` procedures.

2.2.3 Synoptic models and the initial model

The Synoptic model is a *partition graph* of the trace graph. Given a partitioning of the original vertices, each vertex in the model is one partition. Directed edges in the model are formed through existential abstraction. That is, a directed edge between two vertices indicates that there exists a pair of event instances in the corresponding partitions that are connected by an edge in the trace graph. A further constraint is that each partition contains event instances of only one particular event type. The resulting relational model makes minimal assumptions about the underlying process that produced the logged event instances.

An important property of Synoptic models is that each trace in the input log is accepted by a model constructed from the corresponding event instances (in the sense that each trace maps to a valid path in the model). However, a Synoptic model is also generative — it may accept traces that were not present in the log.

The BisimH algorithm starts with an *initial model* (constructed using `partition` on line 4 of

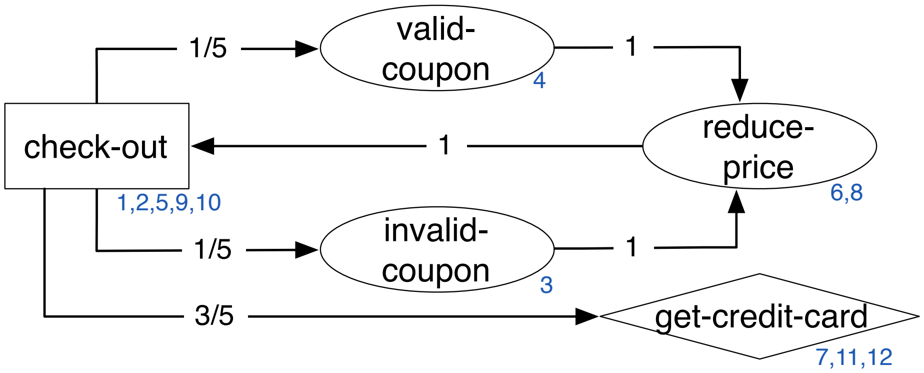


Figure 2.5: Initial model corresponding to the trace graph in Figure 2.2.

Figure 2.4). In this model, there is one partition per event type containing all the event instances of that type. Figure 2.5 shows the initial model for the trace graph in Figure 2.2.

By construction, the initial Synoptic model captures two important kinds of temporal properties for any two adjacent event instances in a trace in the log. First, if an event instance of type *a* is at some point immediately followed by an event instance of type *b* in the log, then there must be an edge from *a* to *b*. Second, if an event instance of type *a* is never immediately followed by an event instance of type *b* in the log, then there is no edge from *a* to *b*.

The initial model is therefore the most compact or abstract model plausible, based on the logged traces. The least compact (and most concrete) model is the trace graph, in which each partition contains a single event instance. This model makes no generalizations and overfits to the input traces.

2.2.4 Refinement and coarsening

Coarsening and refinement are dual operations on a Synoptic model. Starting with the initial model, Synoptic first performs model refinement, shown as an iterative process in lines 5–13 of Figure 2.4. This algorithm is a modification of a partition refinement algorithm introduced by Elomaa [50]. Synoptic refines (i.e., splits) partitions until it reaches a model that satisfies all the mined invariants. Next, Synoptic uses coarsening to merge those partitions that were needlessly refined due to an imperfect splitting heuristic (line 14 in Figure 2.4). The coarsening step is constrained to not violate the mined invariants satisfied during refinement. Synoptic outputs the model when it is unable to coarsen it any further.

Refinement

The refinement goal of BisimH is to pick a minimal sequence of splits, so that the resulting graph is the coarsest graph that satisfies a set of invariants. This problem is NP-hard [32], so an efficient algorithm might not yield the optimal result. For example, a split partition may need to be split again to help eliminate another counterexample, even though a single split might suffice to help eliminate both counterexamples. Figure 2.9 illustrates refinement suboptimality visually. More generally, here are three reasons why refinement is suboptimal:

- Invariants are considered independently.
- Counter-examples for an invariant are considered one at a time.
- A counter-example is resolved by isolating the *conflicting* incoming and outgoing event instances from one another, without considering other event instances in the same partition.

To counteract suboptimal refinements, BisimH uses coarsening, which is explained in Section 2.2.4.

BisimH performs splits as long as there exists some mined invariant that is not satisfied. BisimH uses an FSM-based model checker to check whether a model satisfies a mined invariant. It converts each invariant into a small FSM that accepts traces satisfying the invariant. It then updates the FSMs as it traverses the model graph. If the model does not satisfy an invariant, the model checker outputs a counterexample path. For example, the invariant *valid-coupon* $\not\to$ *invalid-coupon* mined from the log in Figure 2.1 is not true in the model in Figure 2.5 — Figure 2.6 shows a counterexample path.

Having identified a set of counterexamples that violate the mined invariants, BisimH follows the counterexample guided abstraction refinement (CEGAR) approach [32] to determine a set of *candidate partitions*, for each of which there exists a split that removes at least one of the counterexamples. BisimH identifies these partitions heuristically by tracing each counterexample, stepwise, in parallel, in the input traces and in the model. In the traces, only a prefix of the

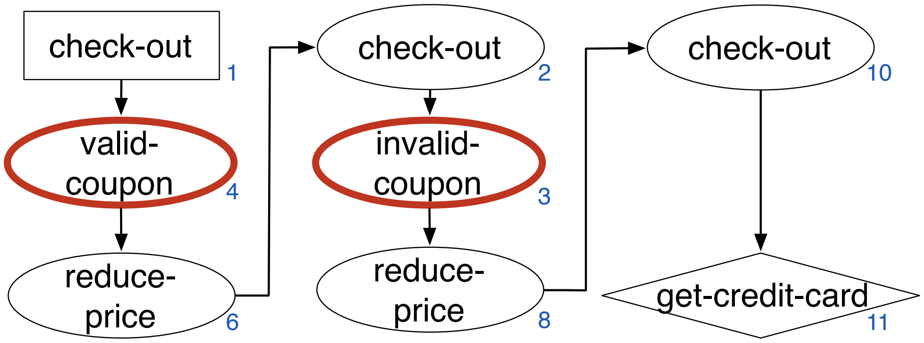


Figure 2.6: A path through the initial model in Figure 2.5 that violates the mined *valid-coupon* $\not\to$ *invalid-coupon* invariant.

counterexample path will be present (otherwise the counterexample would not violate an invariant). BisimH finds the longest such prefix, and the last partition of this prefix in the model becomes a candidate for refinement — this partition allows a spurious transition in the model that allows for the counterexample path to exist. For example, the longest such prefix for the counterexample path in Figure 2.6 ends in the *check-out* partitions. This is because *check-out* stitches together two traces from the log (two left-most traces in Figure 2.2) into a trace that violates the *valid-coupon* $\not\to$ *invalid-coupon* invariant.

To refine a candidate partition (i.e., to eliminate the counterexample path), the event instances in this partition are divided into two sets based on whether they can or cannot be reached from the partition immediately preceding the candidate partition in the prefix. In line 7 of Figure 2.4, `selectSplit` obtains a predicate p that distinguishes these two event instance sets, and lines 8 and 9 introduce two new partitions, π_1 and π_2, corresponding to these two sets. Figure 2.7 illustrates a refinement of the initial model in Figure 2.5 to eliminate the counterexample in Figure 2.6. In this case, the predicate p separates the event instances in the *check-out* partition into those that can or cannot be reached from the *reduce-price* partition.

We experimented with two kinds of predicates. Synoptic uses the one described above: it separates event instances in the candidate partition based on an incoming edge from a partition that immediately precedes the candidate partition. We also tried a predicate that separates event instances in the candidate partition based on an outgoing edge representing the spurious transition — it separates event instances in the candidate partition into sets based on whether they can or cannot make the spurious transition. Though we did not show this formally, in practice, we found the second strategy to be less optimal than the first. It is also possible to split the candidate partition simultaneously on an incoming and on an outgoing edge. Though we have not tried this, we think this may work best. In our future work, we intend to further study the splitting predicate's impact on the algorithm.

Typically, the refined model violates several invariants and candidate partitions must be ranked to decide which one to split first. Synoptic employs a two-class ranking: it examines all counterexamples in an arbitrary order and performs the first split that validates an invariant (i.e., eliminates the last counterexample for that invariant). If no such split is available (because more counterexamples exist for each invariant), BisimH picks a split nondeterministically. This ranking introduces nondeterminism and BisimH might perform unnecessary splits.

Coarsening

BisimH may end up refining more than it needs to. When this happens, the model will contain partitions that can be merged without violating the satisfied invariants. After refinement, BisimH coarsens the model to merge such partitions (line 14 in Figure 2.4).

For coarsening, BisimH uses *kTail-equivalence* [23]. kTail is a coarsening algorithm that starts with the most fine-grained model. It stops once there is no pair of k-equivalent partitions, i.e., no two partitions that are roots of sub-graphs identical up to depth k. At each step, the algorithm merges one pair of *kTail-equivalent* partitions, chosen nondeterministically. BisimH runs kTail with $k = 0$ (label equivalence) to produce the most concise models. It starts with the final refined graph, under the extra constraint that all merges do not unsatisfy any invariants. The resulting merged model is locally minimal: merging any two partitions will violate some invariant.

2.2.5 The impact of mined invariants on BisimH

BisimH uses the mined invariants to establish a well-defined termination criterion for refinement, and also to guide refinement in its choice of partition to refine. This use of invariants is an important

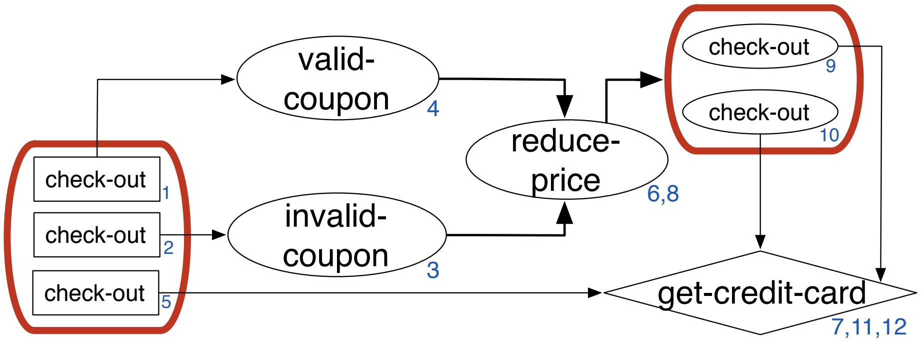


Figure 2.7: A refinement of the model in Figure 2.5 that eliminates the counterexample path in Figure 2.6. The edge between the *reduce-price* partition and the *check-out* partition induces a split of *check-out*: the *check-out* event instances reachable from *reduce-price* are split out. The two new *check-out* partitions, with the contained *check-out* event instances, are shown in bold. This model is equivalent to the final model shown at the bottom of Figure 2.1.

feature of BisimH. To see this, suppose the set of invariants is empty. In this case, refinement would terminate with a model that is the quotient under label-equivalence, i.e., the initial model. This model is often too compact to capture key properties of the log and is overly generative. On the other hand, suppose that the invariant set includes all possible temporal log invariants expressible in LTL. Then the algorithm will terminate when, for all partitions A, if an event instance in A has a successor event instance in a partition B, then every event instance in A has a successor event instance in B in the model. In this case, the final model is the quotient under bisimulation, i.e., a graph that satisfies the same set of LTL formulae as the trace graph. In our experience, the bisimulation quotient is usually too similar to the trace graph, and thus too fine-grained to be considered concise.

Our choice of the three invariant types (listed in Figure 2.8 along with their LTL formulas) is a compromise between the above two extremes. In our experience, the models derived using this set of invariants are accurate, yet sufficiently generative for the kinds of applications we are considering (e.g., improving developer understanding of how the system operates). These invariant types are also exactly the most frequently observed specification patterns formulated by Dwyer et al. [47], with scope constrained to a trace (i.e., global scope). The translation is not one-to-one: $a \rightarrow b$ is Dwyer's

Invariant		**LTL formula**	**Type**
$x \rightarrow y$	(x Always Followed by y)	$\square(x \rightarrow \lozenge y)$	liveness
$y \leftarrow x$	(y Always Precedes x)	$\lozenge y \rightarrow \neg y \text{ U } x$	safety
$x \nrightarrow y$	(x Never Followed by y)	$\square(x \rightarrow \square \neg y)$	safety

Figure 2.8: Event invariants mined by Synoptic, with corresponding LTL formula and classification. LTL properties must hold over the each trace in the input log and are specified using the operators: *always* ($\square$), *eventually* ($\lozenge$), and *until* (U). For example, the formula $\lozenge y \rightarrow \neg y \text{ U } x$ requires x to occur before y. Without the premise $\lozenge y$, x would be required to appear at least once, even if the trace does not contain y.

Existence pattern when a is *START* (see Definition 2.2 below), and is otherwise Dwyer's Response pattern. Another example is $\forall b, a \leftarrow b$, which is Dwyer's Universality pattern. In our experience, these invariants were sufficient for capturing key temporal properties of the systems that produced the logs we considered.

Users can write Java code to define custom Synoptic invariants. However, all of the users in our case studies (Section 2.5) successfully used Synoptic without even knowing about its use of invariants.

In the next section, we define log and model formalism and prove important positive results about the BisimH algorithm.

2.3 Formal description

This section proves the correctness of our algorithm, and explains why it is efficient and is able to infer *concise* models in practice. Sections 2.3.1 and 2.3.2 define the formalisms. Section 2.3.3 proves that BisimH always halts and that the final model satisfies exactly the invariants mined from the input log. Section 2.3.4 proves an important result for improving model search efficiency, and Section 2.3.5 deals with model size.

2.3.1 Definitions

Two special event types — *START* and *END* — are added internally by Synoptic to keep track of initial and terminal events in the traces.[1]

Definition 2.1 (Event Types)**.** A set of event types is a finite set (alphabet) $E \supseteq \{START, END\}$.

Definition 2.2 (Trace)**.** Let E be a set of event types. Then for all $n \in \mathbb{N}_{\geq 2}$, a finite trace is an ordered sequence of event types $l \in E^n$ such that the first element of l is *START* and the last element is *END*. The length of l is $n-2$.

Definition 2.3 (Log)**.** A log L is a set of traces.

The set of event instances in a log is the collection of elements in the traces in that log. Each trace element is a unique event instance, indexed by its trace and position within that trace.

Definition 2.4 (Event Instances)**.** Let E be a set of event types. Let L be a log over E. Then an *event instance* is a triplet $\langle e, l, i\rangle$ such that $e \in E$ occurs in the log trace $l \in L$ at position $i \in \mathbb{N}$. $\hat{E}$ denotes the set of all such event instances for L.

An event instance relation is a set of pairs of elements of $\hat{E}$. For example, one representation (using "**0**" to represent the event instance $\langle \mathbf{0}, l, 1\rangle$, etc.) of the event instance relation "next" on the log trace $l = \langle 0,1,2,3,4\rangle$ is $\{\langle 0,1\rangle, \langle 1,2\rangle, \langle 2,3\rangle, \langle 3,4\rangle\}$. The examples in this chapter only use this "next" relation, although all the results generalize to arbitrary relations.

Definition 2.5 (Event Instance Relation)**.** Let $\hat{E}$ be a set of event instances. Then $r \subseteq \hat{E}^2$ is an event instance relation.

A partitioning of a finite set of event instances $\hat{E}$ is a finite set of disjoint, exhaustive subsets of $\hat{E}$. Each subset is called a partition and contains event instances of the same event type.

Definition 2.6 (Partitioning)**.** Let $\hat{E}$ be a set of event instances. Then $P \subset \mathcal{P}(\hat{E})$ is a partitioning of $\hat{E}$ if $\forall$ distinct $p,q \in P$, $p \cap q = \emptyset$, and $\hat{E} = \bigcup_{p \in P} p$, and $\forall p \in P$, all $\hat{e} \in p$ are of the same event

[1] When viewing a model, the user can optionally hide these nodes, and instead have Synoptic specially mark the partitions containing any initial and terminal events as rectangles and rhombuses, respectively. The models pictured in this chapter were all generated in this way.

type. Each $p \in P$ is called a partition. We enforce the condition that for a valid partitioning, all instances of the *START* event type are in a single partition and all instances of the *END* event type are in a single partition. That is, for all $\hat{e_1} \in p_1$, $\hat{e_2} \in p_2$: $\hat{e_1}, \hat{e_2}$ instances of $START \Rightarrow p_1 = p_2$ and $\hat{e_1}, \hat{e_2}$ instances of $END \Rightarrow p_1 = p_2$.

A relational model is a partition graph. The largest (most nodes) relational model for a log L is the trace graph: the set of disconnected subgraphs, one for each trace $l \in L$, with a vertex for each event instance and edges only between consecutive event instances in l. Other relational models can be generated by merging vertices that represent event instances of the same event type (and removing redundant edges).

Definition 2.7 (Relational Model)**.** Let $\hat{E}$ be the set of event instances in a log. Let R_r be a family of relations over $\hat{E}$ indexed by r. Then the relational model is a directed graph $M = \langle M_V, M_A \rangle$, such that M_V is a partitioning of $\hat{E}$ and $a = \langle p_1, p_2, r \rangle \in M_A \subseteq M_V \times M_V \times R_r$ iff $p_1, p_2 \in M_V$ and $\exists \hat{e}_1, \hat{e}_2 \in \hat{E}$ such that $\hat{e}_1 \in p_1$, $\hat{e}_2 \in p_2$, and $\langle \hat{e}_1, \hat{e}_2 \rangle \in R_r$.

Definition 2.8 (Complete Path)**.** A path in a model is complete if it starts at the *START* partition and ends at the *END* partition.

A relational model M accepts a trace if the event instances of the trace form a complete path in M.

Definition 2.9 (Trace Acceptance)**.** Let $n \in \mathbb{N}$ and let $l = \langle START, \hat{e}_1, \ldots, \hat{e}_n, END \rangle$ be a trace of length n. Then a relational model M accepts l iff $\exists \Pi = \langle p_{START}, p_1, \ldots, p_n, p_{END} \rangle$ such that $\forall 1 \leq i \leq n$, $\hat{e}_i \in p_i$ and Π is a complete path in M.

Note that by construction a relational model M for a log L accepts all traces in L. To see this, consider a trace $l = \langle START, \hat{e}_1, \ldots, \hat{e}_n, END \rangle \in L$. The "next" relation, corresponding to R_{next}, holds for all pairs of adjacent event instances, that is $\forall 1 \leq i < n, \langle \hat{e}_i, \hat{e}_{i+1} \rangle \in R_{next}$ as well as $\langle START, \hat{e}_1 \rangle \in R_{next}$ and $\langle \hat{e}_n, END \rangle \in R_{next}$. This means that l maps to a complete path in M and therefore M accepts l.

2.3.2 Invariants

We consider three invariants that relate pairs of event types:

Definition 2.10 (Event Invariant)**.** Let a and b be two event types. Then an event invariant is a property that relates a and b in one of the following three ways:

$a \rightarrow b$ **:** In a model, if some partition on a complete path contains an event instance of type a, then at least one later partition along that path contains an event instance of type b.

$a \not\rightarrow b$ **:** In a model, if some partition on a complete path contains an event instance of type a, no later partition along that path contains an event instance of type b.

$a \leftarrow b$ **:** In a model, if some partition on a complete path contains an event instance of type b, at least one earlier partition along that path contains an event instance of type a.

Definition 2.11 (Invariant Satisfiability)**.** Let M be a relational model, and let i be an event invariant. M satisfies i iff $\forall$ Π, a complete path in M, i is true of Π.

Each of the three event invariants may relate any pair of event types. Thus, for a set of event types E there can be at most $3|E|^2$ invariants.

Synoptic mines the above invariants by collecting three kinds of counts across all the traces. Each trace is traversed once in the forward and once in the reverse direction to count:

- $\forall a$,Occurrences$[a]$: the number of event instances of type a,
- $\forall a, b$ Follows$[a][b]$: the number of event instances of type a that are followed by at least one event instance of type b, and
- $\forall a, b$ Precedes$[a][b]$: the number of event instances of type b that are preceded by at least one event instance of type a.

The invariants are then determined by using the following equivalences:

$$a \rightarrow b \;\Leftrightarrow\; \text{Follows}[a][b] = \text{Occurrences}[a]$$
$$a \not\rightarrow b \;\Leftrightarrow\; \text{Follows}[a][b] = 0$$
$$a \leftarrow b \;\Leftrightarrow\; \text{Precedes}[a][b] = \text{Occurrences}[b]$$

2.3.3 BisimH termination

This section proves that BisimH terminates (Theorem 2.1) and that the final model satisfies all the event invariants in the log (Theorem 2.2) and no others (Theorem 2.3).

Theorem 2.1 (BisimH Terminates)**.** *BisimH makes a finite number of iterations (no more than the number of event instances).*

Proof of Theorem 2.1. After every BisimH iteration, the model has more partitions than the model before the iteration, as some partition is split into two new partitions. The maximal model has a finite number of partitions (each event instance is in its own partition, except the *START* and *END* instances) and satisfies all the mined invariants. Therefore, BisimH can make no more iterations than there are event instances. □

Theorem 2.2 (True Invariant Satisfiability)**.** *BisimH produces a final model that satisfies all the event invariants that are true for a log.*

Proof of Theorem 2.2. The termination condition for BisimH is that the final model satisfies all the invariants mined from the log. The maximal model trivially satisfies all those invariants. Therefore, BisimH either terminates with that model or a smaller model that also satisfies the invariants. □

Theorem 2.3 (False Invariant Unsatisfiability)**.** *Let L be a log with event instances $\hat{E}$ and event types E. Let $M = \langle M_V, M_A \rangle$ be a relational model over $\hat{E}$ with a family of relations R_r. Let i be an event invariant that is not true of L. Then M does not satisfy i.*

Proof of Theorem 2.3. Since i is not true for L, there must be a trace $l \in L$ for which i is not true. Since there exists a path in M corresponding to l, that path must start with *START* and end with *END* and must not satisfy i. Therefore, M does not satisfy i. □

2.3.4 Improving model search efficiency

The previous section proved that regardless of which partitions BisimH splits, the algorithm always finds a model that satisfies exactly the log invariants. This is an important theoretical result, but a splitting strategy must be efficient in practice since refinement is expensive — a log with dozens of event types will generally satisfy hundreds of invariants of the types we are considering. Checking these invariants is costly, especially when the model grows to a large size. In this section, we prove that once BisimH satisfies an invariant, it never again violates it (Theorem 2.4). Therefore, invariants that have been satisfied do not need to be re-checked in finer models. This reduces the number of model checking runs BisimH needs to perform, making it more efficient.

Theorem 2.4 (Invariant Preservation)**.** *Let L be a log with event instances $\hat{E}$ and event types E. Let $M = \langle M_V, M_A \rangle$ be a relational model over $\hat{E}$ with a family of relations R_r. Construct a new relational model $M' = \langle M'_V, M'_A \rangle$ as follows:*

1. *Select one partition $p \in M_V$, $|p| \geq 2$.*
2. *Split p into two nonempty partitions p' and p''.*
3. *Let $M'_V = (M_V \setminus \{p\}) \cup \{p', p''\}$.*
4. *Compute M'_A using the family of relations R_r.*

For all event invariants i:

$$[(M \textit{ satisfies } i) \;\wedge\; (i \textit{ is true for } L)] \;\Rightarrow\; M' \textit{ satisfies } i.$$

It is an immediate corollary that M' satisfies all invariants that M does and that are true in the log.

Proof sketch: The proof considers the differences between M and M', and relies on the fact that these differences are confined to the region around the refined partition p. Consider some path Π' in M' that might violate invariant i. That path is made up of edges, each of which comes from some trace, which means for each edge, there is a corresponding edge in M. Therefore, there is a corresponding path Π in M that goes through partitions of the same event types. Since the event types along both paths are identical, then either both or none of the paths satisfy i. But since M satisfies i, so must M'.

Proof of Theorem 2.4. Without loss of generality, let $\langle a, b \rangle$ be the pair of event types that i relates. We will now show that all paths in M' must satisfy i.

Consider a complete path Π' in M'. For any two partitions connected by an edge in Π' there must exist at least one pair of event instances, one in each partition, that is related by some relation in some trace. For each edge in Π' choose such a pair of event instances to construct a sequence of event instances s.

Now consider the unique path Π in M that corresponds to s. Every partition in Π contains event instances of the same type as the corresponding partition in Π' (in fact, Π's partition is a superset of Π''s).

Assume Π' violates i. Consider three cases:

Case 1: i is $a \rightarrow b$. There must be some partition in Π' with an event instance of type a such that no subsequent partition in Π' contains an event instance of type b. Therefore, no subsequent partition in Π contains an event instance of type b. But M satisfies i. Contradiction.

Case 2: i is $a \not\rightarrow b$. There must be some partition in Π' with an event instance of type b that follows a partition with an event instance of type a. Then Π must also violate i. Contradiction.

Case 3: i is $a \leftarrow b$. There must be some partition in Π' with an event instance of type b such that no earlier partition in Π' contains an event instance of type a. Then no earlier partition in Π contains an event instance of type a. But M satisfies i. Contradiction.

□

2.3.5 Maintaining a small model size

Synoptic's aim is to present to a developer the smallest model (fewest nodes) satisfying the mined invariants. Large models are often too complex and no better than the raw log. In this section, we explain how the CEGAR [32] approach leads BisimH towards concise models. We argue that refinement always makes provable progress towards satisfying an invariant. Therefore BisimH rarely performs splits that make the model larger than it needs to be.

As a reminder, the CEGAR approach (detailed in Section 2.2.4) works as follows. First, BisimH generates a counterexample trace (e.g., Figure 2.6) that is accepted by the model and violates a mined invariant (*valid-coupon* $\not\rightarrow$ *invalid-coupon*). BisimH then traces along the counterexample trace in the model and in the input traces to find the longest prefix of partitions that exists as a sequence of corresponding event type instances in at least one input trace. The last partition in this prefix is refined (Figure 2.7). Two cases are possible:

Case 1: The refined model does not accept the counterexample trace. Consider the set of trace equivalence classes: two traces are in the same class if the paths of the two traces in the model are equivalent after removing all iterations through loops in the model. A split that eliminates one loop-free trace from an equivalence class, eliminates all traces in that class. Thus, eliminating a counterexample always eliminates an entire class of counterexamples that violate the invariant. Since there are a finite number of loop-free paths in a model, eliminating a class makes progress toward satisfying the invariant.

Case 2: The refined model accepts the counterexample trace. Consider the prefix corresponding to the counterexample trace in the refined model. This prefix is shorter than the previous prefix by at least one partition (the partition that was refined). Because any prefix must be finite, the refinement makes progress toward eliminating the counterexample trace from the model (towards Case 1).

2.3.6 BisimH as a model space exploration algorithm

The BisimH algorithm can be thought of as a strategy for exploring a large space of potential models. Figure 2.9 represents this process visually. The figure demonstrates how BisimH uses the mined invariants to take a specific path through the space of potential models that fit the observed log.

2.4 Tool implementation

We implemented and deployed Synoptic as a web service that is accessible through a browser, and as a stand-alone desktop application Synoptic that can be downloaded and run on a user's personal computer. These two approaches provide different trade-offs, and we support both for greater flexibility.

Web service. A web service (Figure 2.10) allows us to transparently update the code and to improve the user's experience without requiring users to download a new software version. Another important benefit is that we can transparently parallelize many of the Synoptic algorithms on the back-end, thus providing users with better performance than if Synoptic were to run on a single machine. Users can also more easily share Synoptic output with others (e.g., by sharing a URL). We also provide users with the option of downloading and running a Synoptic web service instance of their own.

Desktop application. We also have a fully functioning and platform-independent desktop version of Synoptic, which can be invoked from a command line and via a GUI. Figure 2.11 shows a screenshot of the GUI with the model from Figure 2.1. One drawback of a web service is the latency

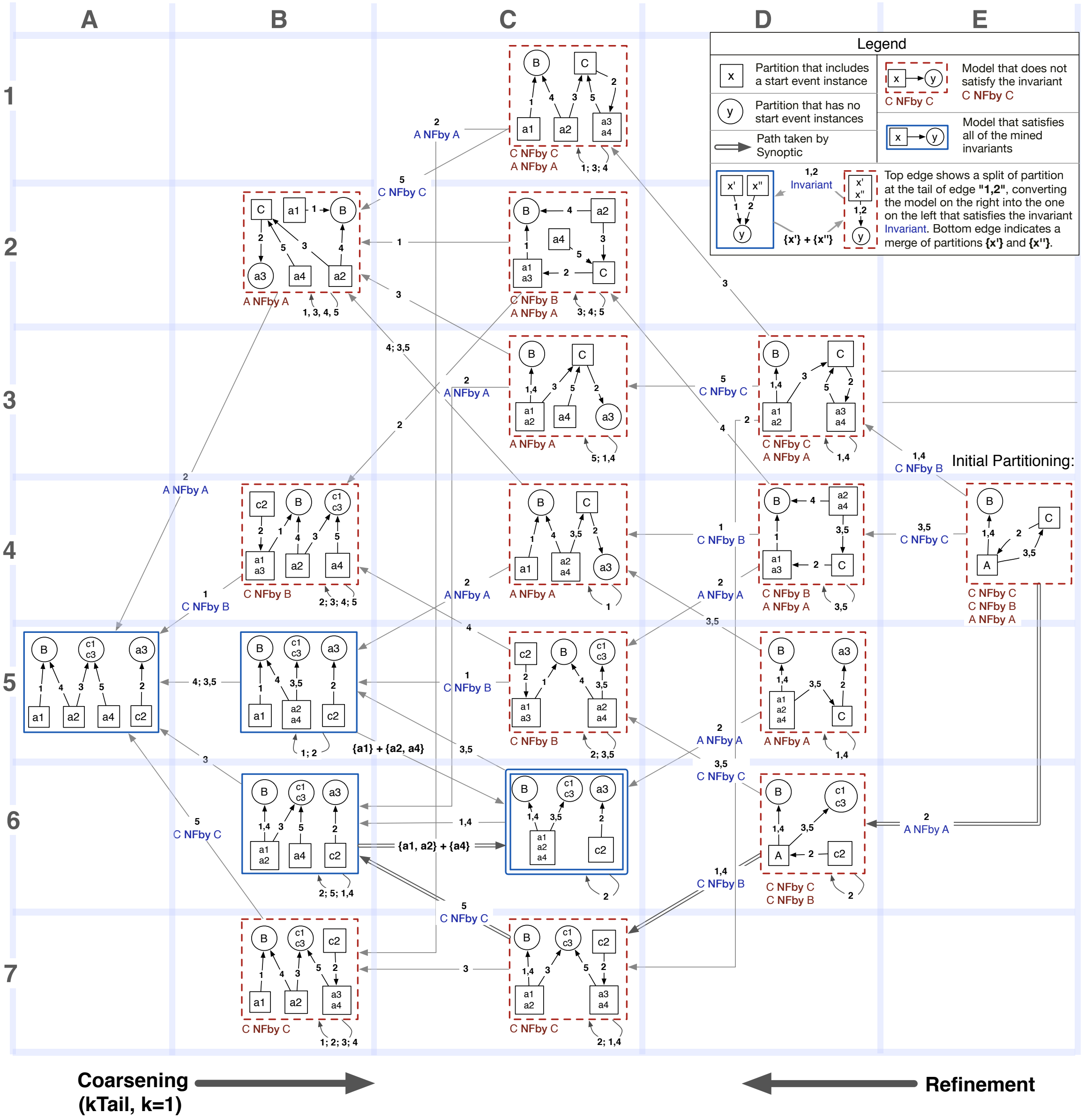
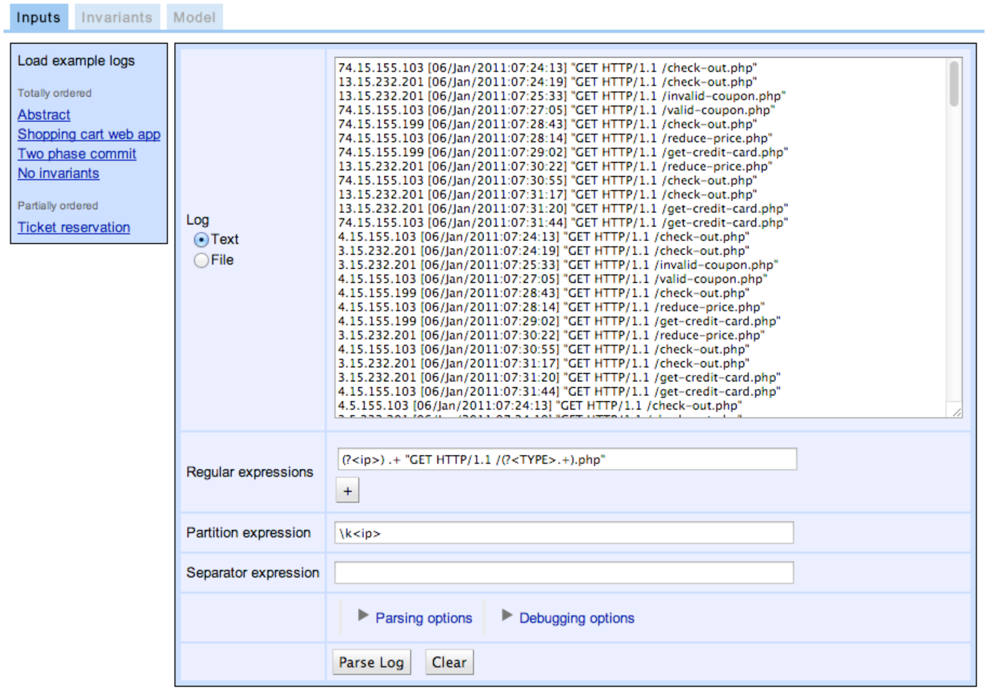

A
B
C
D
E
1
2
3
4
5
6
7
Legend
Partition that includes a start event instance
Partition that has no start event instance
Path taken by Synoptic
Model that does not satisfy the invariant
C NFby C
Model that satisfies all of the mined invariants
Top edge shows a split of partition at the tail of edge "1,2", converting the model on the right into the one on the left that satisfies the invariant Invariant. Bottom edge indicates a merge of partitions {x'} and {x''}.
1,2
Invariant
{x'} + {x''}
Initial Partitioning:
C NFby C
C NFby B
A NFby A
2 A NFby A
5 C NFby C
1 C NFby B
3,5 C NFby C
1,4 C NFby B
{a1} + {a2, a4}
{a1, a2} + {a4}
4; 3,5
Coarsening (kTail, k=1)
Refinement

Figure 2.9: A depiction of BisimH as a model-space exploration algorithm. BisimH has two phases: **refine** (right to left) from the initial partitioning at E4 until a representation satisfying all the mined invariants at B6; **coarsen** (left to right) towards the final, more compact, representation at C6, which retains the satisfied invariants. This final model is presented to the user. The most compact valid representation at C6 cannot be reached by refinement alone when starting at E4.



Log
● Text
○ File

```
74.15.155.103 [06/Jan/2011:07:24:13] "GET HTTP/1.1 /check-out.php"
13.15.232.201 [06/Jan/2011:07:24:19] "GET HTTP/1.1 /check-out.php"
13.15.232.201 [06/Jan/2011:07:25:33] "GET HTTP/1.1 /invalid-coupon.php"
74.15.155.103 [06/Jan/2011:07:27:05] "GET HTTP/1.1 /valid-coupon.php"
74.15.155.199 [06/Jan/2011:07:28:43] "GET HTTP/1.1 /check-out.php"
74.15.155.103 [06/Jan/2011:07:28:14] "GET HTTP/1.1 /reduce-price.php"
74.15.155.199 [06/Jan/2011:07:29:02] "GET HTTP/1.1 /get-credit-card.php"
13.15.232.201 [06/Jan/2011:07:30:22] "GET HTTP/1.1 /reduce-price.php"
74.15.155.103 [06/Jan/2011:07:30:55] "GET HTTP/1.1 /check-out.php"
13.15.232.201 [06/Jan/2011:07:31:17] "GET HTTP/1.1 /check-out.php"
13.15.232.201 [06/Jan/2011:07:31:20] "GET HTTP/1.1 /get-credit-card.php"
74.15.155.103 [06/Jan/2011:07:31:44] "GET HTTP/1.1 /get-credit-card.php"
4.15.155.103 [06/Jan/2011:07:24:13] "GET HTTP/1.1 /check-out.php"
3.15.232.201 [06/Jan/2011:07:24:19] "GET HTTP/1.1 /check-out.php"
3.15.232.201 [06/Jan/2011:07:25:33] "GET HTTP/1.1 /invalid-coupon.php"
4.15.155.103 [06/Jan/2011:07:27:05] "GET HTTP/1.1 /valid-coupon.php"
4.15.155.199 [06/Jan/2011:07:28:43] "GET HTTP/1.1 /check-out.php"
4.15.155.103 [06/Jan/2011:07:28:14] "GET HTTP/1.1 /reduce-price.php"
4.15.155.199 [06/Jan/2011:07:29:02] "GET HTTP/1.1 /get-credit-card.php"
3.15.232.201 [06/Jan/2011:07:30:22] "GET HTTP/1.1 /reduce-price.php"
4.15.155.103 [06/Jan/2011:07:30:55] "GET HTTP/1.1 /check-out.php"
3.15.232.201 [06/Jan/2011:07:31:17] "GET HTTP/1.1 /check-out.php"
3.15.232.201 [06/Jan/2011:07:31:20] "GET HTTP/1.1 /get-credit-card.php"
4.15.155.103 [06/Jan/2011:07:31:44] "GET HTTP/1.1 /get-credit-card.php"
4.5.155.103 [06/Jan/2011:07:24:13] "GET HTTP/1.1 /check-out.php"
```

Regular expressions: `(?<ip>) .+ "GET HTTP/1.1 /(?<TYPE>.+).php"`

Partition expression: `\k<ip>`

Separator expression:

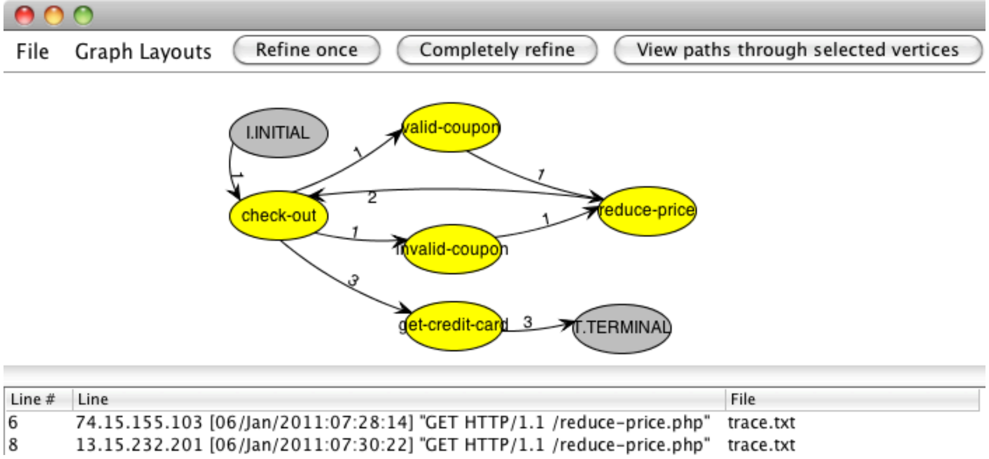
Figure 2.10: Screenshot of the inputs page of the web service version of Synoptic.

associated with uploading large logs — the user may want a quick analysis of a log that is local. Users with proprietary logs may also want to avoid a public web service. The Synoptic desktop application is useful in both of these cases.

Next, we detail how a user interacts with Synoptic to interpret a log. We focus on the Synoptic web service as it includes more features than the desktop version.

2.4.1 Model exploration

Synoptic presents users with interactive models. Users can tweak and explore them in pursuit of goals ranging from a more complete understanding of their system to identifying the source of unexpected behavior. A number of features are available to aid users in manipulating the models for these purposes.



Line #	Line	File
6	74.15.155.103 [06/Jan/2011:07:28:14] "GET HTTP/1.1 /reduce-price.php"	trace.txt
8	13.15.232.201 [06/Jan/2011:07:30:22] "GET HTTP/1.1 /reduce-price.php"	trace.txt

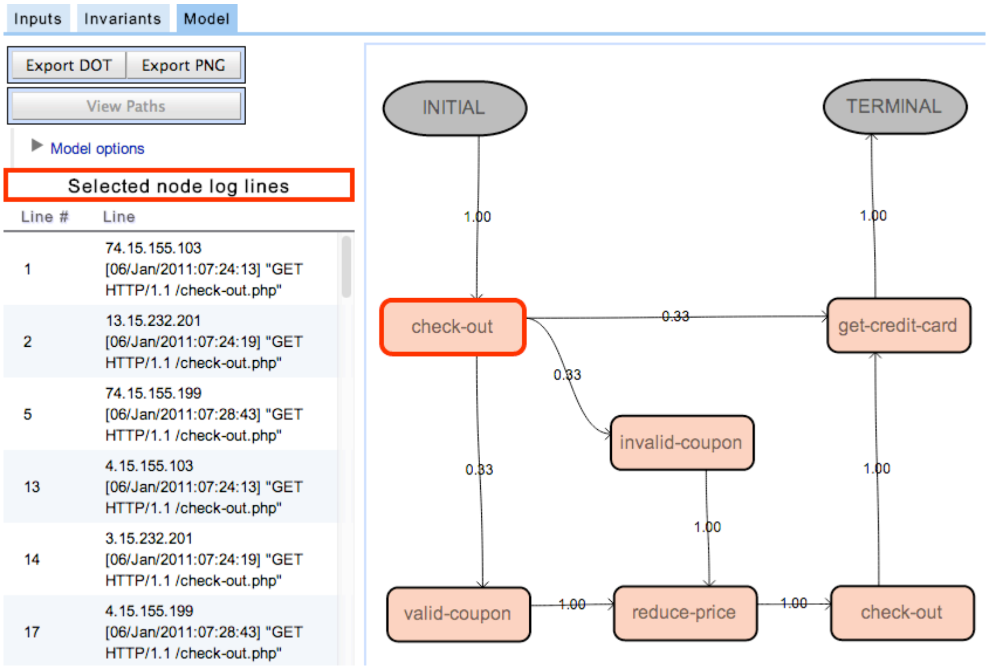
Figure 2.11: Screenshot of the Synoptic desktop application showing the model from Figure 2.1.

Matching abstract and concrete information

The user can select a partition in the model and view the log lines that correspond to this partition (e.g., *check-out* node in Figure 2.12). This is useful when the user wants to unpack the partition and identify the set of events that were actually logged at this point. The user can use this information to browse to a specific line in the log file that contains the interesting event. This operation is a kind of drilling down, mapping abstract information in the model to concrete events in the log.

Synoptic models are generative — they may accept traces that are not present in the input log. A user may want to know if a trace accepted by the model was observed in the log or not. For example, to a user, a generated trace may resemble buggy behavior, and the user may want to know whether the behavior actually occurred (if so, the system contains a bug). If a generated trace is invalid, it indicates that the input log is incomplete. This may lead the user to expand the test suite to invalidate an overfitted temporal invariant, which both improves the test suite and allows Synoptic to exclude the invalid generated path from the model.

To help users distinguish these two kinds of traces, users may select multiple partitions and consider the set of concrete traces that pass through the selected partitions. Synoptic either lists all the observed traces that pass through these partitions or lets the user know that the selected sub-trace was not observed. For example, the user may select the top-left *check-out* node and the *invalid-coupon*



Line #	Line
1	74.15.155.103 [06/Jan/2011:07:24:13] "GET HTTP/1.1 /check-out.php"
2	13.15.232.201 [06/Jan/2011:07:24:19] "GET HTTP/1.1 /check-out.php"
5	74.15.155.199 [06/Jan/2011:07:28:43] "GET HTTP/1.1 /check-out.php"
13	4.15.155.103 [06/Jan/2011:07:24:13] "GET HTTP/1.1 /check-out.php"
14	3.15.232.201 [06/Jan/2011:07:24:19] "GET HTTP/1.1 /check-out.php"
17	4.15.155.199 [06/Jan/2011:07:28:43] "GET HTTP/1.1 /check-out.php"

Figure 2.12: Screenshot of the Synoptic web service showing the model from Figure 2.1.

node in the model in Figure 2.13, and find out that there is indeed an observed trace that passes through these nodes (the middle trace in Figure 2.2). This more advanced capability proved to be of particular use to developers in practice. It helped them understand unexpected paths in the model by exposing the associated concrete traces from the input log.

Invariant selection

Synoptic mines three types of temporal invariants (Section 2.2.2) to guide refinement. The choice of invariants is important because they constrain the executions a derived model may generate. By default, Synoptic uses all of the mined invariants. However, users may know that some invariants are false because the logs may not sufficiently represent all possible system behaviors. If this is the case, the user may mark some of the mined invariants as false so that Synoptic does not use them to over-fit the model to the log. Figure 2.14 illustrates a Synoptic screen with which the users can deselect some of the mined invariants.

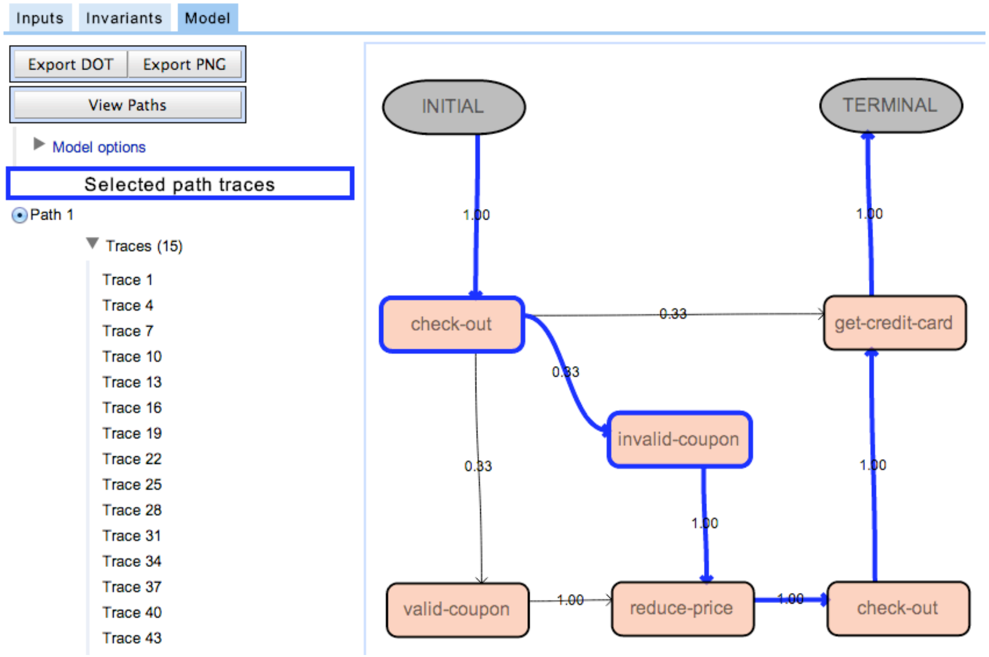


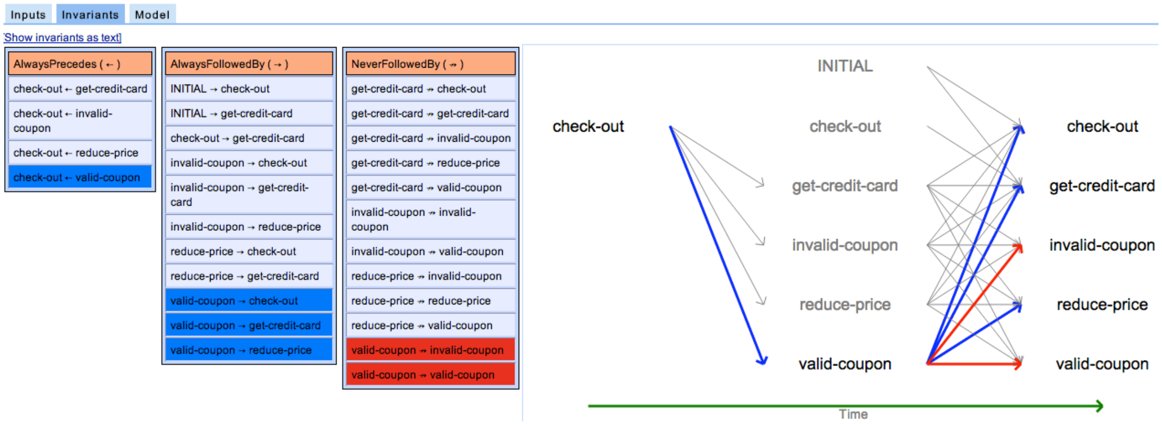
Figure 2.13: A screenshot that demonstrates path exploration functionality in a Synoptic web service on the model from Figure 2.1.

2.5 Experience with Synoptic

We performed a short proof of concept study and two detailed case studies to evaluate Synoptic's ability to produce concise and useful models. Our proof of concept study involved Peterson's leader election algorithm [102] (Section 2.5.1).

Of the more detailed studies, first, we carried out a user study with a developer working on the *reverse traceroute* system that determines the likely reverse route from an arbitrary destination on the Internet to a source host [14] (Section 2.5.2). Synoptic analyzed the coordinator node logs that contained debugging event instances generated by the system.

Second, we introduced Synoptic as a tool for use in an undergraduate distributed systems class of 45 students (Section 2.5.3). The students were tasked with designing and implementing a cache coherence protocol and had to (1) draw a finite state machine of their design, (2) run Synoptic on their implementation, and (3) explain any observed differences.



AlwaysPrecedes (←)
check-out ← get-credit-card
check-out ← invalid-coupon
check-out ← reduce-price
check-out ← valid-coupon

AlwaysFollowedBy (→)
INITIAL → check-out
INITIAL → get-credit-card
check-out → get-credit-card
invalid-coupon → check-out
invalid-coupon → get-credit-card
invalid-coupon → reduce-price
reduce-price → check-out
reduce-price → get-credit-card
valid-coupon → check-out
valid-coupon → get-credit-card
valid-coupon → reduce-price

NeverFollowedBy (↛)
get-credit-card ↛ check-out
get-credit-card ↛ get-credit-card
get-credit-card ↛ invalid-coupon
get-credit-card ↛ reduce-price
get-credit-card ↛ valid-coupon
invalid-coupon ↛ invalid-coupon
invalid-coupon ↛ valid-coupon
reduce-price ↛ invalid-coupon
reduce-price ↛ reduce-price
reduce-price ↛ valid-coupon
valid-coupon ↛ invalid-coupon
valid-coupon ↛ valid-coupon

Figure 2.14: Screenshot that illustrates the invariants page in the Synoptic web service. Users may use this page to inspect and deselect invariants to control how Synoptic derives the model. (Left) The three tables list all of the mined invariants. Users may deselect invariants by clicking on them in the table. (Right) A visualization of the mined invariants. The blue/red colors in both the tables and in the diagram highlight those invariants that involve the (moused-over) *valid-coupon* event type.

The students used Synoptic during development and testing, while the reverse traceroute developer used Synoptic on logs generated in production. We therefore believe that Synoptic can be helpful during all stages of a typical software engineering process. Overall, we found that Synoptic was useful for finding new bugs (Section 2.5.4), for increasing developer confidence (Section 2.5.5), and for building understanding (Section 2.5.6).

2.5.1 *Peterson leader election*

The Peterson leader election algorithm [102] allows an asynchronous unidirectional ring network of nodes to elect a leader. All nodes start as *active* and with a random unique node ID. In each round, at least half of the active nodes become *relays* through exchange and comparison of node IDs. A relay node forwards all messages it receives. When an active node receives its own messages via a ring of relay nodes, it becomes the *leader*.

We implemented a simulator of the Peterson algorithm that logs all messages sent and received by a node, as well as node state transition debug messages. This log includes a variety of message

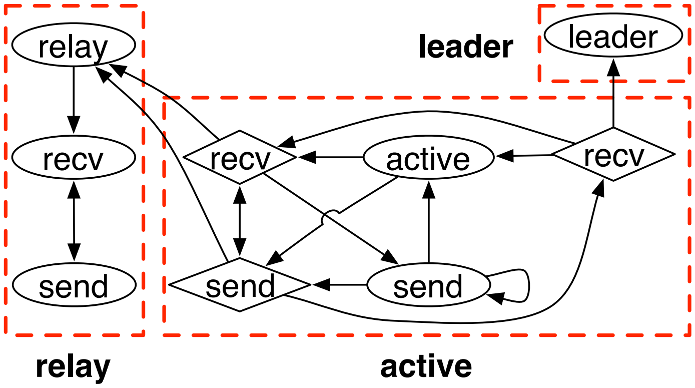


Figure 2.15: Synoptic's output for a Peterson log with 3,308 events, generated by simulating 5 nodes. The manually-added, labeled, dotted regions group nodes into the states a node may take on in the algorithm.

interleavings as the simulator allows concurrent node execution. Messages are timestamped and partially ordered using Lamport vector clocks [83]. Figure 2.15 shows Synoptic's output for an execution with 5 nodes that generated 3308 log events. This graph is useful in understanding node behavior as nodes take on different states. For example, Synoptic correctly captures the fact that a relay node cannot send before receiving while an active node may first send and then receive, depending on the timing of incoming messages.

2.5.2 Reverse traceroute study

Reverse traceroute [14] is a distributed system that determines the likely reverse traceroute from an arbitrary destination on the Internet to a source host. Reverse traceroute relies on a distributed set of Internet vantage points hosted by PlanetLab [4], and uses a variety of methods to find each segment of the reverse route, such as IP record route and timestamp options [1, 2], and relies on IP spoofing from PlanetLab hosts.

Reverse traceroute has been in live deployment for over 7 months and since that time it has had hundreds of distinct users and has handled a total of 3.6 million requests. Today, it gets tens of thousands of requests per day. Recently, a large, popular, and ubiquitous Internet company has

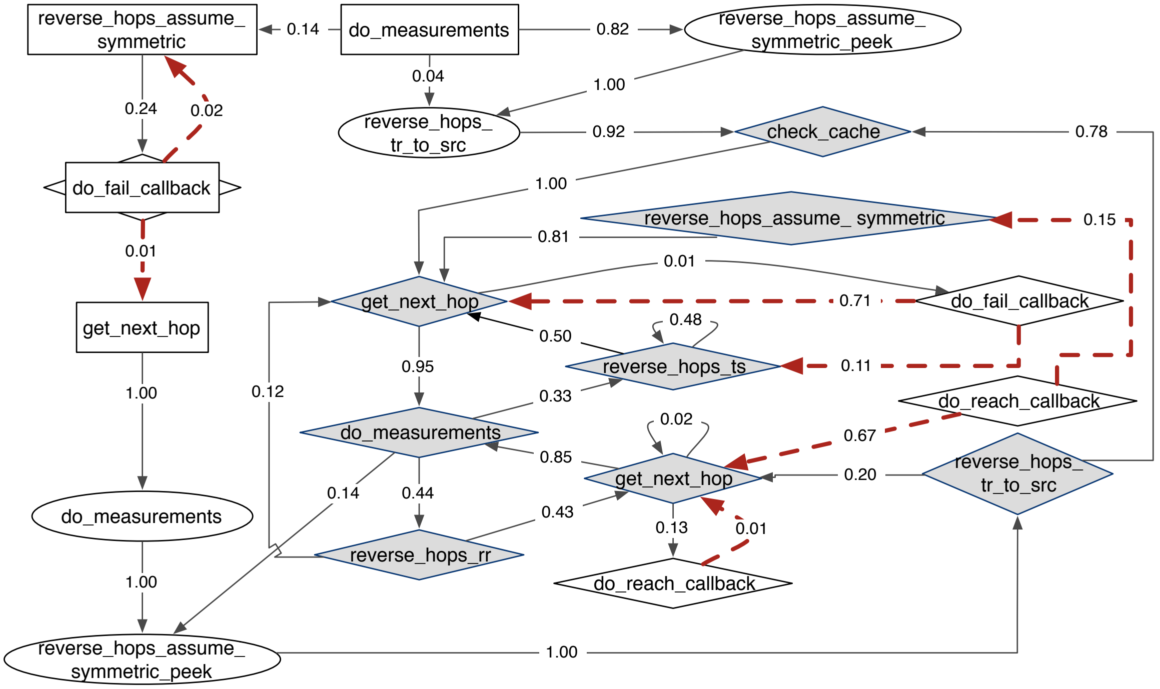


Figure 2.16: Synoptic model for a reverse traceroute input log of 900,000 event instances. Rectangular nodes are start nodes, and diamond nodes are terminal nodes. Edge labels indicate transition probability. For clarity certain edges and nodes are omitted. The (manually) bold, dashed lines indicate a new bug that was discovered by the developer. The (manually) shaded terminal nodes make up the set of methods exhibiting a bug known to the developer. Before viewing the Synoptic graph, the developer thought the bug affected only two of these eight nodes.

deployed the system internally.

We carried out a user study with a developer working on the system to study a log of 900,000 event instances. To generate this log, the developer spent a total of 15 minutes to add a total of 16 lines of logging code to the system. We then wrote four regular expressions to process the log. The log was divided into traces by measurement-based method names — a single trace corresponded to a sequence of method calls made to determine the reverse route for a particular ⟨source, destination⟩ pair.

Synoptic took 11.5 minutes to generate the final graph from the input log on an Intel i7 (2.8 GHz) OS X machine with 8GB of RAM. Because this graph contained many rare edges (i.e., edges with low transition probabilities), we showed the developer both the full graph, as well as a graph that omitted 62 edges with low transition probability. The second type of graph is shown in Figure 2.16. We then performed a talk-aloud user study with the developer by showing him Synoptic-derived graphs, explaining to him what they represent, and asking him to talk through his observations.

2.5.3 Distributed systems course assignment

In the University of Washington undergraduate distributed systems course[2], groups of 2–4 students designed and implemented a peer-to-peer Facebook-like social network. The project was divided into multiple assignments, one of which was to implement the distributed version of a cache coherence protocol [84] between a single master node and some number of replica clients. For this assignment, the students were to (1) record their design as a FSM diagram, (2) implement their design, (3) apply Synoptic to logs generated by their implementation, and (4) observe and explain any differences between the Synoptic output and their initial FSM diagram.

For testing, the students used a simulated environment in which all nodes executed in a single process, and communicated via a centralized simulator manager. The simulator provides the option of reordering, losing, and duplicating messages, as well as randomly failing and restarting nodes.

The simulator logged common event types like message sent, received, and lost and file read and written, and also allowed student node code to log user-defined event types. Although the simulated system was distributed, the simulator produced totally ordered logs — event instances were serialized through the central simulator manager. The students were also given a set of Synoptic regular expressions for processing logs generated by the simulator.

All 18 groups completed the assignment. The following sections quote just a few of the student reports on their experiences with Synoptic and showcase one of the Synoptic diagrams generated by the students (Figure 2.17).

[2] http://www.cs.washington.edu/education/courses/cse490h/11wi

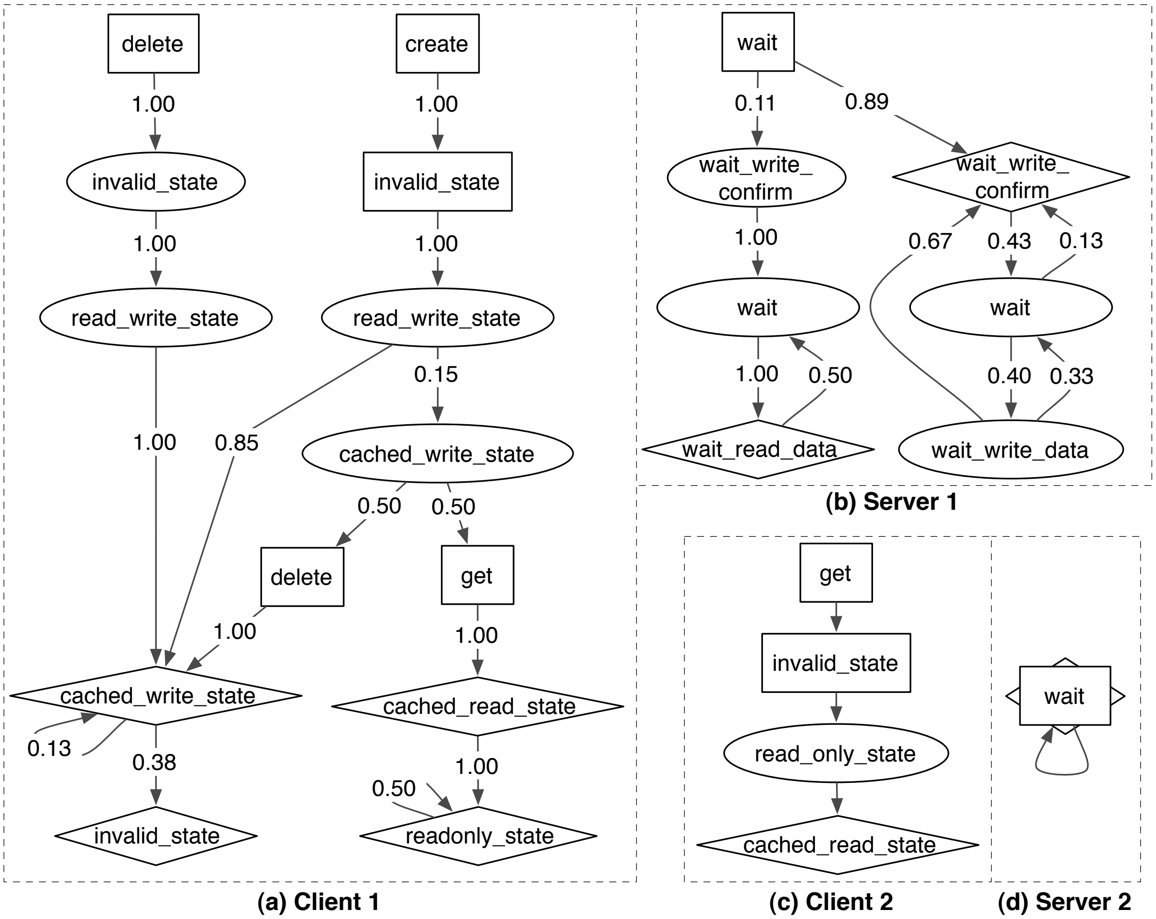


Figure 2.17: Two pairs of Synoptic models generated by a group of students in the distributed systems class for a distributed cache coherence assignment. Each pair of models has a server model and a client model. The **Client1** and **Server1** models correspond to a scenario in which the client host starts, and then deletes the file if it exists. Alternatively the client creates the file, and then either deletes it or reads from it. The **Client2** and **Server2** models correspond to the simpler scenario in which the client reads a file that is not currently accessed by any other client.

2.5.4 Finding bugs in code

Synoptic models capture event type orderings and co-occurrence frequencies among event types. The absence of an edge could indicate that the log is incomplete. However, if the behavior is supposed to occur at all times or with high frequency, an unexpected graph topology can be an indicator of a latent bug.

Reverse traceroute study The reverse traceroute developer identified one new and important bug using the Synoptic model within the first two minutes of seeing the model. All measurements made by the system must eventually terminate in either the `do_reach_callback` or the `do_fail_callback` methods. The developer thought that all traces reaching these methods terminated. The graph showed otherwise — some of the traces continued past these callbacks. The model in Figure 2.16 illustrates these buggy transitions with bold, dashed emphasis. The developer hypothesized that this bug is caused by concurrency in the measurement code. The developer also observed that the tool offers a light-weight means of verifying that some previously observed buggy behavior is not present after a bug fix, and that it may help to rule out bug fixes that fail to eliminate buggy behavior.

Distributed systems course Of the 18 groups, 3 groups found bugs in their implementations with Synoptic. Synoptic models effectively capture event type orderings and all three of the bugs had to do with illegal message orderings. One group observed that a transition that was expected never occurred — the node seemed to never execute the write command after processing it. They then fixed the bug and used Synoptic to confirm that it did not appear in the traces:

> “We did find a bug in the graph. If you follow the append path in the final graph you can see that it goes from append→send→write. In the old graph the append→send, but dies instead of passing it onto write.”

A different group found a bug in which they mistakenly sent the wrong type of packet:

> “We had few places where we sent the wrong type of packet in the code. For example, we sent RDC when we had to send WDC. When looking at the Synoptic diagram, these kinds of mistakes were easy to find.”

2.5.5 Increasing developer confidence

Synoptic models can succinctly represent thousands of execution traces with a few nodes in a graph. A single compact diagram that consolidates many executions gives developers confidence that they will not overlook any behaviors present in the log. Moreover, developers often recognized expected patterns and considered them to be important evidence that the system worked as expected.

Reverse traceroute study The reverse traceroute developer was often prompted by the model to try to explain various patterns. Patterns that were simpler and more noticeable, like self-loops on nodes, elicited more attention. For example, upon noticing a self-loop on one of the nodes the developer mentioned that this indicated that a specific type of measurement was re-tried and that this was correct behavior.

Distributed systems course Of the 18 groups in the class, 11 reported that Synoptic increased their confidence in their implementations. In many cases the students recognized expected patterns. Figure 2.17 illustrates two sets of diagrams generated by a group that felt that they acquired additional confidence in their system by using Synoptic. The figure shows four models, with each pair corresponding to a *Client* and *Server* processes in the system. The group decided to use a combination of messages and states for their event types — e.g., `create` is a file create request message sent by the client to the server, while e.g., `readonly_state` is the state of the client when it holds a shared read lock on the file. To more easily follow the sequence of event types the students generated two sets of models for two distinct scenarios (see Figure 2.17). By inspecting these models, the group confirmed that the messages were exchanged in the appropriate order and that the nodes transitioned between states correctly.

The following are some student quotes that indicate that Synoptic increased developers' confidence in their systems:

> "[Synoptic] definitely let us know for sure that our code was functioning correctly."

> "Using Synoptic did not help us find any bugs with our code, but it did help us to clarify that our code is doing what it should."

> "We can confidently say that Synoptic helped confirm the correct behavior of our program, and certainly

> made us feel better about our code."

2.5.6 Building system understanding

Using the Synoptic model, the reverse traceroute developer was able to solidify his understanding about the system. For example, he knew that the system had a bug in which a reverse traceroute measurement terminates prematurely. Using the model, he was able to verify that this bug occurred — methods terminating prematurely appeared as terminal nodes in the model. However, as it turned out, the developer did not understand the full extent of this bug. He assumed that it affected only two methods. By inspecting the model, he found out that other methods were impacted as well. The model in Figure 2.16 illustrates the set of all the methods impacted by this known bug with a darker shading. This experience solidified the developer's understanding of where the bug manifested and he felt better prepared to resolve it.

Overall we found that Synoptic was useful for finding new bugs, for increasing developer confidence, and for building systems understanding.

2.5.7 Threats to validity

Our two user studies are limited in scope and have a number of inherent biases for which we were unable to control. The reverse traceroute system has been developed by about five developers, all of whom understand the entire system. Consequently, Synoptic models are straightforward for them to interpret. Developers who are new to a project or are working on a larger project may find it difficult to interpret Synoptic models, which may be larger and more complex. However, we believe that Synoptic may also be used to gain insight into components of larger systems, and because most sizable systems are developed in a modular fashion, there may still be value in using Synoptic in large projects. Lastly, our study with the developer implicitly emphasized bug findings, which may have primed the developer into thinking more about bugs. In a different context, he might have been less successful in identifying bugs.

Because the students in the distributed systems course were required to use Synoptic as part of the assignment, it is unknown whether they would have been motivated enough to learn about and use the tool without a mandate. Students might have also been attempting to please us and thereby reported only positive experiences with the tool. Finally, students are not representative of experienced developers and we do not know whether the bugs they found using Synoptic are problems for expert developers.

2.6 Discussion

While working on Synoptic, we observed a number of its limitations. Here, we detail the most important of these and connect some of them to our future work.

Applicability. Synoptic models capture ordering relationships between events observed in a log. It does not handle algebraic and logical relationships that may also be useful in modeling software (e.g., this.next $\neq$ this.prev). Synoptic is therefore best suited for studying logs of systems whose execution can be modeled as a sequence of elements, with the ordering, the presence, and absence of elements encoding some useful semantics about the system. We have observed that Synoptic can help with problems whose root causes can be deciphered using such semantic information. More advanced issues, however, would require richer and more complex models than Synoptic currently provides.

Invariants. Synoptic relies on three temporal invariants to determine when to terminate and how to proceed during refinement. A rigorous evaluation of the limitations and advantages of these invariant types is necessary. For example, we know that the invariants constrain Synoptic models in ways that are sometimes undesirable. For instance, the $\not\to$ invariant constrains Synoptic models to be less generative: e.g., if $a \not\to b$ is true, then the model is restricted from generating a path between a and b, even though this behavior might be valid and can appear in an execution that is not present in the input log. However, we do not know what kinds of systems or uses these invariants favor, and whether we should expand this set, or make it smaller.

Synoptic invariants are temporal. They do not involve the *data values* that are often present in logs. Extending Synoptic to mine and then preserve value-based invariants is a part of our future work.

Filtering rare and common behavior. Sometimes Synoptic-generated models are large and contain more information than is necessary. For example, a developer may be interested in a section of the model when seeking to pinpoint a rarely-occurring behavior. To support this use-case, Synoptic allows the developer to filter out high/low probability edges from her view of the model. More generally, the user may be interested in high/low probability traces admitted by the model. Synoptic lets the user select a start and end node, and specify the maximum/minimum path probability to use for hiding all paths between the two nodes that have a path probability outside of the desired range.

Comparing models. A common use-case for Synoptic models is to study how the models change with different log inputs. Log inputs may differ because of additional executions in the log, a change to the mined set of invariants, a modification to the codebase, or because of added or removed logging statements.

Synoptic displays two models side by side and highlights their differences. These may be topological (e.g., the node count is different), or statistical (e.g., the transition probability of certain edges may have increased/decreased). By studying model differences, developers can check whether system behavior is the same or different — either over different traces or different system settings.

Reliance on logs. To work well, Synoptic needs the input log to include as many different system executions as possible. This is because Synoptic models are at most as detailed as the input logs. If the user failed to log an important behavior, then this behavior will usually not be present in the Synoptic-generated model. However, generating all possible system behaviors is notoriously difficult, and may be infeasible, as illustrated by the following student quote:

> "However, we had to run specific simulation cases in order to produce the log, so while Synoptic was very useful, most of the debugging process involved trying commands in the simulator. We knew what cases we were testing, so running them through the terminal was an easier way to test for the bug. But Synoptic did confirm that we have the right message flows."

Fault localization/advanced tool support. The feasibility of fault localization using Synoptic-generated models depends on the density and quality of the logging statements. Reconstructing execution paths based on logs is an active research area (e.g., SherLog [136]), and we hope to leverage this existing work in developing more automated fault localization techniques. However, fault localization fundamentally requires human insight. To this end, the existing Synoptic web service can be extended with more advanced functionality. For example, we would like to allow users to work with collections of models of varying abstraction, to deal with evolving log formats, and ultimately make a better link between the inferred models and the underlying system code.

2.7 Summary

Logging is a popular debugging methodology. Unfortunately, large logs are often complex and difficult to analyze manually. This chapter presented the design and evaluation of a tool called Synoptic, which builds a system model from the system's execution logs. Unlike other tools, Synoptic requires few inputs from the developer and can be applied to pre-existing logs.

The key to Synoptic's algorithm is its use of three types of mined temporal invariants to guide the model space exploration. Our formal evaluation showed that Synoptic's algorithm always makes progress and always finds a model that satisfies the mined invariants. Our case studies showed that Synoptic graphs improved developer confidence in the correctness of their systems, and were useful for finding existing, as well as previously unknown bugs.

Synoptic bridges the gap between systems developed by developers with little to no training in formal methods, and a suite of methods developed by the formal methods community.

Chapter 3

Dynoptic: inferring models of networked systems

Most model inference algorithms, including Synoptic, focus on inferring models of *sequential* systems. This requires that the executions recorded in the log are totally ordered — for every pair of events in an execution, one precedes the other. A total order has important advantages, such as the ability of humans to directly inspect and understand the logged information and the applicability of simple and powerful log analysis techniques, such as Synoptic. Further, many well-established logging formats are totally ordered.

However, in domains where an execution may be concurrent, such as in distributed or networked systems, events are partially ordered, and do not have a total order. As a result, most model inference work cannot be applied to infer a model of a distributed or a *networked system* in which events at different nodes may occur without any happens-before relationship [83].

Concurrency is increasingly used to improve performance in clusters and in multi-core architectures. Even simple applications are becoming concurrent, with more functionality migrating into the cloud and with widespread use of Ajax to mask latency. It is therefore important to develop model inference techniques and tools to support developers who work on concurrent system.

This chapter describes a model inference technique and a corresponding tool, called Dynoptic, which infers a communicating finite state machine (CFSM) [27] model of the processes that generated the log. This model type is especially well suited to describing networked systems — systems in which a number of processes communicate over channels. For example, Dynoptic can be used to model distributed systems, protocols, execution of AJAX events in a web-browser, and other behavior traces that record events at multiple communication processes.

Dynoptic models can support developers with a variety of tasks — developers can inspect, query, and check Dynoptic's models against their own mental model of the system. These uses are the focus of this chapter. However, Dynoptic-generated models also support other use-cases, such as model-based testing of networked systems, and detection of anomalous behavior as systems are

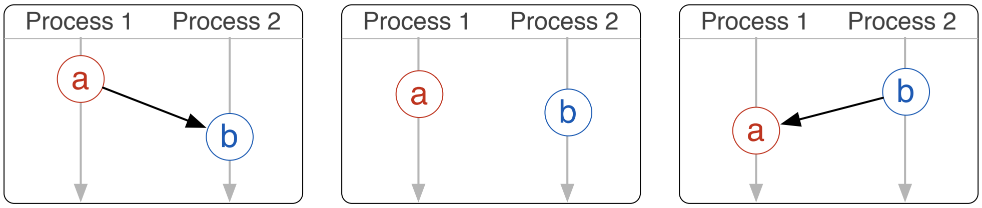


Figure 3.1: Three partial orderings, represented as time-space diagrams, that are equally implied by a serialized (totally ordered) log containing two events — *a* and *b*, in this order. This illustrates the ambiguity of a totally ordered log in a distributed setting.

exposed to new workloads or environments.

To evaluate Dynoptic we applied it to logs of three different systems — the stop-and-wait protocol, opening/closing handshakes of TCP, and the replication strategy in the Voldemort [125] distributed hash table. Second, we performed a user study with a class of 39 undergraduates to evaluate the efficacy of CFSM models in finding bugs. This evaluation indicates that Dynoptic produces models that are accurate and useful in finding implementation bugs.

The next section introduces partial order in the context of networked systems, discusses how it can be tracked, and discusses time-space diagrams. Section 3.2 introduces and motivates the CFSM formalism. Section 3.3 overviews the Dynoptic approach. Then, Sections 3.4 – 3.11 formally define the Dynoptic algorithm. Section 3.12 proves that Dynoptic returns models that are accurate, in the sense that they satisfy the mined log properties we define. Section 3.13 presents an evaluation of the Dynoptic tool on logs generated by three systems. Section 3.14 discusses issues raised by our work on Dynoptic, and Section 3.15 provides a summary of this chapter.

3.1 Partial order in networked systems

A partially ordered log captures concurrency information explicitly, making it more useful than a totally ordered log for studying concurrent systems. For example, consider Figure 3.1, which illustrates this extra information with a simple system of two communicating processes. Process 1 generates an *a* event, and process 2 generates a *b* event. If these two events are serialized into a totally ordered log in the order *a* and then *b*, then then it is unclear whether *a* preceded *b* or just happened to be logged before *b*. Even if *a* and *b* appear in different orders in multiple traces, it is still possible that they can never occur concurrently. Figure 3.1 shows the three possible partial orderings. Unlike a totally ordered log, a partially ordered log of these two events will make concurrency explicit: the partially ordered log will indicate whether *a* and *b* have some order, in which case they are not concurrent, or if there is no ordering between the two events and the two events are concurrent. A partially ordered log will be able to distinguish between the three scenarios in Figure 3.1.

Next we explain time-space diagrams — a common technique for visualizing a partial order. Then, we describe how a partial order can be tracked in a distributed setting[1].

3.1.1 Visualizing a partial order as a time-space diagram

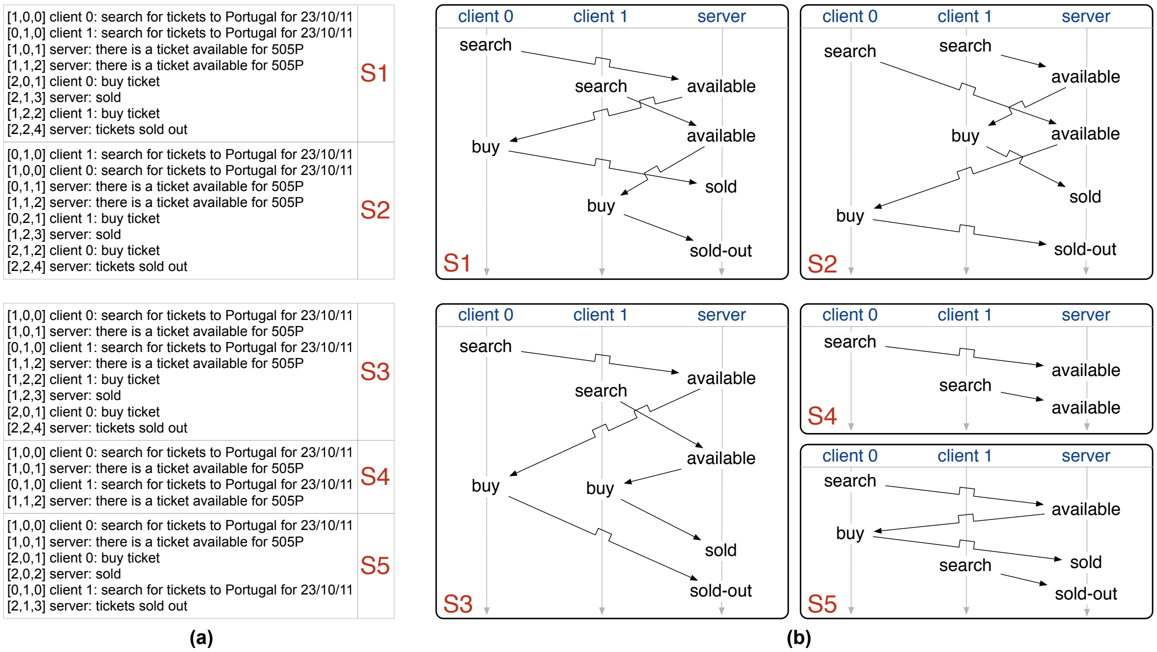
A *time-space diagram* is a representation of a single distributed execution and can be used to understand a single trace in a partially ordered log. In a time-space diagram there is one timeline for each host in the system (with time proceeding visually down). Local events for a host are plotted along the host's timeline, and communication events (i.e., message sends and receives) are represented as arrows that connect the timelines of the communicating hosts at points of time when they communicated.

Figure 3.2(a) is a log of traces with vector times and events from a system with three hosts. In this system two clients concurrently buy tickets from a ticket server that has just one ticket available. The log captures several scenarios in which the clients and the server interact: different orders of checking for ticket availability, attempts to buy a ticket, and so on. This log is partially ordered because the clients issue requests independently.

Figure 3.2(b) shows five time-space diagrams[2] representing the five partially ordered traces in Figure 3.2(a). In each time-space diagrams the two left-most timelines (grey vertical lines) depict the events executed at the clients — client 0 and client 1; the third timeline shows events executed by the ticket server. An arrow between two host timelines in a time-space diagram denotes communication

[1] This explanation corresponds to a system that uses message passing. Vector timestamps can also be used for ordering event instances in a system that uses other mechanisms for inter-host communication, such as shared memory.

[2] For compactness, the diagrams in Figure 3.2(b) bundle message receipt, message processing, and message send events into one event.



```
[1,0,0] client 0: search for tickets to Portugal for 23/10/11
[0,1,0] client 1: search for tickets to Portugal for 23/10/11
[1,0,1] server: there is a ticket available for 505P
[1,1,2] server: there is a ticket available for 505P          S1
[2,0,1] client 0: buy ticket
[2,1,3] server: sold
[1,2,2] client 1: buy ticket
[2,2,4] server: tickets sold out

[0,1,0] client 1: search for tickets to Portugal for 23/10/11
[1,0,0] client 0: search for tickets to Portugal for 23/10/11
[0,1,1] server: there is a ticket available for 505P
[1,1,2] server: there is a ticket available for 505P          S2
[0,2,1] client 1: buy ticket
[1,2,3] server: sold
[2,1,2] client 0: buy ticket
[2,2,4] server: tickets sold out

[1,0,0] client 0: search for tickets to Portugal for 23/10/11
[1,0,1] server: there is a ticket available for 505P
[0,1,0] client 1: search for tickets to Portugal for 23/10/11
[1,1,2] server: there is a ticket available for 505P          S3
[1,2,2] client 1: buy ticket
[1,2,3] server: sold
[2,0,1] client 0: buy ticket
[2,2,4] server: tickets sold out

[1,0,0] client 0: search for tickets to Portugal for 23/10/11
[1,0,1] server: there is a ticket available for 505P          S4
[0,1,0] client 1: search for tickets to Portugal for 23/10/11
[1,1,2] server: there is a ticket available for 505P

[1,0,0] client 0: search for tickets to Portugal for 23/10/11
[1,0,1] server: there is a ticket available for 505P
[2,0,1] client 0: buy ticket                                   S5
[2,0,2] server: sold
[0,1,0] client 1: search for tickets to Portugal for 23/10/11
[2,1,3] server: tickets sold out
```

Figure 3.2: (a) Five system traces (S1-S5), each comprised of log lines and corresponding vector clock timestamps for a web application that sells airplane tickets. In the traces, two clients access a single server. **(b)** A visualization of the five system traces as time-space diagrams. Time flows down, and events at each host are shown in a single column.

— the event at the start of an arrow is a send message event, and the event that an arrow points to is a receive message event.

Informally, the vertical space between events represents time. More formally, the time-space visualization captures the ordering of events in a distributed setting. Two events in the system are ordered if there is a path (along all the edges in the diagram — both grey and black) from one event to the second, that never goes upward. For example, in system trace S2 in Figure 3.2, the send of `buy` at client 1 happens before the `sold` at the server. Two events are considered to execute concurrently if there is no such path between the two events. For example, in all of the executions in Figure 3.2(b) the `buy` event at client 0 and the `buy` event at client 1 are always concurrent, while the events `sold` and `sold-out` at the server are always ordered.

As another example, in system trace S1 a `search` event instance at client 0 has a timestamp of $[1,0,0]$, which immediately precedes the first `available` event instance at the server, timestamped with $[1,0,1]$. The time-space diagram encodes this precedence information as a directed edge between the two event instances. However, in S1 the same `search` event instance at client 0 is not ordered with the `search` event instance at client 1, which has a timestamp of $[0,1,0]$. Correspondingly, there is no path in the time-space diagram between these two event instances.

Time-space diagrams are frequently drawn by developers to understand a distributed execution. We therefore compare the efficacy of CFSM models inferred by Dynoptic against time-space diagrams. Section 3.13.5 reports on user study that evaluates the two representations.

3.1.2 Tracking partial order with vector time

Vector time [52, 96] is logical clock mechanism for tracking event ordering in a concurrent system. For this, each logged event in a trace is associated with a vector timestamp. These timestamps can then be used to reconstruct the partial ordering of events.

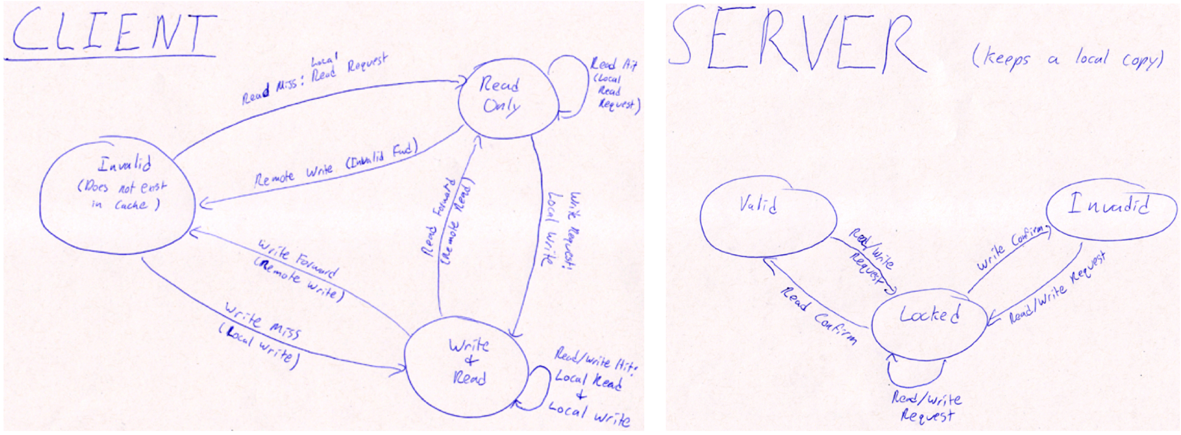
Formally, in a distributed system of h hosts, each host maintains an array of clocks $C = [c_0, c_1, \ldots, c_{h-1}]$, in which a clock value c_j records the local host's knowledge of (equivalently, dependence on) the local logical time at host j. We denote a timestamp's C clock value for host j as $C[j]$.

When generating an event, the host increments its own clock in the array. When sending a message to another host, the sender shares its clocks array with the receiver, who updates its local clocks array to the most recent clock values (and increments its local clock, as message receipt is an event). More formally:

1. All hosts start with an initial vector clock value of $[0, \ldots, 0]$.

2. When a host i generates an event instance, it increments its own clock value (at index i) by 1, i.e., $C_i[i]++$.

3. When host h communicates with host h', h shares its current clock C_h with h', and h' updates its local clock $C_{h'}$ so that $\forall i$, $C_{h'}[i] = \max(C_h[i], C_{h'}[i])$. h' also updates its local clock value as in (2), since message receipt is considered an event.


CLIENT
Read Miss: Local Read Request
Read Only
Read Hit (Local Read Request)
Invalid (Does not exist in Cache)
Remote Write (Invalid Fwd)
Read Forward (Remote Read)
Write Request: Local Write
Write Forward (Remote write)
Write Miss (Local write)
Write & Read
Read/write Hit: Local Read & Local write
SERVER (keeps a local copy)
Valid
Invalid
Read/Write Request
Read Confirm
Locked
Write Confirm
Read/Write Request
Read/Write Request

(a)

(b)

Figure 3.3: A mental model drawn by an undergraduate student for a systems course. The model depicts a client-server system with **(a)** the client process sub-model, and **(b)** the server process sub-model. The process models are represented as distinct finite state machine that process remotely generated events.

Note that in the above procedure we assume that the hosts know the number of participants (hosts) in the system, and that the set of participants does not change over time.

Using the above procedure, each event instance in the system is associated with a *vector timestamp* — the value of C immediately after the event instance occurred. Vector timestamps can be partially ordered with the relation $\prec$, where $C \prec C'$ if and only if each entry of C is less than or equal to the corresponding entry of C', and at least one entry is strictly less. More formally: $C \prec C'$ iff $\forall i, C[i] \leq C'[i]$ and $\exists j, C[j] < C'[j]$. This ordering is partial because some timestamp pairs cannot be ordered (e.g., $[1,2]$ and $[2,1]$).

3.2 Networked systems as communicating finite state machines

Dynoptic uses the CFSM formalism for models because a CFSM is similar to the widely known and easily understood FSM formalism. CFSMs are well-established in the formal methods community, and we believe that a CFSM is intuitive and simpler for developers to comprehend than an alternative model type (e.g., a Petri net). For example, a single process FSM in a CFSM can be inspected and understood without needing to understand the activity of other processes in the system.

The CFSM model is a natural fit for developers of networked systems because such systems are typically modular, and one of the most useful modularity boundaries is the process abstraction. For example, when we asked students in a distributed systems course to draw their mental model of a distributed system, many students drew a model that resembles a CFSM (e.g., Figure 3.3). Such models had two key features: one distinct finite state machine per process, and each process consuming events generated by remote processes. These features happen to be the defining characteristics of a CFSM. Further, prior work on fault-tolerance services, with its emphasis on per-process FSMs [113], also supports our CFSM model choice.

A communicating finite state machine (CFSM) is a model that depicts a fixed set of processes. Processes communicate with one another by sending and receiving messages on uni-directional channels. Each process in a CFSM is represented as a finite state machine. For example, Figure 3.4 shows a CFSM with two processes — process A and process B. These two processes are connected with channels c and d.

In a process finite state machine, circles represent states (e.g., process A has two states — $a0$ and $a1$). And, arrows depict transitions on events between states. The process machines execute discretely and asynchronously by taking transitions from one state to the next, starting from an initial state and terminating in the terminal state. The initial state is indicated by a floating arrow that points to it (e.g., $a0$ is the initial state at process A). The terminal state is indicated by two concentric circles (e.g., $b0$ is the terminal state at process B).

Processes in a CFSM have two kinds of communication events that allow them to exchange messages. The send of a message m on channel c is expressed as $c!m$ (the exclamation mark indicates a send). While the receive of a message m on channel c is indicated by $c?m$ (the question mark indicates a receive).

For example, in the CFSM above, process A can send a message m to process B with the event $c!m$. This would transition process A from state $a0$ to state $a1$. Process B can receive m on channel c by transitioning from $b0$ to $b1$ (event $c?m$). Next, process A must stay in $a1$, waiting for an ack message from process B. Process B can reply with an ack on channel d – $d!ack$, which would transition it from $b1$ to $b0$. Finally, process A can receive the ack and transition back to $a0$.

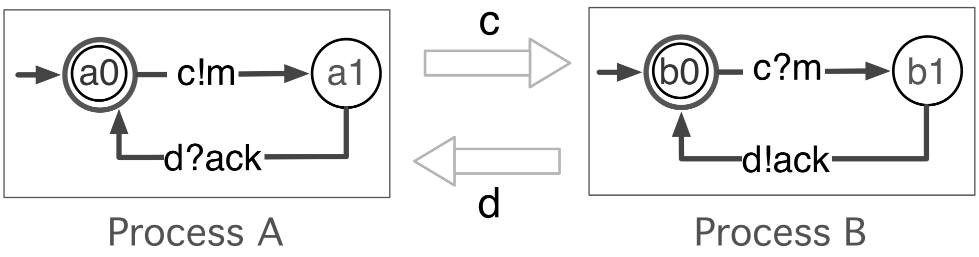


Figure 3.4: A simple CFSM model with two processes — A and B, and two channels — c and d.

The above sequence is a valid execution of the system because it starts with all processes in initial states, terminates with all processes in terminal states, and has legal intermediate transitions. This execution can be represented more compactly as the sequence $[c!m, c?m, d!ack, d?ack]$. On the other hand, the execution $[c?m, d!ack, d?ack]$ is not a valid execution (process B cannot receive message m since process A never sent it).

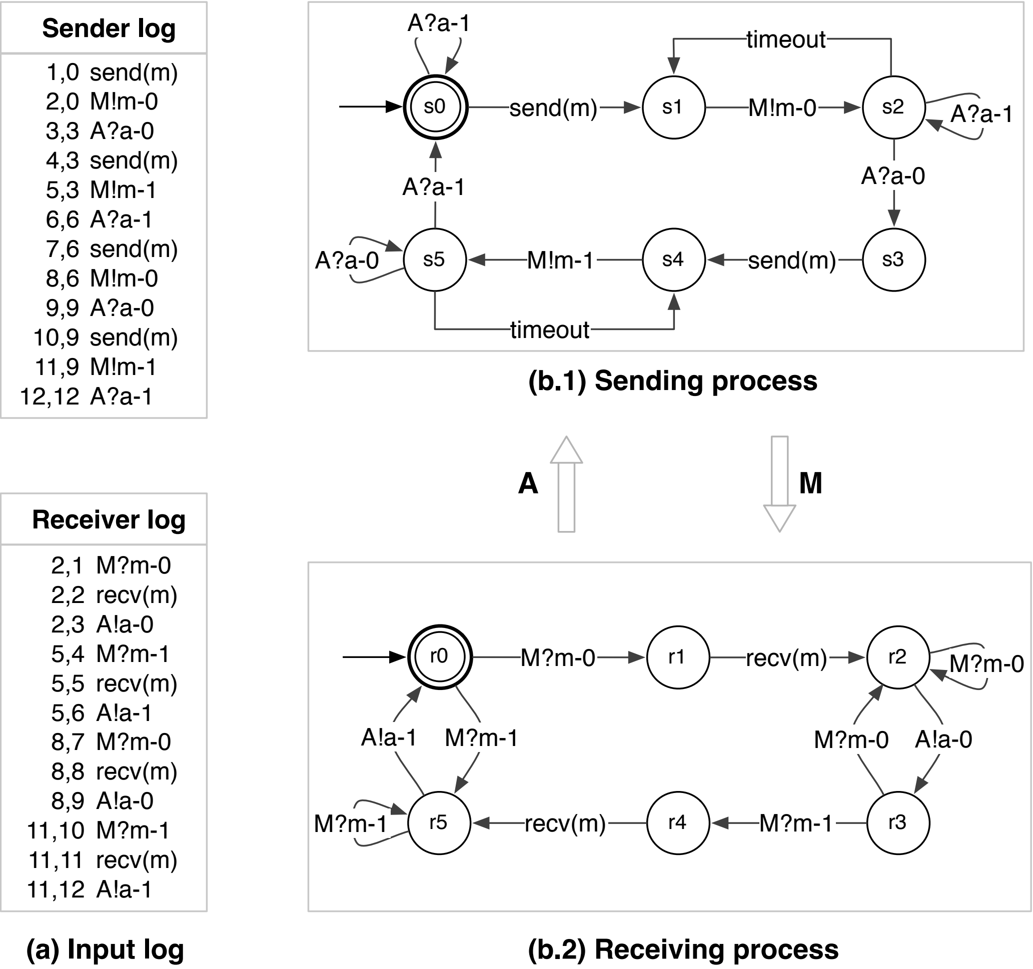
In the standard CFSM formalism, processes communicate with one another via message passing over reliable FIFO channels. However, unreliable channels can be simulated by replacing each unreliable channel with a lossy "middlebox" FSM that non-deterministically chooses between forwarding and losing a message.

3.3 Overview of approach

Dynoptic's input is a log, and its output is a CFSM model that describes the distributed system that generated the log. An input log consists *execution traces* of the system. Each execution trace is a set of events, and each event has an attached vector timestamp [52, 96].

Figure 3.5(a) shows an example input log generated by two processes executing the stop-and-wait (SAW) protocol. In the protocol, a *sender* process communicates a sequence of messages to a *receiver* process over an unreliable channel. The receiver must acknowledge an outstanding message before the sender moves on to the next message. If a message is delayed or lost, the sender re-transmits the message after a timeout.

In the system that produced the log in Figure 3.5(a) the sender transmits messages to the receiver



Sender log

```
 1,0  send(m)
 2,0  M!m-0
 3,3  A?a-0
 4,3  send(m)
 5,3  M!m-1
 6,6  A?a-1
 7,6  send(m)
 8,6  M!m-0
 9,9  A?a-0
10,9  send(m)
11,9  M!m-1
12,12 A?a-1
```

Receiver log

```
 2,1  M?m-0
 2,2  recv(m)
 2,3  A!a-0
 5,4  M?m-1
 5,5  recv(m)
 5,6  A!a-1
 8,7  M?m-0
 8,8  recv(m)
 8,9  A!a-0
11,10 M?m-1
11,11 recv(m)
11,12 A!a-1
```

(b.1) Sending process

(a) Input log

(b.2) Receiving process

Figure 3.5: Example Dynoptic inputs and outputs. **(a)** Example input log with a single vector-timestamped trace of two processes running the stop-and-wait protocol (SAW). **(b)** The CFSM model of SAW derived by Dynoptic, with **(b.1)** the sender process model and **(b.2)** the receiver process model. These models were inferred by Dynoptic for this system on a more complete log (not shown).

using channel M, and the receiver replies with acknowledgments through channel A. Notation Q!x means enqueue message x at head of channel Q, and event Q?x means dequeue message x from the head of channel Q. The event send(m) is a down-call to send m at the sender, and recv(m) is an up-call at the receiver indicating that m was received. The timeout event at the sender triggers a message re-transmission after some internal timeout threshold is reached. The “alternating bit” is associated with each message and is appended to a message’s channel representation as -0 or -1. For example, the first (and every odd) message sent by the sender is represented as m-0 and the corresponding response is represented as a-0.

Figure 3.5(b) shows Dynoptic’s output — a communicating finite state machine [27] (CFSM) model — for a more complete log of this protocol (not shown). The model in Figure 3.5(b) handles message loss, but the lossy middlebox is not shown in the diagram.

Dynoptic transforms its input through a series of representations and analyses to produce its output (see Figure 3.6). First, Dynoptic converts the *input log* into a set of *execution DAGs*. Those are combined into a single *concrete FSM*, which is abstracted into a more concise *abstract FSM*. Dynoptic iteratively refines (enlarges) the concise, general abstract FSM until it satisfies a set of temporal invariants that Dynoptic mined from the execution DAGs. The final abstract FSM is converted to the output CFSM. We now give more details about each of these steps. The following sections will then formally describe the Dynoptic process.

Dynoptic first parses the input log into a set of execution DAGs (step ① in Figure 3.6). Each DAG captures the partial ordering of events in one of the executions in the log. Figure 3.10(a) illustrates three DAGs parsed from a log (not shown) of an example system with two processes (Section 3.8 describes the example system).

Dynoptic uses the collection of all of the parsed DAGs to mine *temporal invariants* that are true across all executions in the log (step ②). For example, for the DAGs in Figure 3.10(a), one invariant Dynoptic mines is: event *b* at process p1 is never followed by event *x* at process p2. As another example, one mined stop-and-wait protocol invariants is M?m-0 is always followed by A?a-0.

Dynoptic next uses the execution DAGs to construct a single *concrete FSM* (step ③). This FSM represents the observed, or concrete, executions of the whole system. Each state in the concrete FSM is a tuple of all the individual process states and all the channel contents. A channel’s content is computed from the sequence of observed message sends and receives in the trace. Process states, however, must be inferred, and are assumed to be uniquely determined by the process’s history. In other words, for a specific sequence of events at a process, Dynoptic creates a single unique *anonymous* process state. Figure 3.10(b) illustrates the process states that Dynoptic creates based on the DAGs in Figure 3.10(a). These local process states are then combined with observed channel contents to generate the concrete FSM (Figure 3.10(c)).

The concrete FSM describes all of the observed executions in the log: it accepts all serializations of events across all the execution DAGs. For example, the concrete FSM in Figure 3.10(c) accepts the sequence $[a, c, !m, ?m, y]$, which is exactly trace 1 from Figure 3.10(a). However, the concrete FSM also generalizes based on state equivalence across traces. For example, the concrete FSM in Figure 3.10(c) accepts the sequence $[a, c, !m, ?m, x]$, which is not an observed trace.

The concrete FSM is first used to filter, or validate, the mined invariants (step ④). This step is necessary because of incompleteness of the input traces (see Section 3.8).

Second, the concrete FSM is used to construct an *abstract FSM* (step ⑤). The concrete FSM models the system, but generalizes in a very limited way. Moreover, the concrete FSM is not concise: it is a DAG whose longest path is as long as the longest execution. Because the concrete FSM is neither concise nor sufficiently abstract, Dynoptic generates a more concise *abstract FSM* model, using a process we call *state abstraction*.

A state in the abstract FSM corresponds to multiple states in the concrete FSM. Dynoptic derives the abstract FSM from the concrete FSM by *merging* concrete states into abstract states, with transitions between abstract states derived from the transitions between the underlying concrete states. The resulting FSM is concise (because there are far fewer states than in the concrete FSM) and accepts all of the sequences accepted by the concrete FSM. But, the partition graph also generalizes and accepts sequences of events that were not observed. In fact, the initial abstract FSM model that Dynoptic builds is often too abstract. Next, Dynoptic uses the mined invariants to *refine* the abstract FSM into a model that limits abstraction in a way that is consistent with patterns observed in the executions (i.e., by satisfying the mined invariants). Figure 3.7 overviews, at a high-level, this refinement process, which we briefly describe next.

Dynoptic achieves refinement via a counter-example-based abstraction refinement loop [32]. Dynoptic uses the McScM model checker [71] to check the CFSM model corresponding to the abstract FSM against each of the mined invariants (steps ⑥ and ⑦). If the abstract FSM model does not satisfy an invariant, the model checker returns a counter-example path in the model. Dynoptic uses this path to refine, or split, a set of partitions in the abstract FSM to create a larger model that invalidates the counter-example (step ⑧). The new model is less abstract (not as general) and does

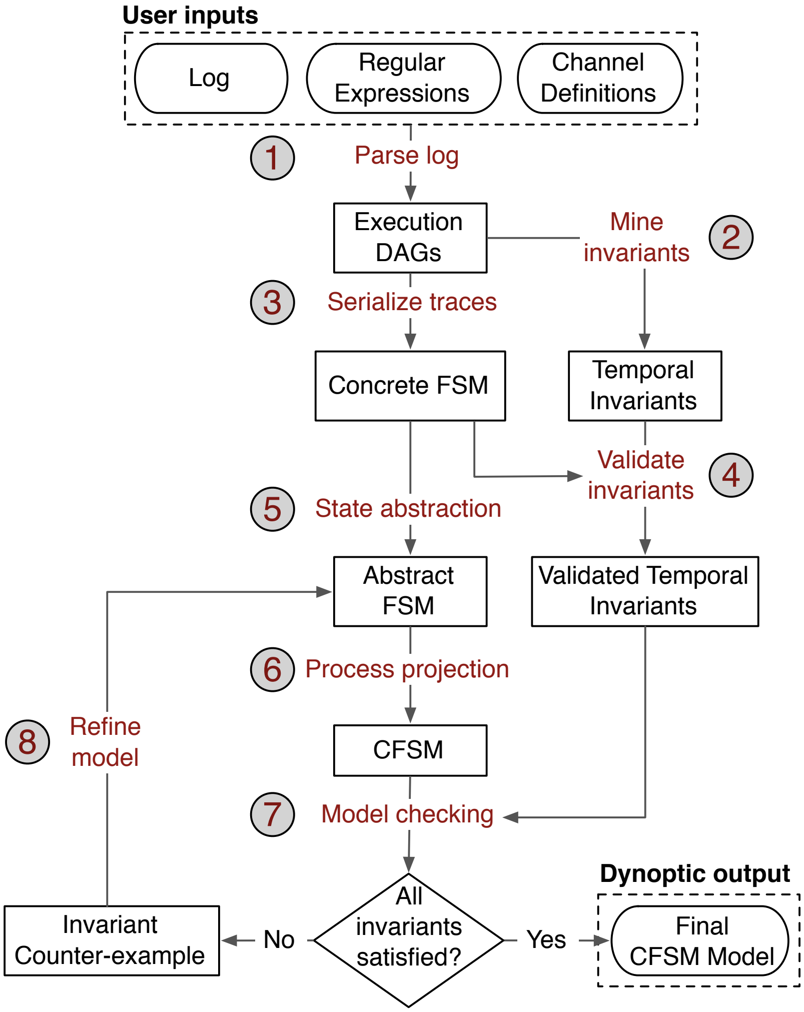


Figure 3.6: Dynoptic process flow chart. This is an elaboration of the more abstract process description in Figure 1.5(a).

not contain the counter-example path. Eventually, after potentially several refinements, the model will satisfy the invariant.

Once all of the invariants are satisfied, Dynoptic generates the output CFSM, which corresponds to the final abstract FSM. The output model can be used directly by an end-user, or fed into other automated tools (e.g., for test-generation).

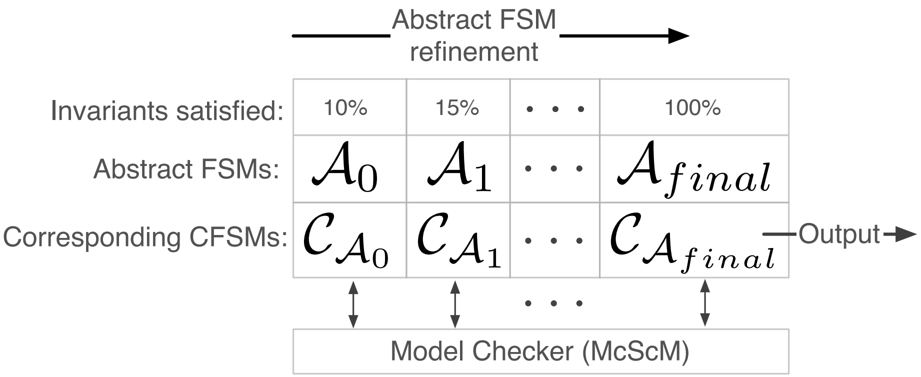


Figure 3.7: A summary of the Dynoptic refinement process. Dynoptic starts with an initial abstract FSM, $\mathcal{A}_0$, which is gradually refined to a final abstract FSM, $\mathcal{A}_{final}$, which satisfies all of the invariants.

3.4 Formalizing a log of events

The next couple of sections formalize Dynoptic's model inference process (overviewed in Section 3.3 and Figure 3.6). This formalism enabled us to formally prove important properties of our approach (Section 3.12). A reader who wishes to get an intuitive understanding of the approach can skim these sections.

We start by defining a log of events and describing the inputs to Dynoptic (Section 3.5). Then, we describe the invariants that Dynoptic mines, and how it mines them (Section 3.6). Then, we specify how Dynoptic converts a log into a concrete FSM (Section 3.7), and uses this FSM to validate the mined invariants (Section 3.8). Next, we explain how Dynoptic abstracts the concrete FSM into the initial abstract FSM (AFSM) (Section 3.9). Finally, we describe how Dynoptic model-checks an AFSM (Section 3.10); and then refines the AFSM to satisfy the valid invariants and converts the AFSM into a CFSM (Section 3.11).

We begin by describing the notation used in the rest of the chapter. Given an n-tuple t, let $|t| = n$, and let $t[i], 0 < i \leq n$ refer to the i^{th} component of t; for $i > n$, let $t[i] = \varepsilon$. Given two tuples t_1 and t_2, $|t_2| \leq |t_1|$, we say $t_2 \subseteq t_1$ (t_2 is a sub-tuple of t_1) if and only if we can generate t_2 by deleting some number of components from t_1. Let $t' = t[i \leftarrow c]$ represent the n-tuple that is identical to t except possibly in the i^{th} component, which equals c. That is, $t'[i] = c$ and $\forall j \neq i$, $t'[j] = t[j]$. We write $t'' = t \cdot t'$ to denote concatenation of two tuples: t'' has length $|t| + |t'|$, and $\forall i \in [1, |t|]$, $t''[i] = t[i]$ and $\forall i \in [|t|+1, |t'|]$, $t''[i] = t'[i - |t|]$. We call t a *prefix* of t'' iff $\exists t'$ such that $t'' = t \cdot t'$. Finally, let the projection function π map a tuple t and a set S to a tuple t' that is the maximum length sub-tuple of t with elements from S. That is, $t' = \pi(t, S)$ is the largest tuple such that $t' \subseteq t$ and $\forall i, t'[i] \in S$.

A *log* is a set of *system traces*, each of which represents one system execution. Each system trace consists of a set of *process traces* (one process' view of the execution) of *instances of events* and a partial ordering over those instances. The event instances can be classified into *types* and some types (*send* and *receive*) are associated with *channels* between processes. We now formally define these concepts.

We assume a distributed system that is composed of h processes, indexed from 1 to h, and there is a fixed set of channels, each of which is used to connect one sender process to one receiver process.

Definition 3.1 (Channel)**.** A *channel* c_{ij} is identified by a pair of process indices (i, j), where $i \neq j$ and $i, j \in [1, h]$. Indices i and j denote the channel's *sender* and *receiver* process, respectively.

For each channel c_{ij}, a (possibly empty) finite set of *messages* M_{ij} are the only messages that can be *sent* and *received* on c_{ij}.

The execution of each process generates a sequence of event instances, each of which has an event type from a finite alphabet of process event types. These event types can be local, or message send or receive events. Each of the send and receive events is associated with a channel (we use ! to label send events and ? to label receive events).

Definition 3.2 (Channel send and receive event types)**.** For a channel c_{ij}, the channel *send* (S_{ij}) and *receive* (R_{ij}) event types are the finite sets (alphabets) $S_{ij} = \{c_{ij}!m \mid m \in M_{ij}\}$ and $R_{ij} = \{c_{ij}?m \mid m \in M_{ij}\}$.

Definition 3.3 (Process event types)**.** For a process i, the *process event types* set E_i is a finite set (alphabet) of event types that can be generated by process i. E_i is composed of all send and receive events that the process can generate, as well as a set of *process local* (L_i) event types: $E_i = (\cup_{\forall j} S_{ij}) \cup (\cup_{\forall j} R_{ji}) \cup L_i$.

Definition 3.4 (Event instance)**.** For a process i with the process event types E_i, an *event instance* is a triple $\langle e, i, k \rangle$, where $k \geq 1$ is an integer that indicates the order (position) of the event instance, among all event instances generated by process i. (In other words, no two event instances will share the same k.) When the value of k is not important, we denote this triplet as $\hat{e}_i$.

A process trace is the set of all event instances generated by a process. We assume that each event instance is given a unique position in a consecutive order, from 1 to length of the trace.

Definition 3.5 (Process trace)**.** For the process i, a *process trace* is a set T_i of event instances, such that $\forall k \in [1, |T_i|]$, $\exists \langle e, i, k \rangle \in T_i$, and $\langle e, j, k \rangle \in T_i \implies j = i$.

A system trace corresponds to a single execution of the system. It consists of the union of a set of process traces (one per process), and a strict partial ordering over event instances in the process traces. The partial ordering captures the *happens before* relation [83] and must totally order event instances in each process trace according to their positions.

In addition, a system trace must satisfy three basic communication consistency constraints: (1) every sent message is received, (2) only sent messages are received, and (3) sent messages on the same channel must be received in FIFO order. These constraints are trace assumptions[3] and removing them is part of our future work (see Section 3.14). We express these three constraints with a bijection between message send and message receive instances for a pair of processes.

Definition 3.6 (System trace)**.** A *system trace* is the pair $\langle T, \prec \rangle$, where $T = \cup T_i$, and $\prec$ is a strict partial order such that $\forall \hat{e}_i = \langle e, i, k_1 \rangle \in T$, $\hat{f}_i = \langle f, i, k_2 \rangle \in T$, $\hat{e}_i \prec \hat{f}_i \iff k_1 < k_2$.

For all i, j let $\hat{S}_{ij} = \{\hat{s} \mid \hat{s} \in T \text{ and } s \in S_{ij}\}$ and $\hat{R}_{ij} = \{\hat{r} \mid \hat{r} \in T \text{ and } r \in R_{ij}\}$. A system trace must have a corresponding bijection f that satisfies the three communication consistency constraints mentioned above. More formally, $\forall i, j, \exists$ a bijection $f : \hat{S}_{ij} \to \hat{R}_{ij}$, such that:

1. $f(\hat{s}) = \hat{r} \implies \hat{s} \prec \hat{r}$

2. $\forall \hat{s_1}, \hat{s_2} \in \hat{S}_{ij}, \hat{s_1} \prec \hat{s_2} \implies f(\hat{s_1}) \prec f(\hat{s_2})$.

A *log* is a set of system traces.

[3] The validity of the no message drops and no reorderings assumptions depends on the underlying transport protocol. TCP satisfies both assumptions, but for UDP, these assumptions are invalid. Generally, TCP is used for complex protocols that Dynoptic targets.

3.5 Log parsing

To use Dynoptic, the user must provide three inputs: (1) a log, (2) a set of regular expressions, and (3) a set of channel definitions.

1. **Log.** The input log was formalized in the previous section. Figure 3.5(a) lists an example log.

2. **Regular expresisons.** The user must supply a set of user-defined regular expressions for log parsing. These expressions determine the subset of lines that are parsed, what part of a line corresponds to a vector timestamp, and which local, send, or receive event instance the line represents. For example, the log in Figure 3.5(a) requires a single regular expression — `(?<VTIME>)(?<PID>)(?<TYPE>.+)`. In this expressions the `VTIME` keyword indicates that the first field should be treated as a vector timestamp, while the `TYPE` keyword indicates that the last field should be considered an event type[4].

3. **Channel definitions.** The user must define all of the channels referenced in the log. For example, the event instance `M!m-0` in the log in Figure 3.5(a) uses the `M` channel. This is a channel from process 0 to process 1. Another channel used in the same log is channel `A` (for acknowledgement messages), which is a channel from process 1 to process 0. For the log in Figure 3.5(a) the input channel definitions are specified with the following string: `M:0->1;A:1->0`.

Given the three inputs described above, Dynoptic first parses each system trace in the log into an *execution DAG* (step ① in Figure 3.6). Each such DAG captures the partial order of events in the corresponding system trace. Figure 3.10(a) shows three example execution DAGs. Next, Dynoptic mines a set of temporal invariants from these execution DAGs.

3.6 Invariant mining

Dynoptic mines a set of temporal invariants that relate events in the log (step ② in Figure 3.6). These invariants, defined next, are true for all of the execution DAGs and are central to Dynoptic — for the

[4] Message send/receive events, like `M!m-0` and `M?m-0`, are automatically parsed into channel name (`M`), action type (send or receive), and message type (`m-0`).

invariants to be accurate, the log must have executions representative of all possible executions of the modeled system. Note that the current version of Dynoptic considers just the top three of the five invariants defined below ($\rightarrow$, $\not\rightarrow$, and $\leftarrow$).

A key property of Dynoptic models is that it guarantees that the final inferred CFSM model satisfies all of the mined invariants that are valid (Theorem 3.2 in Section 3.12). Section 3.8 explains which invariants are considered valid.

Definition 3.7 (Event invariant)**.** Let L be a log, and let a_i and b_j be two event types whose corresponding event instances, $\hat{a}_i$ and $\hat{b}_j$, appear at least once in some system trace in L. Then, an *event invariant* is a property that relates a_i and b_j in one of the following three ways.

$a_i \rightarrow b_j$ **:** An event instance of type a at host i is **always followed by** an event instance of type b at host j. Formally:
$\forall \langle T, \prec \rangle \in L, \forall \hat{a}_i \in T, \exists \hat{b}_i \in T, \hat{a}_i \prec \hat{b}_j$.

$a_i \not\rightarrow b_j$ **:** An event instance of type a at host i is **never followed by** an event instance of type b at host j. Formally:
$\forall \langle T, \prec \rangle \in L, \forall \hat{a}_i \in T, \nexists \hat{b}_j \in T, \hat{a}_i \prec \hat{b}_j$.

$a_i \leftarrow b_j$ **:** An event instance of type a at host i **always precedes** an event instance of type b at host j. Formally:
$\forall \langle T, \prec \rangle \in L, \forall \hat{b}_j \in T, \exists \hat{a}_i \in T, \hat{a}_i \prec \hat{b}_j$.

$a_i \parallel b_j$ **:** An event instance of type a at host i is **always concurrent** with an event instance of type b at host j. Formally: $\forall \langle T, \prec \rangle \in L, \forall \hat{a}_i, \hat{b}_j \in T, (\hat{a}_i \not\prec \hat{b}_j \wedge \hat{b}_j \not\prec \hat{a}_i)$.

$a_i \nparallel b_j$ **:** An event instance of type a at host i is **never concurrent** with an event instance of type b at host j. Formally: $\forall \langle T, \prec \rangle \in L, \forall \hat{a}_i, \hat{b}_j \in T, (\hat{a}_i \prec \hat{b}_j \vee \hat{b}_j \prec \hat{a}_i)$.

We term invariants that relate host event types on the same host (i.e., $i = j$) as *local*, and those that relate host event types on different hosts ($i \neq j$) as *distributed*. Local invariants can be evaluated independently of event instances on other hosts, solely by using the total ordering of event instances on the host. In contrast, distributed invariants capture dependency between event types on different hosts — their evaluation requires the use of the partial ordering.

Synoptic and previous work on specification patterns [47] only consider local invariants as they study specification patterns of sequential systems. For each of the five invariants above, the event instances may occur on different hosts or on the same host. For example, one invariant of the stop-and-wait protocol model in Figure 3.5 is `M?m-0` → `A?a-0`.

As with Synoptic, the $\rightarrow$, $\not\rightarrow$, and $\leftarrow$ invariants capture particular kinds of ordering dependency. The $\parallel$ and $\nparallel$ invariants, however, are more general. The $\parallel$ invariant captures the *lack of* ordering, and $\nparallel$ captures the *presence of some* ordering. An example of a $\parallel$ invariant is "$\text{search}_{c_0} \parallel \text{search}_{c_1}$" for the log in Figure 3.2(a), which means that ticket search requests from the two clients are never ordered. The $\parallel$ and $\nparallel$ invariants are also commutative — e.g., $a_i \parallel b_j$ iff $b_j \parallel a_i$.

We now explain how these five invariants types are mined.

3.6.1 Mining invariants

The task of mining invariants involves taking a log (e.g., Figure 3.2(a)) as input, and outputting the set of invariants that are true of the log.

To simplify our discussion of invariant mining algorithms, we use the directed acyclic graph (DAG) representation of a system trace. The time-space diagrams in Figure 3.2(b) illustrate the basic idea of the DAG representation (except that in these diagrams, edges between events generated at the same host are implicit). Formally, a system trace $\langle T, \prec \rangle$ can be represented as a DAG with nodes corresponding to event instances in T, and an edge from $\hat{e}$ to $\hat{f}$ iff $\hat{e}$ is a direct predecessor of $\hat{f}$ (i.e., iff $\hat{e} \prec \hat{f}$ and $\nexists\, \hat{g},\ \hat{e} \prec \hat{g} \prec \hat{f}$).

Transitive-closure-based algorithm

One invariant-mining algorithm computes the forward and reverse transitive closures of each trace DAG in L and then determines which invariants are valid from those transitive closures as follows:

- $a_i \rightarrow b_j$ iff in each forward DAG transitive closure, every $\hat{a}_i$ node (instance of event type a_i) has an edge to a $\hat{b_j}$ node.

- $a_i \not\rightarrow b_j$ iff in each forward DAG transitive closure, every $\hat{a}_i$ node has no edge to a $\hat{b_j}$ node.

- $a_i \leftarrow b_j$ iff in each reverse DAG transitive closure, every $\hat{b_j}$ node has an edge to a $\hat{a}_i$ node.

- $a_i \parallel b_j$ iff there are no edges between $\hat{a}_i$ and $\hat{b_j}$ nodes in either the forward or the reverse DAG transitive closures, and the two kinds of nodes both occur in some DAG.

- $a_i \nparallel b_j$ iff there are edges between all $\hat{a}_i$ nodes and all $\hat{b_j}$ nodes in either the forward or the reverse DAG transitive closures whenever both nodes occur in the DAG.

This algorithm performs poorly on sparse DAGs, for which transitive closure construction is expensive. Next, we describe two algorithms that do not explicitly generate the transitive closures, but instead mine invariants implicitly by collecting event type co-occurrence counts.

Co-occurrence counting algorithm v1

The idea behind the co-occurrence counting algorithm is to avoid explicit construction of the trace DAGs' transitive closures. Instead, the algorithm walks through the trace DAGs and counts specific values, such as the number of times an event instance of type a_i is followed by an event instance of type b_j. After counting, the algorithm uses a set of rules (derived from the invariant definitions above) to infer the true invariants:

- $a_i \rightarrow b_j$ iff the number of $\hat{a}_i$ occurrences is equal to the number of times that $\hat{a}_i$ was followed by $\hat{b_j}$.

- $a_i \not\rightarrow b_j$ iff the number of times that $\hat{a}_i$ was followed by $\hat{b_j}$ was 0.

- $a_i \leftarrow b_j$ iff the number of $\hat{b_j}$ occurrences is equal to the number of times $\hat{b_j}$ was preceded by $\hat{a}_i$.

- $a_i \parallel b_j$ iff $\hat{a}_i$ and $\hat{b_j}$ co-occurred at least once in a system trace (otherwise calling the two event types concurrent does not make sense); and the number of times that $\hat{a}_i$ followed $\hat{b_j}$ and the number of times $\hat{b_j}$ followed $\hat{a}_i$ was 0 (the events were never ordered).

```
Input: event log $L$, as a set of event instance DAGs

for dag $\in L$ {
  let dagOcc[] // Maintains DAG event counts per event type
  // Traverse the DAG in the forward direction:
  foreach node $\in$ dag, in topological order:
    let node.predecessors $= \cup_{p \in node.parents}$p.predecessors
    let $b_j =$ node.type
    dagOcc$[b_j]$ ++
    let seenTypes $= \{\}$
    for nodeP $\in$ node.predecessors:
      let $a_i =$ nodeP.type
      PrecPairs$[a_i][b_j]$ ++
      if $a_i \notin$ seenTypes:
         CoOcc$[a_i][b_j] = true$
         Prec$[a_i][b_j]$ ++
         seenTypes = seenTypes $\cup \{a_i\}$

  // Traverse the DAG in the reverse direction:
  foreach node $\in$ dag, in reverse topological order:
    let node.successors = $\cup_{c \in node.children}$ c.successors
    let $a_i =$ node.type
    let seenTypes $= \{\}$
    for nodeS $\in$ node.successors:
      let $b_j =$ nodeS.type
      FollowsPairs$[a_i][b_j]$ ++
      if $b_j \notin$ seenTypes:
        Follows$[a_i][b_j]$ ++
        seenTypes = seenTypes $\cup \{b_j\}$

  // Accumulate this DAG's event instance counts:
  for $a_i \in$ dagOcc.keys:
    Occ$[a_i] +=$ dagOcc$[a_i]$
    for $b_j \in$ dagOcc.keys:
      TraceCountProductsSum$[a_i][b_j] +=$
         (dagOcc$[a_i] *$ dagOcc$[b_j]$)
}

// Use the counts to derive the invariants:
let invariants = []
for $a_i, b_j \in$ eventTypes :
  if Follows$[a_i][b_j] =$ Occ$[a_i]$:
     invariants.append($a_i \rightarrow b_j$)
  if Follows$[a_i][b_j] = 0$:
     invariants.append($a_i \not\rightarrow b_j$)
  if Prec$[a_i][b_j] =$ Occ$[b_j]$:
     invariants.append($a_i \leftarrow b_j$)
  if CoOcc$[a_i][b_j] \wedge$ Follows$[a_i][b_j] = 0 \wedge$ Follows$[b_j][a] = 0$:
     invariants.append($a_i \parallel b_j$)
  if CoOcc$[a_i][b_j] \wedge$ TraceCountProductsSum$[a_i][b_j] =$
    PrecPairs$[a_i][b_j] +$ FollowsPairs$[b_j][a_i]$:
     invariants.append($a_i \nparallel b_j$)

Output: invariants
```

Figure 3.8: The co-occurrence counting algorithm v1 described in Section 3.6.1.

```
for dag $\in L$ {
  // Traverse the DAG in the forward direction:
  foreach node $\in$ dag, in topological order:
    let node.typePred $= \cup_{p \in node.parents}$p.typePred
    let $b_j =$ node.type
    Occ[$b_j$]++
    for $a_i \in$ node.typePred:
      CoOcc[$a_i$][$b_j$] $=$ true
      Prec[$a_i$][$b_j$]++

  // Traverse the DAG in the reverse direction:
  foreach node $\in$ dag, in reverse topological order:
    let node.typeSucc = $\cup_{c \in node.children}$ c.typeSucc
    let $a_i =$ node.type
    for $b_j \in$ node.typeSucc:
      Follows[$a_i$][$b_j$]++
}
```

Figure 3.9: A different *for* loop body for the pseudocode in lines 3–29 of Figure 3.8, which generates a simpler and more efficient algorithm (co-occurrence counting algorithm v2) for computing all the invariants except $\nparallel$. As well, the new algorithm would omit lines 50–52 in Figure 3.8.

- $a_i \nparallel b_j$ iff $\hat{a}_i$ and $\hat{b}_j$ co-occurred at least once in a system trace; and in every trace each $\hat{a}_i$ instance is followed or preceded by every $\hat{b}_j$ instance. That is, in a trace the number of $\hat{a}_i$ followed by $\hat{b}_j$ pairs plus the number of $\hat{a}_i$ preceded by $\hat{b}_j$ pairs must equal the count of $\hat{a}_i$ in the trace times the count of $\hat{b}_j$ in the trace.

The pseudocode in Figure 3.8 outlines this procedure. The algorithm starts by building a set of data structures to hold various occurrence counts:

- Occ[a_i] : the count of $\hat{a}_i$ across all traces.

- CoOcc[a_i][b_j] : whether or not $\hat{a}_i$ and $\hat{b}_j$ co-appeared in a trace.

- Prec[a_i][b_j] : the count of $\hat{b}_j$ instances that were preceded by at least one $\hat{a}_i$.

- Follows$[a_i][b_j]$: the count of $\hat{a}_i$ instances that were followed by at least one $\hat{b}_j$.
- FollowsPairs$[a_i][b_j]$: the count of all $(\hat{a}_i, \hat{b}_j)$ pairs for which $\hat{a}_i$ was followed by $\hat{b}_j$.
- PrecPairs$[a_i][b_j]$: the count of all $(\hat{a}_i, \hat{b}_j)$ pairs for which $\hat{a}_i$ precedes $\hat{b}_j$.
- TraceCountProductsSum$[a_i][b_j]$: the sum across all traces of the product of the number of $\hat{a}_i$ in a trace and the number of $\hat{b}_j$ in a trace.

To collect these counts, the trace DAG is first traversed in the forward and then in the reverse directions. Both traversals are in topological order (e.g., on the forward traversal a node is visited after all of its parents). The topological order guarantees that all the nodes that precede (respectively follow) the node's parents (respectively children) are aggregated correctly. Once the DAG is traversed in both directions, the algorithm infers invariants from the data structures. Each *if* statement on lines 42–52 of the pseudocode corresponds to an informal description given above.

Because the algorithm traverses each edge once, its base traversal time for a single trace DAG is $\Theta(|E|)$, where E is the set of edges in the DAG. On traversing an edge, the algorithm needs to merge two sets whose sizes are at most $|V|$, where V is the set of nodes in the DAG. Therefore, in processing a single trace DAG, the algorithm has a running time of $\Theta(|E||V|)$. Because it does not need to explicitly maintain a transitive closure, this algorithm performs especially well on sparse trace DAGs.

Co-occurrence counting algorithm v2 (without $\nparallel$)

In both of the previous algorithms, the cost of computing the $\nparallel$ invariant is significantly higher than that of computing each of the other invariant types. This is because evaluating the invariant $a_i \nparallel b_j$ requires an algorithm to consider every pair of instances $(\hat{a}_i, \hat{b}_j)$. This overhead prompted us to consider an algorithm that mines all of the invariants except the $\nparallel$ invariant.

Figure 3.9 lists a different *for* loop body for the pseudocode in lines 3–29 of Figure 3.8. The resulting algorithm — co-occurrence counting algorithm v2 — is significantly faster (see Section 3.13.1). The reason for this is that instead of maintaining the set of all *event instances* that precede (respectively follow) a node, the algorithm maintains only the set of *event types* that precede (respectively follow) a node. Because of this, the per-edge cost drops from $\Theta(|V|)$ to $\Theta(|ETypes|)$ where $ETypes$ is the set of event types in the trace DAG. Therefore, this algorithm's running time is $\Theta(|E||ETypes|)$.

As mentioned at the beginning of this section, Dynoptic uses three of the five invariants that are currently mined: $\rightarrow$, $\not\rightarrow$, and $\leftarrow$. In the rest of this chapter we mean exactly these three invariants types when we talk about invariants.

3.7 Deriving a concrete FSM

In a system trace $\langle T, \prec \rangle$, the ordering $\prec$ is *partial*. Dynoptic derives this partial ordering from the vector timestamps associated with each event instance. Any linear extension of events in T that respects $\prec$ is a totally ordered interleaving of the process traces. That is, a linear extension is one possible order in which the events could have taken place to generate the system trace.

Definition 3.8 (Linear extension)**.** For a system trace $S = \langle T, \prec \rangle$, a linear extension $<$ of $\prec$ is a total order over T such that $\forall \hat{a}_i, \hat{b}_j \in T, \hat{a}_i \prec \hat{b}_j \implies \hat{a}_i < \hat{b}_j$.

Again, a linear extension is one possible serialization of a partially ordered execution. We will now introduce formalisms that will enable us to define a concrete FSM (Definition 3.13) whose language includes all linear extensions of all system traces in a log: $\cup_{S=\langle T,\prec\rangle \in L} \mathcal{L}(S)$, where $\mathcal{L}(S)$ is the set of *all* linear extensions of $\prec$. Before we can define a concrete FSM, however, we first need to introduce the notion of a system state, which consists of a global process state and a global channel state.

To simplify definitions we first introduce $\hat{E}_i$ — the set of all process i event instances in a log, and L_i — the set of process i traces in a log L. More formally, let $\hat{E}_i = \{\hat{e}_i \mid \exists \langle T, \prec \rangle \in L$, such that $\hat{e}_i \in T\}$, and let $L_i = \{\pi(T, \hat{E}_i) \mid \exists \langle T, \prec \rangle \in L\}$. Note that L_i is the set of all process traces, T_i, that appear in the log. Also, because $\forall T_i \in L_i$, T_i is totally ordered (Definition 3.6), we will treat T_i as a tuple. That is, $T_i[j] = \langle e, i, j \rangle$, for some e.

We let Q_i be a finite set of *local* process states for a process i. Each process begins execution in an initial state, q_0^i, and after executing a sequence s of process i event instances, the process enters state q_s. More formally, $Q_i = \{q_0^i\} \cup \{q_s \mid \exists T_i \in L_i$, s is a prefix of $T_i\}$.

We call q_s a *terminal* state for process i if and only if $s \in L_i$. That is, q_s corresponds to a sequence of process i event instances, s, that are all of the process i event instances in some trace. The state q_0^i

is a *terminal* state for process i if and only if there is a trace with no process i event instances (i.e., $\varepsilon \in L_i$).

Now, we define global process state and global channel state that together make up the system state. We use Q to denote the set of all *global process states*. As there are h process, global process states are h-tuples, $Q = Q_1 \times \cdots \times Q_h$.

Definition 3.9 (Global process state)**.** A *global* process state $q \in Q$ is a h-tuple that represents the state of all processes in the system, with $q[i] \in Q_i$ denoting the local process state of process i.

To describe a system's state, we must also include the contents of channels, which contain all sent messages that have not yet been received. We first define the state of a single channel and then use this definition to define the global channel state of the system.

Definition 3.10 (Channel state)**.** For a channel c_{ij} the *channel state* of c_{ij} is a tuple of variable-length, w_{ij}, whose entries are messages that can be sent/received along c_{ij}. That is, $w_{ij} \in (M_{ij})^*$.

Definition 3.11 (Global channel state)**.** A *global channel state* w is a tuple whose entries are channel states. More formally, $w = w_{1,2} \cdots w_{ij} \cdots w_{h-1,h}$ where $w_{ij} \in (M_{ij})^*$. We write $w[ij]$ to denote the tuple w_{ij} in w, and reuse ε (the empty string) to also stand for a global channel state with all channels empty. Further, we let M be the set of all possible global channel states, $M = (M_{1,2})^* \times \cdots \times (M_{h-1,h})^*$.

Finally, we represent the *system state* as a pair of global process state and global channel state, $\langle q, w \rangle \in Q \times M$.

Now that we have a notion of state, we need to specify how a sequence of event instances impacts the state of the system. For this, we define a process transition function, δ_i, which maps a state at process i and a process i event instance to a new state. This function is defined on sub-sequences of the totally ordered process i traces (Definition 3.5).

Definition 3.12 (Process transition function)**.** Let i be a process index, and L be a log. The *process transition function* for a process i is $\delta_i : Q_i \times \hat{E}_i \rightarrow Q_i$, such that

- $\delta_i(q_0^i, \hat{e}) = q_{\hat{e}} \iff \exists T_i \in L_i, \hat{e} = T_i[1]$
- $\delta_i(q_s, \hat{e}) = q_{s \cdot \hat{e}} \iff \exists T_i \in L_i, (s \cdot \hat{e})$ is a prefix of T_i.

As an example that illustrates δ_i, assume that states are the natural numbers: $Q_i = \mathbb{N}$, and event types are totally ordered: $\exists g, g : E_i \rightarrow \mathbb{N}$. Then, we can define $\delta_i(q, \hat{e})$ as a number that is formed by concatenating q and $g(e)$.

Notice that δ_i has two distinguishing properties:

1. $\forall \hat{e}, \hat{f}, \hat{e} \neq \hat{f} \implies \delta_i(q, \hat{e}) \neq \delta_i(q, \hat{f})$

2. $\delta_i(q, \hat{e}) = \delta_i(q', \hat{e}) \iff q = q'$

These properties encode an *assumption* that we make about process state: we consider two local process states to be different if they were generated by two distinct sequences of events. We discuss the implications of this choice in Section 3.8.

Given a log L we now define a concrete FSM for L, $\mathcal{F}_L$. Dynoptic derives this FSM in step ③ in Figure 3.6. A key property of $\mathcal{F}_L$ is that it accepts all linear extensions of all traces in L, as well as all possible traces that are *stitchings* of different concrete traces that share identical concrete states.

Definition 3.13 (Concrete FSM)**.** Given a log L, a *concrete FSM* $\mathcal{F}_L$ for L is a tuple $\langle S, S_I, \hat{E}, \Delta, S_T \rangle$ whose states (S) are system states. The FSM has one initial system state ($|S_I| = 1$) and the transition function Δ is defined as a composition of the individual process transition functions, except that Δ also handles communication events. Finally, the terminal states (S_T) are states with empty channels, and with each process in a terminal state that was derived by executing all of the event instances for that process in some system trace in L. More formally,

- $S = Q \times M$

- $S_I = \{\langle [q_0^1, \ldots, q_0^h], \varepsilon \rangle\}$

- $\hat{E} = \{\hat{e} \mid \exists \langle T, \prec \rangle \in L, \hat{e} \in T\}$

- $\Delta : Q \times M \times \hat{E} \rightarrow Q \times M$,
  $\Delta(\langle q, w \rangle, \hat{e}_i) = \langle q', w' \rangle$, where:

  - $q' = q[i \leftarrow \delta_i(q[i], \hat{e}_i)]$.

  - $w' = \begin{cases} w[ij \leftarrow (w[ij] \cdot m)] & e_i = c_{ij}!m \\ w[ji \leftarrow \text{tail}] & e_i = c_{ji}?m \\ & w[ji] = m \cdot \text{tail} \\ w & \text{otherwise} \end{cases}$

- $S_T = \{\langle t, \varepsilon \rangle \mid \forall i, t[i] \text{ terminal}\}$

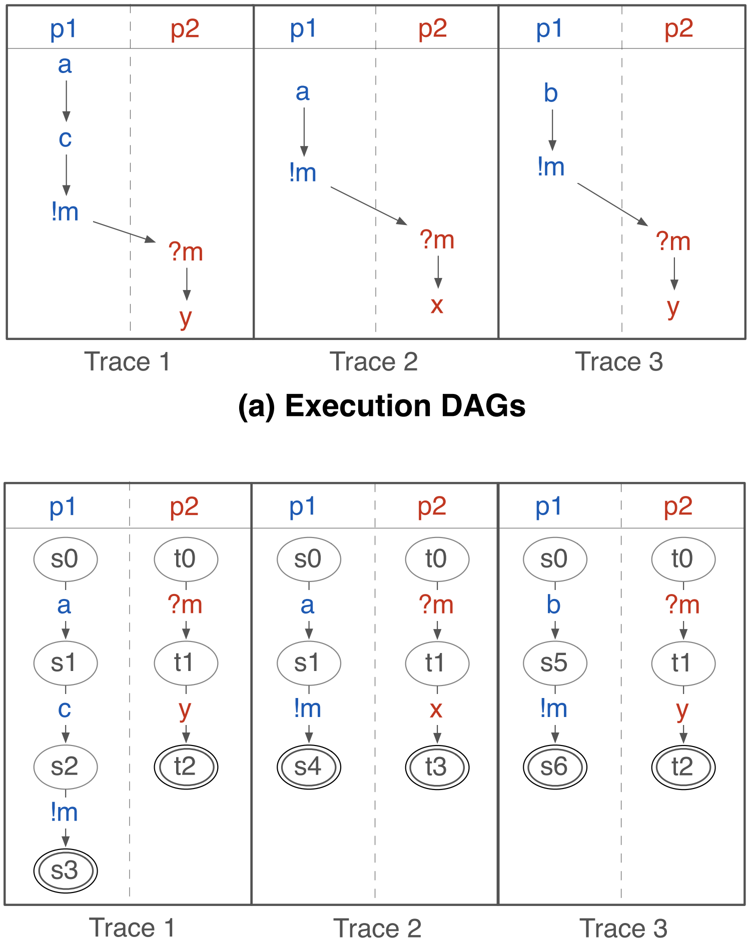
3.8 Validating the mined invariants

Although the invariants Dynoptic mines (described in Section 3.6) are valid for the input traces, it might not be possible to construct a model that satisfies the invariants and accepts all of the input traces. As an example consider the set of traces in Figure 3.10(a) for a system of two processes, p1 and p2, with a single channel from p1 to p2. These traces satisfy the invariant $b_1 \not\to x_2$. However, from these traces we can also observe that after p2 executes the $?m$ event it has no way to decide between the x and y events; from the point of view of p2, both are legal and equally valid, and it is impossible to model p2 in a way that satisfies the $b_1 \not\to x_2$ invariant.

This situation arises due to incompleteness of the traces. This can be caused by insufficient executions ($b_1 \to x_2$ is possible but was not observed) or insufficient information in the trace (the m events that precede x_2 differ from those that precede y_2, in a way that is not recorded in the trace). In either case, the $b_1 \not\to x_2$ invariant is an undesirable false positive.

Dynoptic overcomes this problem by detecting and omitting invariants of the above form, a process we term *invariant validation* (step ④ in Figure 3.6). Figure 3.10(b) shows the traces from Figure 3.10(a) with intermediate anonymous states (note that the state of p2 after executing $?m$ is identical across *all* three traces). These anonymous local process states, along with concrete channel contents, are used to derive the concrete FSM (Figure 3.11) corresponding to the above execution DAGs. This concrete FSM is checked for paths that violate the mined invariants, and these invariants are reported to the user and omitted from the set of Dynoptic steps that follow. The output of this process is a set of *valid* invariants. Figure 3.12 details the `ValidateInvariants` procedure.

It is worth noting that invariant validation is necessary because of how Dynoptic generates anonymous local process states: the local process state is completely determined by the set of



(a) Execution DAGs

(b) Traces with anonymous states

Figure 3.10: **(a)** Execution DAGs parsed from an input log with three traces, each one containing events from two processes: p1 and p2. Communication events omit the channel name as all messages flow from p1 to p2. Note that the execution DAGs satisfy the invariants $c_1 \rightarrow y_2$ and $b_1 \not\rightarrow x_2$. **(b)** The traces from (a) with added per-process anonymous states. Note the reuse of states s0, s1, t0, t1, and t2.

executed *local* events (the assumption mentioned in Section 3.7). If a process state was determined not by just the local process events, but by the global history of events at all processes in the system then we would not need to perform invariant validation (all invariants would be satisfied in the

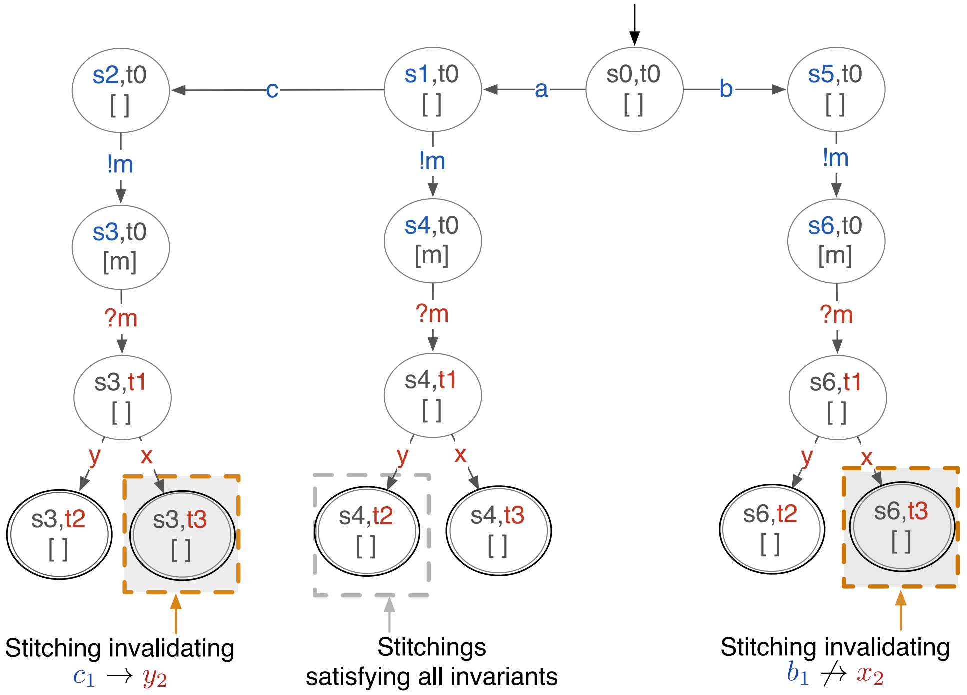


Figure 3.11: The concrete FSM for the execution DAGs in Figure 3.10(a). The unshaded box in the middle highlights a stitching that satisfy all of the mined invariants. The two shaded boxes highlight stitchings that invalidate both the $c_1 \rightarrow y_2$ and the $b_1 \not\rightarrow x_2$ invariants. During *invariant validation*, Dynoptic model-checks the concrete FSM to determine which of the mined invariants are *valid*. In this example, both $c_1 \not\rightarrow y_2$ and $b_1 \not\rightarrow x_2$ are invalid.

concrete FSM). However, an advantage of our design choice is that it mitigates the much more difficult problem of state explosion, as states would be differentiated to a very fine degree.

One concrete method by which an invalid invariant can be made valid is by refining the existing event types in the log based on remote events. For example, in the traces in Figure 3.12(a), if message m was differentiated into m_{ac}, m_a, and m_b then all of the mined invariants would be valid.

```
function ValidateInvariants($\mathcal{F}_L$,$Invs$):
  let $Invs' = Invs$
  foreach complete path $p$ in $\mathcal{F}_L$:
    foreach $inv \in Invs$:
      if ($p$ violates $inv$):
        $Invs' = Invs' \setminus \{inv\}$
  return $Invs'$
```

Figure 3.12: `ValidateInvariants` model-checks the mined invariants *Invs* in the concrete FSM $\mathcal{F}_L$ and returns a subset of invariants, $Invs' \subseteq Invs$, that are valid for $\mathcal{F}_L$.

3.9 Abstracting a concrete FSM

The concrete FSM $\mathcal{F}_L$ accepts all possible serialized sequences of event *instances* across all executions in a log L. We let $\mathbf{P}$ represent a partitioning of states in $\mathcal{F}_L$, (i.e., a partitioning of $Q \times M$). Dynoptic's aim can now be defined as deriving an *abstract* FSM (AFSM) $\mathcal{A}_L(\mathbf{P})$ whose states are partitions in $\mathbf{P}$ and which accepts sequences of *events* — rather than *event instances*. That is, transitions between states (partitions) in $\mathcal{A}_L(\mathbf{P})$ are generated through existential abstraction — there is a transition from $P_1 \in \mathbf{P}$ to $P_2 \in \mathbf{P}$ on event e iff there are concrete states s_1 and s_2 such that $s_1 \in P_1, s_2 \in P_2$, and there is a transition from s_1 to s_2 on some event instance $\hat{e}$, corresponding to e.

Note that $\mathcal{A}_L(\mathbf{P})$ will trivially accept the serialized event instance sequences. But, we also want the abstract FSM to generalize, or accept unobserved sequences of events that satisfy the valid invariants.

Now, we formally define $\mathcal{A}_L(\mathbf{P})$.

Definition 3.14 (Abstract FSM (AFSM))**.** For a log L let $\mathcal{F}_L = \langle S, S_I, \hat{E}, \Delta, S_T \rangle$ be the concrete FSM for L, and Let $\mathbf{P}$ be a partitioning of S. That is, $S = \cup_{P \in \mathbf{P}} P$, and $\forall s \in S, \exists! P \in \mathbf{P}, s \in P$. Then, an *abstraction of* $\mathcal{F}_L$, or an *abstract FSM* of L is an FSM $\mathcal{A}_L(\mathbf{P}) = \langle \mathbf{P}, P_I, E, \Delta', P_T \rangle$, where

- $P_I = \{P \in \mathbf{P} \mid S_I \cap P \neq \emptyset\}$
- $E = \{e \mid \hat{e} \in \hat{E}\}$

- $\Delta'(P,e) = P' \iff \exists q \in P, q' \in P', \hat{e} \in \hat{E}, \Delta(q,\hat{e}) = q'$

- $P_T = \{P \in \mathbf{P} \mid S_T \cap P \neq \emptyset\}$

An important feature of an AFSM is that it generalizes observed system states. A partition contains a finite number of observed system states, but through loops with transitions that modify channel state, an AFSM can generate arbitrarily long channel contents, leading to an arbitrarily large number of system states. We may not have observed these system states, but an AFSM model generalizes to predict that they are feasible.

Dynoptic uses a top-*k* partitioning strategy for generating an initial AFSM for a concrete FSM (step ⑤ in Figure 3.6). This partitioning assigns two system states to the same partition if and only if the top-*k* message sequences in the channel states of the two states are identical. For example, if the system has two channels, c_{12} and c_{21}, and there are three concrete states: s_1, s_2, s_3, and $s_1.channels = \{c_{12} : [], c_{21} : [m]\}$, $s_2.channels = \{c_{12} : [], c_{21} : [m,m]\}$, $s_3.channels = \{c_{12} : [l], c_{21} : [m]\}$ then the top-1 contents of s_1 and s_2 are $\{c_{12} : [], c_{21} : [m]\}$ (the second m in $s_2.c_{21}$ is not included), while the top-1 contents of s_3 are $\{c_{12} : [l], c_{21} : [m]\}$. Therefore, in a top-1 partitioning strategy, s_1 and s_2 would map to the same partition, while s_3 would map to a different partition.

Definition 3.15 (Top-*k* partitioning)**.** Let S be a set of system states. $\mathbf{P}_k$ is a top-*k* partitioning of S if $\forall P \in \mathbf{P}_k$, $\langle q,w \rangle, \langle q',w' \rangle \in P \iff \forall c_{ij}, \forall g, 1 < g \leq k, w[ij][g] = w'[ij][g]$.

A CFSM is a set of per-process FSMs that communicate over FIFO channels.

Definition 3.16 (Communicating FSM (CFSM))**.** A CFSM of h processes is a tuple of process FSMs $\langle F_i \rangle_{i=1}^{h}$, where $F_i = \langle Q_i, I_i, E_i, \Delta_i, T_i \rangle$ is a process i FSM.

An AFSM is an abstraction of concrete FSM. Dynoptic's goal is to construct a communicating FSM (CFSM). The AFSM can be thought of as a cross product of the per-process FSM, and the CFSM can therefore be reconstructed from the AFSM. Figure 3.13 details the `AFSMtoCFSM` procedure for converting an AFSM into a CFSM (step ⑥ in Figure 3.6).

3.10 Model-checking an AFSM

Dynoptic uses the McScM [71] model checker to check if an invariant holds in the CFSM that corresponds to an AFSM (step ⑦ in Figure 3.6). Since McScM reasons about state (un-)reachability,

```
function AFSMtoCFSM(AFSM $\mathcal{A}$):
  $\mathcal{A} = \langle \mathbf{P}, P_I, E, \Delta, P_T \rangle$
  foreach $i \in [1, \ldots, h]$:
    // $\mathcal{A}_i$ is $\mathcal{A}$ with non pid $i$ events replaced with $\varepsilon$.
    let $\mathcal{A}_i = (\mathbf{P}, P_I, E_i, \Delta_i, P_T)$, where
          $E_i = \{e_i \in E\}$, and
          $\Delta_i(q, \varepsilon) = q' \iff \Delta(q, e_j) = q', i \neq j$
          $\Delta_i(q, e_i) = q' \iff \Delta(q, e_i) = q'$
    let $F_i$ = eliminate-$\varepsilon$ ($\mathcal{A}_i$)
  CFSM $C_{\mathcal{A}}$ = $\langle F_i \rangle_{i=1}^{h}$
  return $C_{\mathcal{A}}$
```

Figure 3.13: AFSMtoCFSM translates an AFSM $\mathcal{A}$ into a CFSM $C_{\mathcal{A}}$; eliminate-ε performs standard ε transition elimination [74].

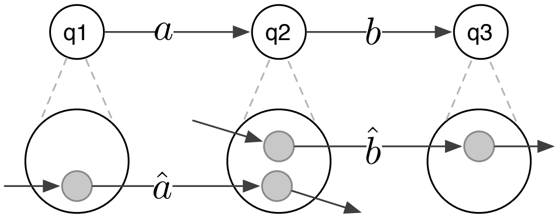


Figure 3.14: An example of an AFSM path, $[a, b]$, that can be eliminated by refining the abstract state $q2$, separating the two concrete states that generate the abstract path.

Dynoptic encodes a temporal invariant in terms of states that can only be reached if the sequence of executed events violates the invariant. We now describe this encoding and the overall CFSM model checking procedure.

Model checking an event invariant (Section 3.6) in a CFSM (Section 3.9) is a three step process. First, the CFSM must be augmented with synthetic "tracking" transitions that record when an event type mentioned by the invariant is executed by the model checker. Second, the negation of the invariant must be encoded as a set of "bad states" of the CFSM. If the model checker finds an execution that can reach one of these states, it will emit the corresponding path. The third, and final,

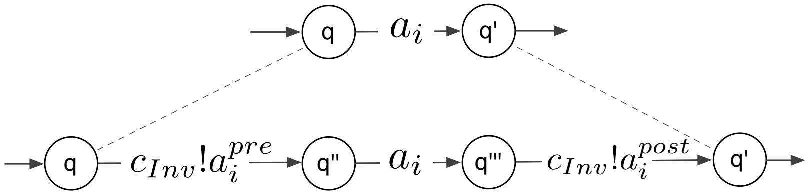


Figure 3.15: Transforming an edge in a CFSM to track a_i.

step is to rewrite this path into a valid counter-example in the GFSM model. We now describe each of these steps in more detail.

3.10.1 Preparing a CFSM for model checking

Consider an event invariant *Inv*, such that $Inv = a_i \mathcal{T} b_j$, with $\mathcal{T} \in \{\rightarrow, \not\rightarrow, \leftarrow\}$. And, let $\mathcal{C}$ be a CFSM in which we want to check if *Inv* holds.

We modify $\mathcal{C}$ to produce a new CFSM model, $\mathcal{C}'$, which will be used as input to the McScM model checker. We transform $\mathcal{C}$ into $\mathcal{C}'$ in two ways — we convert local events into send events on a synthetic channel, and we add synthetic "tracking events" to track the execution of a_i and b_j event types.

Handling local events. The McScM model checker does not model local events (it only supports message send and receive events). We cannot omit local events from $\mathcal{C}'$, as we need to be able to unambiguously map an McScM counter-example path in $\mathcal{C}'$ to a path in $\mathcal{C}$. We therefore add a synthetic "local events" channel, c_{local}, and allow all processes to send messages on c_{local}. This channel is write-only — processes never receive on c_{local}. Finally, each process local event type, e_i, in $\mathcal{C}$ is translated into the event type $c_{local}!e_i$ in $\mathcal{C}'$. That is, we replace transitions of the form $\Delta_i(q, e_i) = q'$ with $\Delta_i(q, c_{local}!e_i) = q'$.

Tracking events necessary for checking Inv. Given a CFSM and a set of "bad states", the McScM model checker checks if there is an execution of the input CFSM that causes it to reach one of the bad states. A bad state is a kind of a *system state* (introduced in Section 3.7); it consists of a global process state and a global channel state. In McScM, the global channel state is expressed with regular expressions over channel contents.[5]

To check *Inv* with McScM, we therefore need to generate a set of system states of $\mathcal{C}'$, each of which encodes a violation of *Inv*. To generate this encoding for $Inv = a_i \mathcal{T} b_j$, we need to track when a_i or b_j occur. We do this with "tracking events", which track the occurrence of a_i and b_j as send events to a synthetic *Inv*-specific channel, c_{Inv}. This channel is used exclusively for checking *Inv*. As with the synthetic local events channel, all processes can send on c_{Inv}, and no process ever receives from this channel. Figure 3.15 illustrates this tracking. More formally, if Δ_i is the process i FSM transition function in $\mathcal{C}$, then we track a_i and b_j event types as follows:

- Replace transitions of the form $\Delta_i(q, a_i) = q'$ with:
  - $\Delta_i(q, c_{Inv}!a_i^{pre}) = q''$
  - $\Delta_i(q'', a_i) = q'''$
  - $\Delta_i(q''', c_{Inv}!a_i^{post}) = q'$

  Where, q'', q''' are synthetic states with just the transitions above, and $c_{Inv}!a_i^{pre}$ and $c_{Inv}!a_i^{post}$ are synthetic event types that maintain a record of when a_i occurs as messages in c_{Inv}.

- Similarly, to keep track of b_j, replace $\Delta_j(p, b_j) = p'$ with:
  - $\Delta_j(p, c_{Inv}!b_j^{pre}) = p''$
  - $\Delta_j(p'', b_j) = p'''$
  - $\Delta_j(p''', c_{Inv}!b_j^{post}) = p'$

Note that both of the transformations retain the event that is being tracked (a_i or b_j). This is necessary as we want $\mathcal{C}'$ to have identical behavior to $\mathcal{C}$.

To see why we need to augment both a_i and b_j with a pre and a post event, consider other strategies, with less overhead (fewer synthetic events). For example, assume that $Inv = a_i \rightarrow b_j$ and we repaced transitions of the form $\Delta_i(q, a_i) = q'$ with $\Delta_i(q, c_{Inv}!a_i^{pre}) = q''$ and $\Delta_i(q'', a_i) = q'$; and we replaced transitions of the form $\Delta_j(p, b_j) = p'$ with $\Delta_j(p, c_{Inv}!b_j^{pre}) = p''$ and $\Delta_j(p'', b_j) = p'$.

[5] Therefore, an McScM bad state is, in fact, a *set* of system states, as defined in Section 3.7.

Then, consider the following contents of c_{Inv}: $[a_i^{pre}, b_j^{pre}]$. These two tracking events imply one of three executions:

1. $c_{Inv}!a_i^{pre} \rightarrow a_i \rightarrow c_{Inv}!b_j^{pre} \rightarrow b_j$ *Inv* true

2. $c_{Inv}!a_i^{pre} \rightarrow c_{Inv}!b_j^{pre} \rightarrow a_i \rightarrow b_j$ *Inv* true

3. $c_{Inv}!a_i^{pre} \rightarrow c_{Inv}!b_j^{pre} \rightarrow b_j \rightarrow a_i$ *Inv* false

The above example indicates that the simpler strategy results in ambiguity — given the contents of channel c_{Inv} we cannot tell if *Inv* satisfies the execution that produced the tracing events in c_{Inv}. Likewise, we can show that other strategies that do not include all of the four pre and post events lead to ambiguity.

3.10.2 Expressing invariants as McScM bad states

To model check *Inv* in $\mathcal{C}'$ we need to construct bad states for *Inv*. Each such state needs to specify the states of all of the processes in the system and describe the contents of all channels in $\mathcal{C}'$ as regular expressions. First, we overview the McScM regular expression syntax for denoting channel contents:

• $a\,.\,b$	concatenation of *a* and *b*
• $a\hat{}*$	zero or more *a*
• $a\hat{}+$	one or more *a*
• $a \mid b$	either *a* or *b*
• ( )	grouping
• _	empty channel

Using the above syntax, we now define some basic patterns that we will reuse in specifying bad states.

- $\texttt{ANY} = (c_{Inv}!a_i^{pre} \mid c_{Inv}!a_i^{post} \mid c_{Inv}!b_i^{pre} \mid c_{Inv}!b_i^{post})\hat{}*$

  Accepts an arbitrary sequence of tracking events.

- $\texttt{A} = (c_{Inv}!a_i^{pre} \mid c_{Inv}!a_i^{post})$

  Implicitly accepts an a_i via the corresponding tracking events.

- $\texttt{B} = (c_{Inv}!b_i^{pre} \mid c_{Inv}!b_i^{post})$

  Implicitly accepts a b_j via the corresponding tracking events.

Now, let Acc be the set of all accepting global process states. That is, $Acc = \{a|a[i]$ accept state for process $i\}$. We will now form bad states based on the kinds of invariant we are checking. Note that we only specify the contents of c_{Inv} channel; the contents of all other channels are: (1) all system channels (non-invariant and non-local-event channels) are specified as being empty – "_", and (2) the local events channel is specified as containing an arbitrary sequence of local events[6].

Now, we can specify the set of bad states B for each invariant type:

- $\mathcal{T} = \rightarrow \qquad B = \{\langle a, \texttt{ANY.A}\hat{}+\rangle | a \in Acc\}$

- $\mathcal{T} = \not\rightarrow \qquad B = \{\langle a, \texttt{ANY.A.ANY.B.ANY}\rangle | a \in Acc\}$

- $\mathcal{T} = \leftarrow \qquad B = \{\langle a, \texttt{B}\hat{}+\texttt{.ANY}\rangle | a \in Acc\}$

3.10.3 Post-processing the counter-example path

The invariant counter-example path returned by McScM for $\mathcal{C}'$ must be post-processed to derive a valid counter-example for $\mathcal{C}$. The path is transformed in two ways, that counter the transformations described in 3.10.1: (1) remove all tracking events, and (2) replace events that are send events to the local channel, c_{local}, to be local events in $\mathcal{C}$.

[6] This is $(e^1|\ldots|e^n)\hat{}*$, where e^i is a local event in $\mathcal{C}$.

3.11 Refining an AFSM to satisfy invariants

Model-checking an invariant (previous section) can produce one of three results:

(1) McScM fails to terminate (CFSM state reachability is an undecidable problem [27]). Dynoptic stops (times out) McScM executions that exceed a user-defined threshold (defaulted to 5 minutes). Dynoptic then attempts to check a different invariant first before coming back to the invariant that timed out [7]. If checking each of the remaining invariants times out, Dynoptic gives up.

(2) The invariant holds in the model. Dynoptic either moves on to the next invariant, or terminates and outputs the model if all invariants have been satisfied.

(3) The invariant does not hold and McScM finds and reports a counter-example CFSM execution. A *CFSM execution* is a sequence of events that abides by CFSM semantics (e.g., a process can only receive a message if that message is at the top of the channel), and the events sequence leads the system into state in which each of the processes is in a terminal state, and all channels are empty. A *counter-example* CFSM execution is a sequence of events that violates the invariant. In this case, Dynoptic uses counter-example guided abstraction refinement (CEGAR) approach [32] to refine the AFSM to eliminate the counter-example. The rest of this section details this refinement.

Dynoptic uses partition refinement to eliminate a counter-example for an invariant — once all invariant counter-examples have been eliminated, the model satisfies the invariant. Figure 3.14 illustrates how the concrete states from the log may generate a counter-example trace in the AFSM. The McScM invariant counter-example is a CFSM execution, but refinement must occur in the AFSM. So, Dynoptic maps the McScM-generated *CFSM counter-example* into an *AFSM counter-example*. Figure 3.16 describes this translation.

Definition 3.17 (Invariant counter-example)**.** Let L be a log, let Inv be a valid invariant for L, and let $\mathcal{A}$ be an AFSM for L. Then, a *CFSM counter-example* to Inv in CFSM $\mathcal{C}_{\mathcal{A}}$ (derived with `AFSMtoCFSM` in Figure 3.13) is a sequence of events p that does not satisfy Inv and is an execution of $\mathcal{C}_{\mathcal{A}}$. The *AFSM counter-example* corresponding to p is the sequence of sets, S, where each set contains paths in $\mathcal{A}$. The sequence S is derived using $S = \texttt{TranslatePath}(p, \mathcal{C}_{\mathcal{A}})$ (Figure 3.16).

[7] After refining the model to satisfy an invariant, a previously difficult-to-check invariant may become trivial to model-check

```
function TranslatePath($p, \mathcal{C}_{\mathcal{A}}$):
  // $p$ is a complete events path in CFSM $\mathcal{C}_{\mathcal{A}}$
  let AFSM $\mathcal{A} = \langle \mathbf{P}, P_I, E, \Delta, P_T \rangle$
  foreach $i \in [1, \ldots, h]$:
    let $S_i = \{s \mid s$ an events path in $\mathcal{A}$ from $P_1$ to $P_k$,
                 $\pi(s, E_i) = \pi(p, E_i), P_1 \in P_I$,
                 $\exists \langle q, w \rangle \in P_k, q[i]$ terminal, and
                            $\forall j, w[ji] = \varepsilon\}$
  return $[S_1, \ldots, S_h]$
```

Figure 3.16: `TranslatePath` translates an events path p in a CFSM $\mathcal{C}_{\mathcal{A}}$ into S, a sequence of sets of paths in $\mathcal{A}$. Each set in S maps to one event in p.

Note that the AFSM counter-example is a sequence of sets of AFSM paths, one for each process in the system. This is because the process-specific events subsequence of a CFSM execution may be generated by multiple paths in the AFSM (due to the CFSM construction based on ε-transitions in Figure 3.13).

Once the AFSM counter-example is generated, Dynoptic uses partition refinement (`Refine` in Figure 3.17) to eliminate the CFSM counter-example by transforming the AFSM into a more concrete (or less abstract) AFSM. Refinement corresponds to step ⑧ in Figure 3.6. For each AFSM path, `Refine` identifies the set of partitions that stitch concrete observations, as in partition $q2$ in Figure 3.14. It then refines all partitions in a set that is smallest across all processes and returns the refined AFSM.

A refined AFSM is more concrete because the states of the refined AFSM contain fewer system states observed in the log, and therefore the AFSM is closer to the concrete FSM.

Definition 3.18 (AFSM Refinement)**.** An AFSM $\mathcal{A}_L(\mathbf{P}')$ is a *refinement* of AFSM $\mathcal{A}_L(\mathbf{P})$ if $\forall P' \in \mathbf{P}', \exists P \in \mathbf{P}, P' \subseteq P$.

The complete Dynoptic algorithm is listed in Figure 3.18. Next, we prove a few key properties of the Dynoptic process.

```
function Refine($\mathcal{A}, S, L$):
  // $S$ is derived via TranslatePath
  let AFSM $\mathcal{A} = (\mathbf{P}, P_I, E, \Delta, P_T)$
  let $Stitch_{min} = \emptyset$
  foreach $i \in [1, \ldots, h]$:
     foreach $s \in S[i]$:
        let $P_s$ = state sequence for $s$ in $\mathcal{A}$
        // Derive stitching states, e.g., $q2$ in Fig. 3.14
        let $Stitch_s = \{p \mid p$ a stitching state in $P_s\}$
        if $Stitch_s = \emptyset$:
          next $i$
     // Partitions set shared by paths that generate $s$
     $Stitch_i = \cap_{s \in S[i]} Stitch_s$
     // $Stitch_{min}$ is the min size set over all $Stitch_i$
     if $Stitch_{min} = \emptyset$ or $|Stitch_i| < |Stitch_{min}|$:
       $Stitch_{min} = Stitch_i$
  let $\mathbf{P}' = \mathbf{P}$ with all partitions in $Stitch_{min}$ refined
  // Derive $P_I', \Delta'$, and $P_T'$ from $\mathbf{P}'$ as in Def. 3.14.
  let AFSM $\mathcal{A}' = \langle \mathbf{P}', P_I', E, \Delta', P_T' \rangle$
  return $\mathcal{A}'$
```

Figure 3.17: `Refine` removes an invariant counter-example from an AFSM by refining the set of process paths in S that require the fewest refinements.

3.12 Formal evaluation

We begin with an observation: the concrete FSM satisfies all valid invariants. This is true by construction in `ValidateInvariants` in Figure 3.12.

Observation 3.1 (Concrete FSM satisfies valid invariants)**.** Let L be a log, and let $Invs$ be the set of invariants that are valid in $\mathcal{F}_L$. Then, $\forall Inv \in Invs, t \in Lang(\mathcal{F}_L), t$ satisfies Inv.

A key property of the `Refine` procedure in Figure 3.17 is that it eliminates the counter-example path from the CFSM corresponding to the current AFSM. We prove this next.

Theorem 3.1 (Refinement eliminates counter-examples)**.** *Let p be a CFSM counter-example path for Inv in $C_{\mathcal{A}}$ and let $S =$ `TranslatePath`$(p, C_{\mathcal{A}})$, and let $\mathcal{A}' =$ `Refine`$(\mathcal{A}, S, L)$. Then, p is not*

```
function Dynoptic(Log $L,k$):
  let $Invs =$ ValidateInvariants(MineInvariants($L$))
  let $\mathcal{F}_L =$ Concrete FSM for $L$
  let $\mathcal{A} =$ AFSM for $\mathcal{F}_L$ with partitioning $\mathbf{P}_k$
  let $\mathcal{C}_{\mathcal{A}}$ = AFSMtoCFSM($\mathcal{A}$)
  foreach $Inv \in Invs$:
    while ($\mathcal{C}_{\mathcal{A}}$ violates $Inv$): // Call to model checker.
      let $p =$ counter-example path for $Inv$ in $\mathcal{C}_{\mathcal{A}}$
      let $S =$ TranslatePath($p, \mathcal{C}_{\mathcal{A}}$)
      $\mathcal{A}$ = Refine($\mathcal{A},S,L$)
      $\mathcal{C}_{\mathcal{A}}$ = AFSMtoCFSM($\mathcal{A}$)
  return $\mathcal{C}_{\mathcal{A}}$
```

Figure 3.18: The complete Dynoptic algorithm.

a counter-example to Inv in $\mathcal{C}_{\mathcal{A}'}$. That is, p is not a valid execution of $\mathcal{C}_{\mathcal{A}'}$.

Proof of Theorem 3.1. We prove this by contradiction. Assume that p is a sequence of events that is a valid execution of $\mathcal{C}_{\mathcal{A}'}$ and that p violates Inv. Let $Stitch_{min} = Stitch_i$ for some i in the execution of `Refine(`$\mathcal{A},S,L$`)` procedure in Fig. 3.17. Also, let $\mathcal{C}_{\mathcal{A}'} = \langle F_i \rangle_{i=1}^{h}$.

For p to be a valid execution in $\mathcal{C}_{\mathcal{A}'}$, the sub-sequence of process i events p_i, $p_i = \pi(p, E_i)$, must be a valid execution in F_i. The Procedure `AFSMtoCFSM` in Fig. 3.13 constructs F_i to accept p_i iff there is a complete path s in $\mathcal{A}'$, such that $p_i = \pi(s, E_i)$. However, any such s must also be in $Stitch_i$. Therefore, after refining $Stitch_i$, s can no longer be a valid path in $\mathcal{A}'$. Contradiction.

□

Now, we prove that the `Dynoptic` procedure in Figure 3.18 returns a CFSM model that satisfies all of the valid invariants.

Theorem 3.2 (True invariant satisfiability)**.** *For a given log L, `Dynoptic` produces a CFSM model that satisfies all of the event invariants that are valid in $\mathcal{F}_L$.*

Proof of Theorem 3.2. For a log L with a total of n event instances, Dynoptic can refine the initial abstract FSM for L, $\mathcal{A}_L(\mathbf{P}_k)$, at most n times. This is because after n refinements, each partition in the abstract FSM would map to exactly one concrete state, and a singleton partition cannot be refined further.

Let $\mathcal{A}$ be the abstract FSM after n refinements of $\mathcal{A}_L(\mathbf{P}_k)$. Because $\mathcal{A}$ maps each event instance to a unique partition, it is indistinguishable from the concrete FSM it abstracts, $\mathcal{F}_L$. Therefore, $Lang(\mathcal{A}) = Lang(\mathcal{F}_L)$. By Observation 3.1, $\mathcal{F}_L$ satisfies all valid invariants, therefore so does $\mathcal{A}$.

Since Dynoptic does not terminate until all the valid invariants are satisfied in the abstract FSM, it either returns $\mathcal{A}$ after n refinements, or it returns a smaller (and more abstract) $\mathcal{A}'$. In both cases, the returned AFSM satisfies all of the valid invariants.

□

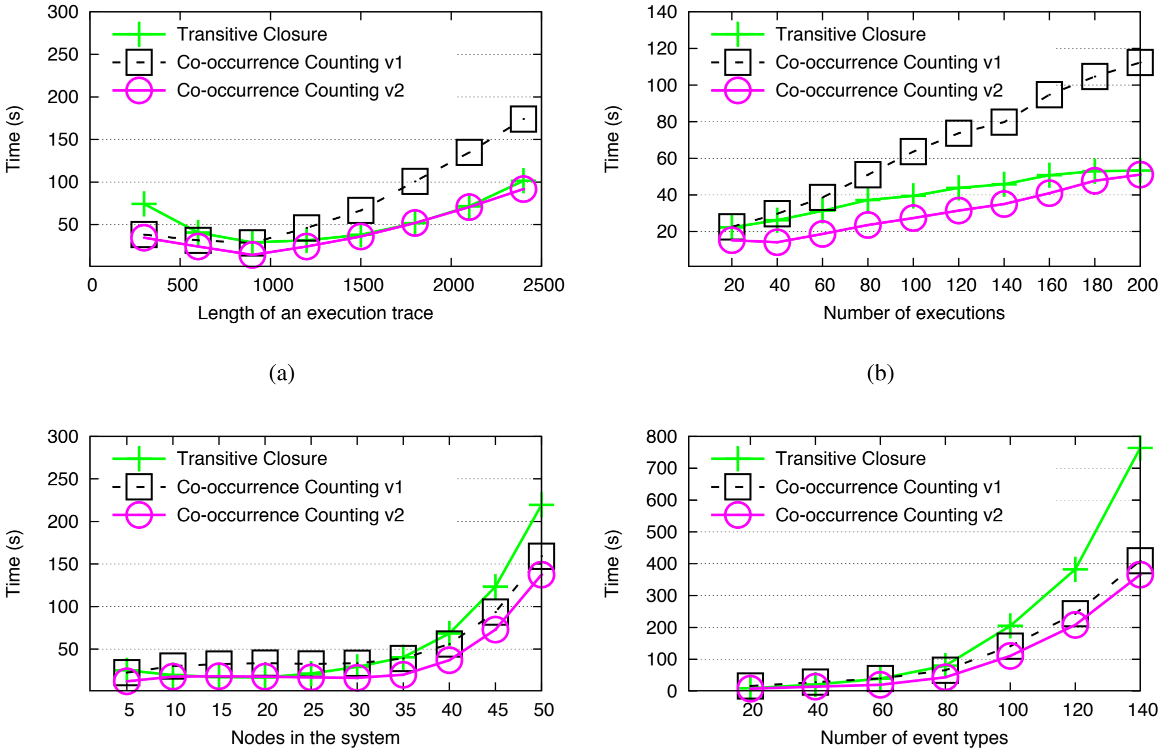
3.13 Experimental evaluation

The Dynoptic prototype is implemented in Java and uses the McScM model checker as a black box. We leverage graphviz [63] for model visualization.

In this section we first present an evaluation of Dynoptic's invariant mining algorithms detailed in Section 3.6. Then, we present results from applying Dynoptic to different networked systems. We evaluated Dynoptic on logs produced by a simulator of the stop-and-wait protocol and by two real systems — the TCP stack of OS X; and Voldemort [125], an open source project that implements a distributed hash table [46] and is used in data centers at companies like LinkedIn. Finally, we present results from a user study in which a group of undergraduate students evaluated the efficacy of CFSM models in finding bugs.

3.13.1 Invariant mining performance

We compare the performance of the transitive-closure-based algorithm with the performance of the two co-occurrence counting algorithms. We evaluate the algorithms on synthetic partially ordered logs that we generated using a discrete-time simulator that simulates a set of concurrent communicating hosts. The evaluation's focus is on mining scalability since system size and log size are a major concern for practical log analysis. We first describe the simulator, and then present and discuss the results.



(a)

(b)

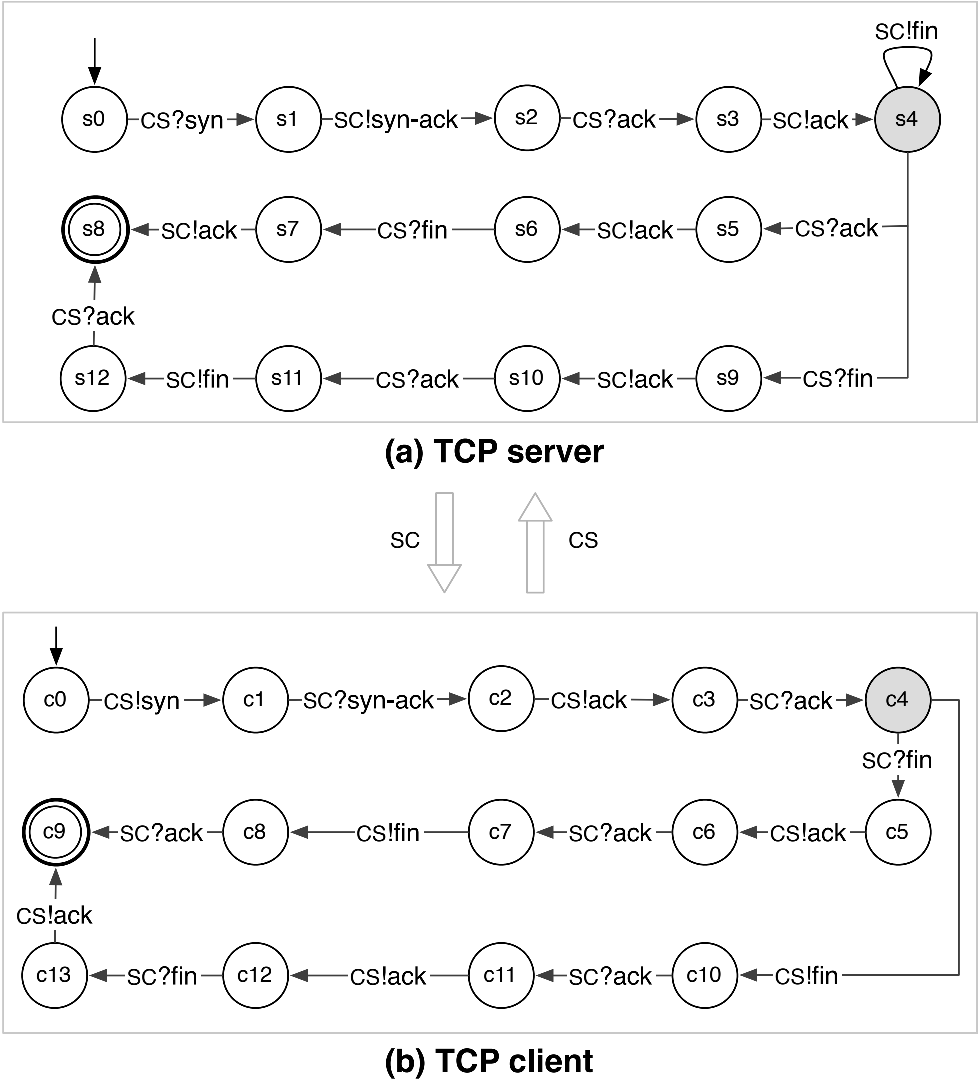
(c)

(d)

Figure 3.19: Invariant mining time for the transitive closure and the co-occurrence counting algorithms on logs generated by the simulator described in Section 3.13.1. In each of the figures, a single log feature is varied: (a) number of hosts, (b) execution length, (c) number of executions, and (d) number of event types. The other log features were held constant in the same figure, and were identical across figures: 30 hosts, 50 host event types per host (= 1,500 total since event types at different hosts are considered different), 1,000 events per execution, and 50 executions.

Log-generating system simulator

The simulator is parameterized by the number of hosts, number of events types, number of events per execution, and the number of executions. For each event, the simmulator chooses the host that will execute the event and the event's type, both with uniform probability. The simulator also decides



(a) TCP server

(b) TCP client

Figure 3.20: The inferred **(a)** TCP server, and **(b)** TCP client state machines. The server communicates with the client over the `SC` channel, and the client communicates with the server over the `CS` channel. Shaded states represent the `connection-established` states.

to either associate the event with sending a message to some other random node (with probability 0.3); or, if the node has messages in its queue, to associate the event with receiving a message (with probability 0.4); or to make the event local to the selected host (remaining probability). Any outstanding messages in the receive queues are flushed when the simulation ends.

The simulator maintains vector clocks, following the procedure from Section 3.1.2. The simulator outputs a log of multiple executions, or system traces, composed of events; each event has a vector timestamp.

Study methodology and results

We implemented the three invariant-inferring algorithms in Java and ran experiments on an Intel i7 (2.8 GHz) OS X 10.6.7 machine with 8GB RAM. Our implementation used the Floyd-Warshall [54] algorithm to compute the transitive closure. As part of our future work, we plan to implement a more efficient transitive closure algorithm special to DAGs (e.g., [62]).

Our evaluation goal was to measure how the two versions of the co-occurrence counting algorithm scale, as compared to the transitive-closure-based algorithm, in four dimensions: (1) with the length of the system trace, (2) with the number of traces in the log, (3) with the number of hosts, and (4) with the number of event types. For each of the dimensions, we first used the simulator to generate a set of logs, varying that dimension and keeping the others constant. The constant values were: 30 hosts, 50 host event types per host (= 1,500 total since event types at different hosts are considered different), 1,000 events per execution, and 50 executions. We ran each algorithm 5 times and report the median value.

Figure 3.19 plots the results of our simulations. Figure 3.19(a) illustrates the algorithms' scalability with respect to the length of the system trace and Figure 3.19(b) with respect to the length of the log (i.e., the number of traces). In both cases, the transitive-closure-based algorithm outperformed the co-occurrence counting algorithm v1. The co-occurrence counting algorithm v2 (without $\nparallel$) performed best.

Figure 3.19(c) illustrates the algorithms' scalability with respect to the number of hosts and Figure 3.19(d) with respect to the number of event types. In both cases, the transitive-closure-based algorithm underperformed the co-occurrence counting v1. Again, the co-occurrence counting algorithm v2 performed best.

3.13.2 Stop-and-wait protocol

We considered traces from a simulator of the stop-and-wait protocol described in Section 3.3. We derived a diverse set of traces by varying message delays to produce different message interleavings. Dynoptic mined a total of 66 valid invariants, and Figure 3.5 shows the Dynoptic-derived model. This experiment was a sanity check to verify that Dynoptic performed as expected on this well-understood protocol, when faced with delays and concurrency-induced non-determinism in interleavings. The model Dynoptic derived is identical to the true model of the stop-and-wait protocol.

3.13.3 TCP

The TCP protocol uses a three-way opening handshake to establish a bi-directional communication channel between two end-points. It tears down and cleans up the connection using a four-way closing handshake. The TCP state machine is complicated by the fact that packet delays and packet losses cause the end-points to timeout and re-transmit certain packets, which may in turn induce new messages. Our initial goal was to model common-case TCP behavior, so we did not explore these protocol corner cases.

We used netcat and dummynet [110] to generate and control TCP packet flow. We captured packets using tcpdump and then annotated the log to include vector timestamps. The resulting traces were fed into Dynoptic, which was used to model just the opening and closing TCP handshakes.

For the captured TCP log, Dynoptic identified 149 valid invariants, some of which are not true of the complete protocol (e.g., because the input traces did not contain certain packet retransmissions). The Dynoptic-derived CFSM model is shown in Figure 3.20. The shaded states `s4` and `c4` represent the server and client `connection-established` states, which is attained when the two end-points have successfully set up the bi-directional channel. Transitions up to these two states model the opening handshake, while transitions after these states model the closing handshake. The closing handshake is split into a server-initiated tear-down sequence (middle row of states) and a client-initiated tear-down (bottom-most row of states).

The derived model is accurate except for the self-loop on state `s4` in Figure 3.20(a). This loop appears because `s4` merges the `connection-established` state with the state after the server has initiated the closing handshake. This loop appears to contradict the `SC!fin` $\not\rightarrow$ `SC!fin` invariant, which *is* mined by Dynoptic for the input traces and is valid. However, the model checker only considers counter-examples that terminate. Note that if the loop in the model is traversed twice then the client will not be able to consume both server `fin` packet copies and will enter an unspecified reception error state and therefore not terminate. Such unspecified reception states are typically undesirable, as they can be confusing. The McScM model checker can be used to detect these states and further refinement will eliminate them. Implementing this elimination remains as part of future work.

Figure 3.20 also illustrates a key feature of Dynoptic— the user decides (by specifying a set of line-matching regular expressions) what information in the log is relevant to the modeling task. Thus, for each use of Dynoptic, the user decides on the trade-off between comprehensibility of the model (e.g., model size) and the amount of information lost/retained by the modeling process. For example, the TCP model in Figure 3.20 is simple to understand, but it omits TCP sequence numbers, data packet, and other details that were present in the input log.

3.13.4 Voldemort distributed hash table

Voldemort implements a distributed hash table with a client API that has two main methods: `put(k,v)` — associate the value `v` with the key `k`, and `get(k)` — retrieve the current value associated with the key `k`. Voldemort is a *distributed system* as it provides scalability by partitioning the key space across multiple machines and achieves fault tolerance by replicating keys and values across multiple machines.

The Voldemort project has an extensive test suite, which we leverage to generate a log of replication messages in a system with one client and two replicas. We logged messages generated by client calls to the synchronized versions of `put` and `get` and captured just the messages between the client and the two replicas[8]. Since Voldemort does not log vector timestamps, we annotated the traces to include them.

Dynoptic mined 112 valid invariants and generated the model in Figure 3.21. This model contains a client FSM and two replica FSMs. As expected, the replica FSMs are identical. Synchronized

[8] The replicas also communicate with each other to maintain key availability, but we excluded this communication from the log

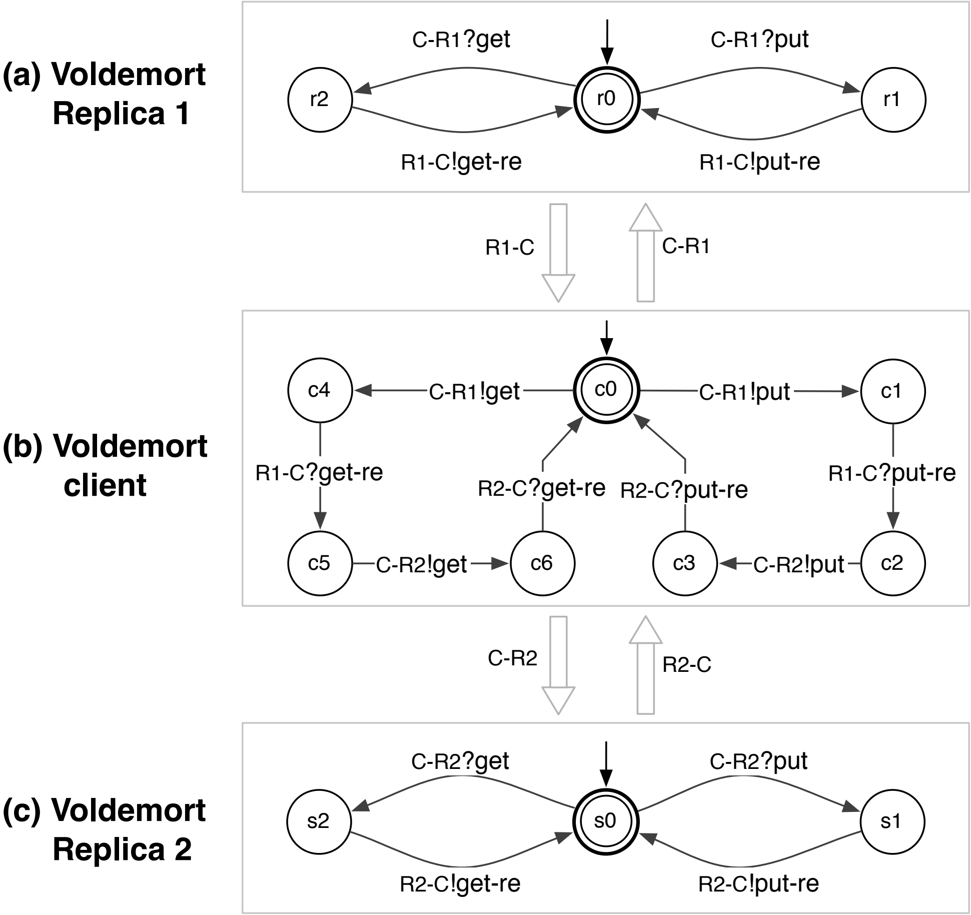


Figure 3.21: The inferred star-topology within a three node Voldemort cluster with **(a,c)** the Voldemort replica models, and **(b)** the Voldemort client model.

Voldemort operations are serialized in a specific order, so the flow of messages for `put` as well as for `get` is identical — the client first executes the operation at replica 1 and then at replica 2.

The model is accurate and provides a high-level overview of how replication messages flow in the system. However, as with the TCP protocol, the model is also abstract and omits numerous details, such as what happens when replicas fail. The advantage of Dynoptic is that a developer can focus on those aspects of system behavior that are important to them (e.g., flow of replication messages) and ignore irrelevant details (e.g., replica failures).

3.13.5 User study

To determine whether CFSM models are useful in finding bugs, we designed and ran a user study that compared the efficacy of CFSM models against time-space diagrams in bug finding tasks. Time-space diagrams were described in Section 3.1.1. Even though developers typically deal with vector-timestamped logs generated by distributed systems, and convert those manually to time-space diagrams, we compared CFSMs against time-space diagrams because developers prefer to visualize the logged executions as time-space diagrams and the conversion process is too laborious for a study. The study consisted of an in-class assignment in a Software Engineering class with 39 Computer Science undergraduate students at a major university. Throughout the study and in the oral feedback session that followed, students were not told the purpose of the study.

The study consisted of two distributed systems: the stop-and-wait protocol and Voldemort. We introduced a bug into each system — the sender in the stop-and-wait protocol failed to retransmit packets on timeout, and the synchronized Voldemort client sent requests to all replicas concurrently, instead of blocking on acknowledgment from each replica. In our experience, both bugs are representative of real bugs faced by distributed system developers. The root cause of the stop-and-wait bug is not taking the right action on an event; the root cause of the Voldemort bug is performing an action at the wrong time. For each buggy system, we generated: (1) a set of representative time-space execution diagrams (8 for the stop-and-wait protocol and 6 for Voldemort), and (2) a CFSM model. Overall, we created four artifacts for the study — *tspace-saw* and *cfsm-saw* (time-space diagrams and CFSM model of the buggy stop-and-wait protocol), and *tspace-vol* and *cfsm-vol* (artifacts for buggy Voldemort).

Students performed the study as an in-class web-based assignment. To account for learning effects, we used a within-participants mixed design across all 39 participants. We considered two factors: the model factor (time-space diagrams vs. CFSM models), and the task factor (stop-and-wait vs. Voldemort). We randomly assigned each student to one of four possible study sequences: ⟨*tspace-saw*, *cfsm-vol*⟩, ⟨*tspace-vol*, *cfsm-saw*⟩, ⟨*cfsm-saw*, *tspace-vol*⟩, ⟨*cfsm-vol*, *tspace-saw*⟩. We considered the two bugs to be independent — finding one bug in one artifact does not help in finding the second bug in a different artifact.

Each task consisted of three steps:

1. To familiarize the students with the idea of time-space diagrams and CFSMs, prior to each task (time-space and CFSM), each student went through a mini-tutorial on the appropriate diagram (time-space and CFSM, respectively). To verify their understanding, at the end of each tutorial, each student had to answer two basic questions correctly. They had to answer both questions correctly to move on and could resubmit their answers until both were correct.[9]

2. Each student was given a *correct* description of the system in English, along with either a set of time-space diagrams or a CFSM model.

3. The student was asked to respond to a single open-ended question. For time-space diagrams (*tspace-saw*, *tspace-vol*) we asked:

   > List all of the time-space diagrams above that you think do not conform to the description of the system above. What made you choose these diagrams?

   For the CFSM models (*cfsm-saw*, *cfsm-vol*) we asked:

   > How does the observed model differ from the intended system description?

As a complete example, the (*tspace-saw*, *cfsm-vol*) sequence mapped to four webpages in the assignment: (1) time-space tutorial and two questions, (2) description of stop-and-wait, *tspace-saw* diagrams, and one open-ended question, (3) CFSM tutorial and two questions, (4) description of Voldemort, *cfsm-vol* model, and one open-ended question.

Results

The 39 students who completed the assignment had, on average, 4.2 years of programming experience, and 76% of the students had never taken a networks course. We manually graded students' answers to the open-ended questions.

Students found bugs about as well with the CFSM model as they did with time-space diagrams. For the stop-and-wait protocol, students who were shown the CFSM were 71.5% successful in getting the right answer, while those who were shown the 8 time-space diagrams were 61.2% successful.

[9] The average time to complete a tutorial was 5 minutes. Students made, on average, 1.8 attempts to answer the tutorial questions correctly. These measures were similar when the time-space and the CFSM tutorials were considered separately.

For Voldemort, the success rate with the CFSM was 72.3% and success rate with the 6 time-space diagrams was 85.7%.

We were surprised that students found CFSMs just as useful (for completing the task) as time-space diagrams. We expected the small collection of time-space diagrams to be simpler to understand and check against a system description than the more complex CFSM model. These results indicate that CFSM models can be used to effectively find bugs. Moreover, in practice, developers have to inspect neither 8 nor 6 executions of the system, but hundreds or thousands. The CFSM models, though, will remain roughly of the same size and complexity. In other words, finding anomalous behavior in the time-space diagrams becomes harder with more executions, whereas the difficulty remains constant with CFSM models. For example, consider if the study presented 8,000 stop-and-wait and 6,000 Voldemort time-space diagrams (the same system executions, but repeated 1,000 times each). The task of manually finding the anomalous time-space diagrams would be infeasible, whereas the CFSM models would remain the same. Since many system executions are, in fact, identical, this scenario is realistic. Therefore, we believe that our results on the utility of CFSMs over time-space diagrams are conservative, and CFSMs would perform even better in practice. Our study finds that they already perform as well as time-space diagrams tightly focused on the buggy behavior.

We collected students' oral feedback on the assignment, which reveals why many of them preferred the CFSM model:

According to many students, time-space diagrams were difficult to follow, especially for long executions:

> "I found the time-space diagrams confusing. It was hard to tell what was happening when. The models were simpler and made it clear what state a system was in and I could keep track of that state." – student 1
>
> "Time-space diagrams were easy to follow for small time segments. For longer time segments, you got lost. The other models were better for longer time." – student 2

Students also preferred a CFSM model because they could compare it to their own mental model, or the model they would draw after reading the system description:

> "I read the description and tried to recreate the model myself first. Then, I compared what I drew to the given diagram and found mistakes. Understanding those mistakes helped me understand the system a lot better." – student 3

Finally, students mentioned that CFSM models were easier to use because they did not explicitly model time, making them simpler for understanding executions in the abstract.

> “For the models, I assumed no time delay in the network messages. It was harder to do in the time-space diagrams because you cannot ignore the delay there. In CFSMs, you can ignore time at first, and then allow for it.” – student 4

3.14 Discussion

The vector timestamps that Dynoptic requires may adversely affect the system’s performance and increase its implementation complexity. Moreover, not all systems use vector timestamps. We are working on a library that will transparently maintain and log vector timestamps for an arbitrary distributed system that uses TCP and UDP sockets. There has also been prior work on optimizing the overhead of maintaining vector timestamps (e.g., [78]). Further, as is typical for production systems, logging vector timestamps can be enabled for a small fraction of the requests. As long as the collected traces are representative of typical system behavior, Dynoptic’s model will too be representative.

CFSM models (Def 3.16) may contain two kinds of error states [27] — unspecified reception and deadlock — neither of which can be eliminated directly by Dynoptic’s model construction. Unspecified reception occurs when a process enters a state with a message m at the head of its channel, but has no reachable future state that receives m. A deadlocked system state occurs when no process can send a message and at least one process cannot reach a terminating state. Currently, Dynoptic does not check if these error states are reachable in the final model. It is possible to extend Dynoptic with such a check, for example, by using the McScM model checker, but the check would be computationally expensive.

Dynoptic uses three of the five mined invariant types and ensures that the final model satisfies the valid invariants subset. While we found these invariant types to be sufficient to infer interesting models in practice, more extensive invariants can lead to more expressive models. For example, a developer might know of a property that the system must satisfy and might want Dynoptic to require that the final model preserves it if this property is true of the input log. In this case, the developer may extend Dynoptic to support a user-defined LTL invariant by (1) implementing a miner to mine invariant instances of this type from the log, and (2) sub-classing and updating a basic template that encodes invariants for model checking with the McScM model checker.

To simplify presentation, we omitted certain details about the mining algorithms. For instance, some invariants are logically equivalent, such as $START_i \rightarrow x_i$ and $x_i \leftarrow END_i$. Others, such as the local versions of $\|$ and $\nparallel$ are trivial. Also, some invariants may be subsumed by others. For example, the distributed versions of $\rightarrow$ and $\leftarrow$ invariants make stronger claims about event ordering and therefore subsume distributed $\nparallel$ invariants. The mining algorithm implementations detect such duplicate, trivial, and subsumed invariants and filter them out.

Dynoptic works for system traces that satisfy certain communication constraints (Def. 3.6). For example, Dynoptic cannot model unclean termination and assumes that each execution terminates with empty channels. However, in practice, system logs may not satisfy these constraints. For example, it might not be possible to acquiesce a production system to process all outstanding messages or the user might be interested in modeling a subset of a trace. Our future work will extend Dynoptic to handle terminal states with non-empty channels. One way to do this is to introduce a terminal state q_i^t for each process i, and append synthetic receive events to each trace to receive all outstanding messages while remaining in the q_i^t state. This drains non-empty channels and converts incompatible traces into ones that Dynoptic can already process.

In generating process states Dynoptic assumes that in each execution trace each process began executing from an identical state. Dynoptic can also run with the assumption that the same process always initiates execution from a unique state in each trace. The first mode is more memory efficient than the first, and may produce more accurate models, however it is not as general as the second mode.

Finally, Dynoptic does not currently scale to very large logs. The scalability bottleneck is the McScM model checker, whose runtime depends on model size and model complexity. To improve scalability, we are integrating the more efficient Spin model checker [72], through Spin is less precise on channels with many messages (beyond a predefined channel capacity bound).

3.15 Summary

Networked systems are hard to implement, debug, and verify. This chapter described Dynoptic, which is a tool that helps with these tasks. Dynoptic uses a partially ordered log of events to infer a concise and precise communicating finite state machine model of the system. Dynoptic's precision comes from its use of mined temporal properties that relate events in the log.

We have evaluated Dynoptic in three ways. First, we formally defined the Dynoptic models and the model derivation process and proved that (1) the Dynoptic process eliminates property counter-examples produced by model-checking, and (2) the final model satisfies the subset of properties that are valid. Second, we implemented Dynoptic and ran it on logs produced by three distributed systems/protocols — the stop-and-wait protocol, TCP, and Voldemort, a distributed storage system. Finally, we carried out a user study to evaluate the utility of CFSM models in finding bugs.

By automatically mining a system model from logs Dynoptic has the potential to ease system understanding, debugging, and maintenance tasks.

Chapter 4

InvariMint: model inference through declarative specification

Model inference is a promising approach to help users make sense of large and complex executions. The previous two chapters focused on two particular model-inference approaches that derive models from logs to support software comprehension. However, numerous related model-inference algorithms and corresponding tools exist to help debug, verify, and validate systems [6, 66, 69, 87, 88, 90, 91, 95, 108, 134]. Unfortunately, it is challenging to apply and extend this rich body of work because model-inference algorithms are primarily expressed procedurally, as algorithms that iteratively modify a representation of the log (e.g., a graph) to infer and output a model that can be shown to a user. The procedural specification of these algorithms makes them difficult to understand, extend, and compare.

This chapter of the thesis presents InvariMint, an approach to specify model inference algorithms declaratively. InvariMint (1) leads to new fundamental insights and better understanding of existing algorithms, (2) simplifies creation of new algorithms, including hybrids that extend existing algorithms, and (3) makes it easy to compare and contrast previously published algorithms. Finally, algorithms specified with InvariMint can outperform their procedural versions.

4.1 Problems with existing model inference algorithms

Existing model-inference algorithms are difficult to understand, extend, and compare.

1. Understand. For most algorithms, it is difficult to understand which temporal and structural properties of the log are preserved in the inferred model. For example, if an inferred model of an email client states that each `login` event is immediately followed by a `check mail` event, a developer may wish to know whether that property is true for all traces in the log, or is an artifact of the model-inference algorithm.

2. Extend. It is difficult to modify existing model-inference algorithms or to compose them to create hybrids. For example, suppose that a developer uses two inference algorithms: one to model exceptional executions and another that identifies executions with sequences of library calls not observed during testing. The developer may want to compose these two algorithms to generate a single model, but combining the procedurally-specified algorithms may require a complete algorithm redesign. Further, it is difficult or impossible to exclude a specific instance of a log property from a specific invocation of the algorithm. If a log has every `login` event followed by a `check mail` event and if the developer decides that this property is an artifact of an incomplete log, the developer may want the inferred model to allow traces that violate this property. However, this may be difficult because a procedural algorithm definition does not explicitly specify the properties that the inferred model will satisfy.

3. Compare. Previously published algorithms lack a common form to aid comparison and juxtaposition. Instead, researchers must reason about pseudocode and work out complex proofs. A declarative approach, in which a model-inference algorithm is specified in terms of log properties that the inferred model will satisfy, allows researchers to, for example, identify when two algorithms with vastly different procedural definitions produce models with identical, or overlapping, sets of properties.

This chapter proposes InvariMint, a technique to specify model-inference algorithms *declaratively*. InvariMint has two key features: (1) it explicitly specifies the types of properties that will be enforced in the final model, and (2) it decouples the mechanism of property *mining* from property *specification*. We illustrate the advantages of InvariMint by specifying two procedural algorithms declaratively. We find that InvariMint alleviates the above problems:

1. Understand. InvariMint expresses an algorithm in terms of the properties that the inferred model must satisfy. This formulation is more clear, concise, and comprehensible. Further, this formulation makes evident certain complexities that may otherwise be hidden, such as non-determinism.

2. Extend. With InvariMint, it is easy to add, remove, and modify both (1) the instances of properties in a specific inference execution (e.g., each `login` event must be followed by a `check mail` event), and (2) the types of properties the algorithm preserves (e.g., an event may only follow another event if it did so in the log). New algorithms can be created, and multiple algorithms can be trivially composed to create hybrid approaches. For example, the Synoptic [22] algorithm uses the kTails algorithm as a final (coarsening) step to derive a more compact final model. Synoptic's

InvariMint specification expresses this by simply merging the kTails property types into the Synoptic specification.

3. Compare. InvariMint makes it easier for those using and developing model-inference algorithms to compare and improve on those algorithms. For example, algorithms with incomparable procedural definitions may enforce overlapping sets of properties on their inferred models. InvariMint makes this overlap evident.

4.2 Overview of approach

A model-inference algorithm outputs a model that accepts a formal language. The model's language is smaller than Σ^*: it is limited by certain temporal or structural properties that the algorithm mined from the log. Some of these properties may be explicit in the algorithm definition, whereas others may be implicit and deeply hidden in the procedural definitions.

InvariMint (Figure 4.1) represents a model-inference algorithm with the types of properties that are expected to be true of the inferred model. InvariMint mines instances of these properties from the log, represents each property as a DFA, and composes the DFAs using standard DFA operations (such as DFA union and intersection). Well-understood work on formal languages allows InvariMint to perform these operations efficiently and to produce minimal models [73].

To evaluate InvariMint, we applied it to two previously-published algorithms. First, we used InvariMint to declaratively and exactly specify the well-known kTails [23] algorithm. From our past experiences with kTails, we know that this algorithm behaves non-trivially on large log inputs. For instance, it is neither apparent which states will be merged, nor what synthetic traces the final kTails-inferred model will accept. The InvariMint formulation decomposes a kTails execution into a set of properties that are easy to inspect to better understand the characteristics of the final kTails-inferred model. The InvariMint kTails specification also provides the user with more fine-grained control over the execution of the algorithm — the user may remove a particular merge (by modifying a property instance) without having to modify the algorithm implementation.

Second, we used InvariMint to approximate Synoptic [22]. Synoptic is a more recent algorithm constructed with explicit log properties in mind. Although Synoptic attempts to make certain properties explicit, we found that it in fact preserves a set of implicit, or hidden, properties in its procedural declaration. Specifically, Synoptic allows a log event type to be immediately followed by another type only if such following occurred in the observed log. For example, Synoptic forbids a `login` event from being immediately followed by a `compose mail` event if, in the log, `login` was *never* immediately followed by `compose mail`. Synoptic's procedural declaration does not allow this property to be removed, altered, or relaxed, and hides this property from the user. In contrast, an InvariMint formulation of Synoptic makes this property explicit and allows a user to remove all properties of this type or to select individual instances of this property for specific log event types to enforce. More importantly, InvariMint makes the algorithm's user and developer explicitly aware of the properties it enforces.

Another feature of Synoptic is that it is a non-deterministic algorithm. Depending on the order with which Synoptic satisfies the mined log properties, the algorithm might produce a different final model. Although our InvariMint Synoptic formulation is an approximation of Synoptic, its advantage is that it is deterministic and highly predictable. In particular, it is easier to check whether two different logs produce identical models.

As a final benefit, the InvariMint versions of kTails and Synoptic with efficient property mining scale linearly with log size and greatly outperform their procedural counterparts.

The rest of this chapter is structured as follows: Section 4.3 uses an example model-inference algorithm to explain the InvariMint approach. Sections 4.4 and 4.5 present InvariMint specifications of kTails and Synoptic, respectively. Section 4.6 discusses implications of our work. Section 4.7 summarizes this chapter.

4.3 The InvariMint approach

This section describes a model-inference algorithm named SimpleAlg and then overviews the InvariMint approach by outlining the InvariMint steps in specifying an example algorithm, called SimpleAlg. Sections 4.4 and 4.5 extend SimpleAlg's specification to derive the kTails and Synoptic algorithms.

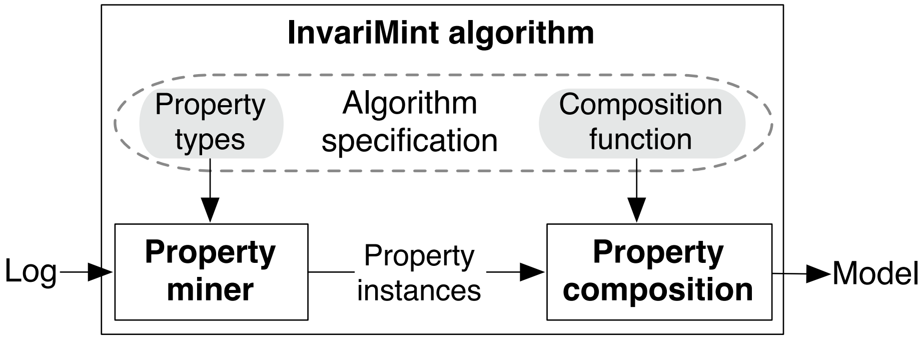


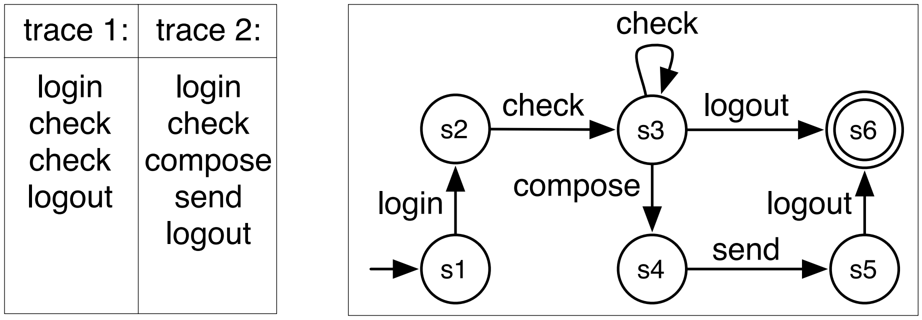
Figure 4.1: An overview of the InvariMint approach. An InvariMint *algorithm* is parameterized by an algorithm specification, which consists of a set of property types and a composition function. The resulting InvariMint algorithm is a model-inference algorithm — it takes a log of traces as input and outputs an inferred model which describes the process that generated the input log. Internally, the algorithm uses property types to mine property instances, and then applies the composition function to the property instances to derive the model. This is an elaboration of the more abstract description in Figure 1.5(b).

4.3.1 SimpleAlg

A model-inference algorithm's input is a **log** — a set of traces of a system's execution. Each **trace** is an ordered sequence of **events** (elements of a finite alphabet) that occurred during execution. SimpleAlg's output is a model — a finite state machine whose language is a set of traces. (Figure 4.2 shows example input and output.) The language corresponding to the model accepts all the traces in the log, as well as other traces. A model-inference algorithm's goal is to infer a model that accurately describes and generalizes the log: the extra accepted traces should be ones that are likely to be generated by the system that produced the log.

SimpleAlg is a model-inference algorithm. It generalizes in the following way: if SimpleAlg ever observes an event e_1 to be immediately followed by an event e_2 in the log, then whenever the system being modeled produces or consumes an e_1 event, SimpleAlg assumes that it is legal for the system to then produce or consume an e_2 event.

Pseudocode for SimpleAlg appears in Figure 4.3. In the generated model, each state represents



trace 1:	trace 2:
login check check logout	login check compose send logout

(a) Input log **(b) Output model**

Figure 4.2: **(a)** An example log of an email client with two traces. **(b)** The model inferred with SimpleAlg (Figure 4.3) for the input log in (a).

an event that has just occurred. The model contains one state for each unique event type that occurs in the log, plus one "initial" state. The model contains a transition from the state for event type e_1 to the state for event type e_2, with the label e_2, iff there exists a trace in the log in which an e_2 event immediately follows an e_1 event.

Figure 4.2(a) lists an email client log with two traces. The event alphabet is {`login`, `check mail` (shortened to `check`), `compose`, `send`, `logout`}. Figure 4.2(b) shows the model SimpleAlg infers for this input log. The model has six states, one for each event type (e.g., s4 corresponds to `compose`) plus the initial state (s1).

SimpleAlg's models are compact — the number of states is one more than the number of unique event types in the log, which is independent of the total number of events in the log. The running time is asymptotically linear in the size of the log. The inferred model's language always contains every trace in the input log, plus other traces SimpleAlg deemed possible.

4.3.2 InvariMint overview

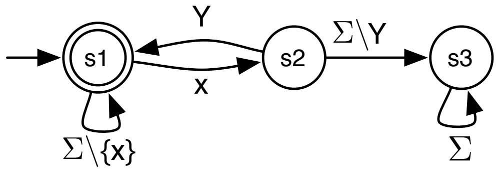
InvariMint is an approach — or a common language — for describing model-inference algorithms, such as SimpleAlg. Figure 4.1 overviews the InvariMint approach. Like other model-inference algorithms, an InvariMint algorithm takes as input a **log** of traces to be modeled, and outputs a

```
Input: Log $L$
let $M$ = new FSM model

// Create states
$M$.addState($init$)
foreach (Trace $t$ in $L$):
  foreach (Event $e$ in $t$):
    let $y$ = Event type of $e$
    if ($\neg M$.hasState($s_y$)) : $M$.addState($s_y$)

// Add transitions among the states.
foreach (Trace $t$ in $L$):
  // Add transition from init state to first event.
  let $f$ = Event type of first event in $t$
  $M$.addTransition(src=$init$, dst=$s_f$, label=$f$)

  // For each pair of adjacent events, add a transition
  // between states of corresponding event types.
  foreach (Event $e$ in $t$):
    if ($e$.hasNext()):
      let $y$ = Event type of $e$
      let $z$ = Event type of $e$.next()
      if ($\neg M$.hasTransition($s_y$, $s_z$)):
        $M$.addTransition(src=$s_y$, dst=$s_z$, label=$z$)

Output: $M$
```

Figure 4.3: Procedural pseudocode of the SimpleAlg algorithm.

model. The common language InvariMint uses to specify an algorithm is: a set of **property types** that describe properties to be mined from the log to derive *property instances*; and a **composition function** that combines the mined property instances into a final model.

Different model-inference algorithms take different approaches to generalizing the traces in the log to infer traces likely traces that are not in the log. What constitutes reasonable generalization is often subjective and depends on features of the system, its environment, and the specific development task that the model will be used for. While typical model-inference algorithms hard-code these



$$Eval(\text{Log } L, \langle x{=}a, Y{=}B\rangle) = \begin{cases} \text{true} : \forall t \in L, \exists b \in B,\ \Diamond(a \rightarrow \bigcirc b) \text{ in } t \wedge \\ \qquad\quad \forall b \in B, \exists t \in L,\ \Diamond(a \rightarrow \bigcirc b) \text{ in } t \\ \text{false} : \text{otherwise} \end{cases}$$

(a) Property type (PFSM and Eval)

$$Compose(Prop_1, \ldots, Prop_n) = Minimize(\cap Prop_i)$$

(b) Composition function

Figure 4.4: An InvariMint specification of SimpleAlg. This is equivalent to the pseudocode in Figure 4.3. **(a)** The property type "event *x* can be immediately followed by an event from set *Y*", represented as a parameterized FSM (PFSM) and a corresponding evaluation function (*Eval*). Given an input log, *Eval* determines the validity of bindings of parameters in the PFSM to event types. **(b)** The composition function, which InvariMint uses to compose a model from mined property instances.

features as assumptions in their procedural definitions, InvariMint uses property types and the composition function to generalize the model-inference process. Property types define desirable properties of the final model. For example, the SimpleAlg-inferred model preserves log properties, such as "event *x* can be immediately followed by an event from set *Y*". A property type consists of a parameterized FSM (PFSM) — an FSM with variable-labeled transitions (e.g., top portion of Figure 4.4(a)) — and an evaluation function to decide which bindings of PFSM variables to event types are valid in the log (e.g., bottom portion of Figure 4.4(a)). Together, the PFSM and evaluation function encode relationships between event types.

Using these evaluation functions, InvariMint mines the log for *property instances*, which are instantiations of the corresponding PFSM. InvariMint then combines the derived property instances into a model using the composition function (e.g., Figure 4.4(b)). The *Minimize* procedure referenced in this composition is the FSM minimization algorithm [73], which guarantees that the final model

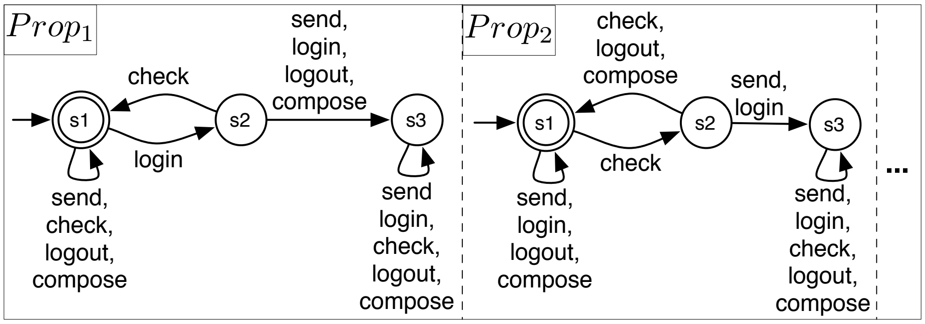


Figure 4.5: Property instances mined by InvariMint from the log in Figure 4.2(a), based on property types in Figure 4.4(a). *Prop*$_1$ represents "event `login` can be immediately followed by an event from set {`check`}". *Prop*$_2$ represents "event `check` can be immediately followed by an event from set {`check`, `logout`, `compose`}".

will be minimal.

We now illustrate InvariMint on the SimpleAlg example.

4.3.3 Specifying SimpleAlg with InvariMint

InvariMint's formulation of SimpleAlg has only a single property type: "event x can be immediately followed by an event from set Y". Figure 4.4 shows the InvariMint specification of SimpleAlg. Figure 4.4(a) shows the property type (a PFSM and an evaluation function). The PFSM is an FSM with variable labels that accepts all traces that relate event x and a set of events Y. The evaluation function defines which bindings of variables to log events result in valid property type instances.

We use LTL to compactly specify evaluation functions. LTL statements use the operators *always* ($\Box$), *eventually* ($\Diamond$), *until* (U), and *next* ($\bigcirc$). For example, the evaluation function in Figure 4.4(a) returns true for event a and events set B whenever a can be immediately followed by only events from B across all traces in the log — that is, there is a trace for every $b \in B$ and there is a $b \in B$ for every trace such that eventually ($\Diamond$), if we observe an a event, then we will observe a b as the next ($\bigcirc$) event.

By indicating how to evaluate a binding of x and Y to event types, the evaluation function specifies

```
Input: Log $L$, Property types $\langle PFSM_1, Eval_1\rangle, \ldots, \langle PFSM_n, Eval_n\rangle$
let $Props = \{\}$
foreach (Property type $\langle PFSM_i, Eval_i\rangle$)
  foreach (Binding of variables in $PFSM_i$, $B$)
    if ($Eval_i$($L$, $B$)):
      $Props = Props \cup \{PFSM_i(B)\}$
Output: $Props$
```

Figure 4.6: The generic property miner algorithm.

how x and Y must relate: an event of type x must be immediately followed by one event from the set Y.

While all bindings can create property instances, the evaluation function determines which instances are valid for a given log. Figure 4.5 lists two of the property instances that are valid for the log in Figure 4.2(a): $\langle x,Y\rangle = \{\langle \texttt{login}, \{\texttt{check}\}\}$, and $\langle x,Y\rangle = \langle \texttt{check}, \{\texttt{check}, \texttt{logout}, \texttt{compose}\}\rangle$. In addition to these two property instances, there are three others (one for each of `compose`, `send`, and `logout`). Note that $\langle \texttt{logout}, \emptyset\rangle$ is necessary to prevent allowing all events to follow `logout` in the inferred model.

Finally, InvariMint composes property instances using the composition function in Figure 4.4(b) to produce the final model. For SimpleAlg, the composition function returns the minimized version of the intersection of the property instances. Therefore, the resulting model is compact and includes only those traces that satisfy all of the mined property instances. This final model is identical to the one produced by SimpleAlg (Figure 4.2(b)). This chapter mostly focuses on composition functions that involve only intersections and minimizations, but this limitation is not inherent to InvariMint. More complex functions may include unions, set differences, and other set operations. For example, an algorithm that uses positive and negative trace example may subtract the model of negative traces from one of positive traces.

```
Input: Property instances $Prop_1, \ldots, Prop_n$, Composition function $C$
let $Model = C(Prop_1, \ldots, Prop_n)$
Output: $Model$
```

Figure 4.7: The generic property composition algorithm.

4.3.4 InvariMint benefits

The InvariMint formulation of SimpleAlg provides three benefits over the SimpleAlg pseudocode: **(1)** The InvariMint formulation helps us understand the key properties of the final model derived with SimpleAlg by decoupling these properties from the mining and composition procedures, while the pseudocode mixes all three. **(2)** We can more easily add new constraints to the model by defining new property types, and eliminate behavior from the model by omitting property instances. For example, if we do not want `login` to only be immediately followed by `check`, we can simply omit $Prop_1$ in Figure 4.5. **(3)** We can, and will, extend the InvariMint formulation of SimpleAlg to construct InvariMint specifications for kTails and Synoptic. The pseudocode for these algorithms looks completely different from SimpleAlg's pseudocode, yet the InvariMint specification reveals that both kTails and Synoptic are based on the same property type (Figure 4.4(a)) used by SimpleAlg. The fact that all three algorithms share this property type is one of the insights gained from specifying these algorithms with InvariMint.

InvariMint's goal is not to produce models, *per se*, but rather to provide a common language for expressing, or specifying, model-inference algorithms. Specifying different algorithms with the same language allows us to understand, combine, and compare the algorithms. InvariMint's common language is property types and composition functions. Once specified, the resulting property mining and property composition procedures (Figure 4.1) are straightforward. Figures 4.6 and 4.7 list the unoptimized pseudocode for these two procedures. Note that in practice, both of these algorithms can be further optimized and tailored to specific choices of property types and composition functions.

Next, we describe and evaluate the InvariMint specification of two previously-published model-inference algorithms — kTails and Synoptic.

```
Input: Log $L$, int $k$
let $M$ = initial FSM model of traces in L

let $merged$ = true
while ($merged$):
  $merged$ = false
  foreach (States $s_1, s_2$ in $M$):
    if ($s_1, s_2$ are $k$-equivalent):
      $M$.merge($s_1, s_2$)
      $merged$ = true

Output: $M$
```

Figure 4.8: Procedural pseudocode of the kTails algorithm. Section 4.4.1 defines *k*-equivalence.

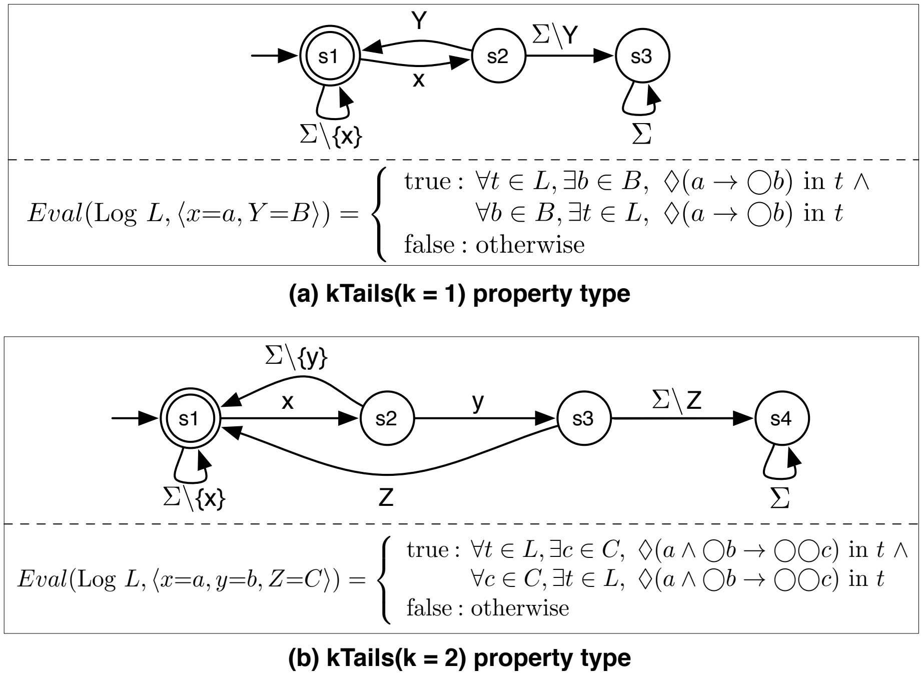
4.4 Expressing kTails with InvariMint

kTails [23] is an extremely popular algorithm that has served as the basis for many modern model-inference algorithms. Unfortunately, there are many procedural descriptions of kTails, and it is difficult to tell if they produce identical or different models.

This section defines the kTails algorithm (Section 4.4.1), demonstrates its InvariMint declarative specification (Section 4.4.2), discusses the insights about kTails that InvariMint reveals (Section 4.4.2), and reports on our empirical comparison of the procedural and declarative implementations of kTails (Section 4.4.3).

4.4.1 kTails

kTails is a state-merging algorithm. kTails takes a log and a parameter k. It represents the log as a DFA composed of linear sub-DFAs, one per trace, that are joined in a parallel fashion, with a single initial state transitioning to the start of each trace, and all traces finishing by transitioning to a single terminal state. kTails then iteratively merges states in the DFA that are "k-equivalent". Two states are k-equivalent if their kTails are identical. A state's kTail is the set of strings, of length k or shorter, that map to valid paths starting from that state. The algorithm terminates and outputs the model when



$$Eval(\text{Log } L, \langle x{=}a, Y{=}B\rangle) = \begin{cases} \text{true}: & \forall t \in L, \exists b \in B,\ \Diamond(a \rightarrow \bigcirc b) \text{ in } t\ \land \\ & \forall b \in B, \exists t \in L,\ \Diamond(a \rightarrow \bigcirc b) \text{ in } t \\ \text{false}: & \text{otherwise} \end{cases}$$

(a) kTails(k = 1) property type

$$Eval(\text{Log } L, \langle x{=}a, y{=}b, Z{=}C\rangle) = \begin{cases} \text{true}: & \forall t \in L, \exists c \in C,\ \Diamond(a \land \bigcirc b \rightarrow \bigcirc\bigcirc c) \text{ in } t\ \land \\ & \forall c \in C, \exists t \in L,\ \Diamond(a \land \bigcirc b \rightarrow \bigcirc\bigcirc c) \text{ in } t \\ \text{false}: & \text{otherwise} \end{cases}$$

(b) kTails(k = 2) property type

Figure 4.9: **(a)** kTails($k = 1$) property type. **(a+b)** kTails($k = 2$) property types. Each of these is equivalent to the pseudocode in Figure 4.8 for the specific value of k.

no two remaining states are k-equivalent. Figure 4.8 lists the kTails pseudocode.

The intuition behind kTails is that if two execution points have identical, k-long sequences of observed events following them, then those points likely represent the same program state. Therefore, to infer a concise model, kTails merges execution points that it considers to represent the same program state. The process stops once all points deemed equivalent are merged. The parameter k determines the size and generality of the inferred model — a smaller k leads to more merges and produces more compact (and more general) models, while a greater k restricts state equivalence.

In InvariMint kTails we introduce a pre- and a post-processing step. We modify each input trace to include an α and ω symbols at the start and end of each of the traces, respectively. After the property instances are composed into a final model we update states in the model with incoming α transition to be initial states, update states with outgoing ω transition to be accept states, and also remove all α and ω transitions from the model.

InvariMint uses property types to capture tail-equivalence and to specify kTails. Figure 4.9(a) lists the $k = 1$ property type for kTails. For $k = 2$, InvariMint requires two property types — the property type for $k = 1$ in Figure 4.9(a) and a new property type shown in Figure 4.9(b). Note that the property type for $k = 1$ kTails in Figure 4.9(a) is identical to the "can be immediately followed by" property type in Figure 4.4(a). This equality is not a coincidence — the k parameter generalizes the "can be immediately followed by" property type to k steps into the future.

The greater k is, the finer the granularity of the properties kTails enforces. For example, the property type in Figure 4.9(b) says that an event x, followed by an event y, must be followed by one — any one — of the events in the set Z. In other words, it corresponds to merging all x, y tails together. Section 4.6 discusses in more detail the granularity of properties and how the wrong granularity may cause the algorithm to overfit to the input log.

An important feature of the InvariMint kTails specification is that it is deterministic. This feature helped us better understand the kTails algorithm and helped to reveal a bug in our procedural implementation, which happened to be non-deterministic.

4.4.2 *Comparing procedural and InvariMint versions of kTails*

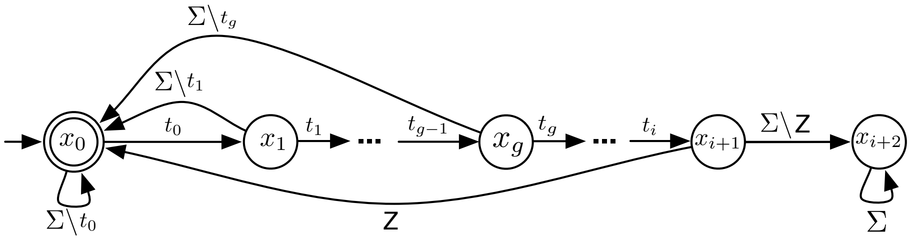
The model produced by the kTails algorithm behaves identically to the model produced by the InvariMint formulation of kTails. Next, we formally define the kTails algorithm based on the formulation in [35], and prove that the two formulations of the algorithm are identical.

Let Σ_k denote the set of all strings of length k or less. Let a trace be a string over alphabet $\Sigma \cup \{\alpha, \omega\}$, and let a log L be a set of traces, each of which starts with an α symbol and terminates with the ω symbol. Let PF_L be the set of all prefixes of strings in L. We use $p \cdot t$ to denote concatenation of string t to p, and refer to t as the *tail*.

For example, consider the log $L = \{\alpha abc\omega, \alpha ab\omega, \alpha cd\omega\}$. Then, the corresponding $PF_L = \{\alpha\varepsilon\omega, \alpha a\omega, \alpha ab\omega, \alpha abc\omega, \alpha c\omega, \alpha cd\omega\}$. And, the string $\alpha abc\omega = \alpha a \cdot bc\omega$, in which $bc\omega$ is a tail.

Definition 4.1 (kTails FSM F_{kTails})**.** The kTails algorithm takes a log L and an integer k as inputs and generates a *kTails FSM F_{kTails}*. The **states** of F_{kTails} correspond to equivalence classes of prefixes from PF_L. An equivalence class E is a set of prefixes such that:

$$\forall (p, p') \in E, \forall t \in \Sigma_k, (p \cdot t) \in PF_L \Leftrightarrow (p' \cdot t) \in PF_L$$



$$Eval(\text{Log } L, \langle a_0, \ldots, a_i, C\rangle) = \begin{cases} \text{true}: & \forall t \in L, \exists c \in C, \\ & \quad \Box\big[a_0 \wedge \bigcirc a_1 \cdots \wedge (\bigcirc^1 \cdots \bigcirc^i a_i) \rightarrow \\ & \qquad \bigcirc^1 \cdots \bigcirc^i \bigcirc c\big] \text{ in } t \\ & \quad \wedge \\ & \quad \forall c \in C, \exists t \in L, \\ & \quad \Box\big[a_0 \wedge \bigcirc a_1 \cdots \wedge (\bigcirc^1 \cdots \bigcirc^i a_i) \rightarrow \\ & \qquad \bigcirc^1 \cdots \bigcirc^i \bigcirc c\big] \text{ in } t \\ \text{false}: & \text{otherwise} \end{cases}$$

Figure 4.10: The kTails($k = i$) property type.

That is, all prefixes in a class E have the same set of tails of length k or less, and every prefix in PF_L is assigned to some equivalence class.

The **transition function** Δ for equivalence classes, or states, in F_{kTails} is defined as follows. Given a state E_i and a symbol $a \in \Sigma$,

$$\Delta(E_i, a) = \bigcup E[p \cdot a], \forall p \in E_i$$

where $E[p \cdot a]$ is the equivalence class of $p \cdot a$.

The **initial state** of F_{kTails} is $E[\varepsilon]$, and an equivalence class E_i is an **accept state** of F_{kTails} if $\exists\ s \in L$, such that $s \in E_i$.

Definition 4.2 (InvariMint kTails FSM $F_{InvMint}$)**.** For a log L and an integer k, let $F_{InvMint}$ be the FSM derived using the InvariMint algorithm specified by the kTails(k) property types and the input log L. We can express $F_{InvMint}$ as a composition of property instances[1]:

$$F_{InvMint} = \bigcap(P_1^1, \ldots, P_{n_1}^1, \ldots, P_1^k, \ldots, P_{n_k}^k)$$

[1] We omit FSM minimization as it does not change the FSM's language.

where $P^i_1, \ldots, P^j_{n_i}$ are the property instances for the PFSM corresponding to kTails($k = i$) property type. Figure 4.10 shows this generalized property type.

Definition 4.3 (Terminal rejection)**.** Let F be an FSM. F terminally rejects s if $\nexists\, t$ such that $s \cdot t$ is accepted by F.

Observation 4.1. $F_{InvMint}$ does *not* terminally reject strings in PF_L.

Proof: Consider a string $s \in PF_L$. Choose a tail t such that $s \cdot t$ is a trace in L. Such a tail must exist since s is a prefix for some trace in L. By construction, $F_{InvMint}$ accepts all strings in L. Therefore, $F_{InvMint}$ accepts $(s \cdot t) \in L$, and does not terminally reject s. □

Theorem 4.1 (InvariMint specification of kTails is exact)**.** *For an input log L and an integer k, let F_{kTails} be the corresponding kTails FSM and let $F_{InvMint}$ be the InvariMint kTails FSM. Then, the languages of the two FSMs are equivalent, or:*

$$Lang(F_{kTails}) = Lang(F_{InvMint})$$

Proof: We prove the two directions of equality in Theorem 4.1 separately.

1) $Lang(F_{kTails}) \subseteq Lang(F_{InvMint})$

Proof by contradiction:

Assume that $\exists\ s \in Lang(F_{kTails})$ and $s \notin Lang(F_{InvMint})$.

Because $s \notin Lang(F_{InvMint})$ there is a non-empty set of rejecting (non-accepting) property instances $\mathfrak{R}$. That is, $\forall P \in \mathfrak{R}, s \notin Lang(P)$. Let r be the *shortest* prefix of s to be rejected by some property instance $P^i_j \in \mathfrak{R}$, with $i \leq k$.

Now consider the prefix string r, which is rejected by P^i_j. We can express r as $r = u \cdot a$ for some $a \in \Sigma$. The property instance P^i_j (in Figure 4.10) can reject r in two ways:

(1a) P^i_j rejects r by terminating in state x_{i+2}, because $a \notin Z$.

In this case, r must be at least $i + 1$ symbols long, and can be expressed as $r = v \cdot t_0 \cdots t_i \cdot a$. Consider the equivalence class $E_v = E[v]$. This class must be non-empty because there exists a transition on t_0 from E_v to $E[v \cdot t_0]$. Since E_v is non-empty, consider a prefix $p \in E_v$. Because $i < k$, and since $t_0 \cdots t_i \cdot a$ is a tail of v, by definition of equivalence classes, $p \cdot t_0 \cdots t_i \cdot a \in PF_L$. However, because $t_0, \ldots, t_i$ matches the tail corresponding to P^i_j, $a \in Z$. Contradiction ($a \notin Z$).

(1b) P^i_j rejects r by terminating in x_h, $0 < h \leq i+1$, and $s = r$.

Note that the LTL formula of the general kTails property type evaluation function (in Figure 4.10) mandates that each a_m bound to t_m must be followed by some a_{m+1} in some trace. Since ω is the last symbol in any trace, it cannot be bound to any a_m in the evaluation function.

The above implies that $\forall g$, $0 < g \leq i+1$ there is no transition on ω *into* x_g. Since every trace terminates with ω, we can express r as $r = v \cdot \omega$. But, this contradicts P^i_j rejecting r in state x_h, since P^i_j can only terminate on $v \cdot \omega$ in states x_{i+2} or x_0.

We have shown that P^i_j cannot reject r since it cannot terminate on r in any non-accepting states. Therefore, by contradiction, $s \in Lang(F_{InvMint})$ and $Lang(F_{kTails}) \subseteq Lang(F_{InvMint})$.

2) $Lang(F_{InvMint}) \subseteq Lang(F_{kTails})$

Note that $s \in Lang(F_{InvMint})$ implies that s is accepted by all property instances that make up $F_{InvMint}$.

Let $s = a_0 \cdots a_n$. By induction on k and n, we will show that if $s \in Lang(F_{InvMint})$ then there exists a valid and accepting path of equivalence classes, $[E_0, \ldots, E_n]$, that corresponds to s, and thus $s \in F_{kTails}$.

Base case (k $=$ 1): We prove this base case by induction on n, assuming $k = 1$.

Base case (n $=$ 2): Show that $s = a_0 \cdot a_1 \cdot a_2 = \alpha \cdot a_1 \cdot \omega$ maps to an accepting path E_0, E_1, E_2, E_3 in F_{kTails}.

Let $E_0 = E[\varepsilon]$.

Since $\alpha \in \Sigma$, there must be a property instance P^1_j, of the property type in Figure 4.9(a), that binds t_0 to α. This P^1_j accepts $\alpha \cdot a_1$, and therefore P^1_j must bind Y to a set B, such that $a_1 \in B$. Next, the LTL formula corresponding to P^1_j tells us that $\exists\, t \in L$ such that $\Diamond(\alpha \rightarrow \bigcirc a_1)$. Since α is the first symbol for any trace, this means that $\alpha \cdot a_1$ is a prefix for this t. Since α is a valid prefix, there must exist a non-empty equivalence class $E_1 = E[\alpha]$. E_0 has a transition to E_1 on α, because α is a valid prefix for traces in L.

Now, consider the string $a_1 \cdot \omega$. Since a_1 appears in some trace there must be a corresponding property instance P^1_m. By the same reasoning as above, P^1_m binds t_0 to a_1 and binds Y to a set B' such that $\omega \in B'$. The LTL formula corresponding to P^1_m tells us that $\exists\, t' \in L$ such that $\Diamond(a_1 \rightarrow \bigcirc \omega)$. We can represent this t' as $t' = p \cdot a_1 \cdot \omega$.

Note that p (a prefix of t') and α have identical 1-tails, namely $\{a_1\}$. By construction of F_{kTails} this means that p and α belong to the same equivalence class E_1. Since, $p \cdot a_1$ is a valid prefix, there must exist an equivalence class $E_2 = E[p \cdot a_1]$, and there must be a transition from E_1 to E_2 on a_1.

Finally, we will use t' to construct E_3. Since $p \cdot a_1$ maps to E_2, there must be a transition on ω to E_3. This E_3 must be terminal because it contains the trace t'.

As a result, we have constructed an accepting path E_0, E_1, E_2, E_3 for the string s.

Inductive hypothesis ($\mathbf{n = i}$): Assume that $a_0 \cdots a_i$ maps to a valid path $E_0, \ldots, E_i$. Show that $a_0 \cdots a_{i+1}$ maps to a valid path $E_0, \ldots, E_{i+1}$.

Consider the string $a_i \cdot a_{i+1}$. Since a_i appears in some trace there must be a corresponding property instance P^1_j. By the reasoning in the base case, P^1_j binds t_0 to a_i and binds Y to a set B such that $a_{i+1} \in B$. The LTL formula corresponding to P^1_j tells us that $\exists\, t \in L$ such that $\Diamond(a_i \rightarrow \bigcirc a_{i+1})$. We can represent this t as $t = p \cdot a_i \cdot a_{i+1}$.

Based on our induction assumption, there exists an equivalence class E_{i-1} that corresponds to a_{i-1}. Since the prefix p is followed by a_i in t, p must also map to E_{i-1}. Therefore, we can extend $E_0, \ldots, E_{i-1}$ with E'_i and E_{i+1}, where $E'_i = E[p \cdot a_i]$ and $E_{i+1} = [p \cdot a_i \cdot a_{i+1}]$.

Inductive hypothesis ($\mathbf{k = j}$): We prove this by induction on n. We assume that the proof statement is true for $k = j$ and perform induction on n to show that the statement is true for $k = j + 1$.

Base case ($\mathbf{n = 2}$): Show that $s = a_0 \cdot a_1 \cdot a_2 = \alpha \cdot a_1 \cdot \omega$ maps to an accepting path E_0, E_1, E_2, E_3 in F_{kTails}.

Since $k = j > 1$, $F_{InvMint}$ includes property instances corresponding to the kTails($k = 1$) property type (Definition 4.2). This means that we can re-use the base case for $\mathbf{k = 1}$ above and construct the path E_0, E_1, E_2, E_3 corresponding to s in F_{kTails} in the same manner. This construction also holds for $k = j + 1$.

Inductive hypothesis ($\mathbf{n = i}$): Assume that $a_0 \cdots a_i$ maps to a valid path $E_0, \ldots, E_i$. Show that $a_0 \cdots a_{i+1}$ maps to a valid path $E_0, \ldots, E_{i+1}$.

Consider the string $t = a_{i-j} \cdots a_i$. Each symbol in t corresponds to a property P, for a particular k value, that makes up $F_{InvMint}$ and which accepts all of the symbols at the tail of t in front of the symbol.

For example, a_{i-j} corresponds to some property P^j, which accepts the tail $a_{i-j+1} \cdots a_i$ of t. Using

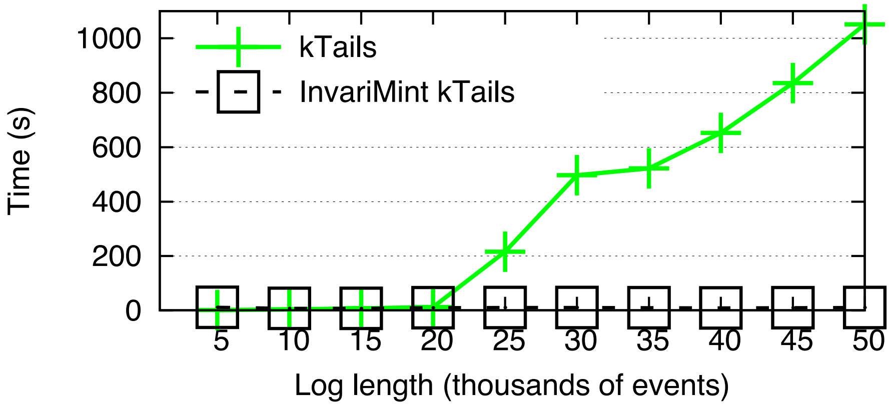


Figure 4.11: The running time of procedural kTails and the declarative InvariMint version of kTails for different log input sizes. The number of property instances true of the log was held constant at 182.

the base case construction of overlapping prefixes, we construct a path $E_0, \ldots, E_{i+1}$ that corresponds to $a_0 \cdots a_{i+1}$. □

4.4.3 Empirical evaluation

We implemented InvariMint and the kTails algorithm in Java and evaluated their relative performance in two experiments. Both experiments were executed on an OS X 10.8 machine with a 2.8GHz Intel i7 processor and 8GB of RAM. In all experiments the bottleneck resource was the CPU. Our experiments used logs with tens of thousands of events. From our previous studies [22] we consider this to be a representative log size for logs generated by developers during debugging sessions.

In the first experiment, we ran both algorithms on logs that ranged in size from 5K to 50K events, but maintained a constant number of property instances per log. Each log ranged over an alphabet of 5 event types, and each log was partitioned into 20 traces of equal length. The number of property instances true for each log was held constant at 182. We performed this experiment three times. Figure 4.11 plots the average runtime of the three runs for each log size.

In the figure, as the log size increases the standard kTails algorithm scales poorly because it needs to perform more merges. The InvariMint kTails algorithm maintains an almost constant running time. This is because for a constant number of property instances InvariMint kTails composes property instances in constant time — composing 182 property instances used in the experiment took about 10 seconds. Although the time to mine property instances does increase linearly with log size, it remains insignificant (for a 50K event log, all property instances are mined in under one second).

In the second experiment, we varied the number of property instances for the log from 108 to 1,480, but maintained a constant log size of 25K events. Logs were drawn from an alphabet that had between 9 and 37 event types. As above, each run was repeated three times and Figure 4.12 plots the average for each set of three running times. Overall InvariMint kTails had a lower running time than procedural kTails. However, the relative ratio between the two running times indicates that InvariMint kTails scales worse than procedural kTails as the number of property instances increases.

Overall we found that our declarative InvariMint kTails implementation outperforms kTails on large logs with few property instances, while procedural kTails scales better with increasing number of property instances.

4.5 Expressing Synoptic with InvariMint

This section describes the Synoptic model-inference algorithm, formulates it with InvariMint, and evaluates the resulting formulation.

4.5.1 Synoptic and its shortcomings

Synoptic is a model-inference algorithm that explicitly infers properties from the log, then constructs a model that satisfies them.[2] Synoptic first infers an overly-general model of the log, which accepts too many traces. Then, Synoptic progressively refines the model until every trace in the language of the model satisfies specific properties mined from the log. Because Synoptic models enforce these observed properties, prior work has found that the models accurately describe the underlying system and can improve understanding and aid debugging [22].

The Synoptic algorithm has four steps: **(1)** Mine three kinds of properties from the log — “*x*

[2] For simplicity, and despite minor differences, we use “property” where the Synoptic literature uses the term “invariant”.

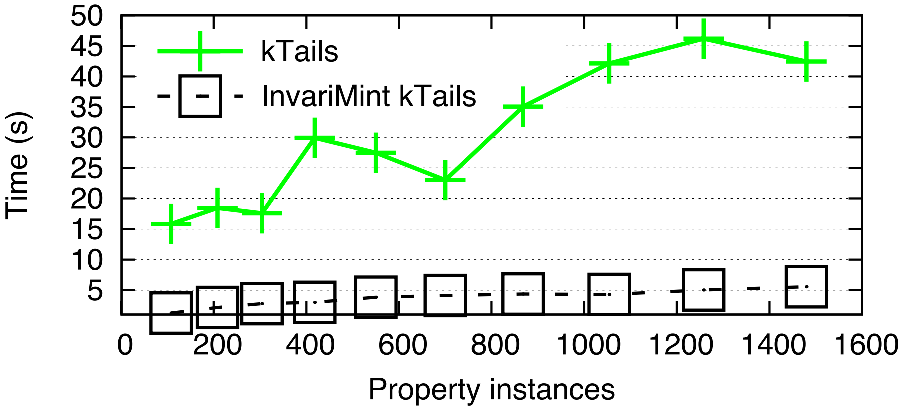


Figure 4.12: The running time of procedural kTails and the declarative InvariMint version of kTails for logs with different number of property instances. The size of the log was held constant at 25K events.

AlwaysFollowedBy *y*" (whenever event *x* occurs in a trace, event *y* also occurs later in the same trace), "*x* AlwaysPrecedes *y*" (whenever event *y* occurs in a trace, event *x* also occurs earlier in the same trace), and "*x* NeverFollowedBy *y*" (whenever event *x* occurs in a trace, event *y* never occurs later in the same trace). **(2)** Build an initial model by merging all anonymous[3] states with the same outgoing event into a single state. **(3)** Iteratively apply counterexample-guided abstraction refinement (CEGAR) [32] to derive a model that satisfies all of the mined properties. Synoptic does this by model checking the current (e.g., initial) model against the mined properties to find counterexample traces in the model's language, which falsify one or more of the properties. Synoptic then traces the found counterexample in the model to find the first state responsible for falsifying the property, and refines (splits) that state to remove the counterexample path. Synoptic repeatedly refines the model to eliminate counterexamples until it reaches a model that satisfies all of the properties. **(4)** Finally, to compact the model, Synoptic applies kTails(*k*=1) to the refined model, but only performs a merge

[3] Synoptic uses an event-based graph model with nodes representing event types and unlabeled edges representing observed event orderings in the log. This model is equivalent to an FSM with anonymous states, which is the model type we use in this chapter.

if it does not un-satisfy any of the properties.[4]

While empirically shown to help developers improve their system understanding and find bugs [22], Synoptic has two features that may cause its users difficulty.

First, Synoptic is non-deterministic. The order in which it resolves the counterexamples may affect the language of the final model it produces. (More generally, the problem Synoptic tries to solve is NP-complete [32, 61, 10], so the non-deterministic algorithm attempts to balance running time against the size of the final model.) If a user makes a change to the input log and Synoptic produces a different model, the user does not know if the input log difference explains the change in the returned model. This makes it difficult to apply Synoptic to verify a bug fix or to check how a new feature impacts the model.

Second, while significantly more efficient on large traces than kTail-based model inference, Synoptic may still be slow. This is because Synoptic must maintain all of the parsed log traces in memory, and it makes repeated model checking invocations and repeatedly traverses the model.

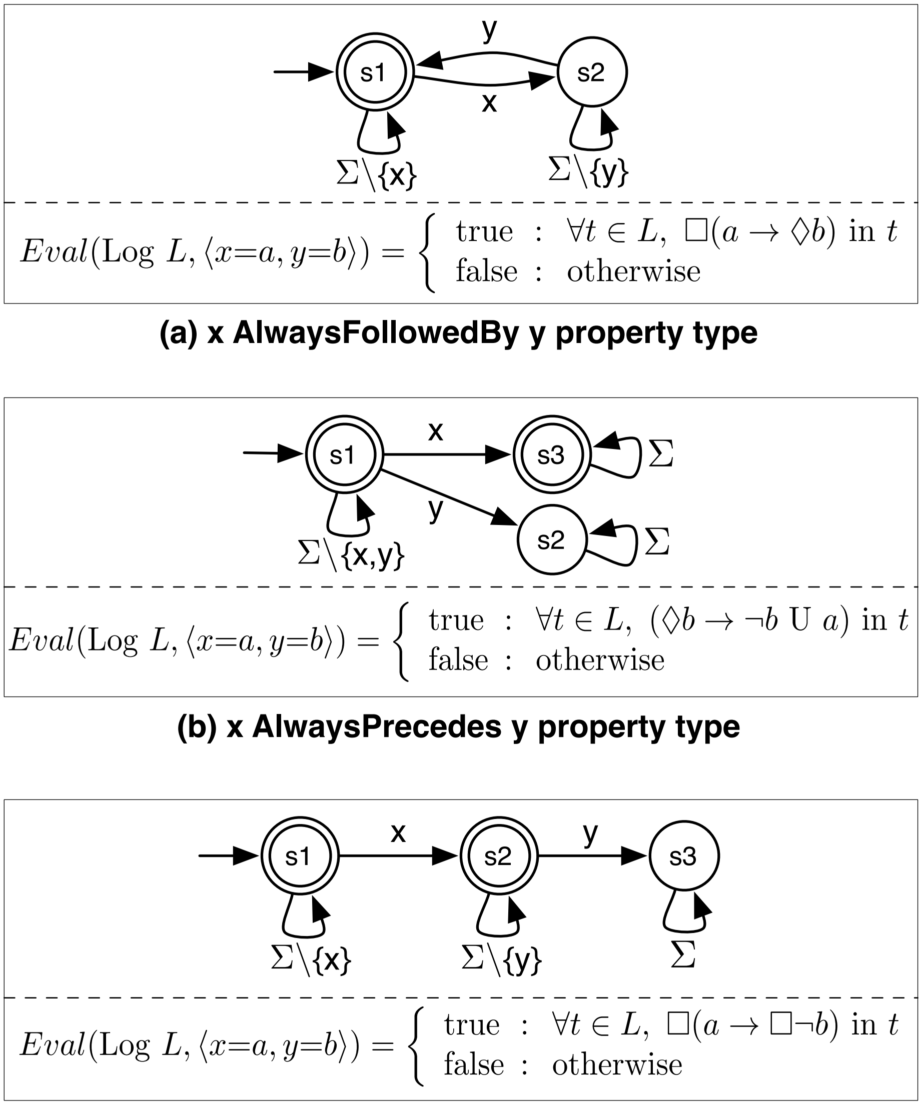
Next, we present an InvariMint formulation that approximates Synoptic. We show that the InvariMint algorithm resolves the above two issues of non-determinism and performance, and discuss insights that we gained about Synoptic through this formulation.

4.5.2 Modeling Synoptic with InvariMint

Synoptic's use of well-defined properties simplifies the task of declaratively specifying it with InvariMint — each of the three mined properties in Synoptic (AlwaysFollowedBy, AlwaysPrecedes, and NeverFollowedBy) has a corresponding property type, shown in Figure 4.13.

However, while Synoptic explicitly specifies some of the log properties that the inferred models will enforce, its original procedural definition imposed a property that was unknown both to Synoptic users and to us, the researchers who developed the algorithm. The process of specifying Synoptic declaratively with InvariMint revealed this property. We found that the initial Synoptic model is not captured by the three explicit properties and the InvariMint formulation requires the additional "immediately followed by" property type, which is exactly SimpleAlg's property type (Figure 4.4(b)).

[4] In an event-based model, Synoptic uses kTails(k=0) to merge nodes with identical event labels. This is equivalent to kTails(k=1) in a state-based model.


s1
s2
y
x
$\Sigma\backslash\{x\}$
$\Sigma\backslash\{y\}$
$Eval(\text{Log } L, \langle x{=}a, y{=}b\rangle) = \begin{cases} \text{true} : & \forall t \in L,\ \Box(a \rightarrow \Diamond b) \text{ in } t \\ \text{false} : & \text{otherwise} \end{cases}$
s1
s3
s2
x
y
Σ
Σ
$\Sigma\backslash\{x,y\}$
$Eval(\text{Log } L, \langle x{=}a, y{=}b\rangle) = \begin{cases} \text{true} : & \forall t \in L,\ (\Diamond b \rightarrow \neg b \text{ U } a) \text{ in } t \\ \text{false} : & \text{otherwise} \end{cases}$
s1
s2
s3
x
y
$\Sigma\backslash\{x\}$
$\Sigma\backslash\{y\}$
Σ
$Eval(\text{Log } L, \langle x{=}a, y{=}b\rangle) = \begin{cases} \text{true} : & \forall t \in L,\ \Box(a \rightarrow \Box\neg b) \text{ in } t \\ \text{false} : & \text{otherwise} \end{cases}$

(a) x AlwaysFollowedBy y property type

(b) x AlwaysPrecedes y property type

(c) x NeverFollowedBy y property type

Figure 4.13: Three of the four property types used by InvariMint to model the Synoptic algorithm. Figure 4.4(a) shows the fourth property type, which captures Synoptic's initial model.

To compose Synoptic property instances, InvariMint uses a composition formula that is similar to SimpleAlg: $Compose(Prop_1, \ldots, Prop_n) = Minimize(\cdots (Minimize(Prop_1 \cap Prop_2) \cap \cdots) \cap Prop_n)$. This composition minimizes intermediate models so as to maintain a small model in memory at

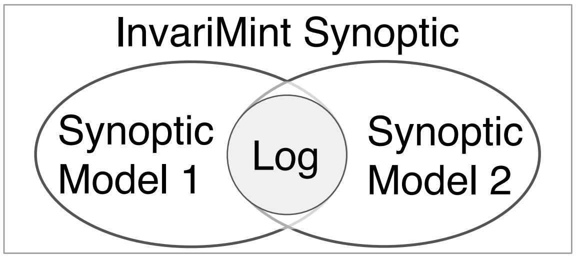


Figure 4.14: The inclusion relationships between an input log, the language of the model derived from the log with InvariMint Synoptic, and the languages of two potential non-deterministically-derived Synoptic models for the log.

runtime. For a large number of property instances, this composition yields a faster algorithm.

Next, we evaluate this InvariMint formulation of Synoptic.

4.5.3 Theoretical evaluation

We were already intimately familiar with Synoptic. Nonetheless, when we modeled Synoptic with InvariMint, we discovered a new feature, demonstrating how InvariMint can improve algorithm understanding. The InvariMint formulation of Synoptic is, in fact, an *approximation* of the Synoptic algorithm. A key feature of Synoptic models is that they have no *spurious* transitions. That is, every transition in the model is associated with some event in the log — there are no uncovered, or *spurious*, transitions. The reason for this feature is that Synoptic models are defined in terms of traces — a transition between two states in the model exists only if there are two observed states in the log that map to the model states and have this transition.

InvariMint models, on the other hand, are specified in terms of event types, so the particular trace-specific constraints are absent from an InvariMint model unless they are explicitly specified with property types. Therefore, InvariMint models may contain spurious transitions. Figure 4.14 summarizes the relationships between the language of the model derived using an InvariMint formulation of Synoptic, the languages of possible non-deterministically-derived Synoptic models, and the input log. The InvariMint formulation is more permissive than Synoptic, and includes the language of all possible non-deterministically-derived Synoptic models. Here, we prove that a Synoptic model's language is a subset of the model derived using InvariMint Synoptic algorithm. We also show that the InvariMint model does not satisfy any Synoptic property instances that are not true of the input log. This result is analogous to Theorem 3 in [22].

Theorem 4.2 (InvariMint specification of Synoptic encompasses Synoptic)**.** *Let L be a log. Let $F_{Synoptic}$ and $F_{InvMint}$ be the FSMs produced by the Synoptic algorithm and the InvariMint Synoptic algorithm on L, respectively. Let $Lang(F_{Synoptic})$ and $Lang(F_{InvMint})$ be the languages of those models. Then $Lang(F_{Synoptic}) \subseteq Lang(F_{InvMint})$.*

Proof: Let t be a trace in $Lang(F_{\text{Synoptic}})$. By construction, Synoptic terminates when all traces accepted by its inferred model satisfy all instances of the AlwaysFollowedBy, AlwaysPrecedes, and NeverFollowedBy property instances mined from L. Therefore, t must satisfy all such property instances.

Consider each of the property instances intersected to form F_{InvMint}. First, each property instance of the three types described in Figure 4.13 is mined from L, and therefore must be true in each trace in L. Since t satisfies all such property instances, the language of each of these instance FSMs must contain t. Second, each property instance of the type described in Figure 4.4(a) accepts all traces whose transitions are pairs of consecutive events observed in L. Since each transition in F_{InvMint} maps to at least one pair of consecutive events in at least one trace in L, a property instance FSM must accept t.

Since every property instance intersected to form F_{InvMint} accepts t, $t \in Lang(F_{\text{InvMint}})$. Therefore, $Lang(F_{\text{Synoptic}}) \subseteq Lang(F_{\text{InvMint}})$. □

Theorem 4.3 (Models produced by InvariMint Synoptic do not include false property instances)**.** *Let L be a log and let $F_{InvMint}$ be the FSM produced by the InvariMint Synoptic algorithm on L. More specifically, let $F_{InvMint} = Compose(P_1, ..., P_n)$.*

Let P_{false} be a set *of property instances, such that $\forall P_f \in P_{false}$, P_f is an instantiation of some Synoptic property type $\langle PFSM, Eval \rangle$, such that $\exists$ a binding B_f, $P_f = PFSM(B_f)$ and $Eval(L, B_f)$ is $false$. That is, P_{false} contains well formed property instances that are not true for the input log L.*

Then $\forall i, P_i \notin P_{false}$.

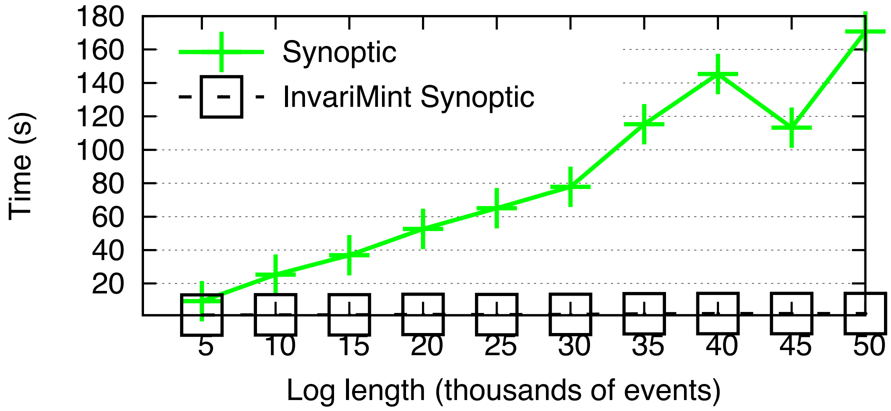


Figure 4.15: The running time of procedural Synoptic and the declarative InvariMint version of Synoptic for different log input sizes. The number of property instances true of the log was held constant at 19.

Proof:

We present a proof by contradiction. Assume the opposite — $\exists P_i$, $P_i \in P_{false}$.

Since P_i is used to construct F_{InvMint}, it must correspond to some property type $\langle PFSM, Eval\rangle$, and by the pseudocode in Figure 4.6, $\exists B$, $P_i = PFSM(L,B)$ and $Eval(L,B)$ is $true$.

However, by definition of the set P_{false}, $Eval(L,B)$ must be false. Contradiction.

□

As discussed in Section 4.5.1, Synoptic is non-deterministic and executing Synoptic on two similar logs may produce different models, even when using identical random number generator seeds. The InvariMint formulation of Synoptic removes this non-determinism because FSM intersection and minimization are commutative. This, in turn, makes it possible to use the algorithm to assist in other development tasks, such as to verify a bug fix or to check how a new feature impacts the model.

4.5.4 Empirical evaluation

We compared the performance of procedural Synoptic against the declarative InvariMint Synoptic implementation. Both algorithms are implemented in Java and we use the same experimental setting

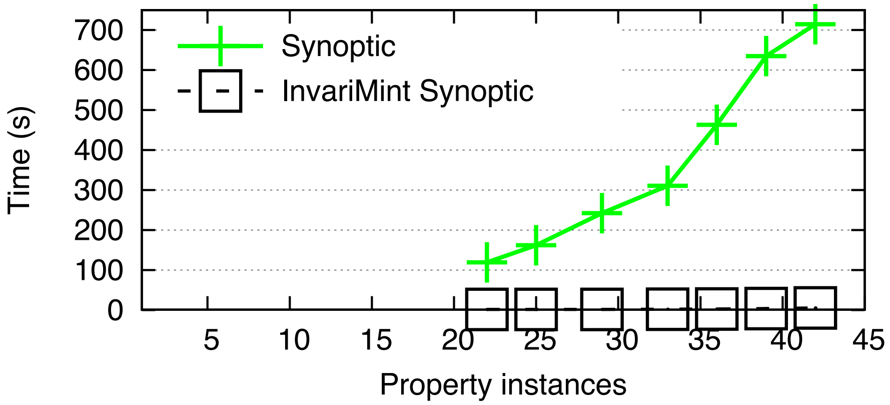


Figure 4.16: The running time of procedural Synoptic and the declarative InvariMint version of Synoptic for logs with different number of property instances. The size of the log was held constant at 25K events.

as in the kTails experiments (Section 4.4.3).

We carried out two experiments to compare algorithm performance across different log sizes (Figure 4.15), and across logs with varying number of property instances (Figure 4.16). As with the kTails algorithm, Figure 4.15 indicates that the declarative version of Synoptic outperforms procedural Synoptic on large logs. As the number of property instances increases (in Figure 4.16), InvariMint Synoptic continues to outperform Synoptic.

4.6 Discussion

Although this chapter has presented insights derived from expressing existing model-inference algorithms with InvariMint, there are other benefits to the InvariMint formulation. If the model is used for model checking or runtime verification, a declarative specification can be more efficiently checked (e.g., in parallel) against a property, and can yield more efficient runtime conformance checking of a trace. A violated property instance can also be more helpful than a path counterexample in understanding why the property does not hold or why a trace does not conform to the model.

As an example of the generality and expressiveness of our approach, an evaluation function may deem a property valid if it is true in *most* of the traces. This can be useful when the properties are probabilistic or the log is incomplete, as when it is not feasible to capture a log from a live, online system's start of execution to its end. For example, some traces at the start may be missing the `login` event while others at the end may be missing the `logout` event. InvariMint can still mine the property that all traces start with `login` and end with `logout`, as long as an overwhelming fraction of the traces satisfy that property. Other kinds of property types include conditional properties (e.g., an event is present only if the username is `root`), properties on resource usage (e.g., time or space), and anomaly-detecting properties (e.g., two events co-occur rarely).

In this chapter we use LTL to compactly specify evaluation functions. As a result, in all of the presented examples the PFSM could be automatically derived from the LTL — the PFSM is a parameterized version of the büchi automaton corresponding to the LTL formula. However, this is not possible for the alternative evaluation functions mentioned above, as these cannot be expressed with LTL.

InvariMint can be robust to specifications with overlapping or conflicting property types. For example, an evaluation function that intersects property instances will ignore overlapping property instances, and will immediately reveal conflicting property instances as their intersection would be the empty set.

4.6.1 Tips for Declaratively Expressing Algorithms with InvariMint

First, identify the right property-type granularity. Do not simply simulate the procedural version of the algorithm with the property types. Instead, consider the properties that the procedural algorithm enforces. Property types that are too fine-grained and too close to the input traces (e.g., union of positive example trace DFAs) lead to models that overfit the log, rather than describe the algorithm. Property types can describe algorithm operations. For example, Section 4.4 showed how a single property type describes merging of all states with the same k-tail.

If the procedural algorithm deals with positive examples of traces (as both kTails and Synoptic do), starting from a formulation that produces a model that is a generalization of the desired model may be easier, as this model may enforce fewer properties. Then, refine this model towards the desired model by introducing new property types or by refining the existing properties.

If the procedural algorithm deals with both positive and negative examples of traces (we have not shown such an algorithm), consider building separate models, one for the positive examples and one for the negative examples. Then, in the composition function, subtract the negative-example model from the positive-example model.

4.7 Summary

Model-inference algorithms can automatically mine models of complex systems. Such models aid numerous development tasks, such as program understanding and debugging. Unfortunately, existing model-inference algorithms are defined procedurally, making them difficult to understand, extend, and compare to one another.

This chapter presented InvariMint, a declarative specification approach for model-inference algorithms. InvariMint enables specification of algorithms in terms of the types of properties they enforce in the models they infer. InvariMint's declarative specifications **(1)** provide insight into how inference algorithms work and how the model relates to the underlying system, **(2)** allow for easy extension of existing algorithms to construct hybrid alternatives, and **(3)** provide a common language for comparing and contrasting the essential aspects of model-inference algorithms.

We demonstrated the benefits of InvariMint by declaratively specifying two existing algorithms, kTails and Synoptic. For example, the InvariMint versions of these algorithms greatly outperform their procedural analogs.

Chapter 5

Related work

This chapter overviews related prior work that this dissertation builds on. This prior work can be placed into four categories: related techniques, particularly other approaches to model inference and related work in formal methods (Section 5.1); prior work on analysis and visualization of logs (Section 5.2); and, work on comprehension, debugging, and verification of systems (Section 5.3).

5.1 Model inference and formal methods

5.1.1 Inference of DFA models

The problem of automata inference from positive examples of executions is computable [24], but is NP-complete [61, 10], and the FSM cannot be approximated by any polynomial-time algorithm [104]. Therefore, polynomial-time algorithms that explore the FSM space are approximation algorithms. Many of the traditional automata inference algorithms are implemented in LearnLib [106], a modular library for automata learning.

One of the first papers to apply automata inference to software engineering for process discovery is by Cook et al. [35]. A key contribution of this paper is to introduce the software engineering community to the kTails algorithm [23]. This algorithm takes a finite state model and produces a more compact one by recursively merging states whose root subgraphs are identical up to a depth of k. kTails can produce precise models for small and simple systems, but when complexity of the system increases, the precision of the inferred models decreases dramatically [87].

The key idea in kTails is to start from a large FSM in which a state represents the state of the system after a single observed event instances in the log. Then, relying on some criterion, depending on the algorithm variant, the states of this FSM are merged. The techniques differ largely in the criterion. kTails is the basis for numerous model-inference algorithms [35, 90, 87, 88, 91, 29, 108, 126]. Many of these algorithms can be modeled with InvariMint to better understand, extend, combine, and compare them.

As an example of an extension made to the kTails algorithm, Lo et al. [90] use temporal properties mined from execution traces to guide state merging and ensure that the final model satisfies temporal constraints. Temporal-invariant-consistency greatly increases the model's precision. Synoptic produces similar high-precision models while leveraging refinement, as opposed to coarsening, to greatly increase the efficiency and scalability of the approach. Krka et al. [81] have proposed, though have not yet implemented, using refinement and mined invariants to improve precision of inferred models beyond that of Lo et al.'s approach.

Numerous techniques leverage developer-written specifications to infer system models. Whittle and Schumann [127] generate component statecharts from scenarios and properties. Damas et al. [38] inductively infer labeled transition system (LTS) models from scenarios interactively provided by the developer. A later extension of this approach reduces the number of questions to the developer [39] by incorporating FLTL properties [60]. However, these techniques can synthesize overspecified models and require significant human input. Uchitel et al. [122] proposed using message sequence charts [75] to infer LTS models and discover implied scenarios. Harel et al. [68] synthesize statecharts from live sequence charts [40]. LTSs can also be constructed based on pre- and post-condition specifications [9, 42]. De Caso et al. [42] generate abstract models to support validation of the specifications. Alarjeh et al.'s technique [9] facilitates refinement of pre- and post-conditions based on system goals and execution scenarios. Similarly, Krka et al.'s algorithm [80] synthesizes behavioral models from pre- and post-condition specifications and execution scenarios and can synthesize component-level models from system-level specification. Uchitel et al. [121] argued that it is crucial to consider the specifications' partiality when using developer-written specifications to infer models. In contrast to all these approaches, the tools in this dissertation require much less input from the developer — the logs that are usually already generated by systems, and a small set of regular expressions, and are suitable for legacy systems. Although some systems are not instrumented to generate logs and may require developers to change the implementation, logging is considered to be generally useful and adding such instrumentation leads to better software. Further, we hope that the utility of the Synoptic, Dynoptic, and InvariMint tools will motivate developers to improve their own logging habbits.

Model-inference frameworks can facilitate algorithm comparison [105]. However, to date, these frameworks have been used to compare model performance and accuracy, not properties of model inference. Further, much of the kTails-based model-inference work compares the recall and precision of inferred models against manually-specified ground-truth models. This process is manual, error-prone, and, again, compares model quality, as opposed to model-inference properties. Model quality is a notoriously challenging aspect of model inference [87]. QUARK, a comparison framework, allows for comparing the quality of models generated by algorithms such as kTails and sk-strings [107]. InvariMint is complementary to these frameworks, as it aims to unify model-inference algorithms with a declarative specification language, facilitating algorithm comparison, and model property comparison.

Li et al. came up with an approach that resembles InvariMint, but targets reactive systems [85]. Their approach is similar to InvariMint in that it uses a set of property templates and mines LTL specifications based on these templates. Unlike InvariMint, the mined properties are not composed into a model.

Walkinshaw et al. [126] propose a model-inference technique in which the user provides a model-inference algorithm with LTL formulae, which are then checked by a model checker and are used as constraints on feasible state merges in the inference algorithm. InvariMint uses LTL differently. Our intent is generalize the specification of model-inference algorithms. To this end, LTL formulae encode valid bindings of variables in a parameterized FSM to event types for a particular log input.

Many model inference techniques require richer model formalisms than the standard FSM models inferred by Synoptic and InvariMint. For example, GK-Tails [91] requires extended FSMs, and RPNI [29] requires probabilistic FSMs. These more general models and non-FSM model inference (e.g., of UML sequence diagrams [138], communicating automata [26], and symbolic message sequence graphs [82]) have even more potential in aiding developers. We believe that InvariMint can be extended to accommodate these other model types. Similarly, InvariMint may be extendable to other types of properties, such as those used to infer behavioral models of web-services [17].

For a broad overview of other FSM model inference algorithms see the survey by Shaukat et al.[8].

5.1.2 Property inference

Temporal properties.

Javert [56] is a specification mining tool that infers complex specifications by composing simpler patterns into larger ones. Javert's invariants are more complex than ours (e.g., it handles invariants over three events). Similarly, Perracotta [134] mines and visualizes temporal properties of event traces. These invariants have been used to study program evolution [132]. All of these systems require TO logs (or observed executions), whereas our work concentrates on PO logs, common in distributed systems.

The properties used by Synoptic and Dynoptic are specified with code (i.e., in Java), however there is prior work that lowers the bar to specifying these properties, for example with graphical approaches [12].

Perracotta [133, 132] mines and visualizes temporal properties from event traces, for a similar purpose. To aid software developers, Gabel et al. developed Javert [56], a specification mining tool that infers complex specifications by composing simpler temporal patterns into larger ones. Their rule-based pattern composition technique is an elegant means of extending a finite temporal invariant set. The mined invariants can then be a step towards other applications, such as increasing the accuracy of model inference.

Jiang et al. [77] proposed approximately mining certain types of invariants that relate flow intensities (e.g., traffic volume) in distributed systems. These invariants capture non-temporal properties. In contrast, our proposal is exact, not approximate, and captures temporal properties. Yabandeh et al. [130] describe Avenger, which mines invariants that hold most of the time. These almost-invariants are helpful for finding bugs that manifest infrequently. Avenger mines a rich set of data invariant types; it does not mine temporal properties.

Finally, Lou et al. [93] define a set of event dependencies that range over events in interleaved traces of independent processes. These include what they term *forward* and *backward* dependencies. The temporal invariants in Dynoptic consider communicating processes, and not just dependent processes.

The Dynoptic evaluation compared three invariant mining algorithms that pre-process the log into DAGs to mine invariants. However, it is also possible to mine invariants directly from a log by first enumerating all the possible invariants based on event types in the log, and then traversing each of the system traces in the log and checking each invariant to eliminate the false invariants. Algorithm developed by Sen et al. [115] for efficiently checking certain kinds of temporal predicates over consistent cuts of a distributed execution could be used for this. However, efficient traversal of the traces without an explicit DAG structure is non-trivial. We plan to implement this more direct algorithm in our future work. More generally, other approaches such as those based on graph reachability could be used for mining invariants [31]. Counting seems to capture the minimum information necessary for our invariant types, but we want to explore other approaches in our future work.

Recent work by Gabel and Zhendong can be applied to validate property instances during an InvariMint execution [57].

Structural properties.

This thesis concentrates on temporal invariants of system behavior. However, there are other types of invaraints, or properties, that have been used in model inference, and that can inspire further work in extending the techniques in the thesis.

Daikon [51] is one of the more popular tools to mine data-value, or structural invariants. Daikon observes system executions and mines data structure invariants as method pre- and post-conditions, and class-level properties over internal program variables. Daikon invariants have been shown to be useful for understanding program evolution. However, they have also been extensively used in work on model-inference.

For example, Lorenzoli et al. [91] developed GK-Tail, a variant of kTails, and applied it to logged sequences of method call invocations. Their technique generates Extended Finite State Machines (EFSMs). In their setting each trace consists of a sequence of method invocations annotated with values for parameters and variables, and is generated by a single execution run of the program being analyzed. The transition arcs of the resulting EFSM are annotated with the relevant constraints on the data to transition from one state to the next state, providing a behavioral view of the program that includes how the input data affects the behavior. Unlike the model inference techniques in this dissertation, the GK-Tail algorithm does not preserve mined trace invariants.

Daikon invariants have also been used to infer better models of distributed systems. Lo et al. have used Daikon to infer more accurate live sequence charts, which are interaction specifications blocks between two concurrent components [89].

5.1.3 Inferrence of networked and concurrency models

CFSMs inferred from executions can demonstrate certain properties, such as absence of deadlocks and unspecified receptions [30, 123]. This prior work is theoretical, and while such properties are important in theory, in practice, we found that they are not necessary to generate models that provide insight into system implementation. Bollig et al. have also explored the problem of inferring a CFSM from message sequence charts [26]. However, these charts must be manually labeled as positive or negative examples of system behavior. In contrast, Dynoptic relies on mined invariants and automates the inference. Recent work by Kumar et al. considers the problem of inferring *class* level specifications of distributed systems, in the form of symbolic message sequence charts [82]. This can be used by Dynoptic to merge identical process FSMs (e.g., the replica models in Figure 3.21(a,c)).

A Petri net [103] is an alternative formalism to a CFSM. Petri nets provide a formal means to model and reason about concurrent systems. Their main advantage over CFSMs is their explicit representation of concurrency and associated concepts like mutual exclusion. However, Petri nets are more difficult to understand and generate than CFSMs and, in our experiments, the ability to express explicit concurrency made it more difficult to understand system behavior.

Dynoptic's goal is to find a short sequence of refinements to produce a small (abstract) AFSM that satisfies all of the valid invariants. This problem is NP-hard [32], and Dynoptic's design finds an approximate solution. Unlike Dynoptic, prior work on model inference from totally ordered logs either excluded concurrency or captured a particular interleaving of concurrent events [36, 108, 34, 6]. Dynoptic captures concurrency as a partial order, which we believe is crucial for understanding a concurrent system's behavior.

Besides Synoptic, the closest total-order log work to Dynoptic is Tomte [5], which also leverages CEGAR [32]. Synoptic mines temporal invariants and infers FSM models from a log of sequential executions while Tomte infers scalarset Mealy machines. In contrast to both of these tools, Dynoptic accounts for richer kinds of invariants, deals with a richer model type, and provides insight into partially ordered logs.

Dynoptic relies on the McScM model checker [71, 70] for checking the validity of invariants in a CFSM model. McScM is one of the most advanced verification tools for communicating systems; building on prior state of the art systems like TReX [11], and symbolic verification of CFSMs with

QDDs [25]. A key property of McScM is that unlike most CFSM verification tools, like SPIN [72], McScM verifies models with no apriori bound on channel size. However, McScM is not guaranteed to terminate. To make model checking more tractable (and faster) we plan to extend Dynoptic to support model checkers that verify CFSM models with bounded channel lengths.

There is significant prior work on software comprehension that is relevant to this dissertation. The most relevant of these is work by Murphy et al. on reflexion models [98]. This work uses user-defined regular expressions to extract an abstract model from the source code, by matching regular expressions against the code statements and reasoning about control flow between the matched statements. The abstract model can then be utilized by developers to better understand the underlying software. A key difference between reflexion models and models inferred by tools like Synoptic is that reflexion models deal directly with code. Synoptic and related tools treat the system as a black box and only require observations of system behavior (i.e., a log).

5.1.4 Model refinement

Synoptic uses counterexample-guided abstraction refinement (CEGAR) [32] to derive a model that satisfies a set of temporal properties.

A bisimulation is a simulation relation that provides a strong notion of similarity for relational structures [112]. Its key feature is to preserve certain properties of the relational structure, for example, two strongly bisimilar transition systems are guaranteed to satisfy the same set of LTL formulae. An important application in model checking is model minimization [53]. Our BisimH algorithm is a modification of a partition refinement algorithm [100], which uses invariants to determine which state to split next and when to stop splitting, resulting in a coarser representation that is not bisimilar to the input structure. Our BisimH algorithm is also related to the partition refinement algorithms in [50], but BisimH uses invariants to guide exploration and termination.

5.2 Log analysis and visualization

5.2.1 Log analysis

Numerous log analysis tools exist; however, we know of no freely-available tool to extract finite state machine models from console logs. A popular log processing tool in the enterprise is Splunk[1], which supports various analyses and understands many common log formats. Splunk's main advantage is the scalability of its analyses due to MapReduce [45]. Popular tools that are similar to Splunk are Sawmill[2] and AWStats[3]. Synoptic supports log exploration, which is a more general goal than what is targeted by tools that have a tighter focus, such as Sisyphus[4], which targets anomaly detection.

Other prior work on mining systems logs focused on detecting dependencies [92], anomalies [76, 94, 129, 136], and performance debugging [111, 118, 119]. That work does not target the problem of finding a concise model for an arbitrary system generating the log. For instance, SALSA [118] and Mochi [119] extract and visualize node behavior of Hadoop [65] node logs to support performance debugging. This line of work is MapReduce-specific.

One area in which log analysis is helpful is debugging. Bugs can manifest themselves via anomalous executions and detecting anomalies in distributed systems is a popular research area [129, 76, 136]. The models generated by the tools presented in this dissertation can be applied to anomaly detection through model differencing.

Bates et al. [15] developed an event definition language that caused programs to generate logs with deep semantics information, such as hierarchical relationships between events. Their approach requires access to the source code. In contrast, our approach does not need access to the source code, and works on the already generated logs. We do require the developer to express a set of regular expressions. However, this set may be mined automatically [124, 137].

MapReduce-specific research — SALSA [118] and Mochi [119] — has created visualizations helpful to performance debugging of Hadoop [65] node logs. Our approach, of course, is generic and

[1] http://www.splunk.com

[2] http://www.sawmill.net

[3] http://awstats.sourceforge.net

[4] http://www.cs.sandia.gov/~jrstear/sisyphus

applicable to a wide range of systems.

Dynoptic assumes the availability of vector timestamped logs. A drawback to using vector timestamps in large systems is their performance penalty — vector length scales linearly with the number of hosts in the system and exchanging them may negatively impact network performance. Though more efficient vector clock mechanisms exist [16], we believe that their application can be made practical by limiting their use to short time periods on large systems, or by using vector clocks exclusively for debugging and during development and testing.

A partially ordered log contains more information, and is therefore more complex, than a totally ordered log. It is generally infeasible to analyze it manually. Prior work on automated partially ordered log analysis has concentrated on visualization for simplifying analysis [33, 117].

5.2.2 Log visualization

A well designed trace visualization tool can significantly improve productivity [37]. One important reason for this is that developers often do not fully understand their systems [97]. Therefore, a key feature of Synoptic, Dynoptic, and InvariMint is a model visualization interface, using which developers explore the generated models.

As an alternative to timeline diagrams for describing distributed system executions Stone et al. propose concurrency maps [117]. A concurrency map is essentially a matrix, with each column corresponding to a single process, and a row corresponding to a time slice during which all the events in the row (across some set of processes) are required to execute. Although concurrency maps are unable to encode arbitrary concurrent executions [49], they have been highly influential as a visualization paradigm. For example, DTVS is one system that builds on concurrency maps [48]. The concurrency map notion of a geometrically rigid temporal dimension can be used in Synoptic to align inter-node edges to point in approximately the same direction.

De Pauw et al. mine communication flow patterns at transaction granularity, and have developed a novel visualization tool to cluster and present the pattern clusters to users [44]. In contrast, Synoptic mines a single compact model to describe all of the input traces. The visualization insight of presenting independent paths that make up the model could be one potential way of simplifying the display of complex Synoptic models. With a different tool, called Zinsight, De Pauw et al. help users make sense logs with millions of events [43]. This work has numerous visualization techniques that Synoptic can build on, such as compact visual encoding of event statistics.

Finally, Hy+ [33] is a system for parallel and distributed systems debugging. This work demonstrated that visualizing distributed executions can help developers identify bugs. Hy+ uses a visualization technique that is a cross between Harel's statecharts [67] and directed hypergraphs. Future work on Dynoptic model visualization can integrate many of the techniques trailblazed by Hy+.

5.3 Comprehension and debugging of systems

Distributed systems are notoriously difficult to get right. This is exemplified by recent efforts that target bug finding in distributed systems [135, 131].

5.3.1 Black-box debugging techniques

A popular black-box approach to debugging distributed systems has been pioneered by Aguilera et al. [7]. In this approach, multiple observations of a distributed executions are used to infer causality, dependencies, and other important characteristics of the system. This approach was somewhat relaxed in later work to produce more informative and application specific results in Magpie [13] and later in X-Trace [55]. However, the approach is still popular. For example, Mysore et al. [99] developed a system that leverages information flow tracking for collecting and then visualizing execution flow in a distributed setting. This is a black-box approach, but the focus is on data. This provides a much finer level of granularity but has significant overhead. Another system — BorderPatrol [79] traces requests among a set of binary modules that cannot be modified.

The tools in this dissertation rely on unstructured log analysis and are therefore black-box techniques. However, these tools leverage existing logging mechanisms that a developer purposefully added to the system. The assumption being that developers log information that they consider useful, and this is therefore a sufficient source of information for making inferences about the system. Another major difference from this prior systems debugging work is that this prior work focuses on *data*, or request flow, while the tools in the dissertation generate models that describe sequences of *events* executed by a system.

5.3.2 General debugging of distributed system.

Quality specification can aid debugging, but as specification tools like CADP [58] have not gained wide adoption by system builders, the community has begun advocating alternate specification styles. Thereska et al. [120] has argued that modeling and specifying systems is hard. Automatically mined invariants can serve as partial specification and can be used to compare the system's implementation to the developers' understanding of the system.

Declarative programming is an alternative system construction model that Singh et al. [116] argue is well suited to building robust systems that are easier to debug and understand. Unlike our work, such a drastic approach requires significant effort from the developers and cannot be applied to legacy systems.

There are also tools that perform run-time checking of distributed systems. These tools monitor the system as it executes and check for user-defined system invariants or properties. If a property is violated the checker might report the incident to the user or developer [109, 59, 86, 41], or the system may automatically attempt to steer away from the violation [131, 101]. Unfortunately, many of the built-in record-and-replay techniques [64] require access, and modifications, to the source code [59, 41]. Others, require access to the binaries for instrumentation [86]. Again, our approach, which can complement these techniques, can help remove the requirement for source code and binary access.

Of all of these alternatives, Pip [109] is the only tool that provides the developer with visualization and logging support to help explore what occurred during an execution. However, Pip's ultimate goal is to diagnose deviations from developer-supplied properties. In contrast, Dynoptic does not check the system, requires few inputs, and is intended to help a developer better understand their implementation by inferring a model that describes the observed executions.

MODIST [135], a transparent model checker for distributed system, does not require a user to specify any properties, and instead explores the space of all possible event interleavings to find bugs that crash the system. Our work addresses precisely the inefficiency of exploring all possible interleavings.

The tools described in this dissertation take a fundamentally different approach from these prior tools on the assumption that some developers will not want to specify properties of their systems, either because this is too difficult or too time-consuming. Instead, our tools mine a model to describe what occurred and display the observed behavior visually. We have observed that developers can often glean numerous details from a visualization, and discover system property violations for properties that they would not have thought important or relevant prior to using the tool.

Chapter 6
Conclusion

Logging is a popular debugging methodology. Unfortunately, large logs are often complex and difficult to analyze manually. This dissertation presented three tools — Synoptic, Dynoptic, and InvariMint — to help developers interpret the observed behavior of their systems through automated model inference.

Synoptic mines logs to derive relational models of sequential systems. Dynoptic mines logs of networked systems to infer communicating finite state machine models. InvariMint leverages the insight of leveraging temporal invariants in Synoptic and Dynoptic to come up with a declarative framework that simplifies the task of creating, understanding, extending, and comparing model inference algorithms more generally.

These tools bridge the gap between systems developed by developers with little to no training in formal methods, and a suite of methods developed by the formal methods community. Overall, this dissertation demonstrates that execution logs generated by large systems can be mined with minimal input from the user to generate models of system behavior that are useful for improving developers' understanding of their systems.

Bibliography

[1] IPv4 Specification, Record Route option. http://www.ietf.org/rfc/rfc791.txt. pg. 20, 21 Accessed april 5, 2011.

[2] IPv4 Specification, Timestamp option. http://www.ietf.org/rfc/rfc791.txt. pg. 22, 23. Accessed april 5, 2011.

[3] Pattern (Java Platform SE 7 b141) http://download.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html. Accessed July 1, 2011.

[4] PlanetLab | An open platform for developing, deploying, and accessing planetary-scale services. http://planet-lab.org/. Accessed April 5, 2011.

[5] Fides Aarts, Faranak Heidarian, Harco Kuppens, Petur Olsen, and Frits Vaandrager. Automata Learning through Counterexample Guided Abstraction Refinement. In Dimitra Giannakopoulou and Dominique Méry, editors, *FM 2012: Formal Methods*, volume 7436 of *Lecture Notes in Computer Science*, pages 10–27. Springer Berlin Heidelberg, 2012.

[6] Mithun Acharya, Tao Xie, Jian Pei, and Jun Xu. Mining API patterns as partial orders from source code: from usage scenarios to specifications. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 25–34, New York, NY, USA, 2007. ACM.

[7] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. *SIGOPS Oper. Syst. Rev.*, 37(5):74–89, October 2003.

[8] Shaukat Ali, Kirill Bogdanov, and Neil Walkinshaw. A comparative study of methods for dynamic reverse-engineering of state models. Technical Report CS-07-16, Department of Computer Science, The University of Sheffield, 2007.

[9] Dalal Alrajeh, Jeff Kramer, Alessandra Russo, and Sebastin Uchitel. Learning operational requirements from goal models. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 265–275, Washington, DC, USA, 2009. IEEE Computer Society.

[10] Dana Angluin. Finding patterns common to a set of strings (Extended Abstract). In *Proceedings of the eleventh annual ACM symposium on Theory of computing*, STOC '79, pages 130–141, New York, NY, USA, 1979. ACM.

[11] Aurore Annichini, Ahmed Bouajjani, and Mihaela Sighireanu. TReX: A Tool for Reachability Analysis of Complex Systems. In *Proceedings of the 13th International Conference on Computer Aided Verification*, CAV '01, pages 368–372, London, UK, 2001. Springer-Verlag.

[12] M. Autili, P. Inverardi, and P. Pelliccione. Graphical scenarios for specifying temporal properties: an automated approach. *Automated Software Engineering*, 14(3):293–340, September 2007.

[13] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using magpie for request extraction and workload modelling. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, page 18, Berkeley, CA, USA, 2004. USENIX Association.

[14] Ethan K. Bassett, Harsha V. Madhyastha, Vijay K. Adhikari, Colin Scott, Justine Sherry, Peter Van Wesep, Thomas Anderson, and Arvind Krishnamurthy. Reverse traceroute. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, page 15, Berkeley, CA, USA, 2010. USENIX Association.

[15] P. Bates. High-level Debugging of Distributed Systems: The Behavioral Abstraction Approach. *Journal of Systems and Software*, 3(4):255–264, December 1983.

[16] Daniel Becker, Rolf Rabenseifner, Felix Wolf, and John C. Linford. Scalable timestamp synchronization for event traces of message-passing applications. *Parallel Computing*, 35(12):595–607, December 2009.

[17] Antonia Bertolino, Paola Inverardi, Patrizio Pelliccione, and Massimo Tivoli. Automatic synthesis of behavior protocols for composable web-services. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 141–150, New York, NY, USA, 2009. ACM.

[18] Ivan Beschastnikh, Jenny Abrahamson, Yuriy Brun, and Michael D. Ernst. Synoptic: studying logged behavior with inferred models. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 448–451, New York, NY, USA, 2011. ACM.

[19] Ivan Beschastnikh, Yuriy Brun, Jenny Abrahamson, Michael D. Ernst, and Arvind Krishnamurthy. Unifying FSM-inference algorithms through declarative specification. In *Proceedings of the 35th International Conference on Software Engineering (ICSE13)*, pages 252–261, San Francisco, CA, USA, May 2013.

[20] Ivan Beschastnikh, Yuriy Brun, Jenny Abrahamson, Michael D. Ernst, and Arvind Krishnamurthy. Unifying FSM-Inference Algorithms through Declarative Specification. Technical Report UW-CSE-13-03-01, University of Washington, 2013.

[21] Ivan Beschastnikh, Yuriy Brun, Michael D. Ernst, and Arvind Krishnamurthy. Making sense of distributed system logs with Dynoptic. Technical Report UW-CSE-12-10-01, http://homes.cs.washington.edu/~ivan/papers/dynoptic_tr.pdf, Univ. of Washington, 2012.

[22] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D. Ernst. Leveraging Existing Instrumentation to Automatically Infer Invariant-Constrained Models. In *Proceedings of the the 8th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '11, 2011.

[23] A. W. Biermann and J. A. Feldman. On the synthesis of Finite-State machines from samples of their behavior. *Computers, IEEE Transactions on*, C-21(6):592–597, June 1972.

[24] L. Blum and M. Blum. Toward a mathematical theory of inductive inference. *Information and Control*, 28(2):125–155, June 1975.

[25] Bernard Boigelot and Patrice Godefroid. Symbolic Verification of Communication Protocols with Infinite StateSpaces using QDDs. *Form. Methods Syst. Des.*, 14(3):237–255, May 1999.

[26] Benedikt Bollig, Joost P. Katoen, Carsten Kern, and Martin Leucker. Learning Communicating Automata from MSCs. *IEEE Trans. Softw. Eng.*, 36(3):390–408, May 2010.

[27] Daniel Brand and Pitro Zafiropulo. On Communicating Finite-State Machines. *J. ACM*, 30(2):323–342, April 1983.

[28] Frederick P. Brooks. *The mythical man-month (anniversary ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[29] Rafael C. Carrasco and José Oncina. Learning Stochastic Regular Grammars by Means of a State Merging Method. In *Proceedings of the Second International Colloquium on Grammatical Inference and Applications*, ICGI '94, pages 139–152, London, UK, UK, 1994. Springer-Verlag.

[30] Xiao J. Chen and Hasan Ural. Construction of Deadlock-free Designs of Communication Protocols from Observations. *The Computer Journal*, 45(2):162–173, January 2002.

[31] Yangjun Chen and Yibin Chen. An Efficient Algorithm for Answering Graph Reachability Queries. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, pages 893–902, Washington, DC, USA, 2008. IEEE Computer Society.

[32] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-Guided abstraction refinement. In E. Emerson and A. Sistla, editors, *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, chapter 15, pages 154–169. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2000.

[33] Mariano C. Consens, Masum Z. Hasan, and Alberto O. Mendelzon. Debugging Distributed Programs by Visualizing and Querying Event Traces. In *Applications of Databases, First International Conference, ADB-94*, volume 819, pages 181–183, 1993.

[34] Jonathan E. Cook, Zhidian Du, Chongbing Liu, and Alexander L. Wolf. Discovering models of behavior for concurrent workflows. *Comput. Ind.*, 53:297–319, April 2004.

[35] Jonathan E. Cook and Alexander L. Wolf. Discovering models of software processes from Event-Based data. *ACM Transactions on Software Engineering and Methodology*, 7:215–249, 1998.

[36] Jonathan E. Cook, Alexander L. Wolf, Jonathan, and Er L. Wolf. Event-based detection of concurrency. In *In Proceedings of the Sixth International Symposium on the Foundations of Software Engineering (FSE-6*, pages 35–45, 1998.

[37] Bas Cornelissen, Andy Zaidman, Arie van Deursen, and Bart van Rompaey. Trace visualization for program comprehension: A controlled experiment. In *Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference on*, pages 100–109, May 2009.

[38] Christophe Damas, Bernard Lambeau, Pierre Dupont, and Axel van Lamsweerde. Generating Annotated Behavior Models from End-User Scenarios. *IEEE Trans. Softw. Eng.*, 31(12):1056–1073, December 2005.

[39] Christophe Damas, Bernard Lambeau, and Axel van Lamsweerde. Scenarios, goals, and state machines: a win-win partnership for model synthesis. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT '06/FSE-14, pages 197–207, New York, NY, USA, 2006. ACM.

[40] Werner Damm and David Harel. LSCs: Breathing Life into Message Sequence Charts. 19(1):45–80, 2001.

[41] Darren Dao, Jeannie Albrecht, Charles Killian, and Amin Vahdat. Live Debugging of Distributed Systems. In *Proceedings of the 18th International Conference on Compiler Construction: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, CC '09, pages 94–108, Berlin, Heidelberg, 2009. Springer-Verlag.

[42] G. de Caso, V. Braberman, D. Garbervetsky, and S. Uchitel. Validation of contracts using enabledness preserving finite state abstractions. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 452–462, Washington, DC, USA, May 2009. IEEE.

[43] Wim De Pauw and Steve Heisig. Zinsight: a visual and analytic environment for exploring large event traces. In *Proceedings of the 5th international symposium on Software visualization*, SOFTVIS '10, pages 143–152, New York, NY, USA, 2010. ACM.

[44] Wim De Pauw, Sophia Krasikov, and John F. Morar. Execution patterns for visualizing web services. In *Proceedings of the 2006 ACM symposium on Software visualization*, SoftVis '06, pages 37–45, New York, NY, USA, 2006. ACM.

[45] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[46] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.

[47] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in Property Specifications for Finite-State Verification. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 411–420, New York, NY, USA, 1999. ACM.

[48] D. Edwards and P. Kearns. DTVS: a distributed trace visualization system. In *Parallel and Distributed Processing, 1994. Proceedings. Sixth IEEE Symposium on*, pages 281–288, 1994.

[49] D. Edwards, S. Simmons, and P. Kearns. Graphical Limits of Concurrency. *Neural, Parallel and Scientific Computations*, 12:219–232, 2004.

[50] Tapio Elomaa. Partition-Refining algorithms for learning finite state automata. In *Proceedings of the 13th International Symposium on Foundations of Intelligent Systems*, ISMIS '02, pages 232–243, London, UK, UK, 2002. Springer-Verlag.

[51] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, February 2001.

[52] Colin J. Fidge. Timestamps in Message-Passing Systems that Preserve the Partial Ordering. In *11th Australian Computer Science Conference*, pages 55–66, University of Queensland, Australia, 1988.

[53] Kathi Fisler and Moshe Y. Vardi. Bisimulation Minimization and Symbolic Model Checking. *Form. Methods Syst. Des.*, 21:39–78, July 2002.

[54] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345+, June 1962.

[55] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. X-Trace: A pervasive network tracing framework. In *4th USENIX Symposium on Networked Systems Design & Implementation*, pages 271–284, 2007.

[56] Mark Gabel and Zhendong Su. Javert: Fully Automatic Mining of General Temporal Properties from Dynamic Traces. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 339–349, New York, NY, USA, 2008. ACM.

[57] Mark Gabel and Zhendong Su. Testing mined specifications. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, New York, NY, USA, 2012. ACM.

[58] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. CADP 2010: A toolbox for the Construction and Analysis of Distributed Processes. In Parosh Abdulla and K. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6605 of *Lecture Notes in Computer Science*, chapter 33, pages 372–387. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2011.

[59] Dennis Geels, Gautam Altekar, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Friday: Global Comprehension for Distributed Replay. In *Networked Systems Design and Implementation (NSDI)*, 2007.

[60] Dimitra Giannakopoulou and Jeff Magee. Fluent model checking for event-based systems. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-11, pages 257–266, New York, NY, USA, 2003. ACM.

[61] Mark E. Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.

[62] Alla Goralč'ıková and Václav Koubek. A Reduct-and-Closure Algorithm for Graphs. In Jir'ı Becvár, editor, *Mathematical Foundations of Computer Science 1979*, volume 74 of *Lecture Notes in Computer Science*, pages 301–307. Springer Berlin / Heidelberg, 1979.

[63] Graphviz. http://www.graphviz.org/, 2013.

[64] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. R2: An Application-Level Kernel for Record and Replay. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 193–208, Berkeley, CA, USA, 2008. USENIX Association.

[65] Welcome to Apache Hadoop! http://hadoop.apache.org/. Accessed April 6, 2011.

[66] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 291–301, New York, NY, USA, 2002. ACM.

[67] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, June 1987.

[68] David Harel, Hillel Kugler, and Amir Pnueli. Synthesis Revisited: Generating Statechart Models from Scenario-Based Requirements. In Hans J. Kreowski, Ugo Montanari, Fernando Orejas, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *Formal Methods in Software and Systems Modeling*, chapter Synthesis revisited: generating statechart models from scenario-based requirements, pages 309–324. Springer-Verlag, Berlin, Heidelberg, 2005.

[69] John Hatcliff and Matthew Dwyer. Using the Bandera Tool Set to Model-Check Properties of Concurrent Java Software. In KimG Larsen and Mogens Nielsen, editors, *CONCUR 2001 — Concurrency Theory*, volume 2154 of *Lecture Notes in Computer Science*, pages 39–58. Springer Berlin Heidelberg, 2001.

[70] Alexander Heussner, Tristan Gall, and Grégoire Sutre. Extrapolation-Based Path Invariants for Abstraction Refinement of Fifo Systems. In *Proceedings of the 16th International SPIN Workshop on Model Checking Software*, pages 107–124, Berlin, Heidelberg, 2009. Springer-Verlag.

[71] Alexander Heussner, Tristan Le Gall, and Grégoire Sutre. McScM: a general framework for the verification of communicating machines. In *Proceedings of the 18th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'12, pages 478–484, Berlin, Heidelberg, 2012. Springer-Verlag.

[72] Gerard J. Holzmann. The model checker SPIN. In *IEEE Transactions on Software Engineering*, volume 23, pages 279–295, 1997.

[73] John E. Hopcroft. An n log n algorithm for minimizing states in a finite automaton. Technical report, Stanford, CA, USA, 1971.

[74] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation - (2. ed.)*. 2001.

[75] Itu. Message Sequence Charts, 2000.

[76] Guofei Jiang, Haifeng Chen, Cristian Ungureanu, and Kenji Yoshihira. Multi-resolution Abnormal Trace Detection Using Varied-length N-grams and Automata. In *Proceedings of the Second International Conference on Automatic Computing*, pages 111–122, Washington, DC, USA, 2005. IEEE Computer Society.

[77] Guofei Jiang, Haifeng Chen, and Kenji Yoshihira. Efficient and Scalable Algorithms for Inferring Likely Invariants in Distributed Systems. *IEEE Transactions on Knowledge and Data Engineering*, 19(11):1508–1523, November 2007.

[78] D. A. Khotimsky and I. A. Zhuklinets. Hierarchical vector clock: scalable plausible clock for detecting causality in large distributed systems. In *ATM, 1999. ICATM &#039;99. 1999 2nd International Conference on*, pages 156–163. IEEE, 1999.

[79] Eric Koskinen and John Jannotti. BorderPatrol: isolating events for black-box tracing. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pages 191–203, New York, NY, USA, 2008. ACM.

[80] Ivo Krka, Yuriy Brun, George Edwards, and Nenad Medvidovic. Synthesizing partial component-level behavior models from system specifications. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 305–314, New York, NY, USA, 2009. ACM.

[81] Ivo Krka, Yuriy Brun, Daniel Popescu, Joshua Garcia, and Nenad Medvidovic. Using dynamic execution traces and program invariants to enhance behavioral model inference. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 179–182, New York, NY, USA, 2010. ACM.

[82] Sandeep Kumar, Siau-Cheng Khoo, Abhik Roychoudhury, and David Lo. Inferring Class Level Specifications for Distributed Systems. In *ACM/IEEE International Conference on Software Engineering (ICSE) 2012*, 2012.

[83] Leslie Lamport. Time Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, July 1978.

[84] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, November 1989.

[85] Wenchao Li, Lili Dworkin, and Sanjit A. Seshia. Mining assumptions for synthesis. In *Proceedings of the Ninth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, July 2011.

[86] Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. D3S: Debugging Deployed Distributed Systems. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 423–437, Berkeley, CA, USA, 2008. USENIX Association.

[87] David Lo and Siau C. Khoo. QUARK: Empirical assessment of automaton-based specification miners. In *Proceedings of the 13th Working Conference on Reverse Engineering*, pages 51–60, Washington, DC, USA, 2006. IEEE Computer Society.

[88] David Lo and Siau C. Khoo. SMArTIC: towards building an accurate, robust and scalable specification miner. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT '06/FSE-14, pages 265–275, New York, NY, USA, 2006. ACM.

[89] David Lo and Shahar Maoz. Scenario-based and value-based specification mining: better together. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 387–396, New York, NY, USA, 2010. ACM.

[90] David Lo, Leonardo Mariani, and Mauro Pezzè. Automatic steering of behavioral model inference. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 345–354, New York, NY, USA, 2009. ACM.

[91] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. Automatic generation of software behavioral models. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 501–510, New York, NY, USA, 2008. ACM.

[92] Jian G. Lou, Qiang Fu, Yi Wang, and Jiang Li. Mining dependency in distributed systems through unstructured logs analysis. *SIGOPS Oper. Syst. Rev.*, 44(1):91–96, March 2010.

[93] Jian G. Lou, Qiang Fu, Shengqi Yang, Jiang Li, and Bin Wu. Mining Program Workflow from Interleaved Traces. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '10, pages 613–622, New York, NY, USA, 2010. ACM.

[94] Jian G. Lou, Qiang Fu, Shengqi Yang, Ye Xu, and Jiang Li. Mining invariants from console logs for system problem detection. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIX ATC'10, page 24, Berkeley, CA, USA, 2010. USENIX Association.

[95] Leonardo Mariani and Mauro Pezzè. Dynamic detection of COTS component incompatibility. *IEEE Software*, 24(5):76–85, September 2007.

[96] Friedemann Mattern. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms*, pages 215–226, 1989.

[97] Johan Moe and David A. Carr. Using execution trace data to improve distributed systems. *Software: Practice and Experience*, 32(9):889–906, 2002.

[98] Gail C. Murphy, David Notkin, and Kevin J. Sullivan. Software reflexion models: Bridging the gap between design and implementation. *IEEE Trans. Softw. Eng.*, 27(4):364–380, April 2001.

[99] Shashidhar Mysore, Bita Mazloom, Banit Agrawal, and Timothy Sherwood. Understanding and visualizing full systems with data flow tomography. *SIGPLAN Not.*, 43:211–221, March 2008.

[100] Robert Paige and Robert E. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16:973–989, December 1987.

[101] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically Patching Errors in Deployed Software. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, pages 87–102, Big Sky, MT, USA, October 2009.

[102] Gary L. Peterson. An O(n log n) Unidirectional Algorithm for the Circular Extrema Problem. *ACM Trans. Program. Lang. Syst.*, 4(4):758–762, October 1982.

[103] James L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[104] Leonard Pitt and Manfred K. Warmuth. The minimum consistent DFA problem cannot be approximated within any polynomial. *J. ACM*, 40:95–142, January 1993.

[105] Michael Pradel, Philipp Bichsel, and Thomas R. Gross. A framework for the evaluation of specification miners based on finite state machines. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ICSM '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[106] Harald Raffelt, Bernhard Steffen, and Therese Berg. Learnlib: a library for automata learning and experimentation. In *Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, FMICS '05, pages 62–71, New York, NY, USA, 2005. ACM.

[107] Anand V. Raman and Jon D. Patrick. The sk-strings method for inferring PFSA. 1997.

[108] Steven P. Reiss and Manos Renieris. Encoding program executions. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 221–230, Washington, DC, USA, 2001. IEEE Computer Society.

[109] Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. Pip: Detecting the Unexpected in Distributed Systems. In *Proceedings of the 3rd conference on Networked Systems Design & Implementation - Volume 3*, NSDI'06, page 9, Berkeley, CA, USA, 2006. USENIX Association.

[110] Luigi Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *SIGCOMM Comput. Commun. Rev.*, 27(1):31–41, January 1997.

[111] Raja R. Sambasivan, Alice X. Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R. Ganger. Diagnosing Performance Changes by Comparing Request Flows. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[112] Davide Sangiorgi. On the origins of bisimulation and coinduction. *ACM Trans. Program. Lang. Syst.*, 31(4):1–41, May 2009.

[113] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[114] Sigurd Schneider, Ivan Beschastnikh, Slava Chernyak, Michael D. Ernst, and Yuriy Brun. Synoptic: Summarizing System Logs with Refinement. In *Proceedings of the 2010 International Workshop on Managing Systems via Log Analysis and Learning Techniques*, SLAML'10, page 2, Berkeley, CA, USA, 2010. USENIX Association.

[115] Alper Sen and Vijay K. Garg. Detecting Temporal Logic Predicates on the Happened-Before Model. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, IPDPS '02, Washington, DC, USA, 2002. IEEE Computer Society.

[116] Atul Singh, Petros Maniatis, Timothy Roscoe, and Peter Druschel. Using Queries for Distributed Monitoring and Forensics. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, number 4 in EuroSys '06, pages 389–402, New York, NY, USA, 2006. ACM.

[117] Janice M. Stone. A Graphical Representation of Concurrent Processes. *SIGPLAN Not.*, 24:226–235, November 1988.

[118] Jiaqi Tan, Xinghao Pan, Soila Kavulya, Rajeev Gandhi, and Priya Narasimhan. SALSA: Analyzing Logs as State Machines. In *Proceedings of the First USENIX conference on Analysis of system logs*, WASL'08, page 6, Berkeley, CA, USA, 2008. USENIX Association.

[119] Jiaqi Tan, Xinghao Pan, Soila Kavulya, Rajeev Gandhi, and Priya Narasimhan. Mochi: Visual Log-analysis based Tools for Debugging Hadoop. In *Proceedings of the 2009 conference on Hot topics in cloud computing*, HotCloud'09, page 18, Berkeley, CA, USA, 2009. USENIX Association.

[120] Eno Thereska and Gregory R. Ganger. IRONModel: Robust Performance Models in the Wild. *SIGMETRICS Perform. Eval. Rev.*, 36(1):253–264, June 2008.

[121] Sebastian Uchitel, Jeff Kramer, and Jeff Magee. Behaviour Model Elaboration Using Partial Labelled Transition Systems. *SIGSOFT Softw. Eng. Notes*, 28(5):19–27, September 2003.

[122] Sebastian Uchitel, Jeff Kramer, and Jeff Magee. Incremental Elaboration of Scenario-Based Specifications and Behavior Models Using Implied Scenarios. *ACM Trans. Softw. Eng. Methodol.*, 13(1):37–85, January 2004.

[123] Hasan Ural and Hüsnü Yenigün. *Towards Design Recovery from Observations*, volume 3235, chapter 9, pages 133–149. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[124] Risto Vaarandi. A Breadth-First Algorithm for Mining Frequent Patterns from Event Logs. In *Proceedings of the 2004 IFIP International Conference on Intelligence in Communication Systems*, volume 3283, pages 293–308, 2004.

[125] Voldemort. http://project-voldemort.com, 2012.

[126] Neil Walkinshaw and Kirill Bogdanov. Inferring Finite-State models with temporal constraints. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 248–257, Washington, DC, USA, September 2008. IEEE Computer Society.

[127] Jon Whittle and Johann Schumann. Generating statechart designs from scenarios. In *Proceedings of the 22nd international conference on Software engineering*, ICSE '00, pages 314–323, New York, NY, USA, 2000. ACM.

[128] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael Jordan. Experience mining Google's production console logs. In *Proceedings of the 2010 workshop on Managing systems via log analysis and machine learning techniques*, SLAML'10, page 5, Berkeley, CA, USA, 2010. USENIX Association.

[129] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. Detecting Large-Scale System Problems by Mining Console Logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 117–132, New York, NY, USA, 2009. ACM.

[130] Maysam Yabandeh, Abhishek Anand, Marco Canini, and Dejan Kostić. Finding Almost-Invariants in distributed systems. In *Proceedings of the 30th IEEE Symposium on Reliable Distributed Systems*, October 2011.

[131] Maysam Yabandeh, Nikola Knezevic, Dejan Kostić, and Viktor Kuncak. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 229–244, Berkeley, CA, USA, 2009. USENIX Association.

[132] Jinlin Yang and David Evans. Automatically Inferring Temporal Properties for Program Evolution. In *Proceedings of the 15th International Symposium on Software Reliability Engineering*, ISSRE '04, pages 340–351, Washington, DC, USA, November 2004. IEEE Computer Society.

[133] Jinlin Yang and David Evans. Dynamically Inferring Temporal Properties. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '04, pages 23–28, New York, NY, USA, 2004. ACM.

[134] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Perracotta: Mining Temporal API Rules from Imperfect Traces. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 282–291, New York, NY, USA, 2006. ACM.

[135] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 213–228, Berkeley, CA, USA, 2009. USENIX Association.

[136] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. SherLog: Error Diagnosis by Connecting Clues from Run-Time Logs. *SIGPLAN Not.*, 45:143–154, March 2010.

[137] Kenny Q. Zhu, Kathleen Fisher, and David Walker. Incremental Learning of System Log Formats. *SIGOPS Oper. Syst. Rev.*, 44:85–90, March 2010.

[138] Tewfik Ziadi, Marcos A. da Silva, Lom M. Hillah, and Mikal Ziane. A Fully Dynamic Approach to the Reverse Engineering of UML Sequence Diagrams. In *Proceedings of the 2011 16th IEEE International Conference on Engineering of Complex Computer Systems*, ICECCS '11, pages 107–116, Washington, DC, USA, 2011. IEEE Computer Society.