UNIVERSITY OF CALIFORNIA,
IRVINE

Specification and Runtime Verification of Distributed Multiprocessor Systems: Languages, Tools and Architectures

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Electrical and Computer Engineering

by

Ahmed Mohamed Ahmed Mohamed Nassar

Dissertation Committee:
Professor Fadi J. Kurdahi, Chair
Professor Rainer Doemer
Professor Ahmed Eltawil

2016

DEDICATION

To The Memory of My Father,
To The Sincerest Love of My Mother,
To My Wife and My Son

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

You think acknowledgments are boring and no one reads them? It is because they are modest expressions of gratitude towards many people dedicating their lives to whom they love, or going out of their way to help someone they may not even know, or feeling excited to share lessons they learned the hard way. My gratitude to my parents, who literally breathe life into every line of this dissertation, cannot be expressed in words. Special thanks to my advisor, Professor Fadi J. Kurdahi, not only for all his guidance and support, but also for nurturing and believing in the worth of this research especially during the many difficult times we have been through.

Thank you my friends and colleagues for making all those years at UC Irvine a matchless learning experience. I would like to thank authors of all books and papers I had the chance to read for the maturity and sophistication I accumulated along the way.

Finally, this work would not have been possible without the constant support of my loving wife and the overwhelming joy of playing with my beloved four-years old son.

CURRICULUM VITAE

Ahmed Mohamed Ahmed Mohamed Nassar

EDUCATION

Doctor of Philosophy in Electrical and Computer Engineering **2016**
University of California *Irvine, CA*

Master of Science in Electrical Engineering **2009**
Cairo University *Cairo, Egypt*

Bachelor of Science in Electrical Engineering **2002**
Alexandria University *Alexandria, Egypt*

INDUSTRY EXPERIENCE

Engineering Intern **2011–2013**
Qualcomm, Inc. *San Diego, CA*

VLSI Design Engineer **2008–2010**
Newport Media, Inc. *Cairo, Egypt*

FPGA Design Engineer **2006–2008**
QuickTel *Cairo, Egypt*

R&D Military Engineer **2003–2005**
Egyptian Armed Forces *Cairo, Egypt*

RESEARCH EXPERIENCE

Graduate Research Assistant **2010–2016**
University of California *Irvine, California*

TEACHING EXPERIENCE

Teaching Assistant **2011–2016**
University of California *Irvine, CA*

EECS40 Object-oriented systems and programming. Spring 2015
EECS229 Low-Power SoC Design (Graduate level). Winters 2015-2016
EECS221 Advanced SoC Design (Graduate level). Winters 2013-2014
EECS181 Senior Design Projects. Summer and Fall 2014
CSE112 Electronic Devices and Circuits. Falls 2012-2013
EECS113 Processor Hardware/Software Interface. Springs 2012, 2016
CSE151 Introduction to Digital VLSI Design. Fall 2011

REFEREED CONFERENCE PUBLICATIONS

Ahmed Nassar, Fadi J. Kurdahi, "Lattice-Based Boolean Diagrams: Canonical, Order-Independent Graphical Representations of Boolean Functions" **Jan 2016**
In Proceedings of Asia and South Pacific Design Automation Conference (ASP-DAC)
[Best Paper Award]

Ahmed Nassar, Fadi J. Kurdahi and Salam Zantout, "Topaz: Mining High-Level Safety Properties from Logic Simulation Traces" **Mar 2016**
In Proceedings of Design, Automation and Test in Europe Conference (DATE)

Ahmed Nassar, Fadi J. Kurdahi, Wael Elsharkasy, "NUVA: Architectural Support for Runtime Verification of Parametric Specifications over Multicores" **Oct 2015**
In proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)

Ahmed Nassar, Fadi J. Kurdahi, "Architectural support for runtime verification on ccNUMA multiprocessors" **Apr 2013**
Proceedings of the international design and test symposium (IDT)

Ahmed Nassar, Fadi J. Kurdahi "ParaMiner: Interactive Parametric Specification Mining for Runtime Verification" **June 2017**
To be submitted to Programming Language Design and Implementation Conference (PLDI)

Ahmed Nassar, Fadi J. Kurdahi "Connecting The Dots: Detection of Sparse Anomalies Using Self-Replicating Automata" **May 2017**
To be submitted to IEEE Symposium on Security and Privacy (S&P)

REFEREED JOURNAL PUBLICATIONS

Ahmed Nassar, A. Emira, A.N. Mohieldin, and A. Hussien, "Multichannel Clock and Data Recovery: A Synchronous Approach" **May 2010**
IEEE Transactions on Circuits and Systems II: Express Briefs

Ahmed Nassar, Fadi J. Kurdahi, "Lattice-Based Function Graphs" **Oct 2016**
To be submitted to IEEE Transactions on Computer-Aided Design (TCAD)

Ahmed Nassar, Fadi J. Kurdahi, "SuperNUVA: Runtime Verification for the Masses" **Dec 2016**
To be submitted to ACM Transactions on Embedded Computing Systems (TECS)

Ahmed Nassar, Fadi J. Kurdahi, "TransMon: Specification and Runtime Verification of Transaction-Level Models" **Mar 2017**
To be submitted to IEEE Transactions on Computer-Aided Design (TCAD)

Ahmed Nassar, Fadi J. Kurdahi, "Trace Origami: Mining Abstract Specifications from Logic Simulation Traces" **Mar 2017**
To be submitted to IEEE Transactions on Computer-Aided Design (TCAD)

SOFTWARE

NUVA, LBFGs, Topaz, ParaMiner, and others `https://github.com/anassar`
Tools, simulation models, Raspberry Pi projects in C++, Java, Perl and Python.

ABSTRACT OF THE DISSERTATION

Specification and Runtime Verification of Distributed Multiprocessor Systems: Languages, Tools and Architectures

By

Ahmed Mohamed Ahmed Mohamed Nassar

Doctor of Philosophy in Electrical and Computer Engineering

University of California, Irvine, 2016

Professor Fadi J. Kurdahi, Chair

Post-Deployment runtime verification (RV) has recently emerged as a complementary technology to extend coverage of conventional software verification and testing methods. This thesis is an attempt to tackle three major barriers that need to be surmounted before RV technologies become in widespread use:

- **Barrier-1**: *Lack of an expressive, yet efficiently monitorable, specification language.* Distributed software behavior is projected onto an observation interface consisting of data-carrying (or parameterized) events, such as Linux system calls including argument values, and self-replicating deterministic finite automata (SR-DFAs) are introduced for RV purposes as well as anomaly-based intrusion detection in embedded and general-purpose software systems based on these parametric traces.
- **Barrier-2**: *The substantial performance and power overhead of pure software RV frameworks.* NUVA, which stands for *nonuniform verification architecture*, a distributed automata-based RV architecture for software specifications in the form of SR-DFAs. NUVA has been implemented over a cache-coherent nonuniform-memory-access (ccNUMA) multiprocessor and can be deployed on the FPGA fabric that will reside on all next-generation processor chips. The core of NUVA is a coherent dis-

tributed automata transactional memory (ATM) that efficiently maintains states of a *dynamic* population of automata checkers organized into a rooted dynamic directed acyclic graph (DAG) concurrently shared among all processor nodes.

- **Barrier-3**: *Formal specifications are hard to formulate and maintain for evolving complex embedded and general-purpose software systems.* Therefore, specification mining has long ago been envisioned to play a key role in software verification, modification and documentation. However, in order to scale beyond simple, library/API-level properties having short temporal spans, specification mining tools need to support more expressive specification languages that can capture complex, application-level properties. This thesis introduces a *bio-inspired* complete specification mining methodology for SR-DFAs using an iterative and interactive mining tool, called ParaMiner. ParaMiner relies on novel mining algorithms invoking multiple-sequence alignment (MSA) techniques to enable learning specifications from temporal slices of software behavior while overcoming the initial-state uncertainty problem.

SR-DFAs and ParaMiner have been leveraged in a new specification-based intrusion detection (ID) framework that protects distributed, reactive computing systems against cyberattacks having very sparse signatures, arbitrarily long time spans and wide attack fronts. Such attacks lie outside the scope of conventional anomaly-based ID methods which typically work with short event windows and ignore manipulated data objects, such as files and sockets. We demonstrate the effectiveness of the constructed SR-DFAs at classifying as well as resolving subtle behaviors typical of cyberattacks with varying evasion parameter values.

Chapter 1

Introduction

1.1 Motivation

Prominence of Runtime Verification. Cyber-physical systems (CPS) harbor strongly interacting physical and computational processes that must usually operate within stringent safety and security constraints. Malfunction due to hardware or software failures, or compromise by an adversary (including insiders) can be unacceptable when lives, large investments (e.g., critical infrastructure) or valuable assets (e.g., environment and natural resources) are at stake. Due to their immense complexity and tight coupling to stochastic (and intrinsically concurrent/distributed) environments, complete verification coverage meant to find all bugs and vulnerabilities of CPS designs is unattainable with conventional testing procedures. Moreover, when a software failure occurs, a difficult blame (or credit) assignment problem arises to correlate the erroneous behavior with its root cause. This is further complicated by the phenomenal proliferation of multiprocessor systems which brought to the forefront elusive *concurrency bugs* that can insidiously build up latent momentum over time until they manifest themselves catastrophically and inexplicably. Conventional debugging

machinery (e.g., breakpoints, watchpoints, etc.), which cannot properly handle *distributed state* [124], fall short of the needs of concurrent application developers. Worse yet, with the nondeterminism inherent in the execution of concurrent code and due to the so-called *probe effect*, faulty executions cannot be reliably reproduced. Even abstract model checking [89] may trade off precision of the analysis for efficiency. Hence, formal reasoning with abstract models of concurrent systems (e.g., SPIN [94]) is in jeopardy of overlooking subtle behaviors that will manifest only in a concrete, detailed model of the system. All this draws attention to the power of *runtime verification* (RV) [6, 22, 38, 48, 112] where systems are continuously monitored and verdicts (or *anomaly scores*) about correctness and/or integrity of their behaviors are issued accordingly at appropriate instants of time. These verdicts indicate the minimal bad prefixes of violating traces and, hence, are *precise* (i.e., localized exactly at fault locations in program executions) and can be easily used to trace failures back to their root causes. These verdicts can also trigger reactive (corrective or preventive) procedures before serious damage is inflicted upon the system.

The interplay between pre-deployment verification (e.g., testing and formal verification) and post-deployment RV is important to delineate. If RV is coupled with recovery mechanisms (which are typically slower than normal execution [124]), then performance of RV starts to depend heavily on the average error rate. Pre-deployment verification helps to reduce the average error rate and, hence, boost post-deployment RV performance.

Dark Silicon. With the emergence of *dark silicon* phenomena [68, 166] in power-constrained systems-on-chip (SoCs) and/or limited-parallelism systems, the trend of increasing the number of identical general-purpose cores on a chip (called *multicore scaling*) became untenable, since they cannot all be powered up at the same time anyway.[1] As a result, the case for a more energy-efficient *computing zoo* of asymmetric, heterogeneous and specialized cores

[1] Unless special arrangements are taken, such as near-threshold voltage (NTV) operation, larger caches, or turbo boosting. However, all these techniques come with many thorny issues, e.g., susceptibility to process variations, assumptions about application level of parallelism, etc.

became stronger. The impetus and room for that specialization on the chip real-estate, dubbed the *dark-silicon gap*, is even growing with every new technology generation. CPS dependability and security has long ago been identified as a major beneficiary of growing specialization on SoCs.

Goals. Motivated by the dependability and accountability benefits brought about by RV, our goal is to bring post-deployment RV into widespread use by many computing systems (general-purpose, data centers, cyber-physical, etc.) through *built-in architectural supporting mechanisms*. The first, and most difficult, step is to find a specification formalism that can serve as an ideal substrate for hardware-assisted RV and possess the following characteristics:

- *Durability*: It must be able to stand the test of time, to justify investment of hardware support by chip vendors and learning time by developers. Due to their wide scope and vast expressive power, we bet on parametric specifications [6, 20, 21, 39, 81, 83, 87, 164] to survive multiple technology generations.
- *Efficiency*: It must be efficiently monitorable in terms of performance overhead, on-chip area and consumed power.
- *Usability*: Specifications are hard to formulate and maintain for complex systems. Specification languages must not make a hard job even harder. That is why, in this thesis, we work with parametric FSMs since they are intuitive and mining algorithms can be developed for them [10, 110]. As shown in Figure 1.1, we envision specification mining to play a key role in RV. Many specification languages in literature do not only have complex notation, but also complex (and, in many cases, counter-intuitive) *semantics*. In this thesis, we adopt what we call *ensemble semantics* of parametric specifications which we think are more intuitive to reason about.

Parametric events are envisaged to serve as the primary abstraction of program executions. Events can be explicit in the programming model. For example, event-driven programming is one of the most widely used programming models for CPS. Alternatively, events

can be implicitly associated with significant changes of program state, or extracted from raw unstructured data streams (e.g., sensory data, text, video, etc.) or exchanged messages. Many anomaly-based intrusion detection (ID) systems [16] operate on system-call traces [92]. In [169], the author laments the absence of syscall arguments from ID systems, which grants attackers the much needed freedom in crafting attacks while avoiding detection. That inspired the use of syscall arguments to increase immunity of ID systems against a wide array of evasion techniques [123]. Representing execution traces with data-carrying events [87], as opposed to event names only, coupled with an expressive specification language of SR-DFAs (capable of dynamic variables binding) enables use of *contextual information* and yields a *spatio-temporal characterization of correct behavior*, since objects, users and threads identifiers are now part of temporal specifications. This promises an increase in model precision which translates to more discriminative (or explanatory) power. Monitoring of parametric and first-order specifications recently received ample attention in literature [6, 20, 21, 39, 81, 83, 87, 164]. Proposed systems have (often subtly different) custom-tailored logics spanning a wide spectrum of expressive power [20] and having different monitoring efficiency levels. Very expressive logics tend to have an undecidable RV problem [21]. Moreover, most proposals lack a naturally distributed structure or representation that scales RV to arbitrarily large multiprocessor systems, which seems to remain as an open problem (to the best of our knowledge). Therefore, we set out to investigate a highly expressive and efficiently monitorable parametric specification formalism that can be supported by hardware. The RV hardware architecture for parametric specifications must also satisfy the following requirements: (I) It must be intrinsically distributed in order to be scalable. (II) It must minimize conflicts among concurrent RV transactions over shared RV data. (III) It must be loosely coupled and minimally invasive to current CPU architectures.

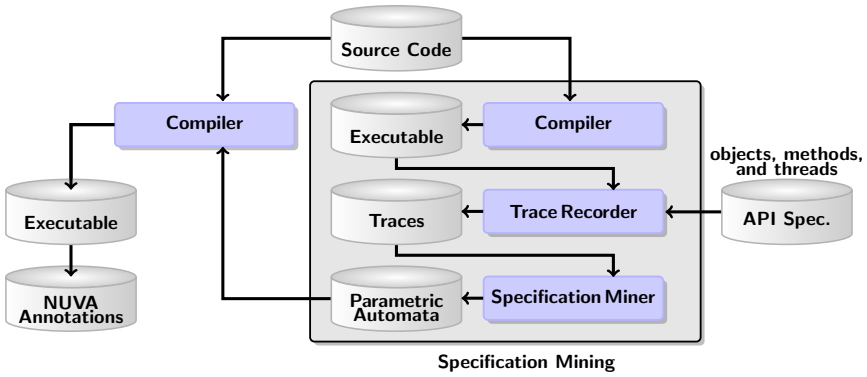


Figure 1.1: Specification-based runtime verification

1.2 Background and Related Work

1.2.1 Dimensions of Runtime Verification

Dependability of computing systems hinges equally on hardware and software, both of which are prone to failures, design flaws as well as attacks and many tools have been devised to verify various aspects *at runtime*. In Table 1.1, we list a sample of all published tools deemed sufficient to reveal the most significant dimensions of the RV tool design space and the categories such tools fall in and to help position NUVA with respect to prior works.[2] According to [112], *executions* (defined as finite traces or prefixes of possibly infinite runs) of a *program under verification* (PUV) are the primary objects in RV and are checked for correctness by monitors which yield a *verdict* whether the trace thus far observed satisfies, violates or not yet decides a given property. The RV monitor shown in Figure 1.2, therefore, adopts a three-valued truth domain for property satisfaction along finite traces, namely $\{\top, \bot, ?\}$ or $\{\mathsf{true}, \mathsf{false}, \mathsf{inconclusive}\}$, respectively. It can also assign anomaly labels (or numerical scores) to observed events while the verdict on correctness of the monitored property is still out. This is useful in intrusion detection applications. Monitoring can be classified as *online*

[2]Dynamic analysis tools intended for pre-deployment testing only are excluded. RV tools that make use of non-standard hardware features (e.g., LogTM used by TxMon [36]) are assumed to be implemented in hardware. A RV tool is labeled as "multicore" if it is implemented with multicore or multiprocessor systems in mind. Tools that use thread-level techniques (such as TLS) do not qualify as multicore-aware.

(examining executions incrementally at runtime) or *offline* (examining recorded executions). In [69], monitors are further classified into *inline* (inserted within the monitored program) or *outline* (running in a thread or process different from the monitored program). The monitors designed in this thesis are intended primarily to make online monitoring more efficient. Moreover, only outline monitoring is supported by virtue of the parallelism inherent in hardware. The checks performed by a monitor can be classified according to [48] as either *precise*, that indicate actual errors in the observed execution trace, or *predictive*, indicating errors that have not occurred in the observed execution but could possibly occur in other executions of the program. In this thesis, runs are considered as sequences of *instantaneous parameterized* (or *data-carrying* [87]) *events* emitted by concurrent threads or processes of a running program. Those events are represented by first-order logic (FOL) predicates and can be associated, for example, with calls of a given API (e.g., system calls, library calls, etc.) or any more elementary operations on primitive values. Moreover, NUVA discovers only bugs that actually occur during program execution, rather than potential bugs lurking in the code, as in the lockset algorithm [156].

1.2.2 Formalism Crisis: Rise of Parametric Specifications

In most specification-based verification applications, *precise specifications* of program behavior are highly desirable. By *precise* we mean a tight characterization of program behavior

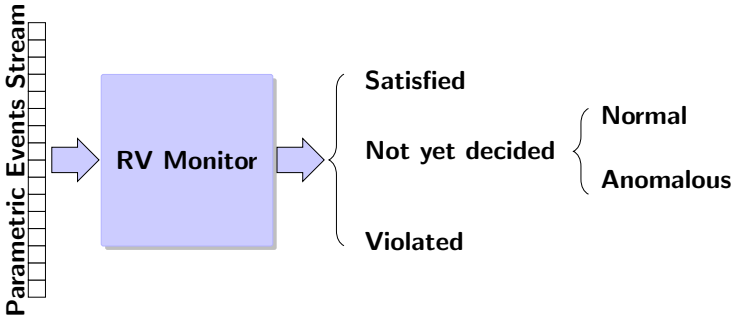


Figure 1.2: Runtime verification of parametric event traces.

Tool		Verifies		Impl.		Multicore	Real-time	Sampling	Faults	Specification	Errors		Action		Predictive
		HW	SW	HW	SW						FP[a]	FN[b]	Report	BER[c]	
NUVA		✖	✔	✔	✖	✔	✖	✖	Functional bugs	Parametric FSMs	✖	✖	✔	✖	✖
P2V	[120]	✖	✔	✔	✖	✖	✖	✖	Functional bugs	sPSL properties	✖	✖	✔	✖	✖
iWatcher	[178]	✖	✔	✔	✖	✖	✖	✖	Functional bugs	Procedural checker code	✖	✖	✔	✔	✖
TxMon	[36]	✖	✔	✔	✖	✔	✖	✖	Data structure misuse	Checker callbacks	✖	✖	✔	✖	✖
ReEnact	[143]	✖	✔	✔	✖	✖	✖	✖	Data races	None	✖	✖	✔	✖	✖
SecMon	[13]	✖	✔	✔	✖	✖	✖	✖	Security vulnerabilities	None	✖	✖	✔	✖	✖
HARD	[177]	✖	✔	✔	✖	✔	✖	✖	Data races	None	✔	✔	✔	✖	✔
AVIO-H	[121]	✖	✔	✔	✖	✔	✖	✖	Atomicity violations	None	✔	✔	✔	✖	✖
AVIO-S	[121]	✖	✔	✖	✔	✔	✖	✖	Atomicity violations	None	✔	✔	✔	✖	✖
SVD	[174]	✖	✔	✖	✔	✔	✖	✖	Serializability violations	None	✔	✔	✔	✔	✖
StackGhost	[74]	✖	✔	✖	✔	✖	✖	✖	Stack overflow	None	✖	✖	✔	✖	✖
CORD	[142]	✖	✔	✖	✔	✔	✖	✖	Data races	None	✖	✖	✔	✖	✖
CoPilot	[141]	✖	✔	✖	✔	✖	✔	✔	Violations in value streams	Stream eqns. + ptLTL	✔	✖	✔	✖	✖
TraceMatches	[6]	✖	✔	✖	✔	✖	✖	✖	Functional bugs	Parametric regular patterns	✖	✖	✔	✖	✖
JavaMOP	[38]	✖	✔	✖	✔	✖	✖	✖	Functional bugs	Parametric properties	✖	✖	✔	✖	✖
QEAs	[20]	✖	✔	✖	✔	✖	✖	✖	Functional bugs	Quantified event automata	✖	✖	✔	✖	✖
SAs	[21]	✖	✔	✖	✔	✖	✖	✖	Functional bugs	FOLTL properties	✖	✖	✔	✖	✖
DAs	[87]	✖	✔	✖	✔	✖	✖	✖	Functional bugs	Data automata	✖	✖	✔	✖	✖
Eraser	[156]	✖	✔	✖	✔	✔	✖	✖	Data races	None	✔	✖	✔	✖	✔
DVMC	[130]	✔	✖	✔	✖	✔	✖	✖	Memory consistency (MC)	MC model	✖	✖	✔	✖	✖
Phoenix	[155]	✔	✖	✔	✖	✖	✖	✖	Logic design defects	Defect bit-level patterns	✖	✖	✔	✖	✖
FRCL	[170]	✔	✖	✔	✖	✖	✖	✖	Logic design defects	Defect bit-level patterns	✖	✖	✔	✖	✖
ReVive	[144]	✔	✖	✔	✖	✔	✖	✖	HW failures	Availability	✖	✖	✔	✔	✖
SafetyNet	[162]	✔	✖	✔	✖	✔	✖	✖	HW failures	Availability	✖	✖	✔	✔	✖
DIVA	[15]	✔	✖	✔	✖	✖	✖	✖	HW transient faults	Availability	✖	✖	✔	✖	✖
Argus	[129]	✔	✖	✔	✖	✖	✖	✖	HW transient faults	Availability	✖	✔	✔	✔	✖
SWIFT	[147]	✔	✖	✖	✔	✖	✖	✖	HW transient faults	Availability	✖	✖	✔	✖	✖
RSUH	[146]	✔	✖	✖	✔	✖	✖	✖	HW transient faults	Availability	✖	✖	✔	✖	✖
iSWAT	[151]	✔	✖	✖	✔	✔	✖	✖	HW permanent faults	Availability	✔	✖	✔	✔	✖

[a]: FP=False positives.
[b]: FN=False negatives.
[c]: BER=Backward Error Recovery.

Table 1.1: Capabilities of runtime verification proposals

that can be used to accurately detect anomalies while minimizing the likelihood of missing subtle, but significant, deviations from correct behavior. More precise specifications usually require more expressive specification languages that can describe more complex behaviors. This increased precision may come at the price of reduced usability, undecidability or harder decision procedures.

Many RV frameworks [112] followed the conventional wisdom of model checking [19] and revolved only around propositional temporal logic (PTL) and some RV algorithms even

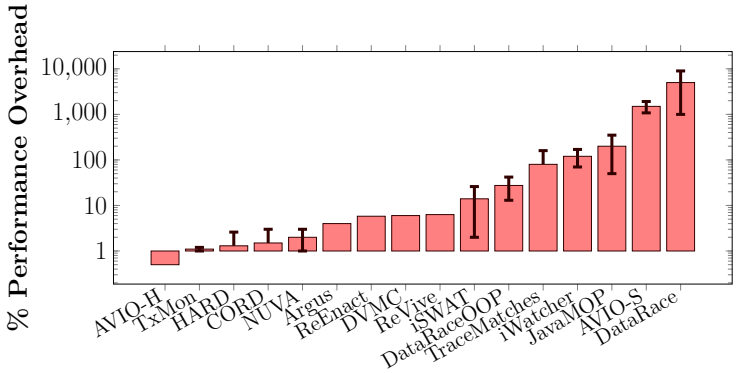


Figure 1.3: Percentage performance overhead for NUVA compared to selected RV tools and architectures. Logarithmic scale is used.

made use of the established theory of ω-automata and ω-regular languages underlying PTL to construct RV monitors [48]. PTL proved to be adequate for the purposes of verifying digital circuit designs of moderate size. However, with everything encoded into propositions, PTL soon becomes impractical for software verification, especially in view of the dynamic nature and complexity of software objects. PTL collapses the programming model primitives (e.g., threads and objects) so that the program state is abstracted into a set of atomic propositions that hold in that state [48]. This spurred a recent interest in the use of first-order linear temporal logic (FOLTL) [105] and parametric specification languages [6, 21, 110, 122] to write concise specifications at a level higher than individual memory accesses. First-order logic (FOL), by virtue of its use of sorts, predicates, functions and quantification, offers more abstract representations closer to the language of the program domain. Programmers can specify the granularity of data accesses (e.g., using an *object* abstraction) and the events of interest to monitor and possibly record for subsequent replay.

The *tracematches* framework [6] enables programmers to invoke code based on matches between regular patterns and event traces. Free variables are allowed in matching patterns and, hence, event matching depends on the values of these free variables. As pointed out in [20], tracematches, like JavaMOP [38], suffers from the limitation that each event name is associated with a unique list of variables. This thesis proposes a distributed representation of what tracematches calls *binding constraints*. This yields a RV solution that will be shown to scale well to large concurrent and distributed systems. Binding constraints will also arise here, but as compile-time constructs, when techniques are investigated to reduce the conflict rate between concurrent RV transactions in a *distributed automata transactional memory* (ATM). A more precise notion of multiple matches is given in terms of *self-replicating automata* and partial variable bindings are allowed and matching subtraces are allowed to overlap (not just interleave). Moreover, a logical framework is used instead of regular expressions. Quantified event automata (QEAs) [20] are strictly more expressive than JavaMOP [38], since QEAs allow event names to accept multiple distinct variable lists and allow quantification. How-

ever, the domain of each quantified variable is derived from the observed trace. Transition relations of QEAs allow overriding previous bindings during the course of RV. In this thesis, overriding does not happen; only self-replication and *bindings augmentation*. We believe that this furnishes more intuitive semantics than QEAs. Guard conditions used by QEAs can be expressed in FOLe. For example, $\mathsf{start(x)}^{\mathsf{x}\neq\mathsf{y}}$ can be expressed as $\mathsf{start(x)} \wedge \neg\mathsf{start(y)}$ in FOLe. In [87], data automata (DAs) are introduced to monitor sequences of data-carrying events. DAs are an explicit approach (where states are named and have formal parameters bound when an event with the same name as the state occurs) different from automata-based RV (where states summarize, but do not completely characterize, an arbitrarily long history of past events). In [83], interactions between client applications and Web services are ensured to proceed as intended given a well-defined and enforceable *interface contract* in the form of LTL-FO+ formulas. The logic LTL-FO+ extends LTL with first-order quantification over XML message elements. LTL-FO+ runtime monitoring uses an on-the-fly algorithm to only create the states corresponding to values that have actually been observed. The number of states grows logarithmically with trace length. Although $\log(x)$ is a very slowly increasing function, it is not clear from [83] whether logarithmic growth is a general phenomenon or only peculiar to the LTL-FO+ properties examined in that thesis. In [21], a logic called LTL^{FO} is used as a specification language for automata-based RV and two important conditions for proper monitors are elucidated, namely *trace-length independence* (i.e., consistency of RV efficiency as observations keep unfolding) and *monotonicity* (i.e., persistence of satisfied/violated verdicts once issued). Monotonicity is the main requirement distinguishing RV (i.e., the prefix problem) from the word problem (i.e., satisfaction over finite words). While the word problem for LTL^{FO} is decidable, the prefix problem for LTL^{FO} is not. Hence, the constructed (spawning) automata checkers are sound, but necessarily incomplete. In [21], a pair $(p, \mathbf{d})$ of predicate p and a tuple of values $\mathbf{d}$ is called an *action*, whereas an *event* is a finite set of actions. In this thesis, we basically restrict the number of actions per event to one. Moreover, we preclude the use of rigid functions and predicates altogether. That comes

at the price of expressiveness, but ensures decidability of the RV problem (i.e., monitoring completeness). Quantification is restricted in [21] to actual values observed in traces (called *domain independence*), which is a severe restriction the limits expressiveness of LTL^{FO} in important ways, but is necessitated by allowing rigid functions and predicates (which would need to be evaluated for all possible tuples of domain values). Spawning automata (SAs) [21] may spawn a positive Boolean combination of child SAs, because *spawning is intended to reflect the syntactic structure of* LTL^{FO} *formulas and is not triggered by dynamic variable binding*. In this thesis, semantics of self-replicating automata are different from spawning automata, and self-replication does not involve Boolean combinations. Rather, replicas are organized into a genealogical DAG only to make discovery of recombination opportunities more efficient.

1.2.3 Performance Crisis: Architectural Support to the Rescue

Most of the prior work on RV of parametric and FOLTL specifications was focused on inline (intrusive) monitoring frameworks implemented in software and usually relied on aspect-oriented programming (AOP) facilities (e.g., AspectJ as in [165, 38, 6]) or dynamic binary instrumentation or rewriting [17, 156] to automatically weave the checker code into the PUV. However, an unaffordable performance and power penalty is usually incurred. In Figure 1.3, performance overhead of RV is peaked by software tools such as DataRace [17], JavaMop [38] and TraceMatches [28]. This evokes architectural support for RV to mitigate these costs and enable *post-deployment RV*. A reasonable maximum overhead for RV architectures is 5%, as recommended in [124]. The probe effect becomes less of a concern if the deployed program execution can be constantly monitored with such reasonably low performance and power penalties so that most bugs eventually show up over practically long executions. Specifications can be compiled with (and unobtrusively bound to) the PUV (and even shipped with it to the field) and can be disabled without any performance penalty whatsoever.

Architectural support for RV has been gaining more interest both in industry and academia. However, many of the attempted approaches to architectural support for RV depend on automatic inference of functional intent from the program execution [178, 143, 177, 121]. Being limited to specific synchronization primitives (e.g., locks), these approaches are neither general nor extensible, and take a considerable toll on the design silicon area and power consumption. Explicit description of the functional intent enables the RV hardware to be more goal-directed rather than striving to automatically infer the implicit intent of the programmer. This also: (1) reduces the number of false positives usually resulting from inferences inconsistent with the implicit functional intent, (2) reduces the RV hardware cost, and (3) abstracts away any dependence of the hardware mechanisms on the synchronization primitives used to implement the programmer's intent. Although the above inference-based approach affords a significant automation advantage over a specification-based approach (where a designer needs to manually write specifications as shown in Figure 1.1), we bet on two technologies that can aid programmers automatically discover and write elaborate specifications: (1) Specification patterns [60], and (2) specification mining techniques [10].

An exception to the inference-based architectural support is [120] where a simple subset (called sPSL) of the standard *property specification language* (PSL) is extended to the verification of C programs. PSL assertions about properties of a program are compiled into hardware checkers that can be loaded into a dynamically extensible processor to be executed concurrently and nonintrusively with the program. However, sPSL assertions are static compile-time entities, rather than being instantiable at run-time according to program execution. Moreover, the claim of zero monitoring overhead results from the tight coupling between the monitor and the processor pipeline. This premise is easily invalidated even on a single-chip multiprocessor (CMP) where monitoring must introduce CPU stalls to enforce a globally consistent order of events emitted by distributed processor nodes (more on that later). Finally, the atomic propositions are restricted to Boolean-typed C expressions, which limits the scope of application of this scheme. In [178], iWatcher associates program-specified

monitoring functions with particular memory locations. When any such location is accessed, the monitoring function is automatically invoked. iWatcher can be used to detect buffer overflow, accessing freed locations, memory leaks, stack smashing and value-invariant violations. In [124], the major challenge in RV over multiprocessors is identified to be observing and managing *distributed state*. Problems of distributed state management, such as consistency and communications, are addressed in this thesis with the help of an *automata transactional memory* (ATM) that takes advantage of all characteristics of the RV problem of FOLe-based parametric specifications to maximize efficiency.

1.2.4 Specification Mining

Automated software verification and testing techniques decide correctness of a design against an explicit or implicit notion of a formal specification that describes *what* a computing system does or does not do (in the form of *properties*, assertions, etc.) independently of *how* the system will be implemented, and in a formal language with well-defined semantics. A good specification [107] must be complete (i.e., cover all aspects of system behavior), internally consistent (i.e., free from contradictions and, hence, satisfiable or implementable) and unambiguous (i.e., uniquely determining design intent without unduly restricting implementations). Unfortunately, formal specifications are not frequently used mainly because of their high development and maintenance cost and complexity, since they require substantial expertise in formal specification languages and their decision procedures, as well as abstraction techniques. Formal specification languages usually present different paradigms (declarative, logical, etc.) than the more familiar *imperative* development languages. Most programmers could understand and appreciate the meaning of a *given* specification, but only a few can compose or come up with specifications from scratch.[3] This difficulty hin-

[3]This is reminiscent of the widely held belief that P≠NP in complexity theory, where many problems in the NP class are hard to solve but their solutions can be efficiently verified. This correspondence is apt, since the automata learning problem is indeed NP-hard [79].

ders formal specifications from coping with agile, fast-paced development environments that foster rapidly evolving systems. Additionally, specifications of *existing* software systems are needed to verify *new* systems built on top of them [176]. Software maintenance cost ranges from 50% to 90% of the total cost [117, Chapter 1] mainly due to difficulty of understanding existing code. In that regard, a distinction can be made [10] between the (typically simple) language-specific or library/API-specific properties (e.g., properties of standard library classes) whose development cost is amortized over a large number of projects, and *program-specific* properties relevant only to one particular design. Unless the prohibitive specification development cost is reduced, program-specific properties will remain unjustified except for costly safety-critical systems.

A key insight is that specification of a software module can be implicit or hidden in how it is being used by, and reacting to, high-quality client code [176]. Therefore, *specification mining* [10, 117] emerged as an automated technique used to discover/infer/learn formal specifications of programs/systems from examples or samples of their executions, source code, change logs or any associated artifact. Inferred properties can take many forms [117, Chapter 1], such as value invariants [66], FSMs [5, 8, 10, 43, 47, 88, 116], patterns and temporal properties [63, 64, 115, 118], or sequence diagrams [31]. A specification mining tool [10, 110, 117] helps programmers understand programs and iteratively develop specifications for them, where the behavior of a program is *projected* onto an observation interface in order to reduce specification complexity (at the cost of precision) by isolating or decoupling different aspects of a system. Specification mining is premised on the availability of fairly high-quality (but not necessarily bug-free) software that can be exercised to reveal the most relevant aspects of behavior *without triggering bugs*. Another premise is that observed behavior of a program exhibits distinctive or characteristic statistical regularities that can be expressed in a concise formal specification (e.g., state machines [10]). Inferred specifications can then be examined, refined, abstracted or corrected by designers and verification specialists. Once validated, these specifications will drive verification, regression testing, or serve as invariants to be

preserved as the design evolves, or can be published as part of documentation. Specification mining helps to alleviate the specification completeness problem by discovering specifications a system designer never thought about. Traces can be obtained from executed test cases or symbolic execution [117, Chapter 1]. Due to the wide scope and vast expressive power afforded by parametric and first-order logic (FOL)-based specifications [6, 10, 20, 21, 39, 117, 81, 83, 87, 110, 136, 164], we present a tool, called *ParaMiner*, that discovers software properties of arbitrary, tunable complexity and precision from large execution traces. The target specification formalism of ParaMiner is self-replicating deterministic finite automata (SR-DFAs) [136] that detect violations of automatically learned program properties *at runtime*.

1.3 Contributions

This thesis makes the following contributions:

- Specification and RV of distributed programs using the highly expressive specification language of finite automata based on a *first-order logic of events* (FOLe).
- A representation of these automata in terms of a hypothetical countably infinite ensemble of automaton replicas (*checkers*) as well as a novel finite, dynamic graph representation of that ensemble in terms of *self-replicating automata*. It is through the medium of dynamic checker population that the behavior of the entire system is verified.
- A scalable distributed RV architecture, NUVA, that enables low-overhead RV for parametric specifications over distributed multiprocessor systems.
- A novel specification mining, and language learning, technique that is the first (to the best of our knowledge) to introduce and use *parametric multiple sequence alignment* (pMSA) that extends classical MSA [59] to handle parameterized (i.e., data-carrying)

alphabets.[4] Sound theoretical underpinnings of using MSA as a language learning tool are presented here. The use of MSA obviates many limiting assumptions made by prior methods and resolves the long-standing *initial-state uncertainty* problem in *offline* specification mining [154]. MSA naturally adopts a scoring scheme that can be used to quantify the statistical significance of inferred formal specifications and reject spurious properties (i.e., those properties scoring too low to be significant).

- An automated specification mining tool flow, called *ParaMiner*, for for specifications in the form of SR-DFAs that detect potential violations, not only of API usage patterns, but also of object manipulations by a given thread as well as thread interleavings on a given object. ParaMiner also enables controlling abstraction levels of mined properties by taking advantage of user-defined alphabet of data-carrying events or API, which enables extraction of high-level specifications. Finally, ParaMiner automates and visualizes trace segmentation, folding or alignment of trace slices, SR-DFA construction and validation and exposes control knobs for programmers to tune specification complexity and precision.

The graph representation is crucial for the efficiency of RV in FOLe for two reasons: (1) It is used to detect when two checkers can *recombine* and, hence, keep the checker population size small. (2) It is used to prune the set of checkers that need to be updated by a given RV event and, hence, make the state-update procedure of the checker population almost independent of the population size. The effectiveness of the proposed graph representation is demonstrated by the very low conflict rate between concurrent *RV transactions* and will be justified based on a *locality of reference* property confirmed by simulation of real benchmarks. To the best of our knowledge, this thesis is the first to investigate *architectural support* for RV of parametric specifications. Using MSA, ParaMiner can reconstruct properties with *abstract* state spaces which do not merely duplicate the hidden program state space and whose sizes are dictated solely by the complexity of observed execution traces. Each *abstract*

[4]Classical MSA was recently used in specification mining from digital logic simulation traces [137].

state of the constructed SR-DFA may stand for many *concrete* design states and can capture unbounded temporal relations among widely separated events.

1.4 Organization

After this brief account of RV and review of prior work, the rest of the thesis is organized as follows. Chapter 2 develops terminology and notation for subsequent discussions by expounding first-order logic of events (FOLe), which is the principal formalism underlying self-replicating automata and ensemble semantics. Also, examples of parametric properties are used to motivate subsequent discussions. Next, Chapter 3 develops the theory of graphical representations of functions over (downward directed) partially ordered sets, which is crucial to representing the (potentially infinite) ensemble of replicas semantically equivalent to a given SR-DFA. Then in Chapter 4, we explain NUVA, the set of architectural elements needed to accelerate RV with self-replicating automata for workloads having high event densities. If the set of workload programs possess moderate event densities with respect to the monitored interface (e.g., Linux system calls), the architectural implementation may become unnecessary and can be replaced by a distributed network of pure software agents. Chapter 5 is the main chapter on mining specifications from parametric event traces. The entire flow is becoming increasingly more important in the domain of anomaly-based intrusion detection, exploiting the enhanced precision of SR-DFA specifications as well as the automation afforded by ParaMiner. Finally, Chapter 7 concludes the thesis with a summary of results and starts a discussion on open problems as venues for future investigations.

Part I

Formalism

Chapter 2

Self-Replicating Automata

In this chapter, we develop the terminology and notation used throughout the thesis. A formal specification consists of one or more *properties*, where each property establishes a relation between events in temporal sequences. The most widely used specification formalism is *propositional linear temporal logic* (PTL) [19]. The specification formalism of ParaMiner is SR-DFAs. This chapter develops a rigorous mathematical basis for SR-DFAs and their semantics.

2.1 Self-Replicating DFAs at a Glance

Intuitively, a SR-DFA is like an ordinary DFA, but with each transition labeled with a FOL formula that may contain *free variables*, such as $\mathsf{open}(\mathsf{t}, \mathsf{x}_1) \wedge \neg\mathsf{open}(\mathsf{t}, \mathsf{x}_2)$. The motivation behind this is that many program properties are described in natural language in terms of events involving one or more *unspecified* program objects and/or values. These unspecified objects are intended to refer to every possible program object of a given type. For example, a property of a program may dictate that if objects x_1 and x_2 are swapped at any instant of

time, they cannot be swapped with each other again until at least one of them is swapped with a different object x_3 (i.e., $x_3 \neq x_1$ and $x_3 \neq x_2$). SR-DFAs are able to directly capture these natural-language specifications where each transition formula encodes a set of parametric events which trigger that transition. A SR-DFA has one failure (trap) state that is visited when the associated property is violated. In that sense, a SR-DFA accepts (i.e., detects) *minimal bad prefixes* of violating traces. In general, such parametric properties cannot be efficiently captured in PTL [19] because the domains of abstract program objects (e.g., files, sockets, database records, threads, users, etc.) are unbounded, determined only at runtime or, at best, too large to be exhaustively encoded into Boolean propositions. Hence, conventional automata used to statically or dynamically verify PTL properties cannot be used.

In general, the above properties cannot be *efficiently* captured in propositional temporal logic (PTL) [19] because the object domains (e.g., graph nodes, threads, etc.) are unbounded and determined only at runtime or, at best, too large to be exhaustively encoded into Boolean propositions. Hence, classical automata [19] used to statically or dynamically verify PTL properties cannot be used directly. Therefore, to check for parametric property violations in parametric event traces at runtime, we introduce, in this thesis, SR-DFAs that accept, in some sense, *minimal bad prefixes* of violating traces. That is, once an accepting state of SR-DFA $\mathcal{A}^{\varphi}$ of parametric property φ is reached, then φ cannot be satisfied along any extension of the trace thus far observed.

A SR-DFA $\mathcal{A}$ stands for an infinite ensemble $|\mathcal{A}|$ of virtual replicas of itself, each corresponding to a particular valuation of the free variables of $\mathcal{A}$. A finite graph representation $\mathcal{G}^{\mathcal{A}}$ was presented in [136] to capture states of all SR-DFA replicas *at any finite time*. Intuitively, $\mathcal{G}^{\mathcal{A}}$ is a subgraph of a partially-ordered set (poset) structure $\mathcal{PV} \subseteq 2^{|\mathcal{A}|}$ induced by the set $\mathcal{PV}$ of partial variable valuations (or *bindings*). When the monitored program starts, $\mathcal{G}^{\mathcal{A}}$ begins as a single vertex with no variable bindings and, hence, stands for *all* replicas in $|\mathcal{A}|$. When an

event σ occurs, it results in state transitions for a subset $\mathcal{R}^\sigma \subseteq |\mathcal{A}|$ of replicas. The set $\mathcal{R}^\sigma$ can itself be represented in terms of partial variable bindings with the help of a lattice-based function graph (LBFG) representation.[1] As a result, new vertices of $\mathcal{G}^{\mathcal{A}}$ are created while others become redundant and may be annihilated (or garbage-collected). Vertices of $\mathcal{G}^{\mathcal{A}}$ comprise a *dynamic* population of automata checkers.

To get a flavor of SR-DFAs and to illustrate the expressive power of parametric specifications,[2] consider the following examples, which also illustrate the need for dynamic variable binding and self-replication in checker automata. In the Barnes-Hut (BH) benchmark (based on Lonestar BH implementation [106]), the BH tree construction phase was parallelized and each builder thread is assigned a set of bodies to insert into a BH tree. To enhance performance, a fine-grained locking scheme is used in which every tree node is protected by a lock and the fairly complex locking discipline used is checked by the SR-DFA in Figure 2.1, where state q_6 is a *failure state* or a *trap state* for which the locking discipline is violated. In a state diagram of a SR-DFA, an initial state is indicated by an incoming arrow and accepting (failure) states are designated by double circles. The variables $\{t, n_1, n_2\}$ range over BH-tree builder threads and nodes, respectively. In the Canneal benchmark [23], the netlist is a lock-free concurrent graph and multiple annealer threads independently conduct random walks through the netlist and swap elements according to an annealing schedule. A rather contrived, but interesting, property verified by the SR-DFA in Figure 2.2 requires that whenever two elements are swapped, they cannot be swapped again until one of them is swapped with a different element. Note that $\mathsf{Swap}(t, n_1, n_2)$ is the same as $\mathsf{Swap}(t, n_2, n_1)$, since Swap is symmetric.

[1] Lattice-based Boolean diagrams (LBBD) were first introduced in [136] and then elaborated in [135].

[2] The full power of SR-DFAs manifests itself in using them as a behavioral modeling tools with properties automatically extracted from program executions. In that case, extracted properties are complex and are not intended for program understanding. Rather, the main purpose is anomaly detection.

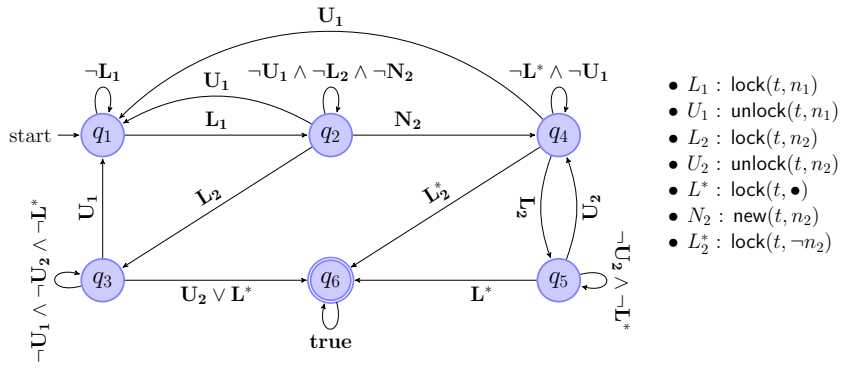


Figure 2.1: Parametric FSM checker for BH locking discipline. We use these abbreviations $p(\bullet) \equiv \forall x.p(x)$ and $p(\neg x) \equiv \forall x'.p(x') \wedge (x' \neq x)$.

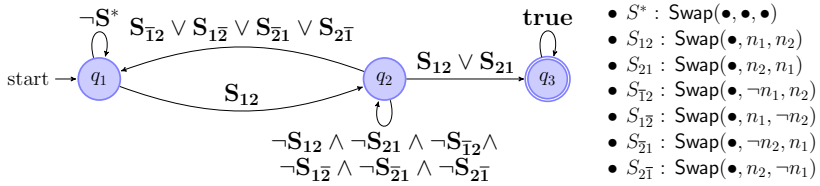


Figure 2.2: Parametric FSM checker for the Canneal element-swapping property (q_3 is the failure state).

2.2 First-Order Logic of Events

To define parametric events and traces, we employ the language of FOLe [136], which is FOL without function symbols[3], with restricted quantifier usage, and with predicates endowed with instantaneous asynchronous event semantics by axiomatically restricting the set of possible interpretations, as detailed next. This allows the expression of more complex properties than other formalisms such as regular expressions [6] and patterns [39] and naturally supports an unlimited number of parameters. Events taking part in a property specification are formally represented by a set $\mathbf{P}$ of FOL predicates. Temporal relations among predicate values can be expressed using temporal operators (such as `next`, `eventually`, `always`, `until`, `before`, etc.) [105] or finite automata (such as SR-DFAs). Predicates from $\mathbf{P}$ may stand for explicit program events (e.g., *method-call events*) or derived from changes in program state. For example, every object class may have a set of methods that can be called on any

[3]Except for zero-arity functions used, by convention, to represent constants.

instance of that class. To maximize concurrency, most specifications need to allow method calls by different threads to overlap (rather than treat them as serialized atomic events) and to describe nested method calls by the same thread. That is why each method **func** has two predicates associated with it, a *method-start predicate* **func.start** and *method-end predicate* **func.end**. A predicate can have any number of arguments, with the first always being the calling thread or process. The return value of a call to method **func** can be considered as an argument to **func.end**.

We borrow the logical notation from [105]. Let $\mathbf{S}^*$ (resp. $\mathbf{S}^\omega$) be the set of finite (resp. infinite) strings over an arbitrary set $\mathbf{S}$ and let ε be the empty string. A many-sorted FOLe signature $\mathsf{SIG} = (\mathbf{S}, \mathbf{P}, \mathbf{C})$ consists of:

- A set $\mathbf{S}$ of sorts or data types (e.g., primitive types, object classes, and thread function types).
- A nonempty set of predicate symbols $\mathbf{P} = \bigcup_{\overline{s} \in \mathbf{S}^*} \mathbf{P}^{(\overline{s})}$ where $\mathbf{P}^{(\overline{s})}$, for every $\overline{s} \in \mathbf{S}^*$, is a (possibly empty) set of predicate symbols and $\overline{s}$ is a predicate prototype giving the tuple of data types accepted as arguments by elements of $\mathbf{P}^{(\overline{s})}$. The first element of $\overline{s}$ is always the calling thread or process type.
- A set of constant symbols $\mathbf{C} = \bigcup_{s \in \mathbf{S}} \mathbf{C}^s$ where $\mathbf{C}^s$, for every $s \in \mathbf{S}$, is a (possibly empty) set of constant symbols of sort s.

Also, let the set $\mathcal{X} = \bigcup_{s \in \mathbf{S}} \mathcal{X}_s$ be a countable set of *variables* of various sorts. Variables can be used as predicate arguments and by quantifiers $\exists$ and $\forall$. Let $\mathfrak{F}(\mathsf{SIG})$ be the set of well-formed formulas [105] in SIG defined by the following syntactic rules:

$$\mathsf{formula} := \mathsf{atom} \mid \mathsf{formula} \wedge \mathsf{formula} \mid \mathsf{formula} \vee \mathsf{formula} \mid \neg\mathsf{formula} \mid (\mathsf{formula}) \tag{2.1}$$

$$\mathsf{atom} := p(x_1, \ldots, x_n) \mid \exists x_k.\mathsf{atom} \mid \exists x'_k.\mathsf{atom} \wedge (x'_k \neq x_k) \tag{2.2}$$

Let Atoms be the set of all atoms defined above. By way of syntactic sugaring, we have:

$$\begin{array}{lll} \exists x_k.p(\ldots,x_k,\ldots) & \equiv & p(\ldots,\bullet,\ldots) \\ \exists x'_k.p(\ldots,x'_k,\ldots) \quad \wedge(x'_k \neq x_k) & \equiv & p(\ldots,\neg x_k,\ldots) \end{array} \tag{2.3}$$

2.2.1 Constant Arguments

Some events may have a finite-valued argument, such as a Boolean or enumeration. In that case, using variables during MSA to abstract the values of these arguments in the observation traces is undesirable, since each individual value may indicate or trigger a characteristically different behavior and distinction needs to be preserved. Therefore, we use a FOL constant for each possible value of a finite-valued argument type to prevent alignment of distinct values. Hence, the use of constants increases the resolving power (or precision) of extracted specifications.

2.3 Semantics

The set $\mathbf{P}$ of predicates represents *instantaneous, asynchronous parameterized events* that, at any time, involve a particular tuple of *values* (e.g., objects and threads) and where events can only interleave and are impossible to occur simultaneously. Software systems that fulfill this condition are termed *asynchronous systems* [19]. This requirement can be axiomatized as follows:

$$q_1 \neq q_2 \Rightarrow \neg q_1(\mathbf{x}) \vee \neg q_2(\mathbf{y}), \tag{2.4}$$

$$\mathbf{x} \neq \mathbf{x}' \Rightarrow \neg q(\mathbf{x}) \vee \neg q(\mathbf{x}') \tag{2.5}$$

where $\mathbf{x} = (x_1, .., x_m)$ and $\mathbf{y} = (y_1, .., y_n)$ are vectors of argument variables and constants, and $\mathbf{x} \neq \mathbf{x}'$ means that $(x_1 \neq x'_1) \vee \ldots \vee (x_n \neq x'_n)$. Formulas in $\mathfrak{F}(\mathsf{SIG})$ are interpreted (i.e., assigned truth values) given a *structure* and a *variable valuation*. A first-order structure S consists of:

- A *data component*, or universe, $|\mathsf{S}| = \bigcup_{s \in \mathbf{S}} |\mathsf{S}|_s$, where $|\mathsf{S}|_s$ is a nonempty set, called *domain*, for every sort $s \in \mathbf{S}$.
- A set of mappings $p^{\mathsf{S}} : |\mathsf{S}|_{s_1} \times \ldots \times |\mathsf{S}|_{s_n} \rightarrow \{\mathsf{false}, \mathsf{true}\}$ for every predicate symbol $p \in \mathbf{P}^{(s_1,\ldots,s_n)}$.
- A mapping $\mathsf{C}^{\mathsf{S}} : \mathbf{C} \rightarrow |\mathsf{S}|$ such that for every $s \in \mathbf{S}$ and every $c \in \mathbf{C}^s$, we have $\mathsf{C}^{\mathsf{S}}(c) \in |\mathsf{S}|_s$.

Moreover, to interpret formulas containing free (i.e., unquantified) variables, a *variable valuation* $\xi : \mathcal{X} \rightarrow |\mathsf{S}|$ with respect to data component $|\mathsf{S}|$ is used to assign to each variable a value compatible with its data type (i.e., $x \in \mathcal{X}_s \Rightarrow \xi(x) \in |\mathsf{S}|_s$). Let Ξ^{S} be the set of all such variable valuations.

Models of FOLe Formulas - The Alphabet Set. For all subsequent discussions, we consider structures with a fixed data component $|\mathsf{S}|$ and fixed constant values C^{S}. We will also use a predicate symbol $p \in \mathbf{P}$ and its interpretation p^{S} interchangeably. Axioms 2.4, 2.5 restrict the possible structures (i.e., *models*) of a FOLe signature SIG to sets of mappings $p^{\mathsf{S}} : |\mathsf{S}|_{s_1} \times \ldots \times |\mathsf{S}|_{s_n} \rightarrow \{\mathsf{false}, \mathsf{true}\}$ such that there is at most one $(s_1, \ldots, s_n) \in \mathbf{S}^*$ and at most one $p \in \mathbf{P}^{(s_1,\ldots,s_n)}$ and at most one tuple of values $(v_1, \ldots, v_n) \in |\mathsf{S}|_{s_1} \times \ldots \times |\mathsf{S}|_{s_n}$ with $p^{\mathsf{S}}(v_1, \ldots, v_n) = \mathsf{true}$. Let Σ^{SIG} (or simply Σ, since SIG is fixed) be the set of all possible models of a FOLe signature SIG. Intuitively, $\Sigma \subseteq \mathbf{P} \times |\mathsf{S}|^*$ is the (infinite) *alphabet set* (i.e., the set of all parametric events) obtained from each $p \in \mathbf{P}$ as its arguments range over their respective domains. A structure S and a variable valuation ξ with respect to S inductively assign a truth value $\mathsf{S}^{(\xi)}(f)$ to every FOLe $f \in \mathfrak{F}(\mathsf{SIG})$. We say $\sigma \in \Sigma$ is a model of $f \in \mathfrak{F}(\mathsf{SIG})$, denoted by $\sigma \models f$, if there is a variable valuation ξ such that $\mathsf{S}^{(\xi)}(f) = \mathsf{true}$.

The FOLe restrictions on the use of quantifiers ensure that the set of variable valuations that make any FOLe formula true in a given model is described by an equality-logic formula.

Theorem 2.1. *Given a FOLe formula $f \in \mathfrak{F}(\mathsf{SIG})$ and a model $\sigma \in \Sigma$, the set of variable valuations that make f true in σ is described by an equality-logic formula.*

Proof. Let $\odot$ be a binary operator with the left operand given by a FOLe model (i.e., a parametric event), the right operand given by a FOLe formula and the result given by a logical description of the set of all variable valuations that render the right operand formula true in the model given by the left operand. This theorem follows directly from the following *unification rules*:

$$
\begin{array}{lcll}
p(x_1,\ldots,x_n) & \odot & (f_1 \vee f_2) & = (p(x_1,\ldots,x_n) \odot f_1) \vee (p(x_1,\ldots,x_n) \odot f_2) \\
p(x_1,\ldots,x_n) & \odot & (f_1 \wedge f_2) & = (p(x_1,\ldots,x_n) \odot f_1) \wedge (p(x_1,\ldots,x_n) \odot f_2) \\
p(x_1,\ldots,x_n) & \odot & (\neg f) & = \neg(p(x_1,\ldots,x_n) \odot f) \\
p(x_1,\ldots,x_n) & \odot & p(y_1,\ldots,y_n) & = (x_1 = y_1) \wedge \ldots \wedge (x_n = y_n) \\
p(x_1,\ldots,x_n) & \odot & p(y_1,\ldots,\neg y_n) & = (x_1 = y_1) \wedge \ldots \wedge (x_n \neq y_n) \\
p(x_1,\ldots,x_n) & \odot & p(y_1,\ldots,\bullet) & = (x_1 = y_1) \wedge \ldots \wedge (x_{n-1} = y_{n-1})
\end{array}
$$

□

In fact, Theorem 2.1 is a characterization of FOLe as the fragment of FOL for which every member logical formula has models consistent only with variable valuations described by equality-logic formulas. FOLe is the fragment of FOL whose models imply equality-logic constraints over variable valuations. For every FOLe formula $f \in \mathfrak{F}(\mathsf{SIG})$, let $\mathcal{M}(f) \subseteq \Sigma$ be the set of models of f.

Parametric traces. A parametric trace is an infinite sequence $\boldsymbol{\sigma} : \mathbb{N} \to \Sigma = (\sigma_0, \sigma_1, \sigma_2, \ldots)$ of parametric events. Given a variable valuation ξ with respect to $|\mathsf{S}|$ and a parametric trace $\boldsymbol{\sigma}$, we can assign a truth value $\mathsf{K}_{\sigma_t}^{(\xi)}(f)$ for every formula $f \in \mathfrak{F}(\mathsf{SIG})$ at every time instant

$t \in \mathbb{N}$ in consistence with the familiar semantics of logical operators. Since every parametric event takes the form $p(c_1, \ldots, c_n)$ where $p \in \mathbf{P}^{(\overline{s})}$ for some $\overline{s} \in \mathbf{S}^*$ and $c_i \in |\mathsf{S}|_{s_i}$ for all i, it follows that the base cases are given by:

$$\begin{aligned}
\mathsf{K}^{(\xi)}_{\sigma_t}(p(x_1, \ldots, x_n)) &= \mathsf{true} \text{ iff } \sigma_t = p(c_1, \ldots, c_n) \text{ and } \quad \xi(x_i) = c_i \text{ for } i = 1, \ldots, n \\
\mathsf{K}^{(\xi)}_{\sigma_t}(p(x_1, \ldots, \neg x_n)) &= \mathsf{true} \text{ iff } \sigma_t = p(c_1, \ldots, c_n) \text{ and } \quad \xi(x_n) \neq c_n, \xi(x_i) = c_i \text{ for } i \neq n \\
\mathsf{K}^{(\xi)}_{\sigma_t}(p(x_1, \ldots, \bullet)) &= \mathsf{true} \text{ iff } \sigma_t = p(c_1, \ldots, c_n) \text{ and } \quad \xi(x_i) = c_i, \text{ for all } i \neq n
\end{aligned}$$

2.3.1 State Transition Systems.

The set of all observable parametric event traces of the program under verification (PUV) can be described by a state transition system TS [105], given by a tuple $(S, \rightarrow, I, \Sigma, L)$, where S is the set of states, $\rightarrow \subseteq S \times S$ is a (possibly nondeterministic) total transition relation, $I \subseteq S$ is a set of initial states, Σ is the FOLe alphabet set, and $L : S \rightarrow \Sigma$ is a labeling function.[4] Nondeterminism of $\rightarrow$ might be due to abstraction, concurrency or lack of constraints on external program inputs. The set $\mathbf{P}$ is chosen depending on the properties to be verified, and summarizes the *observable* events of interest emitted by TS. A *path* in TS is a (possibly infinite) sequence of states $s_0, s_1, s_2, \ldots$ such that $s_0 \in I$ and $s_i \rightarrow s_{i+1}$ for $i \geq 0$. For every path $s_0, s_1, s_2, \ldots$, there is a *trace* $L(s_0), L(s_1), L(s_2), \ldots$. The set $Traces(TS)$ is the set of all traces of TS starting from an initial state.

2.3.2 Linear-Time Parametric Properties.

Let Σ^ω be the set of infinite words over Σ. A linear-time (LT) property φ over Σ is a subset of Σ^ω. Transition system TS satisfies LT property φ, written as $TS \models \varphi$, iff $Traces(TS) \subseteq \varphi$.

[4] Clearly, TS thus defined is infinite and each state may have infinitely many successors. So this representation is only conceptual and, for the purposes of runtime verification only, we will devise decision algorithms that are guaranteed to terminate.

A LT property φ is a *safety property* if every violating trace $\tau \in \Sigma^\omega$ (i.e., $\tau \notin \varphi$) has a *finite bad prefix* (i.e., a prefix all of whose infinite extensions also violate φ). A regular safety property is a safety property whose bad prefixes can be recognized by a SR-DFA.

2.4 Self-Replicating DFAs

Motivated by the intuitive nature of finite automata and the availability of automata mining algorithms [110], we adopt SR-DFAs as the specification formalism. Unlike Büchi automata [19] which operate on infinite words, a SR-NFA operates only on finite words. A SR-NFA is a tuple $\mathcal{A} = (Q, \mathsf{SIG}, \delta, Q_0, F)$, where Q is a finite nonempty set of states, $Q_0 \subseteq Q$ is the set of initial states, SIG is a FOLe signature, $F \subseteq Q$ is the set of final (or accepting) states indicating violation of the monitored property. Finally, $\delta \subseteq Q \times \mathfrak{F}(\mathsf{SIG}) \times Q$ is the transition relation. The set F of final states is terminal (or a trap). That is:

$$\forall (q_1, f, q_2) \in \delta : q_1 \in F \Longrightarrow q_2 \in F$$

Free variables in transition formulas of a SR-NFA $\mathcal{A}$ is intended to attach an *instance* of $\mathcal{A}$ to every valuation ξ of those free variables. Thus, for every ξ, the transition relation δ yields a concrete relation $\delta^{(\xi)} \subseteq Q \times \Sigma \times Q$ where:[5]

$$(q_1, \sigma, q_2) \in \delta^{(\xi)} \Longleftrightarrow \exists (q_1, f, q_2) \in \delta : \mathsf{K}_\sigma^{(\xi)}(f) = \mathsf{true}$$

A SR-NFA $\mathcal{A}$ is *deterministic* (a SR-DFA) if $|Q_0| \leq 1$ and for every $q \in Q$, $\sigma \in \Sigma$ and every ξ, we have $|\delta^{(\xi)}(q, \sigma, \cdot)| \leq 1$ or, equivalently:

$$\forall (q, f_1, q_1), (q, f_2, q_2) \in \delta : q_1 \neq q_2 \Longrightarrow \neg(f_1 \wedge f_2)_{\text{modulo FOLe}}$$

[5] We will also use $\delta^{(\xi)}$ as a function $\delta^{(\xi)} : Q \times \Sigma \to 2^Q$.

It is *nondeterministic* (a SR-NFA) otherwise. The extended transition function $\Delta^{(\xi)} : Q \times \Sigma^* \to 2^Q$ is inductively defined as:

$$w, w' \in \Sigma^*, \sigma \in \Sigma, w = \sigma w' \implies \Delta^{(\xi)}(q, w) = \Delta^{(\xi)}(\delta^{(\xi)}(q, \sigma), w')$$

In this thesis, we adopt *ensemble semantics* of SR-NFAs, as defined in [136]. Given a parametric property φ, in the form of a SR-NFA $\mathcal{A}^\varphi$, a finite prefix $w \in \Sigma^*$ is accepted by $\mathcal{A}^\varphi$ and is called a *bad prefix* iff there is a variable valuation $\xi : \mathcal{X} \to |\mathsf{S}|$ such that $\Delta^{(\xi)}(Q_0, w) \cap F \neq \emptyset$.

The language $\mathcal{L}(\mathcal{A})$ of SR-NFA $\mathcal{A}$ is the set of all accepted finite words. For any $q_1, q_2 \in Q$, if $q_2 \in \delta^{(\xi)}(q_1, \sigma)$, this is abbreviated as $q_1 \xrightarrow{\xi,\sigma} q_2$. If $\sim \subseteq Q \times Q$ is an equivalence relation, then each equivalence class is an abstract state [42] and $\mathcal{A}/\!\sim$ is the *quotient automaton*, which can be nondeterministic even if $\mathcal{A}$ is deterministic. It then holds that $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A}/\!\sim)$.[6]

Given a SR-DFA $\mathcal{A} = (Q, \mathsf{SIG}, \delta, Q_0, F)$, let $\theta : Q \to \Gamma$ be an *output function* with *output alphabet* $\Gamma = \{\top, \bot, ?\}$, indicating satisfaction, violation or indecision of the monitored property, respectively. We now define the finite-word semantics. A parametric property φ, in the form of a SR-DFA $\mathcal{A}^\varphi$, implements, for every possible variable valuation ξ, a three-valued semantic function $[\bullet \models \varphi]^\xi = \mathcal{F}^{\varphi_{|\xi}} : \Sigma^* \to \{\top, \bot, ?\}$ that assigns to each finite prefix $u \in \Sigma^*$ of an infinite trace one of three truth values ($\top$ or *true*, $\bot$ or *false*, ? or *inconclusive*) as follows:

$$\mathcal{F}^{\varphi_{|\xi}}(u) = [u \models \varphi]^\xi = \begin{cases} \top & \text{if } \forall \sigma \in \Sigma^\omega : u.\sigma \models \varphi_{|\xi} \\ \bot & \text{if } \forall \sigma \in \Sigma^\omega : u.\sigma \not\models \varphi_{|\xi} \\ ? & \text{otherwise} \end{cases}$$

where $\varphi_{|\xi}$ is the parametric property φ after substituting all free variables with their values from ξ. The semantic function $\mathcal{F}^{\varphi_{|\xi}}$ extends the output function $\theta : Q \to \{\top, \bot, ?\}$ of $\mathcal{A}^\varphi$ to finite words. If Ξ is the set of all variable valuations, then a semantic function

[6] This holds if the quotienting operation uses *existential abstraction* [42], which over-approximates the original automaton $\mathcal{A}$.

$[u \models \varphi] = \mathcal{F}^{\varphi} : \Sigma^* \to \{\top, \bot, ?\}$ is defined as:

$$[u \models \varphi] = \bigwedge_{\xi \in \Xi} [u \models \varphi]^{\xi} \tag{2.6}$$

2.4.1 Ensemble State

Let Ξ be the set of all variable valuations. Then an *ensemble state* is given by a mapping $\lambda : \Xi \to Q$. The set of initial ensemble states for a SR-NFA $\mathcal{A}$ is given by the set Λ_0 of all ensemble states λ_0 such that $\forall \xi \in \Xi : \lambda_0(\xi) \in Q_0$. Let $\Lambda^{\dagger}$ be the set of all possible functions $\lambda : \Xi \to Q$. SR-NFA $\mathcal{A}$ defines a transition structure $\to_E \subseteq \Lambda^{\dagger} \times \Sigma \times \Lambda^{\dagger}$ over $\Lambda^{\dagger}$ where:

$$\forall \lambda_1, \lambda_2 \in \Lambda^{\dagger}, \sigma \in \Sigma : \lambda_1 \xrightarrow{\sigma}_E \lambda_2 \Leftrightarrow \forall \xi \in \Xi : \lambda_2(\xi) \in \delta^{(\xi)}(\lambda_1(\xi), \sigma)$$

Let Λ be the set of all possible ensemble states reachable from an initial state $\lambda_0 \in \Lambda_0$ on a finite word $w \in \Sigma^*$.

2.4.2 Formal Verification

Given a SR-NFA $\mathcal{A}^{\varphi} = (Q, \mathsf{SIG}, \delta, Q_0, F)$, the product $TS \bigotimes \mathcal{A}^{\varphi} = (S', \to', I', \Sigma, L')$ is defined as:

$$S' = S \times \Lambda, \tag{2.7}$$

$$I' = \{(s_0, \lambda) | s_0 \in I, \forall \xi \in \Xi . \exists q_0 \in Q_0 : \delta^{(\xi)}(q_0, L(s_0)) = \lambda(\xi)\} \tag{2.8}$$

$$\forall s_1, s_2 \in S, \forall \lambda_1, \lambda_2 \in \Lambda \frac{s_1 \to s_2,\ \lambda_1 \xrightarrow{L(s_2)} \lambda_2}{(s_1, \lambda_1) \to' (s_2, \lambda_2)} \tag{2.9}$$

To verify that a finite initial path $s_0, s_1, \ldots, s_n$ in TS, with $s_0 \in I$, models φ, it is sufficient to check that for all $(s_i, \lambda_i) \in S'$, with $0 \leq i \leq n$ and $(s_0, \lambda_0) \in I'$, there is no $\xi \in \Xi$ such that $\lambda_i(\xi) \in F$, since accepting states in F indicate violation of φ. To verify that $TS \models \varphi$, it is sufficient to check that for all $(s, \lambda) \in S'$ reachable from I', there is no $\xi \in \Xi$ such that

$\lambda(\xi) \in F$.

2.5 Runtime Verification with SR-DFAs

The central question now is how to verify a parametric property given by a SR-DFA $\mathcal{A}^{\varphi}$. The major problem is that the semantic function $[u \models \varphi]$ in equation (2.6) is a conjunction over an infinite ensemble $|\mathcal{A}^{\varphi}|$ of *virtual replicas* of $\mathcal{A}^{\varphi}$, each corresponding to a particular variable valuation ξ of the free variables of φ and using its own semantic map K_t^{ξ} to follow a path $(q_0^{\xi}, q_1^{\xi}, q_2^{\xi}, \ldots)$ in $\mathcal{A}^{\varphi}$. A main contribution of this thesis is an efficient method for calculating $[u \models \varphi]$ at every instant of time using *a finite DAG representation that captures states of all SR-DFA replicas in $|\mathcal{A}^{\varphi}|$ by using partial variable valuations*. More technical details on graph representations of functions over partially ordered sets are provided in Chapter 3 and particular attention to multi-variable functions is in Section 3.8.1.

2.5.1 The Semi-Lattice of Partial Variable Valuations

Let Ξ^* be the meet-semi-lattice of all (possibly partial) variable valuations $\xi^* : \mathcal{Y} \to |\mathsf{S}|$ for all $\mathcal{Y} \subseteq \mathcal{X}$. A partial valuation $\xi_1^* : \mathcal{Y}_1 \to |\mathsf{S}|$ is less specific (i.e., binds less variables) than another $\xi_2^* : \mathcal{Y}_2 \to |\mathsf{S}|$, denoted by $\xi_1^* \preceq \xi_2^*$, iff for every $y \in \mathcal{Y}_1$, $\xi_1^*(y) = \xi_2^*(y)$. It is also said that ξ_2^* *extends* ξ_1^*. An ensemble state $\lambda : \Xi \to Q$ can be *recursively* extended to a function $\lambda^* : \Xi^* \to Q$ as follows:

- $\lambda^*(\xi^*) = q$ if, for all ξ which is a complete extension of ξ^* such that there is no extension $\xi^* \preceq \xi' \preceq \xi$ with $\lambda^*(\xi') \neq q$, we have $\lambda(\xi) = q$.

The space of all extended ensemble states $\lambda^* : \Xi^* \to Q$ is denoted by Λ^*. Theorem 2.2 formally states that, for the purposes of RV, a finite directed acyclic graph Ens_t can be used

to represent the ensemble state defined over a (potentially infinite) infinite ensemble $|\mathcal{A}^{\varphi}|$ of SR-DFA replicas, where $Ens_t = (V_t, E_t, r, \rho_t, q_t)$ with:

- V_t is the vertex set representing the replica population.
- $E_t \subseteq V_t \times V_t$ is the set of directed edges.
- r is the distinguished *root vertex* of the graph.
- $q_t : V_t \to Q$ assigns to each vertex $v \in V_t$ a state $q_t(v)$.
- $\rho_t : V_t \times \mathcal{X} \to |\mathsf{S}| \cup \{\mathsf{nil}\}$ associates with each vertex $v \in V_t$ a (possibly partial) set of variable bindings such that $\rho_t(r, x) = \mathsf{nil}$ for every $x \in \mathcal{X}$ and $t \in \mathbb{N}$.

We will later study conditions under which that graph representation is not only finite, but also bounded.

Theorem 2.2. *At every time instant $t \in \mathbb{N}$, there is a finite subset $\Xi_t^* \subseteq \Xi^*$ such that the ensemble state $\lambda_t : \Xi \longrightarrow Q$ and the extended ensemble state $\lambda_t^* : \Xi^* \longrightarrow Q$ satisfy the relation:*

$$\forall \xi \in \Xi : \quad \xi^* \lessdot \xi \text{ in } \Xi_t^* \Rightarrow \lambda_t(\xi) = \lambda_t^*(\xi^*)$$

The proof of Theorem 2.2 is expounded in the following sections, which also explains how RV with SR-DFAs works in practice, both at compile-time and at run-time. The ensemble graph derives from (and owes its efficiency to) a novel graphical function representation (so-called *lattice-based function graph* or LBFG), defined in Chapter 3, which has also been applied in [135] to the representation of Boolean functions $f : \{0,1\}^n \to \{0,1\}$.

2.5.2 Graphical Representation of Ensemble State.

Initially, all replicas of $\mathcal{A}^{\varphi}$ have the same state. Therefore, the initial ensemble state λ_0 and the extended ensemble state λ_0^* both satisfy the condition $\lambda_0(\xi) = \lambda_0^*(\xi^*) = q_0$ for all $\xi \in \Xi$ and $\xi^* \in \Xi^*$. Thus, Ξ_0^* starts as a single element, the bottom $\perp$ of Ξ^*, that stands for the

entire ensemble Ξ, since $\perp$ initially has no successors in Ξ_0^* and all SR-DFA free variables are left unbound.

2.5.3 Graphical Representation of SR-DFA Transition Function.

Let $\mathcal{FP}$, where $\mathcal{X} \cap \mathcal{FP} = \emptyset$, be a set of *formal parameters*, which are variables used, at compile-time, as *uninterpreted placeholders* for predicate arguments that materialize only at run-time. Now, since we know how (Ξ_0^*, λ_0^*) looks like, let's see how (Ξ_t^*, λ_t^*) evolves from $(\Xi_{t-1}^*, \lambda_{t-1}^*)$ given the current event σ_t. The current event $\sigma \in \Sigma$ is *unified* with every formula $f \in \mathfrak{F}(\mathsf{SIG})$ in the transition function δ of $\mathcal{A}^\varphi$ to find which variable valuations *may* lead to state transitions. The result of unifying an event $\sigma = p(a_1, \ldots, a_n)$ with a formula $f \in \mathfrak{F}(\mathsf{SIG})$ is an equality-logic formula $f|_\sigma$ containing only literals of the form $(x_i = a_i)$ and $(x_i \neq a_i)$, where $x_i \in \mathcal{X}$ is a free variable or a symbolic constant used by $\mathcal{A}^\varphi$ and a_i is a particular value from the domain of x_i. At compile-time, a_i is a formal parameter from $\mathcal{FP}$ to refer to a particular parameter position of a predicate symbol. The actual value held by a predicate argument is substituted at run-time. All variable values satisfying (resp. not satisfying) $f|_\sigma$ stand for a subset of $|\mathcal{A}^\varphi|$ that takes (resp. does not take) the transition enabled by f. The equality-logic formula $f|_\sigma$ can be represented as a LBFG by transforming it into a Boolean formula $e(f|_\sigma)$, called its *propositional skeleton* [104], by encoding every literal of the form $(x = a)$ with a propositional variable $\widehat{xa} \in \{0, 1\}$ and encoding $(x \neq a)$ with $\neg\widehat{xa}$ and adding constraints that prohibit a variable from having multiple binding values (e.g., $(x \neq a) \vee (x \neq b)$).

For every SR-DFA state q and for every FOLe formula f labeling one of its outgoing transitions, we construct a LBFG $\mathcal{G}_1(q, f, p)$ for every FOLe predicate p. For example, if formula $f = swap(x, \neg x)$, the corresponding replication graph is shown in Figure 2.3, where the right-hand graph is obtained from the left-hand graph by using the equation $(x = a_1)$ to

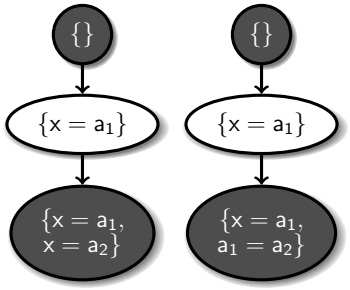


Figure 2.3: An example of the emergence of guard expressions.

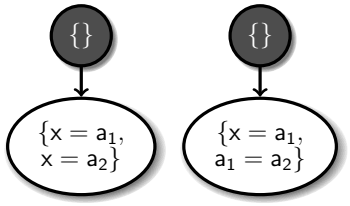


Figure 2.4: An example of the emergence of guard expressions.

substitute in the second equation ($x = a_2$). Figure 2.4 also shows the replication graph for $f = swap(x, x)$ and event $swap(a_1, a_2)$. Equation literals not involving variables (i.e., containing only event arguments, constants, or actual literals) can be thought of as *guard expressions*.

Since, for a SR-DFA, formulas labeling all outgoing transitions $\{f_1, \ldots, f_n\}$ of any state q are mutually exclusive (i.e., every replica with a complete set of variable bindings can only take one transition at any time), all LBFGs $\mathcal{G}_1(q, f_i, p)$ associated with formulas $\{f_1, \ldots, f_n\}$ and predicate symbol $p \in \mathbf{P}$ can be composed[7] into one LBFG $\mathcal{G}_2(q, p)$ associated with the pair (p, q). Then all LBFGs $\mathcal{G}_1(q, p)$ can be composed[8] into one LBFG $\mathcal{G}(\mathcal{A}^\varphi, p)$ associated with SR-DFA $\mathcal{A}^\varphi$ and predicate p.

It follows that the effect of event σ on a LBFG Ξ_t^* representation of ensemble state at time t is given by the composition of local transformation $\mathcal{G}(\mathcal{A}^\varphi, p_t)$ with Ξ_t^*. As a result, some elements of Ξ_t^* *disintegrate* or *shatter* or *self-replicate* into multiple, more specific elements (i.e., elements higher in the partial valuation order) with different states. On the other

[7]Using disjoint-union composition defined in Section 3.5.1.
[8]Using local transformation composition defined in Section 3.6.

hands, some elements of Ξ_t^* become redundant and *recombine* with other elements of Ξ_t^* into less specific elements, which helps keep the size of Ξ_t^* manageable. A crucial aspect to the efficiency of SR-DFAs is the process of *checker recombination* or garbage collection [98] that counters the self-replication process so as to keep the ensemble DAG reasonably small. A single event can cause the collapse of the entire population.

Thus, LBFG representation of the SR-DFA transition relation offers a very concise and efficient way of pre-computing ensemble LBFGs at compile time. This shifts most of the heavy lifting to compile time which substantially reduces hardware area and power overhead.

2.5.4 Interpretation of the Transition LBFG

For a SR-DFA, the LBFGs $\mathcal{G}(\mathcal{A}^\varphi, p)$ represent the ensemble state transition function of a SR-DFA $\mathcal{A}^\varphi$ in response to various event predicates p. Important insights about SR-DFA population growth can be obtained by studying the composition of these local transformations into an *extended ensemble state transition function* $h(\mathbf{w})$ that maps an initial ensemble state λ_0 into a final ensemble state given a finite word $\mathbf{w}$ of FOLe events parameterized with formal parameters from $\mathcal{FP}$. The ensemble state LBFG is bounded if all trajectories reachable from λ_0, under sequences of local transformations $\mathcal{G}(\mathcal{A}^\varphi, p)$, are bounded.

Note that the LBFG representation of $h(\mathbf{w})$ consists of nodes labeled with sets of equality-logic atoms involving free variables and symbolic constants from the transition function of $\mathcal{A}^\varphi$ as well as formal parameters occurring at possibly different moments along $\mathbf{w}$. This points out a salient feature of SR-DFAs:

- Since a SR-DFA uses only a finite set of free variables and constants, these sets of equality-logic atoms (called *cubes*) can be written as one equality between a free variable and a formal parameter with every other atom being an equality between two for-

mal parameters possibly occurring at two different time moments along $\mathbf{w}$. Thus, given a finite word $\mathbf{w}$, the failure state of a SR-DFA is reachable subject to an equality-logic constraint on the formal parameters of events along $\mathbf{w}$. *Thus, a SR-DFA constrains which objects or values can be manipulated by a program at different moments in time.*

- A SR-DFA is characterized by two memory spans, a *sequential memory span* associated with its set of states[9] and a *data memory span* associated with its set of free variables. *The population size is bounded if the data memory span is bounded for all legal event sequences (assuming the number of free variables is finite).*

Given a LBFG representation (Ξ^*, λ^*) of a SR-DFA ensemble state, it is evident that if $\xi_1^* \vee \xi_2^*$ does not exist, where $(\xi_1^*, \xi_2^*) \in \Xi_1' \times \Xi_2'$, then $\xi_i^* \vee \xi_j^*$ also does not exist for all their successors $(\xi_i^*, \xi_j^*) \sqsubseteq (\xi_1^*, \xi_2^*)$. This ensures that the semi-lattice Ξ^* is a hereditary poset in the sense of Section 3.3. *This is the main reason that RV with SR-DFAs is efficient, since state transitions induced every event (as represented by a local transformation) are localized to specific subspaces of Ξ. Hence, in a distributed system, events tend to have minimal conflicts.*

Lemma 2.3. *For every pair $(\xi_1^*, \xi_2^*) \in \Xi_1' \times \Xi_2'$, if $\xi_1^* \vee \xi_2^*$ does not exist, then $\xi_3^* \vee \xi_4^*$ does not exist for every pair $(\xi_3^*, \xi_4^*) \in \Xi_1' \times \Xi_2'$ such that $\xi_1^* \preceq \xi_3^*$ and $\xi_2^* \preceq \xi_4^*$.*

[9]By Myhill-Nerode theorem for DFAs, each state corresponds to a set of equivalent finite words that are indistinguishable by any extension. Thus, each DFA state summarizes the history of all its words.

Chapter 3

Lattice-Based Function Graphs

This chapter presents lattice-based function graphs (LBFGs), a graphical representation of discrete functions over partially ordered sets (posets, for short), as well as symbolic manipulation algorithms. As will be shown in Chapter 4, LBFGs proved to be instrumental to the efficient RV of SR-DFA software specifications over distributed multiprocessor systems using NUVA. A special case of LBFGs is given by lattice-based Boolean diagrams (LBBDs) introduced in [135], a graphical representation of Boolean functions that is *not* derived from binary decision diagrams (BDDs). We later identify a class of Boolean functions where LBBDs are demonstrably more efficient to construct, and reason with, when compared to BDDs and zero-suppressed BDDs (or ZDDs, for short). The case studies include ITC99 and MCNC benchmarks, randomly generated cube covers or sum-of-products (SOP) formulas as well as multi-level Boolean formulas.

3.1 Introduction

Boolean functions constitute an important class of functions. NP-completeness of the Boolean satisfiability problem [44] (K-SAT for $K > 2$) not only implies, on the dark side, that most likely it is intractable in the worst case, but also that a large and important class of decision problems (NP problems) can be reduced to it so efficiently that SAT solvers and graphical representations of Boolean functions became mainstream tools in many areas [9, 24, 49, 76, 82, 126, 127]. Therefore, Boolean functions play the role of a universal (yet primitive and sometimes cumbersome) language in which many interesting problems, from formal verification [102] and logic synthesis [82] to knowledge representation [49] and computational biology [76], could be cast and solved. Decision diagrams, such as BDDs [32], are fundamental data structures used to represent Boolean functions. Much of their power derives from their canonicity which helps in equivalency checking [30], compactness over a large class of Boolean functions, efficient construction and symbolic manipulation procedures, in addition to *compositionality* (which enables sharing common sub-functions). However, the Boolean function space is vast and is not amenable to a single universally efficient representation. That gave rise to a plethora of other more specialized representations. For example, zero-suppressed BDDs (ZDDs) [132] were introduced to represent families of sparse subsets of a given large set (a common case in combinatorics) or, equivalently, to represent cube covers. BDDs have also spawned many other types of graphical representations [18, 33] for more general finite-valued functions. Thus, as the unfettered scope of Boolean functions continues to encompass new applications, the best strategy is to be equipped with an arsenal of different representations that can be efficiently converted among themselves and select the one that best matches every particular task. Decision diagrams (DDs) model an n-ary Boolean function $f : \{0,1\}^n \rightarrow \{0,1\}$ by the set of *sequences* of binary decisions on the n Boolean variables (traversed in a particular order) needed to evaluate that function. However, the Boolean-lattice structure of the set $\{0,1\}^n$ is not utilized and appears only implicitly in the

relation among BDD paths, obscured by the imposed variable ordering.

In [135], we started with a well-known lattice-based representation of *monotonic* Boolean functions and illustrated its clear advantage over BDDs and ZDDs. We then generalized that representation to arbitrary Boolean functions and empirically showed that substantial savings can still be achieved especially as functions approach the monotonic regime. We concluded with detailed benchmarking results using real as well as synthetic Boolean functions. In this chapter, we generalize these *lattice-based Boolean diagrams* (LBBDs) to all discrete functions defined over posets (not necessarily lattices or semi-lattices) with very general requirements. We also develop symbolic algorithms implementing all the pointwise composition operations (e.g., Boolean logic operations) directly on these lattice-based function graphs (LBFGs). A main contribution of this thesis is an algorithm to construct LBFGs *symbolically* rather than by direct reduction from the (exponentially large) partially-ordered domain (e.g., the Boolean lattice $\{0,1\}^n$). LBFGs proved crucial for RV of SR-DFA specifications over distributed multiprocessor systems [136], where they are central to the representation of both SR-DFA transition functions as well as the time-varying states of a (potentially infinite) ensemble of automaton replicas. This LBFG is a concurrent data structure that can be updated incrementally and concurrently by multiple threads with low conflict rates. LBFGs also enabled efficient processing of RV transactions by leveraging temporal and spatial locality of monitored software programs and reducing conflicts among concurrent RV transactions. Other prior work on DDs [77] improved over multi-terminal BDDs (MTBDDs) [75], used to represent functions $f : \{0,1\}^n \to \mathcal{L}$ with $\mathcal{L}$ being any finite set, by imposing lattice structure over the *range set* $\mathcal{L}$. This restricts their possible applications. They also still qualify as decision diagrams, since they use a BDD to represent the Boolean-lattice structure of the *domain set* $\{0,1\}^n$.

3.2 Prior Work

A discrete function $f : X_1 \times \ldots \times X_n \longrightarrow X$, ranging over a finite set X, over a finite set of n discrete variables can be represented by a multi-valued decision diagram (MDD) [163] where the multiplicity of terminal nodes equals $|X|$ and the branching factor of every nonterminal node v labeled with a variable x_i equals $|X_i|$. Binary decision diagrams (BDDs) are special cases of MDDs where the range and all variable domains are binary.

LBFGs and LBBDs are not Decision Diagrams. As mentioned in [135], all *decision diagrams* (DDs), including LVBDDs [77], ROBDDs [32] and MTBDDs [75], model an n-ary Boolean function $f : \{0,1\}^n \rightarrow \{0,1\}$ by the set of sequences of binary decisions on the n Boolean variables (traversed in a particular order) needed to evaluate that function. However, the Boolean-lattice structure of the set $\{0,1\}^n$ is not utilized and appears only implicitly in the relation among BDD paths, obscured by an imposed variable ordering. This is the origin of the dependence of DD size on variable order. A main contribution of this thesis [135] is that, by exploiting the Boolean lattice structure of the set $\{0,1\}^n$, we depart from the BDD orthodoxy. *Unlike LVBDDs [77], which are still binary decision diagrams*, LBBDs [135] are not decision-based. That is why LBBDs are independent of any variable ordering, whereas LVBDDs rely on the existence of a good variable ordering (which implies the need for variable ordering heuristics). In [77], efficiency of LVBDDs for representing finite-valued functions $f : \{0,1\}^n \rightarrow \mathcal{L}$ hinges on the *range* set $\mathcal{L}$ having a distributive lattice structure. On the other hand, our work [135] demonstrated how the Boolean lattice structure of the *domain* $\{0,1\}^n$ may contribute to the efficiency of representation.

Scope of Applications. In [77], imposing a distributive lattice structure over the range set $\mathcal{L}$ restricts its applicability to problems where this is the case, such as alternating finite-state automata, multi-valued logics and abstract interpretation.[1] In [135], for simplicity

[1] These are the specific applications listed in [77].

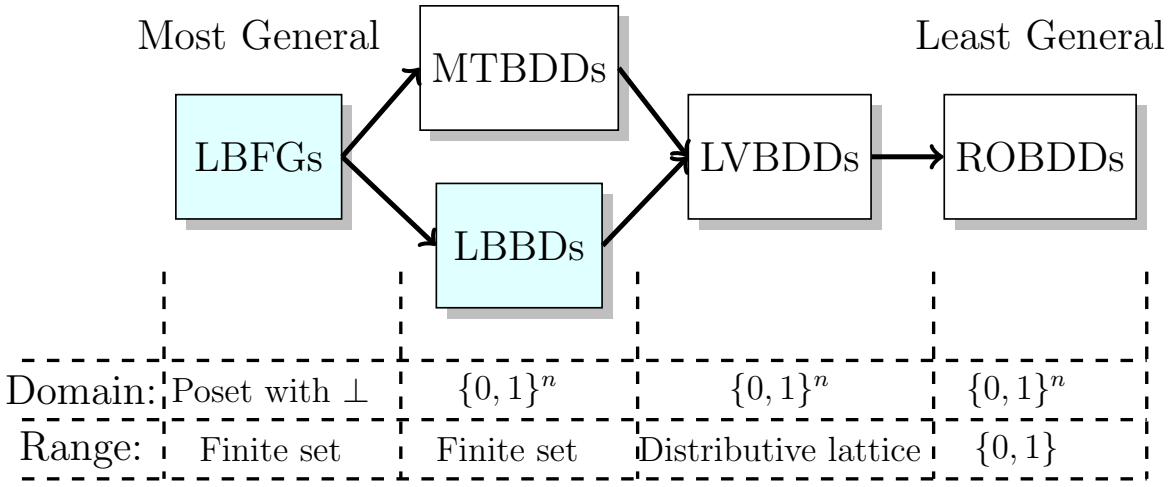


Figure 3.1: Comparison of generality of various representations of finite-valued functions over Boolean variables. ROBDDs, LVBDDs, and MTBDDs are decision diagrams, whereas LBFGs and LBBDs are not decision-based.

of presentation, we focused on *binary-valued* functions $f : \{0,1\}^n \to \{0,1\}$ over Boolean variables. However, the proposed LBBDs apply in a straightforward way to finite-valued functions $f : \{0,1\}^n \to \mathcal{L}$, where $\mathcal{L}$ can be any finite set, not necessarily a lattice. The lattice structure of the domain $\{0,1\}^n$, used by LBBDs [135], is universally common to all functions of the form $f : \{0,1\}^n \to \mathcal{L}$ which does not restrict its applicability in any way. The relative generality of various graph representations is shown in Figure 3.1.

Reductions. LVBDDs [77], as their name implies, generalize BDDs by labeling BDD nodes with values from the range lattice. The reductions used by LVBDDs still work over an underlying BDD and refer only to the lattice structure of the range set $\mathcal{L}$. Whereas reductions used by LBBDs [135] are not constrained by any secondary structure on the range set $\mathcal{L}$.

3.3 Lattice Functions into Finite Sets

Given a partially ordered set (poset) $(X, \preceq)$ with a minimum (and, hence, unique) element $\perp$, let Y^X be the space of functions $f : X \longrightarrow Y$ where Y is some finite set. We now

focus on *lattice-based function graph* (LBFG) representations of functions in Y^X which is advantageous because it enables incremental updates in cases where the function changes with time due to *local transformations*. In all LBFGs we study in the sequel to represent functions, graph vertices are elements of a poset $(X, \preceq)$. Edges in those LBFGs are induced by the partial order inherited from X through transitive reduction. Thus LBFGs edges represent the *covering relation* (Hasse Diagram) of $\preceq$. Therefore, edges will always remain implicit (although they are essential in taming the computational complexity of all symbolic algorithms) and our focus will be on constructing the set of vertices.

An element $a_1 \in X$ which is a *maximal lower bound* (MLB)[2] of another $a_2 \in X$ is denoted by $a_1 \lessdot a_2$. Also, an element $a_1 \in X$ which is a *minimal upper bound* (MUB) of another $a_2 \in X$ is denoted by $a_1 \gtrdot a_2$. A *downward direct set* is a poset where every pair of elements have a (not necessarily unique) lower bound (but not necessarily a greatest lower bound as in a meet semilattice) in the set. The set of minimal upper bounds (i.e., successors) of any element $a \in X$ is an antichain *by the definition of minimal*. For every $a_1, a_2 \in X$, if $a_1 < a_2$ and there is no $a \in X$ such that $a_1 < a < a_2$, then a_1 is said to *cover* a_2, denoted by $a_1 \lessdot a_2$.

All posets X' we construct later to represent functions over X will satisfy the following requirements:

- **Requirement-1**: X will always contain a unique bottom element $\bot \in X$ and, hence, will be a downward directed set. A subset $X' \subseteq X$ of interest will always contain the bottom element $\bot$ and, hence, will also be a downward directed subset of X according to the partial order $\preceq$ inherited from X.
- **Requirement-2**: For every pair of elements $(a_1, a_2) \in X \times X$, a minimal upper bound $a_1 \vee a_2$ may not exist, but if it exists, it must be unique, yielding a least upper bound which is a partial function over $X \times X$.
- **Requirement-3**: If $a_1 \vee a_2$ does not exist, then $a_i \vee a_j$ also does not exist for all

[2]In a general poset, a greatest lower bound need not exist. More generally, a set of MLBs may do.

successors $a_i \succeq a_1$ and $a_j \succeq a_2$.

In this thesis, any poset satisfying these three requirements is called a *hereditary poset*. **Requirement-3** is the main reason that RV with SR-DFAs is efficient on distributed systems, since state transitions induced every event (as represented by a local transformation) are localized to specific subspaces of Ξ. Hence, in a distributed system, events tend to have minimal conflicts. This requirement is also essential for efficiency of symbolic manipulation of LBFGs.

The use of a hereditary poset X' supports the intuition that any two regions of X (represented by elements within X') are *embedded* in (or *generalized* by) a larger region (represented by some other less specific element in X'), and represent *exceptions* to the function value prevailing in that larger region. Thus, a downward directed subset can be viewed as an extensible function representation where any two patches can be embedded in a larger patch. Evaluating a function at any particular point $a \in X$ amounts to taking the limit of a convergent sequence in X', where each element *specializes* (i.e., is more specific than) the preceding one.

The join $\vee$ and meet $\wedge$ operators can be generalized to subsets of X as follows:

$$X' \subseteq X,\ a \in X: \quad X' \vee a \ = \ \{a' \vee a | a' \in X' \text{ and } a' \vee a \text{ exists}\}$$
$$X' \subseteq X,\ a \in X: \quad X' \wedge a \ = \ \{a' \wedge a | a' \in X'\}$$

3.3.1 LBFGs

Theorem 3.1 formally states that a hereditary poset can be compressed or reduced to represent any function $f : X \to Y$. If $X' \subseteq X$, then for every $a \in X$, it is said that $b \in X'$ is a maximal lower bound of a in X', denoted by $b \preccurlyeq_{X'} a$ iff $b \preceq a$ and there is no $c \in X'$ such that $b \preceq c \preceq a$.

Theorem 3.1. *If X is a countable hereditary poset, then for every function $f : X \to Y$,*

ALGORITHM 1: Evaluate $f(a)$

Input: A function $f : X \longrightarrow Y$ where Y is finite and $a \in X$.
Output: The value $f(a)$.

```
1 return f_{X'}(a, ⊥);
```

ALGORITHM 2: Evaluate $f_{X'}(a, u)$

Input: A node $u \in X'$ of a LBFG X' and a variable valuation ξ.
Output: The value $f_{X'}(a, u)$.

```
1 foreach (v ∈ X' : v ≽ u) do
2     if (v ⪯ a) then
3         return f_{X'}(a, v);
4 return f(u);
```

there is a (possibly infinite) ***unique minimal*** *subset $X' \subseteq X$ such that f satisfies the relation:*

$$\forall a \in X : \quad b \in X', b \lessdot_{X'} a \Rightarrow f(a) = f(b)$$

From Theorem 3.1, for every function $f : X \to Y$, there is a subset $X' \subseteq X$ that uniquely determines f. The transitive reduction of this poset is a unique DAG, called a *lattice-based function graph* (LBFG), whose main purpose is to transform the calculation of a maximal lower bound $b \in X'$, $b \lessdot_{X'} a$ for every $a \in X$ to a straightforward graph traversal. In a LBFG, parents of a vertex are the maximal lower bounds of that vertex and its children are its minimal upper bounds. As in domain theory used by denotational semantics of programming languages, a LBFG represents a hierarchy of knowledge where information (i.e., function values) at lower-level elements is less specific than (and is overridden by) information at higher-level elements, but are more widely applicable. Within the set X' of Theorem 3.1, if $a \preceq b$, then $f(b)$ *overrides* $f(a)$ for all $c \in X$ such that $a \preceq c \preceq b$. Algorithm 1 explains how a LBFG representation of an arbitrary finite-valued function f computes the value of f for any valuation of its variables.

Theorem 3.2. *If $a_1 \in X'$ and, for every $a_2 \in X'$ with $a_2 \preceq a_1$, we have $f(a_1) = f(a_2)$, then $X' \setminus \{a_1\}$ provides a more concise representation of f.*

Theorem 3.2 furnishes a recipe for obtaining a reduced set $X' \subseteq X$ that preserves all information about f by starting with X and iteratively eliminating *redundant* elements $a \in X'$. This process eventually terminates (for finite X) at which point, for each $a \in X'$, its successors (i.e., the minimal upper bounds of a in X') yield the *minimal* moves[3] needed to alter the value of f.

Theorem 3.2 essentially says that LBFGs achieve conciseness by grouping together all elements that are successors of a given element[4] in the partial order $\preceq$ and have the same function value and storing that value at the shallowest level possible in the poset X'.

Also, if two elements a_1 and a_2 such that $a_1 \not\preceq a_2$ and $a_2 \not\preceq a_1$ have different function values and also have common extensions (i.e., $a_1 \vee a_2$ exists), then $a_1 \vee a_2$ is used a *resolvent* or an *arbiter* that determines what function value those common extensions actually carry (it could be identical to one of a_1 and a_2 or different from both).

Now, we present theorems stating the three main properties of LBFGs, namely correctness, canonicity and consistency. When a $X' \subseteq X$ is called a LBFG, it is implicitly assumed to be minimal (i.e., reduced in the sense of Theorem 3.2).

Theorem 3.3 (Consistency)**.** *Let X' be a LBFG representing a function $f : X \to Y$. Then for every $a \in X$, if $b, c \in X'$ are maximal lower bounds of a in X' and $b \neq c$, then $f(b) = f(c)$.*

Theorem 3.4 (Canonicity)**.** *Given a function $f : X \to Y$, then any two different LBFGs $X'_1 \subseteq X$ and $X'_2 \subseteq X$ compute two different functions. That is, every function $f : X \to Y$ has precisely one LBFG representation.*

Proof. Assume that the two (minimal) LBFGs X'_1 and X'_2 are different. If $X'_1 = X'_2 = X'$, then we must have $f_1(a) = f_2(a)$ for every $a \in X'$. This means the two LBFGs are identical.

[3]For example, the extra variable assignments, in the case of the semi-lattice of partial variable assignments.

[4]For example, all complete variable valuations that are extensions of the same partial variable valuation, in the case of Ξ^*.

So we now assume $X'_1 = X'_2 \neq X'$ and we now find a value $a \in X$ where the two LBFGs return different values. Let $a \in X'_1 \setminus X'_2$. Also, let $L \subseteq X'_2$ be the set of maximal lower bounds of a in X'_2. Then we either have:

- $f_1(a) \neq f_2(L)$, in which case $f_2(a) = f_2(L) \neq f_1(a)$, or
- $f_1(a) = f_2(L)$, in which case X'_2 is not reduced (a contradiction), since all elements of L are less than a and carry the same value, or
- None of the above, in which case both arguments above apply.

□

Theorem 3.5 (Correctness)**.** *The LBFG X' for a function $f : X \to Y$ returns the correct value of f for every input combination.*

Proof. Assume $a \in X$ is an element of X such that there is a maximal lower bound $b \in X, b \neq a$ of a with $f(a) \neq f(b)$. Moreover, if X' is correctly reduced, then there must be an element $b \preceq c \preceq a$ such that $f(a) = f(c)$, which contradicts that b is a maximal lower bound of a in X'. □

3.3.2 Restriction

Restricting a function $f : X \to Y$ by an element $a \in X$ is equivalent to the function associated with the set $\{a \vee a_1 | a_1 \in X' \text{ and } a \vee a_1 \text{ exists}\}$.

Restriction. Applied to Boolean functions, *restriction* (also called *partial evaluation* or *projection*) tackles cases when a subset $\mathbf{R} \subseteq \mathbf{AP}$ of the propositional variables have values fixed by context. In that case, it is desired to *restrict* the Boolean function (and its LBBD) by that partial valuation to obtain a new LBBD $G|_{\mathbf{R}}$. *Positive restriction*, where all variables $x \in \mathbf{R}$ have value 1, is implemented by noting that a subset $\mathbf{R} \subseteq \mathbf{AP}$ of true variables

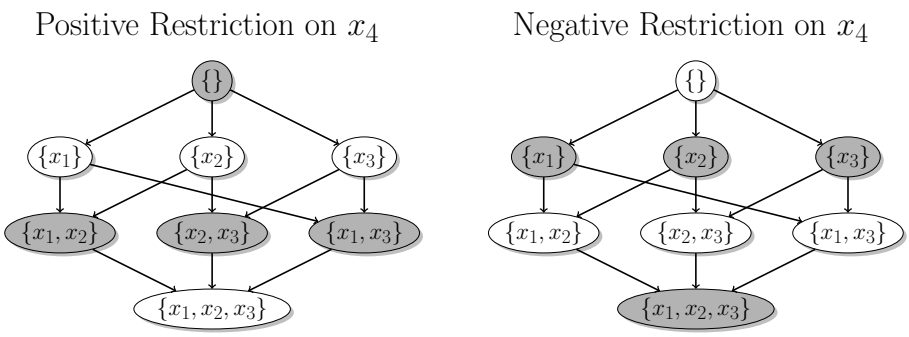


Figure 3.2: Positive and negative restriction of the 4-variable even-parity LBBD over x_4.

corresponds to the upper set $U = \{v \in V | \mathbf{R} \subseteq \mathscr{L}(v)\}$ in G of all graph nodes containing $\mathbf{R}$. All minimal elements of U form an antichain that are then connected to a new root r' of the new LBBD $G|_{\mathbf{R}}$, having $\mathscr{L}|_{\mathbf{R}}(r') = \{\}$. Then for all $v \in U$, we have $\mathscr{L}|_{\mathbf{R}}(v) = \mathscr{L}(v) \setminus \mathbf{R}$. *Negative restriction*, where all variables $x \in \mathbf{R}$ have value 0, is simply implemented by computing the lower set of all LBBD nodes contained in $\mathbf{AP} \setminus \mathbf{R}$, which always includes the root of G. Fig. 3.2 shows the positive and negative restrictions of the even-parity LBBD in Fig. 3.6 on $\mathbf{R} = \{x_4\}$.

3.4 Symbolic Algorithms

Now, we focus on how to construct LBFGs symbolically rather than by reductions applied to the original (possibly exponentially large) poset, such as the Boolean lattice $\{0,1\}^n$. In all the following arguments and algorithms, let (X'_1, f_1) and (X'_2, f_2) be two LBFGs and let $\odot$ be a binary symbolic operation, such as the ones in Section 3.5 for general functions or such as conjunction and disjunction for Boolean-valued functions. Unary operations, such as Boolean negation for Boolean-valued functions, is trivially implemented symbolically by applying it to the value of f at each LBFG vertex $a \in X'_1$. We now study the structure of $(X', f) = (X'_1, f_1) \odot (X'_2, f_2)$.

3.4.1 Co-Stability

Given two functions $f_1, f_2 : X \to Y$, with LBFGs X'_1 and X'_2, and any *pointwise composition* operation $\odot$, how to calculate the LBFG X' corresponding to the composition $f_1 \odot f_2$? For every $a \in X$, there are maximal $a_1 \in X'_1$ and $a_2 \in X'_2$ such that $a_1 \preceq a$ and $a_2 \preceq a$. Conversely, for every $a_1 \in X'_1$ and $a_2 \in X'_2$, there is $a \in X$ such that a_1 and a_2 are maximal lower bounds of a within X'_1 and X'_2, respectively, iff the pair (a_1, a_2) is co-stable. Let $\mathbf{J}(X'_1, X'_2) \subseteq X'_1 \times X'_2$ be the set of all pairs $(a_1, a_2) \in X'_1 \times X'_2$ such that $a_1 \vee a_2$ exists. A pair $(a_1, a_2) \in \mathbf{J}(X'_1, X'_2)$ is *co-stable* iff a_1 and a_2 are maximal lower bounds of $a_1 \vee a_2$ within X'_1 and X'_2, respectively. Let $\mathbf{CO}(X'_1, X'_2) \subseteq \mathbf{J}(X'_1, X'_2)$ be the set of all co-stable pairs.

A pair $(a_1, a_2) \in X'_1 \times X'_2$ is said to be *co-stable* iff $a_1 \vee a_2$ exists and:

$$\begin{aligned} &\forall b_1 \succcurlyeq a_1 : \;\; b_1 \not\preceq a_1 \vee a_2 \\ &\forall b_2 \succcurlyeq a_2 : \;\; b_2 \not\preceq a_1 \vee a_2 \end{aligned} \tag{3.1}$$

Co-stability of two LBFG nodes $(a_1, a_2) \in X'_1 \times X'_2$ means that every pair of maximal paths $(c_1, \ldots, c_m)$ and $(d_1, \ldots, d_n)$ in X'_1 and X'_2 terminating at a_1 and a_2, resp., has a corresponding maximal path[5] $((c_1, d_1), \ldots, (c_m, d_n))$ in $X'_1 \times X'_2$ terminating at (a_1, a_2). Alternatively, lack of co-stability of (a_1, a_2) means that there is no $a \in X$ where the value of $f_1 \odot f_2$ is determined by the combination of $f_1(a_1)$ and $f_2(a_2)$. Hence, for any co-stable pair $a = (a_1, a_2) \in X'_1 \times X'_2$, the value of $f(a)$ can be calculated as $f(a) = f_1(a_1) \odot f_2(a_2)$.

A partial order $\sqsubseteq$ can be defined over the Cartesian product $\mathbf{J}(X'_1, X'_2)$ as follows: Given two pairs $(a_1, a_2), (a_3, a_4) \in \mathbf{J}(X'_1, X'_2)$, we have $(a_1, a_2) \sqsubseteq (a_3, a_4)$ iff $a_1 \preceq a_3$ and $a_2 \preceq a_4$. Note that this includes the special cases $a_1 = a_3$ and $a_2 = a_4$. Note also that the covering relation of that partial order is exactly the Cartesian product of the covering relations of the two constituent partial orders. An *order homomorphism* $h : \mathbf{J}(X'_1, X'_2) \to X'_3 \subseteq X$ can be

[5] Repetitions are possible.

defined as follows:

$$\forall (a_1, a_2) \in \mathbf{J}(X_1', X_2') : h(a_1, a_2) = a_1 \vee a_2$$

In general, the partial order on a homomorphic image $h(X_1)$ of a poset $(X_1, \preceq_1)$ into another poset $(X_2, \preceq_2)$ can be stronger than $\preceq_1$, in the sense that it will contain the same orders as the original poset (perhaps collapsing some), but may also contain more orderings not present in the original poset (i.e., a homomorphism may linearize certain incomparable pairs). By monotonicity of the join operator, we have:

$$(a_1, a_2) \sqsubseteq (a_3, a_4) \Rightarrow h(a_1, a_2) \preceq h(a_3, a_4)$$

Two pairs $(a_1, a_2), (a_3, a_4) \in \mathbf{J}(X_1', X_2')$ are equivalent, denoted by $(a_1, a_2) \sim (a_3, a_4)$, iff $a_1 \vee a_2 = a_3 \vee a_4$. If a pair $(a_1, a_2) \in \mathbf{J}(X_1', X_2')$ is co-stable, this basically means that both a_1 and a_2 have joint complete extensions not subsumed by the children (offspring or descendants) of either of them and, hence, the value of f for these complete extensions must be made available at $a_1 \vee a_2$. As mentioned before, within the set X' of Theorem 3.1, if $a_1 \preceq a_2$, then $f(a_2)$ *overrides* $f(a_1)$ by virtue of the supremum. Thus, even if $a_1 \vee a_2$ contains extensions that fall under any of the descendants of a_1 or a_2, the value $f(a_1 \vee a_2)$ will be overridden appropriately when descendant pairs of (a_1, a_2) are considered.

The quotient set (i.e., the set of all equivalence classes) is denoted by $\mathbf{CO}(X_1', X_2')/\sim$. Let $[a] = \{b \in \mathbf{CO}(X_1', X_2') | a \sim b\}$ denote an element of $\mathbf{CO}(X_1', X_2')/\sim$ for some co-stable pair $a = (a_1, a_2) \in \mathbf{J}(X_1', X_2')$. Note that $\mathbf{CO}(X_1', X_2')$ is nonempty, since the roots $\perp_1$ and $\perp_2$ are co-stable.

Lemma 3.6. *Every vertex of* $X' = X_1' \odot X_2'$ *is an equivalence class of co-stable pairs* $(a_1, a_2) \in \mathbf{CO}(X_1', X_2')$. *That is,* $X' \subseteq \mathbf{CO}(X_1', X_2')/\sim$.

However, not every class in $\mathbf{CO}(X_1', X_2')/\sim$ is a vertex in $X' = X_1' \odot X_2'$ because some classes violate the LBFG alternation condition: $a \succcurlyeq b$ in $X' \Rightarrow f(a) \neq \neg f(b)$. Lemma 3.7 gives the

conditions for existence of covering edges in a LBFG $X' = X'_1 \odot X'_2$ in terms of alternations between walkers over X'_1 and X'_2. It uses the following notation: given a DAG $G = (V, E)$, let $x \xrightarrow{(v_1,\ldots,v_M)} y$ denote a path in G with $v_1 = x$ and $v_M = y$, where $x, y \in V$.

Lemma 3.7. *Let $[a], [b] \in \mathbf{CO}(X'_1, X'_2)/\sim$ be two vertices in $X' = X'_1 \odot X'_2$ with $a = (a_1, a_2)$ and $b = (b_1, b_2)$. Then there is a covering edge $([a], [b]) \in E$ in X' iff:*

- $f_1(b_1) \odot f_2(b_2) \neq f_1(a_1) \odot f_2(a_2)$
- *For all paths $a_1 \xrightarrow{(c_1,\ldots,c_M)} b_1$ in X'_1 and $a_2 \xrightarrow{(d_1,\ldots,d_N)} b_2$ in X'_2, there is no (c_i, d_j) which is a co-stable pair with $f_1(x_i) \odot f_2(y_j) \neq f_1(a_1) \odot f_2(b_2)$.*

This condition is hard to check for every co-stable pair reachable from $(\bot_1, \bot_2)$. Therefore, we add a covering edge $([u], [v])$ to X' if the following condition is satisfied:

- There is at least one pair of paths $a_1 \xrightarrow{(c_1,\ldots,c_M)} b_1$ in X'_1 and $a_2 \xrightarrow{(d_1,\ldots,d_N)} b_2$ in X'_2 such that there is no (c_i, d_j) which is a co-stable pair with $f_1(c_i) \odot f_2(d_j) \neq f_1(a_1) \odot f_2(a_2)$.

This condition results in redundant covering edges in X' that can then be removed using *transitive reduction* of X' (i.e., removing edges without affecting the reachability relation of a graph).[6] The transitive reduction of a finite DAG is unique which preserves canonicity of LBFGs.

Note that $\bot$ must be a member of all subsets $X' \subseteq X$ used to represent functions $f : X \to Y$. Note also that $(\bot, \bot)$ is always a co-stable pair for all admissible sets $X'_1, X'_2 \subseteq X$. However, $\bot \in X'_1$ is not necessarily co-stable with any other elements $a_2 \in X'_2$, and vice versa. Thus, there is no guarantee that $X'_1 \subseteq (X'_1 \odot X'_2)$ or $X'_2 \subseteq (X'_1 \odot X'_2)$ for any pointwise binary operation $\odot$. If $a_1 \in X'_1$ has a narrow *gap* to its minimal upper bounds, that will reduce its chance to be co-stable with other elements $a_2 \in X'_2$, and conversely, the wider the gap between an element and its minimal upper bounds, the larger the territory it reigns over and

[6] For LBFGs, transitive reduction can be performed in time $O(|X'|B^2)$, where B is the average branching factor (number of outgoing edges) of a LBFG node.

the larger its co-stability domain will become.

Lemma 3.8. *Every $a \in X'$ corresponding to $f_1 \odot f_2$ is generated by an equivalence class of co-stable pairs $(a_1, a_2) \in \mathbf{J}(X_1', X_2')$. That is, Xi' is isomorphic to a subset of $\boldsymbol{CO}(X_1', X_2')/\sim$.*

Lemma 3.9 formally states that the function value of an equivalence class in $\mathbf{CO}(X_1', X_2')/\sim$ is well-defined.

Lemma 3.9. *For any two co-stable pairs (a_1, a_2) and (b_1, b_2), if $a_1 \vee a_2 = b_1 \vee b_2$, then $f_1(a_1) \odot f_2(a_2) = f_1(b_1) \odot f_2(b_2)$.*

Proof. Let $a = a_1 \vee a_2 = b_1 \vee b_2$. By co-stability of the pairs (a_1, a_2) and (a_1, a_2), we have: both a_1 and b_1 are maximal sets contained within a. Similarly, both a_2 and b_2 are maximal sets contained within a. Then by Lemma 3.15, we have $f_1(a_1) = f_1(b_1)$ and $f_2(a_2) = f_2(b_2)$. □

Co-stable pairs are the maximal elements within their equivalence classes in the product partial order over $\mathbf{J}(X_1', X_2')$. If $(a_1, a_2) \in X_1' \times X_2'$ is not a co-stable pair (i.e., not maximal within its equivalence class) and all its successors are in the same equivalence class, it is called an *interior pair*. A co-stable pair is covered only by pairs outside its own equivalence class. A non-co-stable pair may be simultaneously covered by pairs from its own equivalence class and from other equivalence classes.

Lemma 3.10. *Every equivalence class contains at least one co-stable pair.*

Proof. The absence of co-stable pairs from an equivalence class would imply the existence of a cycle inside that equivalence class which contradicts the fact that the operand graphs are DAGs. Assume all pairs in an equivalence class are not co-stable. Then each vertex has at least one transition into another pair of the same class. This implies that there is a cycle, contradicting dag assumption. □

Lemma 3.11. *In a LBFG X', members of any co-stable pair $(a_1, a_2) \in \boldsymbol{CO}(X', X')$ must have the same function value:*

$$f(a_1) = f(a_2)$$

Proof. If a_1 and a_2 are co-stable, then there is $a \in X$ such that they are both maximal lower bounds of a within X', in which case the value must be the same due to consistency of LBFGs. □

3.4.2 Structure of Equivalence Classes

The main challenge of symbolic manipulation of LBFGs is efficient construction and representation of the set $\mathbf{J}(X'_1, X'_2)/\sim$. Profile size of the set $\mathbf{J}(X'_1, X'_2)$ relative to size of the set $X'_1 \times X'_2$.

The key to efficiency is representing equivalence classes of $\mathbf{J}(X'_1, X'_2)$ in terms of Cartesian products of subsets of X'_1 and X'_2 because, in that case, the size of the set $\mathbf{J}(X'_1, X'_2)$ will be on average equal to $|X'_1|.|X'_2|$, whereas a product decomposition of its equivalence classes will result in a much smaller complexity $\sqrt{|X'_1|.|X'_2|}$.

Theorem 3.12 (Representing Equivalence Classes)**.** *Every equivalence class $A \in \mathbf{J}(X'_1, X'_2)/\sim$ can be written as a union of Cartesian products of subsets of X'_1 and X'_2 as follows:*

$$A = \bigcup_{a_i} A_{a_i} \times B_{a_i}$$

where $a_i = (a_{i,1}, a_{i,2})$ runs over all minimal elements of A under the Cartesian product order, and $A_{a_i} \subseteq X'_1$ and $B_{a_i} \subseteq X'_2$ are given by:

$$A_{a_i} = \{a \in X'_1 | a_{i,1} \preceq a \preceq (a_{i,1} \vee a_{i,2})\}$$

$$B_{a_i} = \{a \in X'_2 | a_{i,2} \preceq a \preceq (a_{i,1} \vee a_{i,2})\}$$

Proof. For any pairs $(a_{i,1}, a_{i,2}), (a_1, a_2) \in X_1' \times X_2'$, which are not necessarily co-stable, we have:

$$(a_{i,1}, a_{i,2}) \sqsubseteq (a_1, a_2) \Rightarrow (a_{i,1} \vee a_{i,2}) \preceq (a_1 \vee a_2)$$

$$\left.\begin{array}{l} a_1 \preceq (a_{i,1} \vee a_{i,2}) \\ a_2 \preceq (a_{i,1} \vee a_{i,2}) \end{array}\right\} \Rightarrow (a_1 \vee a_2) \preceq (a_{i,1} \vee a_{i,2})$$

So, together, both imply that $(a_{i,1}, a_{i,2}) \sim (a_1, a_2)$. This is the same as saying that $A_{a_i} \times B_{a_i}$ is a subset of the *coset* (i.e., the equivalence class) of $[\![(a_{i,1}, a_{i,2})]\!]$, where:

$$A_{a_i} = \{a \in X_1' | a_{i,1} \preceq a \preceq (a_{i,1} \vee a_{i,2})\}$$

$$B_{a_i} = \{a \in X_2' | a_{i,2} \preceq a \preceq (a_{i,1} \vee a_{i,2})\}$$

Thus, $[\![a]\!]$ is a union of rectangles $A_{a_i} \times B_{a_i}$ as a_i ranges over $[\![a_i]\!]$. However, not all rectangles are needed, since we have:

$$\left.\begin{array}{l} (a_{i,1}, a_{i,2}) \sqsubseteq (a_{j,1}, a_{j,2}) \\ (a_{i,1}, a_{i,2}) \sim (a_{j,1}, a_{j,2}) \end{array}\right\} \Rightarrow A_{a_j} \times B_{a_j} \subseteq A_{a_i} \times B_{a_i}$$

Thus, the union of rectangles $A_{a_i} \times B_{a_i}$ over minimal elements of an equivalence class are sufficient to reconstruct it, since all non-minimal pairs are redundant. □

Given a poset $(X, \preceq)$, the set of minimal upper bounds of a subset $S \subseteq X$ is denoted by $\sqcup S$ and is defined by:

$$\sqcup S = \{x \in X \backslash S | x \text{ is a minimal upper bound of } S\}$$

Thus, by Theorem 3.12, starting from a minimal element within an equivalence class, the two-dimensional search (which has quadratic complexity) may proceed as two independent one-dimensional searches (which have linear complexities). The LBFG obtained from Algorithm 3 is not necessarily reduced, since there are orderings among classes in $\mathbf{J}(X_1', X_2')/\sim$ not implied by the Cartesian product order relation $\sqsubseteq$, and also because two consecutive classes in

ALGORITHM 3: Apply($X_1', X_2', \odot$)

Input: $\bot_1 \in X_1'$ and $\bot_2 \in X_2'$ are the roots of X_1' and X_2', resp.
Output: LBFG $X' = X_1' \odot X_2'$

1 $X' \leftarrow \{\}$, **Visited** $\leftarrow \{(\bot_1, \bot_2)\}$;
2 **Advance**($X_1', X_2', \odot, \bot_1, \bot_2,$ **Visited**, $\{\bot_1\}, \{\bot_2\}$);

the Cartesian product order may possess the same composite function value. Therefore, Algorithm 3 needs to be followed by two reductions: (1) valued-based reduction which guarantees that adjacent vertices in the result LBFG have different function values, and (2) transitive reduction.

Inspired by Theorem 3.12, Algorithm 3 and Subroutine 4 implement the symbolic manipulation algorithm of LBFGs for all binary composition operations $\odot$. **Advance** is recursive and is called on the roots of the operand LBFGs. Algorithm 3 has worst-case time complexity $O(NM)$ where N and M are the sizes of the operand LBFGs. LBFGs produced by Algorithm 3, as its stands, need to go through reduction due to the unnecessary creation of *spurious vertices*.

Given $(a_i, a_j) \in X_1' \times X_2'$, define two operators $\partial_1^{i,j}$ and $\partial_2^{i,j}$ as follows:

$$\partial_1^{i,j} a_i = \sqcup_{X_1'} \{a \in X_1' | a_i \preceq a \preceq (a_i \vee a_j)\}$$

$$\partial_2^{i,j} a_j = \sqcup_{X_2'} \{a \in X_2' | a_j \preceq a \preceq (a_i \vee a_j)\}$$

Note that the set $\{a \in X_1' | a_i \preceq a \preceq (a_i \vee a_j)\}$ is the *closed interval* $[a_i, a_i \vee a_j]$.

Subroutine 4 makes two implicit assumptions about every pair of elements $(a_1, a_2) \in X \times X$ of the underlying poset X:

- A minimal upper bound $a_1 \vee a_2$ may not exist, but if it exists, it must be unique, yielding a least upper bound which is a partial function over $X \times X$.
- If $a_1 \vee a_2$ does not exist, then $a_i \vee a_j$ also does not exist for all successors $(a_i, a_j) \sqsubseteq$

Subroutine 4: Advance($X'_1, X'_2, \odot, a_i, a_j, \textbf{Visited}, A_i, A_j$)

Input: $\bot_1 \in X'_1$ and $\bot_2 \in X'_2$ are the roots of X'_1 and X'_2, resp.
Output: LBFG $X' = X'_1 \odot X'_2$

```
1  if (a_i ∨ a_j ∉ X') then
2  |  X' ← X' ∪ {a_i ∨ a_j};
3  |  f(a_i ∨ a_j) = f_1(a_i ∨ a_j) ⊙ f_2(a_i ∨ a_j);
4  A_i^+ = ∂_1^{i,j} A_i;
5  A_j^+ = ∂_2^{i,j} A_j;
6  foreach (a ∈ A_i^+) do
7  |  if ((a∨a_j) exists and (a, a_j) ∉ Visited) then
8  |  |  Visited ← Visited ∪ {(a, a_j)};
9  |  |  Advance( X'_1, X'_2, ⊙, a, a_j, Visited, {a}, A_j^+ );
10 foreach (a ∈ A_j^+) do
11 |  if ((a_i∨a) exists and (a_i, a) ∉ Visited) then
12 |  |  Visited ← Visited ∪ {(a_i, a)};
13 |  |  Advance( X'_1, X'_2, ⊙, a_i, a, Visited, A_i^+, {a} );
```

(a_1, a_2).[7]

3.5 Function Composition

3.5.1 Disjoint-Union Composition

Composition of functions defined over disjoint sets is called *disjoint-union composition*, denoted by $\uplus$. Given a set $\{f_1, \ldots, f_n\}$ of functions $f_i : X_i \to Y$, where $X_i \cap X_j = \emptyset$ for all $i \neq j$, their disjoint union composition $f_{1,\ldots,n} = f_1 \uplus \ldots \uplus f_n$ is given by:

$$\forall x \in \cup_{k=1}^{n} X_k : x \in X_i \Rightarrow f_{1,\ldots,n}(x) = f_i(x)$$

Note that disjoint-union composition is both associative and commutative. The disjoint-union composition is crucial to representing the transition relation of a SR-DFA. For each predicate p, the set of all transitions outgoing from a given (complete) SR-DFA state is

[7] This is the main reason that RV with SR-DFAs is efficient on distributed systems, since state transitions induced every event (as represented by a local transformation) are localized to specific subspaces of Ξ. Hence, in a distributed system, events tend to have minimal conflicts.

described by a complete set of mutually exclusive equality-logic formulas. A set of equality-logic formulas $\{f_1, \ldots, f_n\}$ is a partition of Ξ if and only if $f_1 \vee \ldots \vee f_n$ is identically true, and $f_i \wedge f_j$ is identically false for all $1 \leq i < j \leq n$. Since each f_i is a map $f_i : \Xi \to \{0, 1\}$, all members of a partition can be composed into a product function $f : \Xi \to Y$, where $Y \subset \{0, 1\}^n$ is the set of one-hot bit-vectors. Each one-hot bit vector is an indicator function $\iota : Q \to \{0, 1\}$ that represents a singleton subset of Q. Thus, we have proved the following theorem:

Theorem 3.13. *The disjoint-union composition of a set of elementary partial functions associated with a complete set of mutually exclusive equality-logic formulas is a well-defined total function.*

In the representation of SR-DFA transition relation, these X_k's are disjoint subsets of the Boolean lattice $2^{EL(p)}$ over the set of all equality-logic atoms consisting of equalities of a variable symbol and an argument of a predicate p. These disjoint subsets are models of mutually exclusive equality-logic formulas.

3.5.2 Product (or Concatenation) Composition

Given two functions $f_1 : X \to Y_1$ and $f_2 : X \to Y_2$ over the same domain X, the *product* (or *concatenation*) *composition* $f = f_1 \otimes f_2 : X \to Y_1 \times Y_2$ can be defined as follows:

$$\forall x \in X : f(x) = (f_1(x), f_2(x))$$

A useful case of product composition is concatenating two word-valued functions over a poset. This is used in converting ensemble states into automaton states during specification mining.

3.5.3 Union Composition

Given two functions $f_1 : X \to Y$ and $f_2 : X \to Y$ over the same domain X, the *union composition* $f = f_1 \cup f_2 : X \to Y$ can be defined as follows:

$$\forall x \in X : f(x) = \{f_1(x), f_2(x)\}$$

This is useful in constructing the language associated with an ensemble state by merging languages associated with its preceding ensemble states (after concatenation with their respective outgoing transition events).

3.6 Local (Pointwise) Transformations

Of particular importance is the representation of transformations $\delta : Y_1^X \longrightarrow Y_2^X$. A transformation is *local* iff it can be represented as a map $\delta' : X \to Y_2^{Y_1}$, where for every $x \in X$, we have $\delta'(x) : Y_1 \to Y_2$ is called the *fiber* of δ' at x. Such transformations are *local* in nature, since the transformed value of a function at a given point $x \in X$ depends only on the value of the function at that point. If $Y_1 = Y_2 = Y$ is a finite set of states, then each fiber $\delta'(x)$ can be thought of as a deterministic transition function over Y. Moreover, the action of a local transformation $\delta : Y^X \to Y^X$ on a function $f : X \to Y$ is given by the composition operation $f' = \delta(f) = \delta \odot f$, called the *contraction* or *section* of δ by f, where $f'(x) = \delta'(x)(f(x))$.

For every $y_1 \in Y_1$, let $c_{y_1} \in Y_1^X$ be a constant function such that, for every $x \in X$, $c_{y_1}(x) = y_1$. Let $\{\delta_{y_1} \in Y_2^X | y_1 \in Y_1, \delta_{y_1} = \delta(c_{y_1})\}$ be the set of transformations of all constant functions. We now seek a representation of δ in terms of this basis. Each δ_{y_1} can be thought of as a local transformation of the form $\delta_{y_1} : X \to Y_2^{\{y_1\}}$. Thus, given a finite set $Y_1 = \{y_1, \ldots, y_n\}$, δ' is merely the composition given by $\delta' = \delta_{y_1} \otimes \ldots \otimes \delta_{y_n}$, where the pointwise composition

operator is the disjoint-union composition of fibers.

3.6.1 Composition of Local Transformations

Let $h_i : X \to Y^Y$, with $1 \leq i \leq n$, be n local transformation over the space Y^X of functions $f : X \to Y$. Let $[1, n]^*$ be the monoid of all words $\mathbf{w} = (i_1, \ldots, i_L)$ of any finite length $L \geq 0$, where $i_k \in \{1, \ldots, n\}$ for all $1 \leq k \leq L$. Let $h_{\mathbf{w}} : X \to Y^Y$ be the composition $h_{i_L} \odot \ldots \odot h_{i_1}$ where the pointwise composition operator is the usual function composition of functions $f_1, f_2 : Y \to Y$.

3.7 Real-Valued Lattice Functions

Let $\mathbb{R}^X$ be the space of real-valued functions $f : X \longrightarrow \mathbb{R}$ over a poset X (with minimum element $\perp$). $\mathbb{R}^X$ is a real vector space under pointwise addition and scalar multiplication. An *indicator function* $\iota : X \longrightarrow \{0, 1\}$ is a special and useful real-valued function that will be used to generate other functions in $\mathbb{R}^X$. Let $\mathcal{B}^X \subset \mathbb{R}^X$ be the set of indicator functions over X. An important application of such functions is *Dempster-Shafer theory* of belief functions defined on Boolean algebras (power sets) of events.

3.7.1 Graphical Representations

Given a function $f : X \longrightarrow \mathbb{R}$ and a tolerance parameter $\epsilon > 0$, it is desired to devise a graph representation $\mathcal{G}_f$ of f that returns the value of f within accuracy ϵ for any $a \in X$. The devised graph representation needs to satisfy the following criteria: (1) It is canonical, (2) compositional (i.e., can be symbolically constructed from simpler functions by composition).

What symbolic operations to support? In addition to vector space addition and scalar multiplication, many *pointwise operations* and *pointwise predicates* over $\mathbb{R}^X$ need to be supported. A binary pointwise operation $\mathfrak{h}$ is a mapping $\mathfrak{h} : \mathbb{R}^X \times \mathbb{R}^X \longrightarrow \mathbb{R}^X$. On the other hand, a pointwise binary predicate $\mathfrak{g}$ is a mapping $\mathfrak{g} : \mathbb{R}^X \times \mathbb{R}^X \longrightarrow \mathcal{B}^X$. Example pointwise binary predicates are $f_1 > f_2$ and $f_1 = f_2$. Two important (nonlinear) operations are the binary least upper bound (or pointwise maximum) $f_1 \vee f_2$ and greatest lower bound (or pointwise minimum) $f_1 \wedge f_2$ of any two functions $f_1, f_2 \in \mathbb{R}^X$. We use the symbols $\vee$ and $\wedge$ to, respectively, represent the maximum and minimum operations over $\mathbb{R}^X$, since $\mathbb{R}^X$ becomes a (vector) lattice with these operations as the lattice join and meet, respectively. This lattice structure over $\mathbb{R}^X$ is induced by the pointwise ordering:

$$f \leq g \Leftrightarrow \forall a \in X : f(a) \leq g(a)$$

By associativity, the maximum $f_1 \vee \ldots \vee f_N$ and minimum $f_1 \wedge \ldots \wedge f_N$ of N functions are well-defined. To quantify degrees of approximation, the set $\mathbb{R}^X$ can be turned into a normed vector lattice using the supremum norm:

$$\forall f \in \mathbb{R}^X :\| f \| = \sup_{a \in X} |f(a)|$$

The central idea in a lattice-based representation is that co-stability of operand graph nodes is a necessary condition for the resulting graph nodes regardless of the actual operation being implemented.

3.8 Multi-Variable Functions

3.8.1 The Semi-Lattice of Partial Variable Valuations

We now study the most important poset from the viewpoint of RV. Let $\mathcal{X} = \{x_1, \ldots, x_n\}$ be a finite set of variables ranging over (possibly different, finite or countably infinite) domains $\{|\mathsf{S}|_1, \ldots, |\mathsf{S}|_n\}$. Also, let Ξ be the product space $|\mathsf{S}|_1 \times \ldots \times |\mathsf{S}|_n$ and let $|\mathsf{S}| = \bigcup_{i=1}^{n} |\mathsf{S}|_i$. Then, every point $\xi \in \Xi$ can alternatively be considered as a variable valuation $\xi : \mathcal{X} \to |\mathsf{S}|$ that assigns to each variable a value compatible with its data type (i.e., $x \in \mathcal{X}_i \Rightarrow \xi(x) \in |\mathsf{S}|_i$). Let Ξ^* be the poset of all *partial functions* $\xi^* : \mathcal{Y} \to |\mathsf{S}|$ for all $\mathcal{Y} \subseteq \mathcal{X}$. A partial function $\xi^* : \mathcal{Y} \to |\mathsf{S}|$ represents a partial variable valuation with dimension $|\mathcal{Y}|$ and co-dimension $|\mathcal{X}| - |\mathcal{Y}|$. A partial valuation $\xi_1^* : \mathcal{Y}_1 \to |\mathsf{S}|$ is less specific (i.e., binds less variables) than another $\xi_2^* : \mathcal{Y}_2 \to |\mathsf{S}|$, denoted by $\xi_1^* \preceq \xi_2^*$, iff for every $y \in \mathcal{Y}_1$, $\xi_1^*(y) = \xi_2^*(y)$. It is also said that ξ_2^* *extends* ξ_1^*. Let $\perp$ be the bottom (i.e., minimum or least) element of Ξ^* corresponding to $\mathcal{Y} = \emptyset$. Then $\perp \preceq \xi^*$ for every $\xi^* \in \Xi^*$. There is no top or maximum element in Ξ^*. However, all *maximal chains* in Ξ^* starting at $\perp$ have the same length equal to $|\mathcal{X}|$.

Theorem 3.14. *Ξ^* is a meet (or lower) semilattice, but not a lattice.*

Proof. If ξ_i^*, with $i = 1, 2$, stand for $f_i : \mathcal{Y}_i \to Y$, then the join or supremum ($\vee$) and meet or infimum ($\wedge$) operators are defined as follows: $\xi_1^* \vee \xi_2^*$ is defined iff for all $x \in \mathcal{Y}_1 \cap \mathcal{Y}_2$, we have $f_1(x) = f_2(x)$. In that case, $\xi_1^* \vee \xi_2^*$ stands for $f : \mathcal{Y} \to Y$ where $\mathcal{Y} = \mathcal{Y}_1 \cup \mathcal{Y}_2$ and:

$$f(x) = \begin{cases} f_1(x) & \text{if } x \in \mathcal{Y}_1 \\ f_2(x) & \text{otherwise} \end{cases}$$

Thus, the join ($\vee$) operator is not defined over all of $\Xi^* \times \Xi^*$. The meet $\xi_1^* \wedge \xi_2^*$ of two elements stands for $f : \mathcal{Y} \to Y$ where $\mathcal{Y} \subseteq \mathcal{Y}_1 \cap \mathcal{Y}_2$ is the subset where $f_1(x) = f_2(x)$. In that case, $f(x) = f_1(x)$ for all $x \in \mathcal{Y}$. □

In a meet semilattice, we can always talk about greatest lower bounds (*infima*), but can only have minimal upper bounds (a greatest lower bound or a supremum need not always exist). Note that complete information about the partial order $\preceq$ is encoded in the meet operator $\wedge$, since $x \preceq y \Leftrightarrow x \wedge y = x$.

A function $f : \Xi \to Y$ can be *recursively* extended to a function $f^* : \Xi^* \to Y$ as follows:

- $f^*(\xi^*) = y$ if, for all ξ which is a complete extension of ξ^* such that there is no extension $\xi^* \preceq \xi' \preceq \xi$ with $\lambda^*(\xi') \neq q$, we have $\lambda(\xi) = y$.

For every $\xi^* \in \Xi^*$, let $[\xi^*] \subseteq \Xi$ be the set of all complete extensions of ξ^*:

$$[\xi^*] = \{\xi \in \Xi | \xi^* \preceq \xi\}$$

In a LBFG $\Xi' \subseteq \Xi^*$, the region $[\![\xi^*]\!] \subseteq \Xi$ be the set of all complete extensions of ξ^* excluding all complete extensions of its children:

$$[\![\xi^*]\!] = [\xi^*] \setminus \cup_{\xi' \succ \xi^*} [\xi']$$

Note that:

$$\xi_1^* \preceq \xi_1^* \implies [\xi_1^*] \supseteq [\xi_1^*]$$

Canonicity and Variable Order. Unlike BDDs, LBFGs do not impose a variable ordering. Yet, LBFGs are canonical representations. However, canonicity relies on the assumption that the set $\Xi' \subseteq \Xi^*$ is chosen in such a way that successors of every $\xi^* \in \Xi'$ include all possible ways to alter or mutate the value of $f^*(\xi^*)$. If we drop that requirement, we may be able get a much more concise LBFG representation at the expense of losing canonicity, which can be restored if we impose a variable ordering similar to BDDs.

3.9 The Boolean Lattice

An important example is the space $Y^{\{0,1\}^n}$ of functions $f : \{0,1\}^n \longrightarrow Y$ over a set X_n of n Boolean variables, where Y is some finite set. The set $\{0,1\}^n$ is a Boolean algebra of bit-vectors partially ordered bitwise. It is isomorphic to the power set 2^{X_n} ordered by set inclusion. Therefore, it is also a downward directed set with a unique bottom element $\bot_n$ given by the all-zero bit-vector (or, equivalently, the empty subset of X_n). Thus, there are two ways to represent a function $f : \{0,1\}^n \longrightarrow Y$:

- Working with the intrinsic partial order over the set $\{0,1\}^n$. In this case, $f : \{0,1\}^n \longrightarrow Y$ can be represented by another simpler function $f^* : V_n \longrightarrow Y$, where $V_n \subseteq \{0,1\}^n$ is a downward directed set containing $\bot_n$.
- *Working with the poset of partial Boolean variable valuations.* This representation is not discussed in this thesis.

3.9.1 Intrinsic Representation

An n-ary Boolean function g is a mapping $g : \{0,1\}^n \to \{0,1\}$. Let $G = (V, E, r, L, T, \mathbf{AP})$ be a rooted finite DAG with a vertex set V, a root vertex $r \in V$, an edge set $E \subseteq V \times V$, and a finite set of atomic propositions $\mathbf{AP}$. Unlike binary decision diagrams (BDDs) [32], instead of terminal nodes for $\mathsf{0}$ and $\mathsf{1}$, a map $T : V \to \{\bot, \top\}$ assigns to each vertex $v \in V$ a truth value $T(v)$ such that $(v_1, v_2) \in E \Rightarrow T(v_1) = \neg T(v_2)$. A labeling map $L : V \to 2^{\mathbf{AP}}$ associates with every vertex $v \in V$ a subset $L(v)$ of $\mathbf{AP}$ such that $L(r) = \emptyset$ and $(v_1, v_2) \in E \Rightarrow L(v_1) \subseteq L(v_2)$. Given a variable assignment $\kappa : \mathbf{AP} \to \{0,1\}$, a vertex $v \in V$ *satisfies* κ, denoted by $v \sqsubseteq \kappa$, iff $\forall p \in L(v) : \ \kappa(p) = 1$. Otherwise, v *conflicts* with κ, denoted by $v \not\sqsubseteq \kappa$. For a variable assignment κ, the value of a Boolean function g can be computed as follows: start at the root and set $v = r$. If there is $u \in V$ such that $(v, u) \in E$

and $u \sqsubseteq \kappa$, then set $v = u$ and repeat the test. Otherwise, the procedure completes and returns $T(v)$. Like BDDs, every Boolean function admits a LBBD representation (proofs omitted for lack of space) and logical connectives ($\vee$, $\wedge$, $\neg$) correspond to graph operations.

A function $f : \{0,1\}^n \to \{0,1\}$ is *conceptually* represented by a rooted finite DAG $G = (V, E, r, \mathscr{L}, \mathscr{T}, \mathbf{AP})$ with a vertex set $V = \{0,1\}^n$, a root vertex $r \in V$ corresponding to the zero vector $\mathbf{0}^n$, an edge set $E \subseteq V \times V$ encoding the covering relation of the Boolean lattice $\{0,1\}^n$, and a finite set of atomic propositions $\mathbf{AP}$. Unlike BDDs [32], instead of terminal nodes for $\mathsf{0}$ and $\mathsf{1}$, a map $\mathscr{T} : V \to \{0,1\}$ assigns to each vertex $v \in V$ a truth value $\mathscr{T}(v) \in \{0,1\}$. A labeling map $\mathscr{L} : V \to 2^{\mathbf{AP}}$ associates with every vertex $v \in V$ a subset $\mathscr{L}(v)$ of $\mathbf{AP}$ such that $\mathscr{L}(r) = \emptyset$ and $(v_1, v_2) \in E \Rightarrow \mathscr{L}(v_1) \subseteq \mathscr{L}(v_2)$. The size of the lattice $\{0,1\}^n$ grows exponentially with n. However, similarly to defining BDDs/ZDDs in terms of reductions applied to exponentially large decision trees, reductions can be iteratively applied to G to obtain a much smaller graph.[8] The key insights are:

- The value of $f(x)$ for any $x \in \{0,1\}^n$ can be thought of as the final value $f(y_m)$ of $f(y)$ along an initial path $(y_0, \ldots, y_m)$ in $\{0,1\}^n$ such that $y_0 = \mathbf{0}^n$, $y_m = x$, and $(y_i, y_{i+1}) \in E$ for all $0 \leq i < m$.
- In many cases, the value of $f(x)$ does not change frequently when moving along an initial path and $f(x)$ may be determined well before reaching x.

The following *conceptual* reductions become apparent: **Reduction(1).** For every pair of points $x, y \in V$ such that $(x, y) \in E$, if $f(x) = f(y)$, then the edge (x, y) is redundant and E can be updated so that x bypasses y directly to its successors:

$$\begin{aligned} &E \leftarrow E - \{(x,y)\} \\ &E \leftarrow E \cup \{(x,z) | z \in V \text{ and } (y,z) \in E\}\} \end{aligned} \tag{3.2}$$

Reduction(2). After the first reduction can no longer be applied, a vertex $y \in V$ might

[8] Later, we will see how to construct that smaller graph directly from Boolean formulas using symbolic algorithms.

become dangling or orphaned and, hence, unreachable from the root. That is, there is no $x \in V$ such that $(x, y) \in E$. Then y can be safely dropped: $V \leftarrow V - \{y\}$.

The above reductions preserve information about values of f and the labeled DAG returned is dubbed a *lattice-based Boolean diagram* (LBBD). Thus, evaluating a Boolean function f on a LBBD for any $x \in \{0,1\}^n$ amounts to following a *maximizing* initial path $(y_0, \ldots, y_m)$ subject to the constraints $y_0 = \mathbf{0}^n$, $y_i \leq x$ and there is no $y \in V$ such that $(y_m, y) \in E$ with $y \leq x$. Example LBBDs along with the corresponding BDDs are shown in Fig. 3.3. Note that no variable ordering is imposed *a priori*. At every step along any initial path in a LBBD G, a *minimal* set of Boolean variables are assigned 1 to alternately flip the value of f from 1 to 0 or vice versa. Thus, the value of f oscillates as we traverse G from the root $\mathbf{0}^n$ along any path.[9] More precisely, for all $v_1, v_2 \in V$, if $(v_1, v_2) \in E$, then $\mathscr{T}(v_1) = \neg\mathscr{T}(v_2)$. *A LBBD represents only the boundaries where a Boolean function f flips its value in the lattice* $\{0,1\}^n$. The zero-LBBD, representing the identically zero function or the empty collection of sets, consists of a single node r with $\mathscr{T}(r) = 0$ and $\mathscr{L}(r) = \{\}$, whereas the one-LBBD consists of a single node r with $\mathscr{T}(r) = 1$ and $\mathscr{L}(r) = \{\}$. Lemma 3.15 formally states that LBBD representations are consistent.

Lemma 3.15. *Let LBBD G represent a Boolean function $f : \{0,1\}^n \to \{0,1\}$. Then for every subset $A \subseteq \mathbf{AP}$, if u and v are maximal nodes of G such that $\mathscr{L}(u) \leq A$ and $\mathscr{L}(v) \leq A$ and $u \neq v$, then we have $\mathscr{T}(u) = \mathscr{T}(v)$.*

Representation Efficiency. To gauge efficiency of LBBD representation of a given Boolean function f versus BDDs and ZDDs, we use the BDD-to-LBBD and ZDD-to-LBBD size ratios as a metric.[10] The size of a graph $G = (V, E)$ is given by $|V| + |E|$.

[9] So basically every Boolean function appears as an alternating sequence of monotonically increasing and monotonically decreasing segments in the Boolean cube $\{0,1\}^n$.

[10] This quantifies the gain in conciseness by switching to LBBDs. The reduction in time complexity is typically the square of this gain, as symbolic operations have complexity $O(NM)$ for BDDs and LBBDs, with N and M being operand graph sizes.

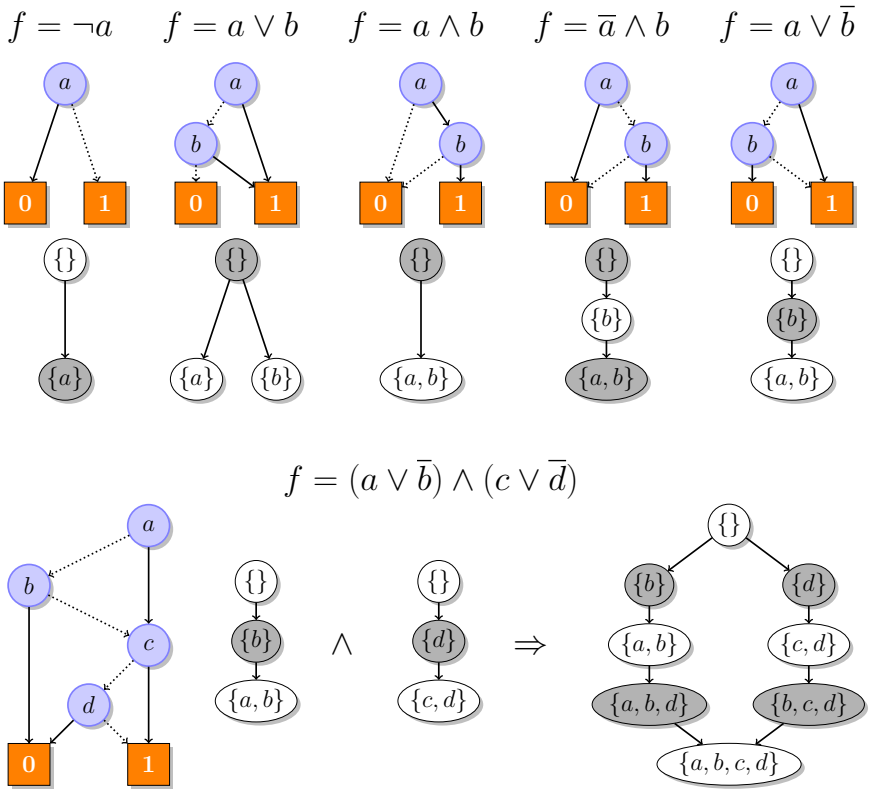


Figure 3.3: BDDs and LBBDs for some primitive Boolean functions. A gray node v has $\mathscr{L}(v) = 0$ and a white node v has $\mathscr{L}(v) = 1$.

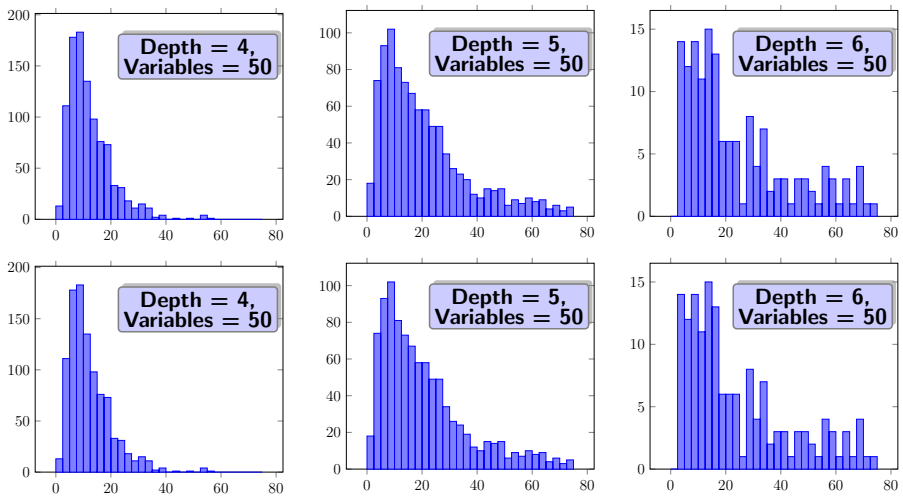


Figure 3.4: Histogram (y-axis is number of instances) of LBBD efficiency (x-axis) with respect to BDDs (top row) and ZDDs (bottom row) for randomly generated *monotonic* Boolean functions. **Depth** refers to the maximum syntax-tree depth of random formulas.

Antichains and Monotonic Boolean Functions. A Boolean function $f : \{0,1\}^n \to \{0,1\}$ is *monotone* (or order-preserving) if, for every bit vector $x \in \{0,1\}^n$, switching one of the bits from 0 to 1 can only (but not necessarily does) flip the value of f from 0 to 1. Equivalently, f is monotone, iff $\forall x, y \in \{0,1\}^n : \; x \leq y \Rightarrow f(x) \leq f(y)$. The negation ($\neg$) and the implication ($\Rightarrow$) are notable examples of non-monotone Boolean functions. Monotonic Boolean functions have many applications [128] such as sorting and matrix multiplication. An antichain representation[11] of monotonic Boolean functions has been known since the time of *Richard Dedekind* [53]. If $\mathbf{AP}$ is a set of n Boolean variables, then a Boolean function f is monotonic if, and only if, there is an antichain $\mathbb{A} \subseteq 2^{\mathbf{AP}}$ such that $f(x) = 1$ if there is some $y \in \mathbb{A}$ with $y \leq x$, and $f(x) = 0$ otherwise. Elements of $\mathbb{A}$ are, thus, the minimal subsets of $\mathbf{AP}$ that can force the value of f to be true. A monotonic Boolean function f has a unique minimum sum-of-products (SOP) representation equal to the disjunction of *all* of its prime implicants [103] corresponding to the minimal points $x \in \{0,1\}^n$ where $f(x) = 1$.

Beyond Monotonic Boolean Functions. LBBDs generalize the antichain representation from monotonic to arbitrary Boolean functions in the following sense. Given an arbitrary

[11] Which is a special case of LBBDs.

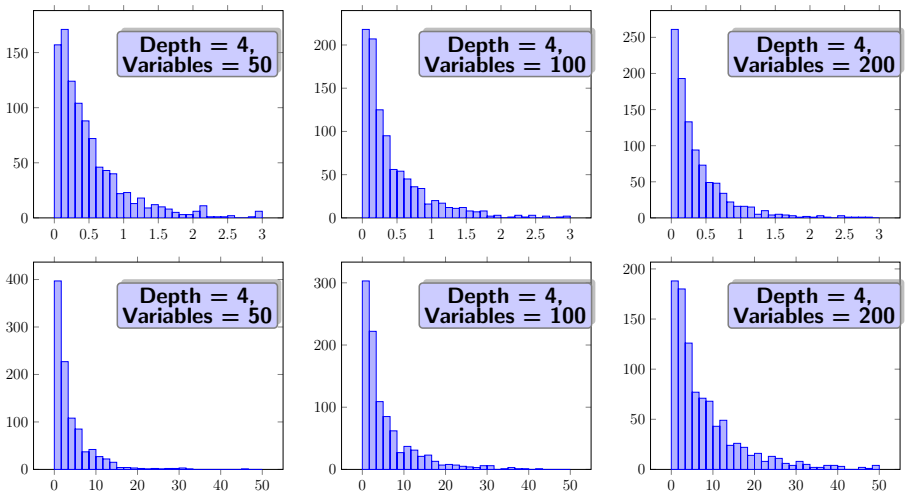


Figure 3.5: Histogram of LBBD efficiency with respect to BDDs (top row) and ZDDs (bottom row) for randomly generated Boolean functions (**Depth** refers to the maximum syntax-tree depth of random formulas).

Boolean function $f : \{0,1\}^n \to \{0,1\}$ represented by a LBBD $G = (V, E, r, \mathscr{L}, \mathscr{T}, \mathbf{AP})$. Then:

- For every $x \in V$, the set $\mathbb{SUCC}(x) = \{y \in V | (x,y) \in E\}$ is an antichain in G.
- For all $x \in V$ and $y \in \mathbb{SUCC}(x)$: $f(y) = \neg f(x)$.

Like BDDs and ZDDs, LBBDs are canonical. However, LBBDs do not rely on any order over **AP**. Histograms of LBBD efficiency for randomly generated monotone Boolean functions are shown in Fig. 3.4. Clearly, antichain (or LBBD) representations of monotone Boolean functions are much more efficient than BDDs and ZDDs.

Counter-Examples. In [32], it was proved that regardless of variables ordering, the Boolean function representing either of the middle two output bits of an n-bit multiplier have BDDs that grow exponentially with n. Similarly, in this section, we give examples of Boolean functions which have exponentially large LBBDs, yet have BDDs of size linear in the number of variables. This helps delimit the scope of applicability of LBBDs. In Fig. 3.6, both the BDD and LBBD for the even-parity function of four Boolean variables is shown. Evidently, the BDD requires only $(2n+1)$ nodes, where n is the number of Boolean variables, whereas

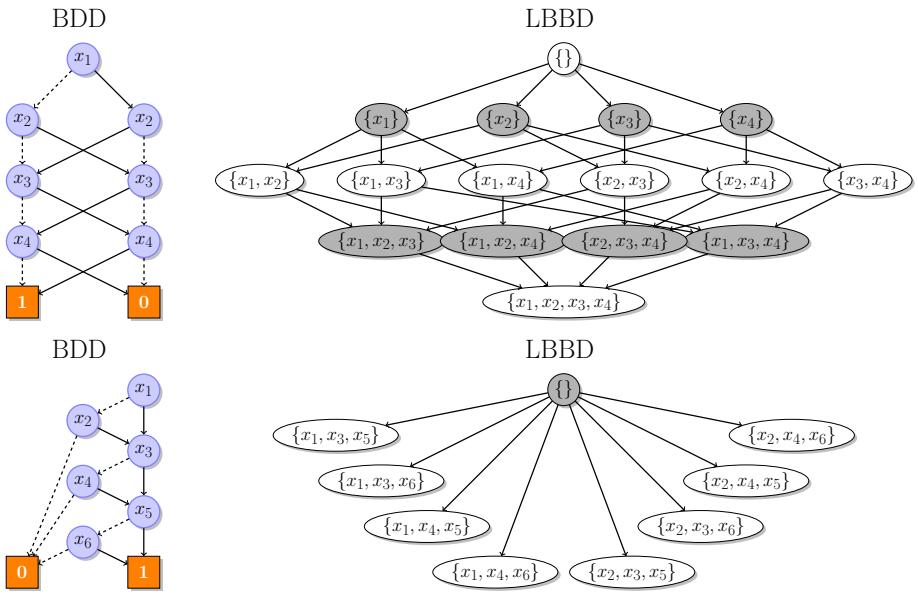


Figure 3.6: BDD and LBBD for the Boolean functions of even parity and $(x_1 \vee x_2) \wedge (x_3 \vee x_4) \wedge (x_5 \vee x_6)$, respectively.

the LBBD requires 2^n nodes (the entire Boolean lattice). The (even or odd) parity function f is exponential in the LBBD representation, since for all $x, y \in \{0,1\}^n$, we have $(x, y) \in E$ implies that $f(x) = \neg f(y)$, which means that none of the above reductions is applicable. Another exponential LBBD example is a monotone 2-CNF function shown in Fig. 3.6.

Boolean Function Decomposition. Shannon expansion gives rise to BDDs [32]. What kind of expansion gives rise to LBBDs? Intuitively, each vertex $x \in V$ of an LBBD represents a Boolean function $\mathbb{L}(x)$ given recursively as follows. The *generate* and *propagate* functions, $\mathbb{G}$ and $\mathbb{P} : V \times \{0,1\}^n \to \{0,1\}$, associate with every vertex $x \in V$ two Boolean functions defined as:[12]

$$\mathbb{P}(x) = \bigwedge_{a \in \mathscr{L}(x)} a, \qquad \mathbb{G}(x) = \mathbb{P}(x) \wedge \overline{\bigvee_y \mathbb{P}(y)} \tag{3.3}$$

$$\mathbb{L}(x) = \Big(\bigvee_y \mathbb{P}(y) \wedge \mathbb{L}(y) \Big) \vee \big(\mathscr{T}(x) \wedge \mathbb{G}(x) \big) \tag{3.4}$$

where y runs over $\mathbb{SUCC}(x) = \{y \in V | (x, y) \in E\}$. Thus, LBBDs are recursive (or *compositional*) representations where common subfunctions/subgraphs can be shared among

[12] If $\mathscr{L}(x) = \emptyset$, then $\mathbb{P}(x) = \mathbf{true}$.

different parts of a Boolean function or of multiple functions stored in the same base. This capability is crucial to efficiency and is not peculiar only to BDDs and ZDDs. Rather, it generalizes to all canonical *compositional* representations recursively defining the Boolean function associated with each graph node in terms of the Boolean functions associated with its child nodes.

3.10 Experimental Results

To evaluate the efficiency of LBBD representations and help position them with respect to other well-known techniques, we use the MCNC [175] and ITC99 [45] benchmark suites of multi-level combinational logic-circuit netlists (in BLIF format). All Boolean functions are extracted from these netlists and converted into BDDs, ZDDs and LBBDs. Every primary output or register/latch input of a MCNC block is handled as a function of all primary inputs and register/latch outputs of that block.

Implementation. All LBBD procedures were implemented in a new Java package along with BDD procedures (without variable reordering heuristics).[13] The C++ interface to the CUDD decision diagram package [160] is used to compute ZDDs for all Boolean functions studied here. From Fig. 3.7, LBBDs can be smaller than the corresponding BDDs/ZDDs for 20–30% of ITC99 and MCNC benchmarks. These results verify that LBBDs can serve only a slice of the Boolean function space. To more clearly understand where LBBDs fit in the Boolean representation toolkit, we consider SOP Boolean functions or cube covers that occur in two-level logic minimization problems [46]. In Fig. 3.8, we randomly generate cube covers and vary the probability of negative literals P. As it turns out, LBBDs become more efficient than both BDDs and ZDDs for SOP representations as more positive literals appear in the formulas, and become inefficient as negative literals prevail. As P tends to

[13]The current implementation uses the Java `HashSet` class to represent variable sets. An implementation where all set manipulations employ the much more efficient bit-vector operations is underway.

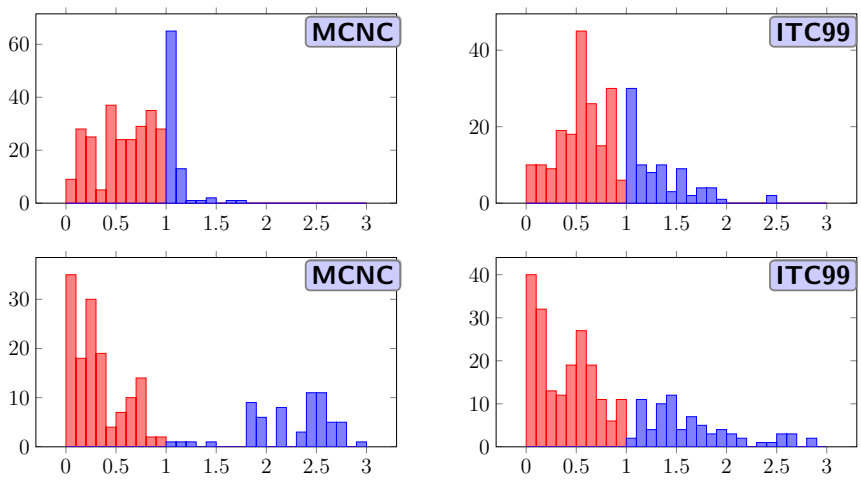


Figure 3.7: Histogram of LBBD efficiency with respect to BDDs (top row) and ZDDs (bottom row) for MCNC and ITC99 benchmarks. Blue indicates the instances where LBBDs are more concise, and red indicates the converse.

zero, the represented Boolean functions approach the monotonic regime where the antichain representation [53] was shown to be superior to BDDs and ZDDs. It is worthwhile noting that for $P > 0.5$, we can replace all negative literals with new positive literals to restore the favorable condition of $P < 0.5$ where LBBDs are more efficient than BDDs/ZDDs. As in [132], for each K, we generate 50 SOP formulas with 50 variables, 10 minterms containing only K positive literals and $50 - K$ negative literals. We vary the number of positive literals K to control sparseness of the represented functions or families of subsets. It can be seen in Fig. 3.9 too that LBBDs seem to dominate ZDDs for a wide range of sparseness levels, since LBBDs need relatively few positive literals to break-even with ZDDs. Therefore, replacing negative literals with positive ones enables LBBDs to be more efficient than ZDDs for all sparseness levels.

3.11 Conclusions

A novel class of graphical representations of Boolean functions, LBBDs, was demonstrated. Out of the vast Boolean function space, a particular subspace where LBBDs perform best

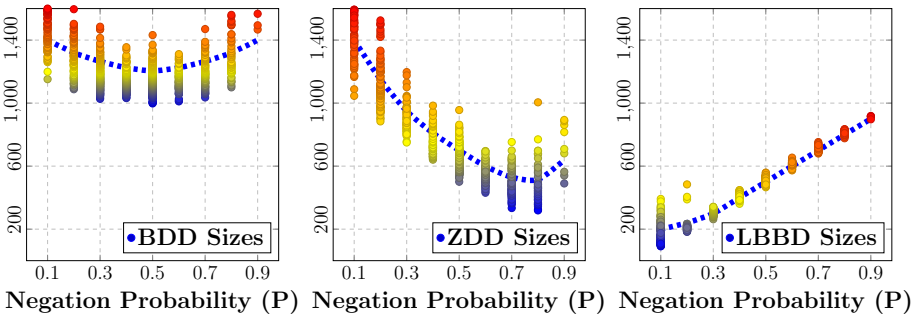


Figure 3.8: BDD vs. ZDD vs. LBBD sizes. For each $0 \leq P \leq 1$, we generate many SOP formulas with 200 variables, 10 cubes, 100 literals each containing negative literals with probability P and positive literals with probability $(1 - P)$.

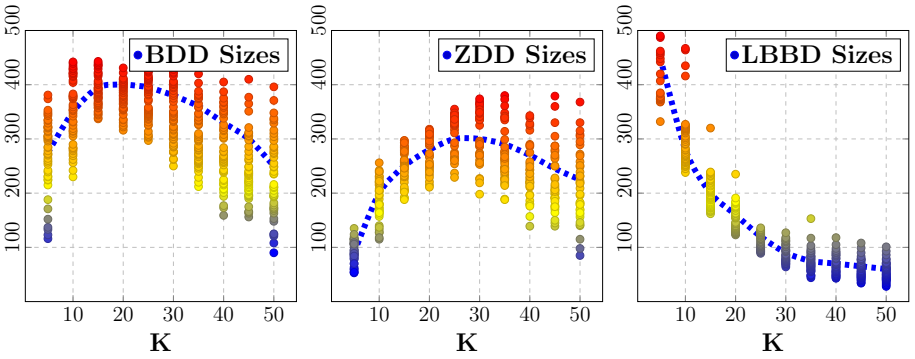


Figure 3.9: BDD vs. ZDD vs. LBBD sizes. As in [132], for each K, we generate 50 SOP formulas with 50 variables, 10 minterms containing only K positive literals and $50 - K$ negative literals.

has been identified by providing both *examples* (attesting to LBBD superiority over BDDs and ZDDs throughout that subspace) and *counter-examples* (establishing or delimiting its boundaries). The Boolean function space will continue to defy attempts to subdue under one overarching representation, and can only be conquered with a range of *interoperable* techniques. That is why it is necessary to develop efficient algorithms to convert LBBDs to and from BDDs/ZDDs so that they can coexist in one comprehensive toolkit.

Part II

Architectures and Tools

Chapter 4

Nonuniform Verification Architecture

This chapter introduces NUVA, which stands for *nonuniform verification architecture*, a distributed automata-based RV architecture for SR-DFA specifications, with a case study over a cache-coherent nonuniform-memory-access (ccNUMA) multiprocessor.

4.1 Architectural Elements

The core of NUVA is a coherent distributed automata transactional memory (ATM) that efficiently maintains states of a *dynamic* population of automata checkers organized into a rooted dynamic directed acyclic graph (DAG), representing the ensemble state LBFG, concurrently shared among all processor nodes. A cycle-accurate model of a ccNUMA multiprocessor confirms that performance slowdown is 1∼3% and NoC message traffic increases by 10∼15% at parametric event density[1] of 0.025 EPI for two compute-intensive scientific benchmarks having irregular concurrent data structures. The detailed architecture and implementation metrics in TSMC 40nm CMOS technology are presented. NUVA can be dimensioned to

[1]EPI stands for events-per-instruction.

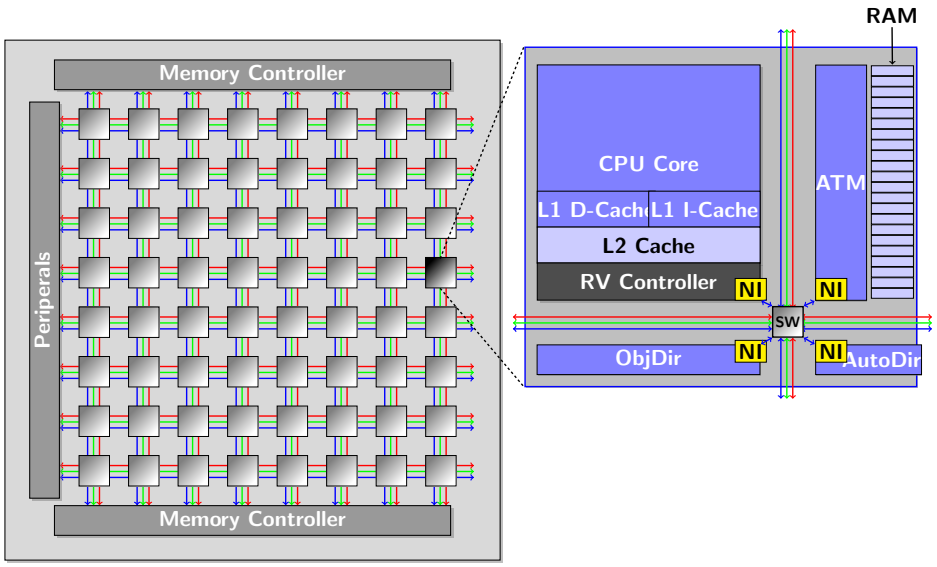


Figure 4.1: RV elements in a single-chip multiprocessor (NI = Network Interface).

incur average total power overhead of less than 140mW and area overhead of 4mm^2 for a quad-core multiprocessor chip, at operating frequency of 250MHz. It is also estimated to incur 1.9∼2.6% area overhead and 0.5∼1% power overhead when integrated with Intel's family of high-end desktop and mobile quad-core processors operating at 1.6∼3.2GHz. Our silicon implementation achieves average performance[2] of 1.5 MEPS/mW. For a processor core with 5 MIPS/mW, this corresponds to a 3.2% drop in energy efficiency at parametric event density of 0.01 EPI.

In this chapter, we identify a minimal set of architectural elements and techniques necessary for efficient RV of SR-DFA specifications over a distributed multiprocessor system. A high-level architecture is shown in Figure 4.1. The RV directories and controllers shown do not have to exist at every processing node (or *tile*). The RV components are completely decoupled and oblivious to the interconnection topology (regular, e.g., a mesh or torus, or irregular) and extent (on-chip or off-chip). However, discussions in this paper focus on single-chip multiprocessors (CMPs).

Components of NUVA can be implemented in software or hardware or both. For performance

[2]MEPS stands for million events per second.

reasons, these components will be tightly integrated with processing nodes, rather than being stand-alone nodes.

4.1.1 Observation Unit

A new RV observation unit (OU), implemented as a *resource* in the CPU core for which instructions in the CPU pipeline can be scheduled, watches for certain sparse *event-carrying instructions* (i.e., instructions designated by the compiler to represent events of interest to automata checkers) or to collect information (such as object and method IDs) needed by the continuous event-building process to be packaged into *event packets*. Event packets are then sent to RV controllers (described later) for verification against the system-wide population of checkers. In Section 4.2, Intel Pin is used to intercept function calls and returns that are used in property specifications of each benchmark.

4.1.2 Object Directories

A fundamental requirement of any distributed RV architecture is to guarantee that all automata checkers distributed throughout the system observe a *consistent execution history for each object*. An *object token* is dynamically allocated for every new software object in the system[3] and *object directories* are used to manage the sharers set or exclusive owner of each token. Each object token is assigned to a *home object directory* (depending on its unique ID). Object tokens do not serialize access to objects and object access methods can still overlap, thus, preserving the application level of parallelism. Rather, object tokens are used to impose a consistent interleaving order on object-access start and end events. A *limited pointer scheme* [161] was employed because the number of sharers of any object is usually

[3]Only data items with unique system-wide identity need to have object tokens. Copyable *values* will not have tokens associated with them which reduces NUVA performance overhead.

small. When the number of sharers exceeds the number of pointers n, a directory entry dynamically switches to a *coarse bit-vector* scheme [161], where each bit stands for a set of K *possible* sharers to which coherence requests need to be directed. Since, most of the time, only a fraction of all objects are shared among one or more nodes, a fully-associative *object directory cache* [161] is used and objects that are not currently shared can be evicted from it. A RV controller is designed to hold only a maximum of J object tokens, where J is selected to cover a sufficient number of events that can occur within the pipeline instruction window W of all hardware threads of a single core. If more than J events are observed within W (which is a rarity by design), the thread causing this overflow is stalled until enough local object tokens are released and can be evicted. Thus, object directory caches throughout the system need only cater for $J \times C$ object entries, where C is the number of RV controllers in the system. This is called *inclusive directory cache organization* [161]. Moreover, if the number of object directories is made to scale linearly $k \times C$ with the RV controller count C, with $k < 1$, the associativity $J \times C/k \times C = J/k$ of object directory caches can be made independent of C.

4.1.3 RV Controllers

A RV controller serves one or more processor nodes by receiving events from OUs embedded in their pipelines, communicating with object directories to request Shared or Exclusive object tokens according to each event type, submitting events to the automata memory controller (see Section 4.1.5) for further processing, and finally relaying back responses to OUs to remove the pipeline stalls inserted. If events involve multiple objects (each with possibly different access type, i.e., READ or WRITE), RV controllers need to request object tokens in a specific order (e.g., by their object IDs) to prevent deadlock.

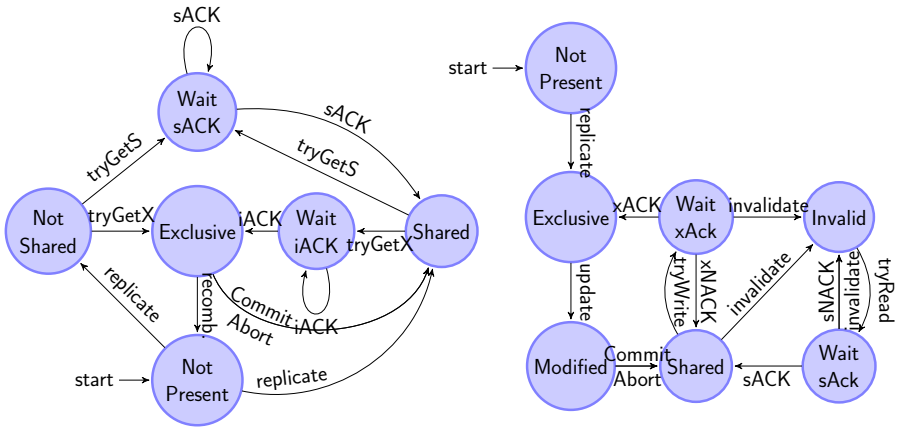


Figure 4.2: States of an automata directory entry (left) and states of an automaton replica (right).

4.1.4 Automata Directories

The population DAG is a concurrent data structure shared among all RV controllers and events are applied to the checker population locally at every RV controller. Therefore, a coherence protocol, implemented by a distributed set of automata directories, ensures that all RV controllers have a consistent view of the checker population by keeping track of the sharing state and the sharers set of each checker replica. Each replica is hashed to a fixed *home directory* based solely on its variable binding values. Coherence states of automata directory entries and automaton replicas at an ATM controller node (see Section 4.1.5) are shown in Figure 4.2.[4]

4.1.5 Automata Transactional Memory

Every event must *appear* to update the states of the entire checker population *atomically* at some point within the lifespan of the *population-update transaction*. To maximize concur-

[4]A copy of a SR-FSM replica has a *sharing state*, which can be Invalid, Shared, Exclusive, or Modified and changes in response to coherence requests (tryGetS, tryGetX, Commit and Abort. It also has a *current SR-FSM state*, which can be any of the states of $\mathcal{M}^{\varphi}$ and changes in response to RV events. Invalid state (I) means that the SR-FSM replica state and its offspring/parents lists are not up-to-date.

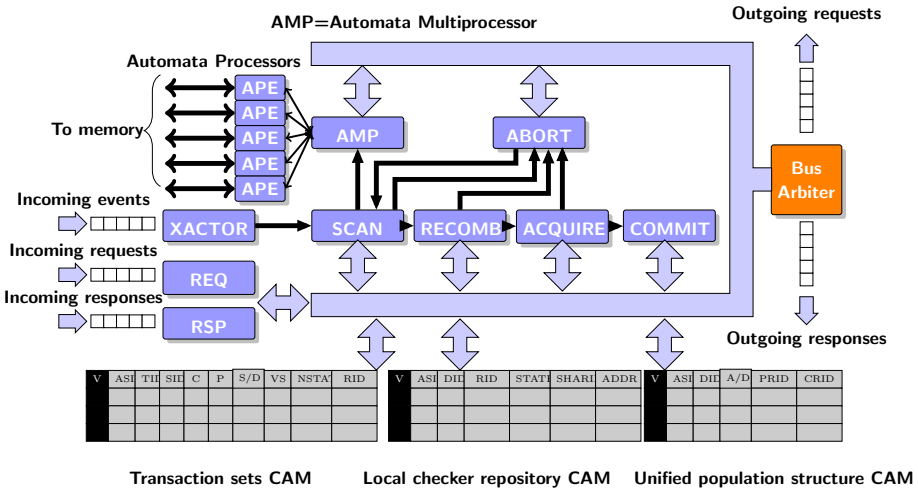


Figure 4.3: ATM event processing pipeline.

rency, state updates due to different events are allowed to overlap as long as they mutate the states of disjoint sets of replicas. An *automata transactional memory* (ATM) is needed on top of the coherence protocol substrate to implement an optimistic concurrency control mechanism that guarantees serializability. A transaction $T(\sigma, \mathcal{M}^{\varphi})$ is associated with every event σ and SR-FSM $\mathcal{M}^{\varphi}$. Given an event σ and its associated transaction (T for short), the set of replicas of $\mathcal{M}^{\varphi}$ that will switch their states in response to σ (without recombination) or have new offspring is called the *write-set* of T, or $WS(T)$. In order to determine $WS(T)$, usually a small fraction of the checker population needs to be read and is called the *read-set* of T, or $RS(T)$. Moreover, since SR-FSM replicas can be created and destroyed at runtime, with every transaction T, we maintain a *replicate-set* (or *create-set*) $CS(T)$ and a *recombine-set* (or *destroy-set*) $DS(T)$. The replicate-set is the set of all new SR-FSM replicas created due to σ at a particular ATM node, and the recombine-set is the set of all replicas that go out of existence by recombination. The internal organization of an ATM node shown in Figure 4.3 illustrates that event processing in ATMs proceeds in a pipelined fashion.

Contention Management and Speculative Update. The state update procedure has *worst-case* time and space complexity of $O(N)$, where N is the checker population size. The read sets of RV transactions can grow very large and cause intolerable RV coherence

traffic and conflicts among concurrent RV transactions. An event usually affects the state of only a few replicas in the population (see Section 4.2.3). Therefore, a *speculative approach* can be used to reduce coherence traffic and make the population update procedure almost independent of population size. Every time an event σ arrives, the entire population DAG is scanned in depth-first order starting at the root. For each visited replica ρ, all replication graphs associated with the predicate symbol of σ and *all states* of $\mathcal{M}^{\varphi}$ are *elaborated*. The elaboration algorithm decides whether or not there is a *possible* state transition. It can then decide whether or not to send a tryGetS RV coherence request for that replica (and add it to $RS(T)$). During this scanning stage, a replica can be in the Invalid state, since its bindings are always valid and do not change with time (unless the replica recombines with its parent). Once in the Shared state, it can be decided whether or not a replica will indeed make a state transition (and be added to $WS(T)$) and/or self-replicate (the new replicas, if any, are added to $CS(T)$). Also, its offspring list is up-to-date and can be traversed. If bindings of ρ and argument values of σ conflict, then ρ just remains in its current (*unknown*) state. Thus, ignorable events are very cheap because they can be detected at the level of the entire population (i.e., the root replica) *even without issuing any RV coherence requests*. Moreover, population scanning does not have to visit children of ρ because they are guaranteed to conflict too. This way, entire subgraphs from the population DAG can be pruned, leading to an efficient population update procedure and reducing the probability of collision between concurrent RV transactions. The scanning stage in Figure 4.3 internally utilizes K *automata processing elements* (APEs) to parallelize the scan.

Recombination stage. The recombination stage resolves all recombination opportunities and updates the recombine set $DS(T)$ of the current transaction T. A *recombination-candidates set* $US(T)$ is initialized with $WS(T)$. For every replica $\rho_c \in US(T)$, its current or new SR-FSM state is compared to that of each ρ_p of its parents. In partial recombination, both the parent ρ_p and child ρ_c are added to $WS(T)$, whereas in case of complete recombination, all parents of ρ_c are added to the $WS(T)$ and ρ_c itself is added to $DS(T)$. At the end

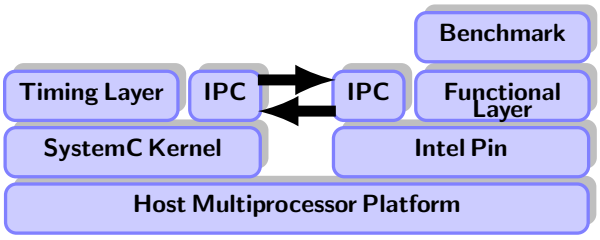
of every iteration, all replicas that acquired new parents (by adoption) are added to $US(S)$ and the process is repeated until no more adopted replicas are discovered.

Atomic Commit. To make all updates in the write-set visible atomically to all other events, a *two-phase commit* protocol is used, in which the initiator node first sends *commit requests* to the automata directories. To prevent deadlock, *try-locks* are used. A tryReplicate request for every newborn replica in $CS(T)$ and a tryGetX request for every replica in $WS(T) \cup DS(T)$ are sent to their respective home directories. Based on the acknowledgments received back from the participant automata directories, the initiator node then decides whether to send a commit or abort (roll-back) message to the participant automata directories. If a newborn replica does not already exist, the tryReplicate request is acknowledged. Later the replicate request will be committed. If the replica already exists, the tryReplicate request is negatively acknowledged (NACK) leading to aborting and restarting the current transaction.

Details of access tracking, version management and recovery, as well as hardware representation of various sets using content addressable memories (CAMs), including local replica identification, is standard and was omitted for lack of space.

4.2 Experimental Validation

Simulation Architecture. A cycle-accurate multiprocessor simulation model for $x86\_64$ instruction set architecture was constructed to have a functional layer (based on Intel Pin [17]) that is completely controlled by a timing layer based on GEM5 [26] translated into SystemC. The two layers communicate via Linux sockets. The GEM5 timing layer features are summarized in Figure 4.4. The NoC is used to support cache coherence controllers and directories, ATMs, RV controllers and object/automata directories on separate virtual nets. The Intel Pin functional layer (PFL) executes the benchmark natively on the host multiprocessor



CPU cores	4/16/64 in-order, 3-way SMT
Cache coherence	MESI, directory-based
Interconnect	Fixed-pipeline Garnet NoC
L1 Cache	I/D$=32/64 kB, 2-way
L2 Cache	2MB, 8-way
Phys. Mem.	512MB
NoC Topology	Torus
APEs	8 per node

Figure 4.4: Simulation platform and parameters

platform and has two functions: (1) It monitors execution of a multithreaded benchmark program on the host machine and uses Linux sockets to send decoded instructions, executed instructions, memory accesses, and RV events to the SystemC timing simulation layer. (2) It controls thread execution on the host machine based on commands from the SystemC timing layer to emulate running the program on the configured target multiprocessor platform.

4.2.1 Benchmarks.

Two benchmarks were completely rewritten to comply with the style expected by the Pin functional layer. In the BH benchmark [106], the BH tree construction phase was parallelized because it offers many opportunities for inter-thread contention and meaningful correctness properties. In the Canneal benchmark [23], the netlist is a lock-free concurrent data structure that was rewritten to use C++11 atomics. More work is ongoing on mining specifications from the PARSEC benchmark suite [23] for more comprehensive evaluation of NUVA.

4.2.2 Bug Detection Capability.

By catching locking discipline violations in the BH benchmark, NUVA helped to refine BH specification in early stages of evaluation. Moreover, the bug detection capability of NUVA was evaluated using *injected violations* and NUVA managed to catch all these violations too.

For example, from Figure 2.1, having two consecutive **Lock** calls to acquire locks on BH tree nodes n_1 and n_2, respectively, the first subsequent **Unlock** operation must release the lock on n_1 before releasing the lock on n_2. Also, two consecutive **Lock** calls cannot be followed by a third **Lock** call unless there is an intervening **Unlock** call. Finally, a new BH tree node n_2 can be created only while holding the lock on another node n_1 and, then, either n_1 is unlocked or n_2 is locked. In Figure 2.2, the property is violated only if two netlist elements $n1$ and n_2 are swapped twice with neither of them having been swapped with a different element in between. Thus, we could craft the benchmark source code to produce violating sub-traces, such as $(\mathsf{Swap}(t, n_1, n_2), \mathsf{Swap}(t, n_1, n_2))$ or $(\mathsf{Swap}(t, n_1, n_2), \mathsf{Swap}(t, n_2, n_1))$, and NUVA could detect all.

4.2.3 Simulation Results.

Figure 4.5 shows the cycles-per-instruction (CPI) adders stack for both benchmarks. The RV overhead is approximately 1∼3% of total execution time and the NoC traffic generated by the RV components accounts for 10∼15% of total NoC traffic. This relatively low impact on performance is due to *temporal locality* in the monitored programs (i.e., threads tend to reuse the same objects) that reflects upon parametric event processing. In the ensemble state LBFG representation, each automaton replica tends to be sensitized only to particular variable values) that has been exploited to its maximum in the transactional RV implementation. To quantify locality, we use the trace of RV read sets $RS(t)$ and calculate the autocorrelation function $\mathcal{R}(s,t)$ of its *working-set trace* $RS^T(t) = \cup_{s=t}^{t+T} RS(s)$ using the concept of *Jaccard's coefficient*, $J(A,B) = |A \cap B|/|A \cup B|$. That is, $\mathcal{R}(s,t) = J(RS^T(s), RS^T(t))$. The high autocorrelation of the read-set trace of a single ATM and the low cross-correlations among read-set traces of different ATMs confirming our locality conjecture are shown in Figure 4.6. The population size at every RV transaction point shown in Figure 4.6 proves that recombination is very effective at reducing the population size that the RV infrastructure has to

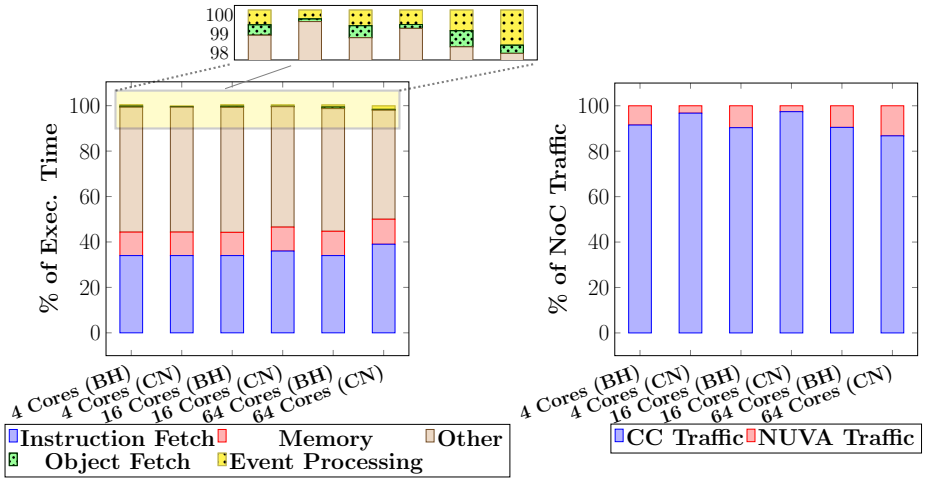


Figure 4.5: CPI adders stack and NoC (data + control) traffic for monitored Barnes-Hut (BH) and Canneal (CN) benchmarks vs. the number of CPU cores.

accommodate and maintain. Sizes of various sets maintained by a RV transaction are shown in Figure 4.6. By virtue of speculative checker update and temporal locality, the abort count per RV transaction in Figure 4.6 is zero most of the time, an indication of low conflict rate among concurrent RV transactions (conflict probability is 0.28). The read-set size and the number of visited replicas during the scanning phase is typically 5-10 and almost all of them are usually found in the Shared state, thus significantly reducing RV coherence traffic. The only limitation apparent from Figure 4.6 is the number of visited replicas during the scanning phase. This problem can be solved by using as many APEs as may be feasible. The APE design implemented here consumes 0.024mm^2 of area and 0.58mW of power at 200MHz in TSMC 40nm technology which implies that we can use as many APEs as desired limited only by number of memory ports.

Sensitivity and Scalability Analysis. To measure NUVA's overhead sensitivity to problem size and scalability to larger multiprocessors, we vary the number of processor cores as well as the benchmark workload (i.e., data set size and number of worker threads), which is Gustafson's scaling model. Figure 4.5 indicates tolerable sensitivity and substantial scalability up to 64 cores.

4.2.4 Synthesis Results.

All the proposed RV components have been modeled at the RTL level using SystemVerilog and synthesized in TSMC 40nm CMOS technology using Synopsys Design Compiler (SDC). The RV architecture and technology parameters are listed in Table 4.1 and synthesis results are shown in Figure 4.8. We then compare the overhead of NUVA with the die size and thermal design power (TDP) of commercial desktop and mobile processors based on publicly available empirical data. NUVA still makes sense even in a single-core context because both event-driven and multithreaded programs can be run on single-core CPUs. Since we are using a standard-cell design flow, it is extremely hard to meet timing constraints at such high frequencies as 2-3GHz unless an unrealistic pipelining depth is used everywhere throughout the design. Such high frequencies as 2-3GHz can only be achieved with semi-custom design flows. Therefore, for fair comparison, with commercial processors, we measure the area and power scaling of the synthesized RV components in our standard-cell design flow as the operating frequency is increased and linearly extrapolate (for simplicity) to 1GHz[5], as shown in Figure 4.7.[6] In Figure 4.8, the area and power of all RV components are shown as estimated by SDC. Evidently, CAMs dominate design area and power, and can be significantly reduced using full-custom design instead of standard-cell design. The area and power of RTL CAM designs can be modeled by:

$$A_{RTL} = a_1 N + a_2 N \log_2 N,\ P_{RTL} = p_1 N + p_2 N \log_2 N$$

where N is the number of CAM entries, a_1 and a_2 are constants depending on CAM width. The $a_1 N$ term is due to the CAM array and the $c_2 N \log N$ is due to iteration logic which is a parallel-prefix tree (PPT). From Figure 4.8, these constants can be estimated to be $a_1 \approx 350\mu m^2/entry$ and $a_2 \approx 7\mu m^2/entry$, $p_1 \approx 13\mu W/entry$, $p_2 \approx 0.4\mu W/entry$ for a

[5]RV components are assumed to work at $\frac{1}{3}$ the CPU core clock rate. The maximum clock rate achieved with TSMC standard-cell flow is 500MHz (without aggressive pipelining not to invalidate performance results).

[6]It should be noted that extrapolated scaling of standard-cell flow is somewhat pessimistic, since a semi-custom flow can improve upon it [41].

52-bit entry. Comparing with [57, 96], a full-custom design of a CAM array can be 3× more efficient than the RTL design in this paper, resulting in area and power given by:

$$A_{FC} = \frac{a_1}{3}N + a_2 N \log_2 N,\ P_{FC} = \frac{p_1}{3}N + p_2 N \log_2 N$$

Constants $a_1 \approx 10\mu^2$ and $p_1 \approx 0.7\mu W/entry$ for a full-custom CAM design, whereas constants a_2 and p_2 remain unchanged because the PPT will still be implemented in RTL. These adjustments are applied to SDC area and power figures to obtain the area and power scaling in Figure 4.7. It should also be noted that the reported power numbers are pessimistic, since Synopsys Power Compiler tool uses a conservative activity propagation technique for power estimation. Area and power of RV (automata and object) directories increase faster than the area and power of RV and ATM controllers as the (object and automata) cache size increases, because a directory entry maintains a sharers set for every object or automaton whereas a RV or ATM controller does not maintain a sharers set per object or automaton.

From SystemC simulations, the average event processing throughput is 43.9 MEPS in a simulated quad-core multiprocessor (with a RV load factor per core[7] of 0.1) at operating frequency of 500MHz. From Figure 4.7, our silicon implementation is estimated to consume an average of 576mW at 1GHz at a load factor of 1. As a very rough measure of energy efficiency,[8] this results in RV energy efficiency of 1.5 MEPS/mW. For a processor core with 5 MIPS/mW, this corresponds roughly to 3.2% drop in energy efficiency at a parametric event density of 0.01 EPI, as shown in Figure 4.7. Of course, as the parametric event density of a program increases, energy efficiency will decrease faster due to NoC conjecstion.

[7]The *RV load factor* of a CPU core is the average number of concurrent RV transactions being processed at any time.

[8]Since the SystemC model and SystemVerilog model are not cycle-equivalent, the NoC power is not accounted for, and the default switching activity annotation and propagation mechanisms of SDC are used.

Argument width	3 bits (max 8 args/predicate)	**Replica ID width**	16 bits
Variable width	4 bits (max 16 vars/property)	**Object ID width**	32 bits
Number of CPU nodes	Up to 64 (CPU ID = 6 bits)	**Supply voltage**	0.99V
Directory max. sharers	4 (bits = 4×6)	**Frequency**	250MHz
Number of APEs	8 per ATM node	**Threshold voltage**	mixed LVt/HVt

Table 4.1: RV architectural and technology parameters

4.2.5 Optimum APE Number

To explore the optimum number of APEs per ATM node, CACTI5.3 [158] (which is a modeling tool for dynamic and leakage power, access time, and area of caches and other types of memories was used to model the effect of the combination of APEs count and the number of SRAM ports on overall energy efficiency, where the SRAM is implemented using a mixed-threshold-voltage 40nm technology. These SRAMs, shown in Figure 4.1, hold the parametric SR-FSMs, replication graphs, etc. In this experiment, each APE is allowed access to the SRAM through an independent port to maximize parallelism, since the elaboration of each replication graph is an independent task. By increasing the number APEs, the total time needed to update a checker population of certain size is reduced proportionately. However, having more read ports to the SRAM substantially raises the memory access time, area, and power consumption. Moreover, increasing the APEs count beyond a certain limit does not reap much gains in execution time and only incurs the excessive area/power overhead of extra read ports. This is illustrated in Figure 4.7, which shows that the total (leakage and dynamic) energy exhibits a minimum at 6-8 SRAM ports.[9] It is clear that, depending on the size of the SRAM, there is an optimum number of read ports or APEs that minimizes the energy per RV operation.

[9]Energy consumed in memory operations to process 1000 (randomly generated) replication graphs (normalized to the maximum).

4.3 Conclusion, Limitations and Future Work

The feasibility of efficient RV of parametric specifications in hardware has been demonstrated. However, a few limitations need to be pointed out. All the CAMs used in this design were entirely modeled at the RTL level. Although that yields practical and parameterizable designs for $< 1k$ locations, the area and power efficiency of NUVA leans heavily on these special memory structures. Therefore, a full-custom implementation of all CAMs is the next logical step. Moreover, mechanisms for spilling into main memory in case the special memory structures also warrant careful consideration in terms of efficient replacement algorithms that exploit locality of reference. Also, object directories can be piggybacked on any contemporary directory-based cache coherence (CC) protocol with minimal changes to the CC protocol. Replica IDs are wide (to support globally large population of checkers). However, any node is expected (as demonstrated in simulations) to host only a much smaller number of replicas. Thus, using smaller local replica IDs will reduce the hardware cost significantly. Therefore, a *bidirectional translation look-aside buffer* (BTLB) is needed to translate back and forth between global and local IDs.

Communication among processing nodes is assumed *secure* and *error-free* and that nodes cannot ignore RV messages or delay them indefinitely. Otherwise, node failures may have occurred. Validating this assumption can be the responsibility of each node, for example, by denying requests whenever there is no buffer space for new requests. Consensus about node failures might be required to avoid those nodes completely.

Moreover, as already shown, NUVA owes its low performance overhead to temporal and spatial locality in monitored programs. Locality is a natural property of most computations. Programs that lack locality will suffer substantial performance hits, for example, due to memory hierarchy and cache coherence latencies. Moreover, NUVA can only detect property violations that manifest themselves at the monitored interface or API. That is why most

intrusion detection systems intercept system calls because it is an interface that cannot be circumvented. Also, it is assumed that no invocations of the monitored interface can be concealed from the checker population, and no extraneous invocations can be inserted. It is also implicit that characteristics (average size, average longevity of a checker replica) of the checker population is dictated both by the monitored property and observed system behavior. *Strong bounds on population size need to be developed.* In this thesis, only information carried by an event (or stored in the free variables or constants of a SR-DFA) can be used in RV. No rigid predicates or functions [105] can be used in NUVA. Support for checker callbacks implementing rigid predicates and functions may be added in the future. Rigid functions and predicates can be implemented by associating them with program-specified functions and allowing the RV hardware to automatically invoke them, as in iWatcher [178].

The current NUVA design inserts stalls into the processor pipeline to fulfill the condition of *precision* (i.e., reporting violations where and when they occur). Although that does not result in crippling performance slowdown, future implementations may give the option of trading off precision (either by increasing latency of violation declaration or by increasing false positives) for even lower performance overhead.

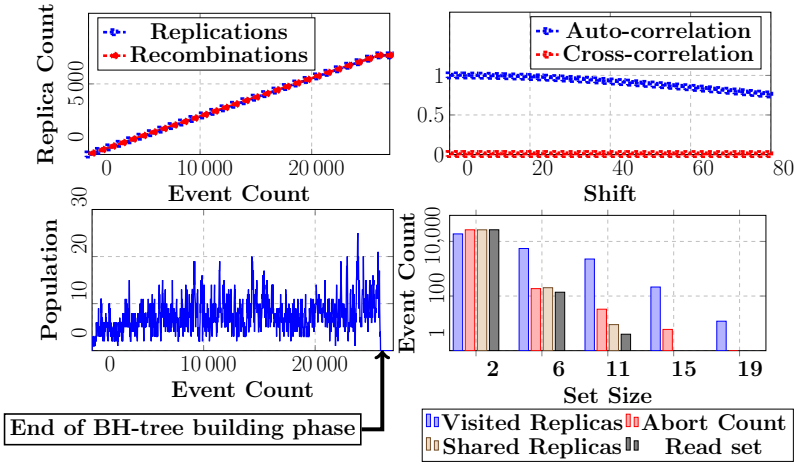


Figure 4.6: Checker population size (= *replications* − *recombinations*), size-histogram (*log-scale*) of various transaction sets, and auto-/cross-correlation of transaction working sets ($T = 40$) for a 16-core/16-ATM system.

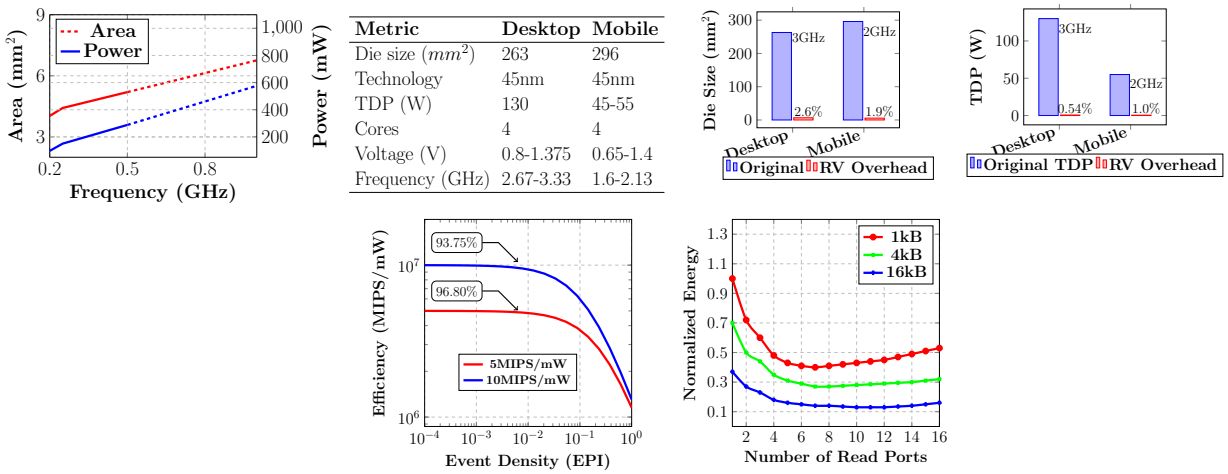


Figure 4.7: Frequency-scaled RV overhead w.r.to quad-core Intel processors, CPU energy efficiency vs. event density, as well as normalized energy per RV operation vs. number of RAM read ports (and for SRAM size = 1kB, 4kB, 16kB). In both the desktop and mobile categories, we use a family of Intel processors covering a narrow range of models, microarchitectures, and applications in the CMOS technology (Intel 45nm) closest to ours (TSMC 40nm).

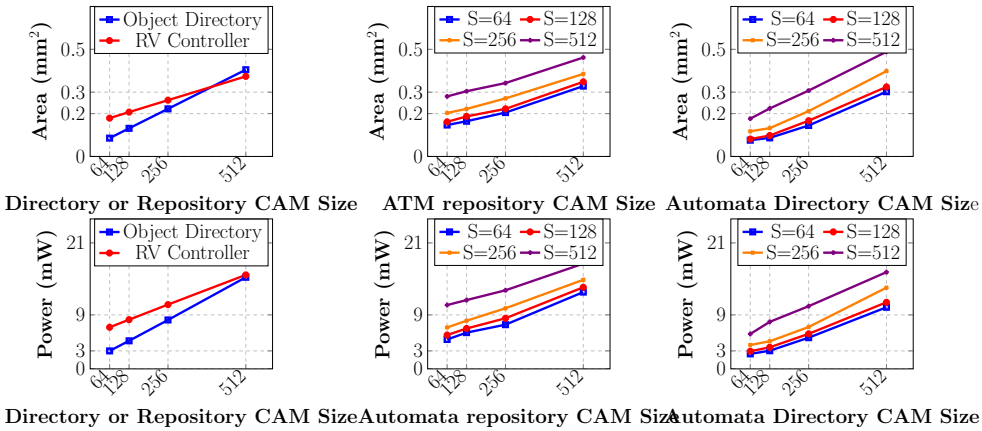


Figure 4.8: Synthesis results of various RV object-layer components, ATM and automata directories at 250MHz (UPS CAM stands for the *unified population-structure CAM* where the population DAG edges are stored as an adjacency list. S stands for UPS CAM size). Leakage power is generally 3-5%.

Chapter 5

Specification Mining

5.1 Specification Mining

Formal specifications, like SR-DFAs, are notoriously hard to formulate and maintain for evolving complex distributed systems, especially at the level of precision mandated by ID applications. Specification mining [10] is an automated approach to extracting specifications from the abundant execution (and operational) traces and audit trails. In this thesis, we utilize a novel and rigorous mining methodology to extract SR-DFAs using an iterative and interactive mining tool, called ParaMiner, from data-carrying event traces.

ParaMiner enables conducting multiple experiments to discover precise specifications that capture salient behaviors of instrumented programs. ParaMiner adopts a SR-DFA formalism intended for runtime verification of embedded and general-purpose software systems as well as anomaly-based intrusion detection in system-call traces. The proposed mining techniques can apply to all of parametric events streams regardless of their origin.

ParaMiner is able to construct properties that capture, and weave together, wildly different

(large-scale) behavioral phases.

With SR-DFAs, the training data is only implicitly memorized within the (limited) capacity of the SR-DFA being mined. A SR-DFA can remember some (but not all) events arbitrarily far in the past. Remembering the entire trace of events observed so far is neither feasible (due to memory constraints) nor desirable or beneficial (due to limited generalization). The goal is to build a SR-DFA that will predict (or generate) all the traces in the training set with good *generalization* performance by allowing recurrent behavioral episodes (possibly with many variants). As explained in [131], a single-state automaton with a self-loop is so weak as to have no predictive power (i.e., it predicts every possible behavior), whereas the prefix-tree acceptor automaton would only predict those traces in the training set.

The problem of identifying a finite-state automaton from samples of its behavior is systematically investigated in the discipline of *Grammatical Induction* [51]. The problem of identifying an automaton from examples is NP-Hard [79, 11, 101], whereas teacher-based identification runs in polynomial time [12]. However, it is empirically demonstrated in [108] that random DFAs are approximately learnable from randomly selected sparse training data. ParaMiner identifies a (not necessarily minimal) SR-DFA without pre-selecting or imposing the form or size of its state space, which depends solely on the training set. In the automata learning literature (e.g., [54]), both positive and negative training examples are required. In this thesis, since a SR-DFA is of interest, the most important criterion is that the training set be sufficiently diverse. The input to ParaMiner is a set of parametric event traces and the output is a SR-DFA that expresses a temporal property satisfied by the training set.

5.1.1 Introduction

Nature of Specification Mining

Specification mining can be viewed as a special form of grammatical inference or induction [50], where a (partially automated or interactive) learning algorithm is given access to *structured-data sets* (e.g., sequential execution traces, program source code in some programming language, change logs, bug reports, etc.) and is expected to return a grammar (i.e., a generative device) that explains or generalizes the training data sets.

The target formalism of ParaMiner is SR-DFAs, in contrast to other formalisms such as value invariants [65] and rules or patterns [125, 172]. The use of finite-state automata implies that behavior approximation is inevitable. Functions in a program may directly or indirectly call each other recursively. Thus, even without taking argument data values into account, API call traces of a program must generally be represented by context-free languages (CFLs) [95]. In order to represent these traces with finite automata, an *approximation* of CFLs with regular languages must be constructed. There are two basic types of behavioral approximations, and mined specifications can be a hybrid of these two types:

- If a property *over-approximates* program behavior (i.e., accepts a superset of correct behavior including impossible behaviors that cannot be exhibited by the program in a correct run), verification of that property might miss latent program bugs (i.e., result in *false negatives*).
- If a property *under-approximates* program behavior (i.e., accepts a subset of correct behavior and rejects legal behaviors that could be observed in a correct run), verification of that property might result in *false positives*.

Typically, a specification mining tool strives to extract tight supersets of the set of system behaviors. Every mined property captures only one aspect of system behavior and, if con-

sidered in isolation, may allow some illegal executions excluded by other properties. Two conflicting requirements are needed in specification mining outcomes [176]:

- *Precision*, which measures how successful a property is at closely approximating *correct* program behavior.
- *Conciseness* or *efficiency*, which measures how successful a property is at *abstracting* away irrelevant behaviors and capturing the most essential (i.e., distinctive) ones.

A key to conciseness is using expressive domain-specific vocabulary or concepts [176], which we achieve in this thesis by focusing on mining specifications based on program-specific API interfaces. Different applications put different weights on precision and conciseness. For example, in anomaly-based intrusion detection (ID) systems [16], model/property precision is of paramount importance[1], whereas conciseness acquires less weight. To mitigate the insider threat (IT), host-based user profiling (e.g., using ParaMiner) is important [153]. However, maintaining and updating user profiles is a challenge to any host-based IDS.

5.1.2 Mining SR-NFAs

ParaMiner extracts SR-DFAs from execution traces to capture regular safety properties, which can then be verified by RV. For a SR-DFA $\mathcal{A}$, let $\mathcal{L}^\dagger(\mathcal{A}) = \mathcal{L}(\mathcal{A}).\Sigma^\omega$ be the set of all infinite extensions of words from the language of $\mathcal{A}$. Then the complementary set $\overline{\mathcal{L}^\dagger(\mathcal{A})}$ is the set of all parametric event traces satisfying the regular safety property φ expressed by $\mathcal{A}$.[2] An extracted SR-DFA $\mathcal{A}$ is intended to *over-approximate* the set $Exec \subseteq Traces(TS)$ of execution traces used in the mining process. If $Exec$ is adequate and $\mathcal{A}$ is not too constrained,[3] it is then hoped that $\mathcal{A}$ also over-approximates the set $Traces(TS)$ of all possible observable behaviors of TS. That is, $Traces(TS) \subseteq \overline{\mathcal{L}^\dagger(\mathcal{A})}$. An extracted SR-DFA

[1]Since precision controls the balance between **false positives** (that can overwhelm system administrators and render the IDS useless) with **false negatives** (missed attacks that can have catastrophic consequences).

[2]Since $\mathcal{L}(\mathcal{A})$ is the set of all bad prefixes of φ.

[3]ParaMiner has many parameters used to control the precision of $\mathcal{A}$.

$\mathcal{A}$ might, however, be proven to be violated during RV of TS. This failure, in the form of a *counterexample*, is a finite trace codifying a possible (but not necessarily correct) system behavior rejected by $\mathcal{A}$. Basically, what we are doing by formally verifying an extracted specification is submitting *equivalence queries* [12] to an oracle that decides whether the extracted specification is satisfied by the real system. Counterexamples can be used to filter out spurious specifications [168], *only if* these counterexamples belong to $Exec$, behaviors already explored by the test suite.[4] Otherwise, if the counterexamples belong to behaviors never explored by the test suite used in specification mining, then either a hole in that test suite has been identified[5] or a design bug has been found.[6]

5.2 Prior Work

Learning properties of software behavior has long history. Many tools [5, 8, 114, 157] rely on static program analysis to construct approximate formal specifications of software behavior, rather than utilize actual execution traces. These tools are, therefore, subject to limitations of static analysis [138] including undecidability, imprecision and aliasing. On the other hand, dynamic analysis tools, such as ParaMiner, can be unsound [139]. That is, they can return specifications not satisfied by the analyzed programs in all situations. In [139], it was empirically shown, with the help of a static checker, that most specifications generated from program runs are satisfied by the implementations, a positive result that increases confidence in adopting dynamic specification mining techniques.

Target Specifications. In many tools, such as DIDUCE [85] and Daikon [67], the focus is on (state) invariants rather than temporal properties.

[4]This cannot happen with a *sound* specification miner, such as ParaMiner, that *over-approximates* the set $Exec$.

[5]Only if the counterexamples represent correct (i.e., intended) design behavior, in which case, the mined property should be rejected.

[6]Only if the counterexamples represent incorrect (i.e., unintended) design behavior.

Static Analysis Methods. An example static analysis tool, JIST [8], extracts the (undocumented) correct sequencing of method calls that client code must invoke on a software component. It employs predicate abstraction of the Java source code and then uses regular language learning algorithms [12, 148] to synthesize a deterministic finite automaton (DFA) that over-approximates the interface specification. In [157], only event sequences *for individual objects* are considered, which ignores or misses many interesting interactions involving multiple objects and/or threads.

Dynamic Analysis Methods. An important and pioneering work on parametric specification mining is [10], which discovers specifications that capture temporal and data-dependence relations among invocations of an API made by a program. However, two assumptions are made in [10] that *a priori* limit inferrable specifications. First, interaction scenarios that manipulate no more than k objects are extracted, where k is a learning parameter. Second, the user is allowed to control which scenarios to extract by supplying a set of scenario seeds or skeletons. In this thesis, we do not limit the number of data objects or threads participating in an interaction and we allow the salient design behaviors speak for themselves. Parameters are provided to control the trade-off between precision and mining time/space complexity, without constraining the *forms* of specifications inferred or capturing temporal relations within a bounded time window.

Another parametric specification mining tool, JMiner [110], extracts parametric specifications[7] from unit tests of Java packages. A *trace slicer* first extracts subsequences or *slices* of related interactions from parametric execution traces. Then the resulting non-parametric trace slices are processed by an off-the-shelf non-parametric property learner (e.g., a PFSA learner [145]). However, JMiner implicitly assumes that all occurrences of a given base event (i.e., method or function name) have the same set of variables used as parameters. This is a serious limitation on the class of properties that can be inferred by JMiner. Many interesting

[7]Defined in [110] as specifications carrying parameters that are bound to concrete objects at runtime.

properties of programs constrain how those programs manipulate different objects at different times with the same method or subroutine. For example, in a simple locking discipline, if a program thread t calls **Lock(x1)** and then **Lock(x2)** on two different objects **x1** and **x2**, it may be required to release locks in the same (or opposite) order they were acquired. In this thesis, ParaMiner does not impose any restrictions on how variables are used as event parameters. Moreover, event parameters can be don't-cares or negated as explained later. One more serious limitation of JMiner is that by separating trace slicing and property learning, the interplay between the sequencing of API calls and choices of manipulated objects by each call is lost, leading to less precise models.

In [119], the interplay between data values and interaction sequences are represented by extended FSMs (or EFSMs), which are FSMs whose edges are annotated with constraints on data values. EFSMs are extracted from interaction traces. However, EFSMs do not have memory other than EFSM states, which means that EFSMs do not have a means to remember values encountered at any transition edge and make subsequent transitions contingent on seeing that value again. The authors in [119] make another limiting assumption, namely that each interaction trace belongs to one thread. Therefore, mining EFSMs cannot capture multithreaded interactions or collaborations over a shared set of objects, which has been a major challenge (and thrust) to verification efforts, especially in today's multiprocessor systems. EFSMs are similar to parametric FSMs used by ParaMiner, but use rigid FOL predicates over parameter data as well as a finite number of other program variables. In RV tools, rigid predicates are implemented by providing callback functions that need to be called on actual parameter data whenever an event occurs. This can significantly increase RV impact on monitored program performance.

Applications. Specifications mined from correct as well as erroneous traces are used to localize or diagnose errors [113].

Complexity. Learning an automaton from samples of its behavior is NP-hard [79]. Multiple

	Universal	Existential
Positive	All extensions of w are in L.	At least one extension of w is in L.
Negative	All extensions of w are not in L.	At least one extension of w is not in L.

Table 5.1: Definitions of a positive (negative) universal (existential) prefix w with respect to an ω-regular language L.

smaller properties can be learned and then composed into more complex specifications. In [90], it was proved that the class of ω-regular languages (i.e., those accepted by Büchi automata) are not identifiable in the limit from samples of their (universal and existential) prefixes, as defined in Table 5.1. However, the class of safety languages is identifiable in the limit from positive existential and negative universal sample prefixes.

Multiple Sequence Alignment. In [116], it is mentioned that global sequence alignment, when used to measure inter-trace distances for clustering purposes, will fail to recognize closely related traces which differ only in the number of loop iterations. They, instead, input regular expressions inferred from traces to the sequence alignment algorithm. However, in this thesis, global sequence alignment proves to be a powerful tool to construct parametric automata from such traces, because it can accommodate these kinds of divergences among traces by inserting gap symbols.

Abstraction of State Space. Identifying the internal program state with interface API calls completely ignores internal state producing FSMs that might have modest predictive power (i.e., modest precision in constraining observable design behavior), which is unacceptable in many applications. In reality, two identical interface API calls might be produced by different internal program states. So collapsing execution traces into sequences of interface API calls incurs too much information loss. Alternatively, interface API calls should be used as an alphabet set Σ annotating *transitions* (rather than *states*) of a SR-DFA whose states abstract the program internal state space while preserving as much information as possible.[8] FSMs mined based on *explicit* state vectors (i.e., bit-vectors that make selective transitions

[8]This is the typical usage model in formal verification where transitions of a Büchi automaton derived from a LTL formula are triggered by the labels of the DUV transition system states.

among a limited set of values) may result only in reverse-engineering the program state space (or parts thereof) and lacks any abstraction or generalization, and is thus not scalable and may duplicate program bugs in the extracted specification. In this thesis, we construct SR-DFAs based on an *implicit* or *hidden* state space whose size is dictated by the complexity of execution traces. This is expected to abstract away many details pertinent only to the implementation and, hence, hopefully extracts more understandable bug-free specifications.

Window-Based Methods. To reduce computational complexity, only relations among episodes within a preset sliding window are considered by many tools. All window-based tools have severely limited ability to discover long-range temporal relations among events[9], which are quite common in many environments. For example, in a web server, communications with clients can introduce wild latency variations.

5.3 ParaMiner Specification Mining Flow

The specification formalism of ParaMiner is SR-DFAs introduced in [136]. In Chapter 2, we developed a rigorous mathematical basis for SR-DFAs and their semantics. A formal specification consists of one or more *properties*, where each property establishes a relation between events in temporal sequences.

5.3.1 A Bio-Inspired Flow

The ParaMiner flow is inspired by protein folding [58], where a newly synthesized chain of amino acids folds into a 3-d protein structure depending on the amino acid sequence and environmental influences. Similarly, in the ParaMiner flow, traces of observations or events extracted from program executions will *self-assemble* into varied finite automata ex-

[9]In order to reduce search space, the window size must be kept small.

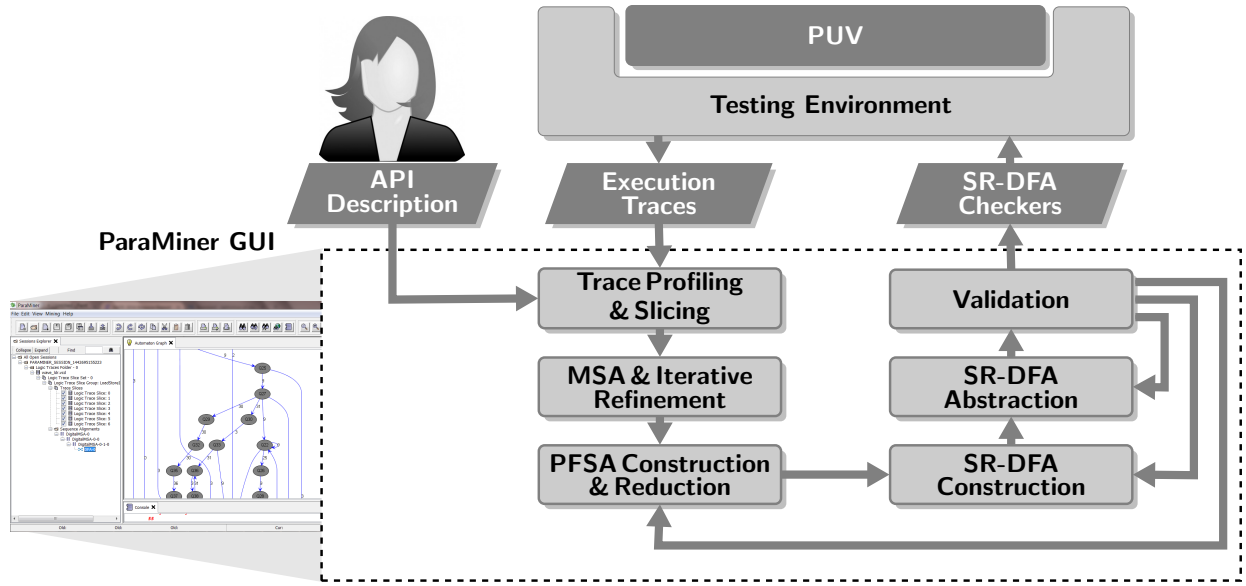


Figure 5.1: ParaMiner specification mining flow. ParaMiner is part of a multi-purpose specification mining Java application with 80,000+ lines of code, that also supports mining DFAs from digital logic simulation traces [137].

pressing different behavioral properties or aspects of a program.[10] There are more parallels between protein folding and ParaMiner. For example, to work with long execution traces in ParaMiner, closely related states will cluster into *superstates* or *macrostates* that can then be used in a similar manner as building blocks for higher-level superstates until the overall structure emerges. Similarly, large protein molecules fold in modules [58], since folding takes place almost independently in different segments (or *domains*) of the long chain of amino-acid residues. Moreover, folding nuclei form among a small number of residues and allow the global topology of the protein structure to rapidly condense around them [58]. This modular mechanism in proteins is also crucial for the assembly of ever larger molecules.

The specification mining flow is shown in Figure 6.2.

[10]The folding mechanisms used by ParaMiner are multiple sequence alignment (MSA) and PFSA state merging.

5.3.2 API Description

The monitored API (e.g., Linux system calls, Java standard libraries, etc.) is described to the ParaMiner in terms of function (or method) names, argument types, and return types. Parametric events and traces naturally follow the language of FOLe.

ParaMiner accepts parametric traces from a variety of sources, from low-level syscall monitors to high-level command-line calls or database accesses.

5.3.3 Trace Recording

ParaMiner is based on dynamic analysis of execution traces (e.g., system calls, Java API method calls, etc.). Specification mining is premised on the availability of fairly high-quality (but not necessarily bug-free) designs that can be exercised to reveal the most relevant aspects of behavior *without triggering bugs*, which can then be captured in a formal specification. ParaMiner needs a database of parametric execution traces providing sufficient coverage of system behavior in various situations (e.g., during system start-up, during normal operation and in response to various transactions) in order to increase model precision. The monitored Linux applications are instrumented at the system-call interface using the `strace` utility that intercepts system calls and records calling thread context, argument values (tracking down `struct` and array pointers) as well as return values. Since we also use the DaCapo-9.12-bach benchmark suite [27] of Java programs and because Java intrinsically supports reflection and byte-code instrumentation, we created an instrumentation agent *ParaTracer* that, with the help of Javassist-3.19 [40], instruments all (public) method calls on standard Java SE and EE packages as used by any benchmark programs. A database of execution traces providing sufficient coverage of system behavior in various situations (e.g., during system start-up, during normal operation and in response to transient events) is needed to increase model precision.

In this thesis, the data set is a collection of finite traces $\{u_1, \ldots, u_n\}$ of events (e.g., database queries or system calls) possibly resulting from interaction of the monitored system with multiple agents exhibiting a sufficiently rich set of behaviors.

On Linux, `ltrace` can be used to intercept and record calls to dynamically linked libraries and system calls made by an executed process and the signals it receives. It should be noted that `ltrace` only traces calls from the instrumented to the dynamically linked libraries and does not trace calls between or within those libraries. This should not be a problem, since the goal is usually to monitor how an application interacts with a library. However, `latrace` can be used to intercept intra-library calls.

5.3.4 Trace Slicing, Segmentation and Folding

It is important to simplify automata-based descriptions of complex systems by effectively introducing *hierarchy* and *concurrency* [86]. Without hierarchy (modularity, state clustering or abstraction) and concurrency (independence or orthogonality), those state-based descriptions become unwieldy, unstructured and incomprehensible.

Untangling Concurrency - Trace Slicing. A single trace may interleave multiple unrelated, but concurrent, activities (or aspects) by the same thread or different threads. These unrelated aspects should be isolated into different *trace slices* that derive separate specifications. Thus, to account for concurrency of many activities inherent in observed traces, *trace slicing* [39, 110] extracts multiple subsequences (or *slices*) of an execution trace that satisfy a *slicing criterion*.

Delimiting Hierarchy - Trace Segmentation. Moreover, a long observation trace of an instrumented program typically exhibits (or cycles through) different *phases*[11] or regions of considerable statistical regularities, as shown in Figure 6.4 produced by ParaMiner for many

[11] Large-scale behavioral features.

of the benchmarks used in this thesis. Within each phase, a trace has a unique composition of event types. Regions having relatively stable statistics, called *phases*, *superstates* or *macrostates*, can be easily delineated when viewed at multiple scales. This implies a *hierarchy* of states. To enable specification mining from very long trace slices without severely suffering from the quadratic complexity of MSA, each phase designated by the user is assigned a weight according to its length. With each phase, we keep only a subset of the folded trace segments in direct proportion with the weight of each phase. The entry and exit segments are always kept, since they carry information about transitions into and out of each phase. We need a segmentation algorithm that identifies *contiguous* homogeneous intervals of a given trace, where homogeneity is a parameter that controls the trade-off between the number of intervals and their average size. An algorithm, such as k-means clustering, which does not take into account contiguity of clustered events may not be suitable. Candidate algorithms include: (1) A top-down split-and-merge segmentation using a binary tree with time complexity of $\mathcal{O}(n \log n)$ (2) A bottom-up unseeded region growing method. (3) Multiscale segmentation based on a scale parameter that generates a hierarchy of segments or regions (e.g., by heat diffusion with position-dependent, anisotropic conduction parameter). In [140], it is proved that no new artificial features[12] are introduced as the scale parameter t is varied from fine to coarse scale (a principle of *causality*). The conduction parameter is chosen[13] locally at each position x along a trace as a monotonically decreasing function $g(\cdot)$ of the (discrete-space) derivative of the feature vector $c(x,t) = g(\| \mathbf{F}(x,t) - \mathbf{F}(x-1,t) \|)$.

State Identification - Trace Folding. Once the lengths and boundaries of all phases have been properly selected, MSA will then be used (as explained in Section 5.4) to *fold* the given trace slice locally within each phase while preserving the transitions among consecutive phases. So a superstate is formed by folding a single long strand or trace within each phase. The instrumented program exhibits recurrent behaviors of possibly different characteristic

[12]Defined in [140] as "blobs" or extrema (i.e., maxima or minima).

[13]With adiabatic boundary conditions, i.e., the conduction coefficient is set to zero at both ends of the trace.

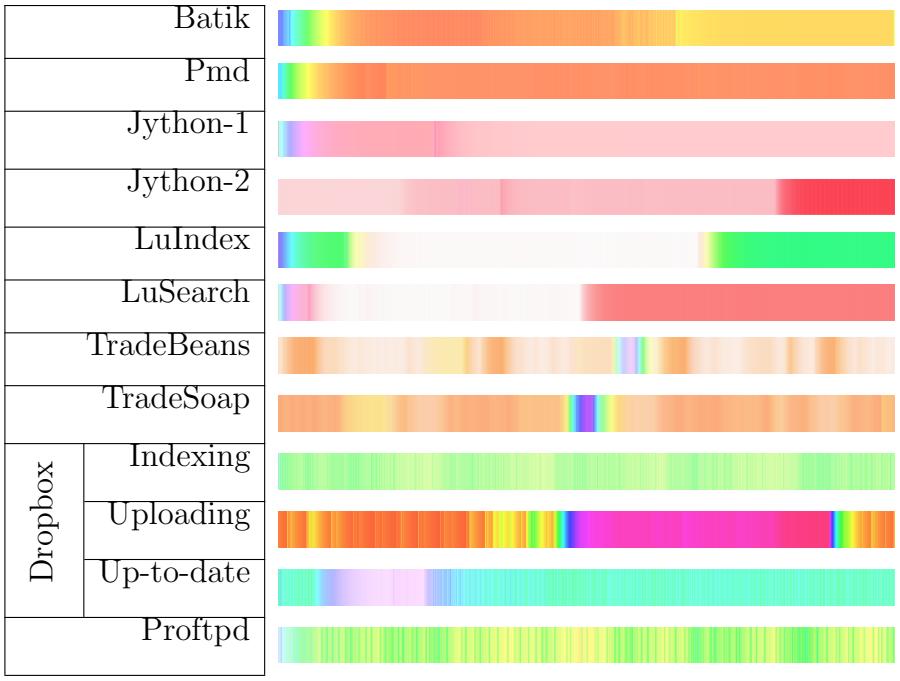


Figure 5.2: Examples of *super-state structure*: distinguishable program phases with time as captured by the largest two principal components (displayed as color hue and saturation, respectively) of method-name histograms within a sliding window of size 200 events.

lengths within each phase and, hence, a different optimum *folding length* should be used for each phase. Thus, the users of ParaMiner need to specify for each trace which regions are considered foldable phases and what is the folding length inside each region, taking into account that complexity of profile-based MSA grows rapidly with folding length and the number of slices.

It is crucial to select slice lengths and boundaries properly so as to ensure that slices contain complete episodes of the behaviors of interest. This helps to improve the quality of subsequent MSA and the final outcome.[14]

Slicing Criteria. Users of ParaMiner may specify any combination of the following slicing criteria:

- Given a user-defined threshold parameter **T** and a parametric trace W, then for every

[14]Slices that are too large are undesirable, since subsequent sequence alignment has time complexity that is quadratic in slice length.

type $s \in \mathbf{S}$, if any value v of type s is observed as an argument to at least $\mathbf{T}$ parametric events in W, then the subsequence of all events containing v is extracted.

- API Groups: To reduce mining noise, we allow users to turn on related API method groups in a given mining run; otherwise, unrelated API methods would clutter the obtained SR-DFAs. ParaMiner organizes APIs (e.g., classes of Java standard library, Linux system calls, etc.) into smaller subgroups so that specification mining may focus on one or more aspects of a program at a time (e.g., file access, file manipulations, memory management, networking, concurrency, access control, etc.).
- Constants: We also allow users to specify certain values to be used in trace slicing, so that coherent cross-sections of system behavior are obtained. For example, in the **proftpd** benchmark, there is a leader process that monitors FTP connection requests and spawns processes that handle these connections. So it may be desired to separate the specification of the leader process from that of its child processes. In that case, the process ID (PID) of the leader process needs to be specified explicitly as a slicing criterion.
- Data Types: The use of data types (by virtue of multi-sorted FOLe) increases the resolving power of extracted specifications (i.e., their ability to resolve anomalous behavior from normal behavior), since objects of different types can be confused for one another. For example, threads (or processes) can be assigned different types according to which thread function (or executable, respectively) they invoke. Thus, behaviors originating from different parts of program code are kept separate in the resulting specification.

Because it is a main contribution of this thesis, ParaMiner's stage following trace slicing is discussed in Section 5.4.

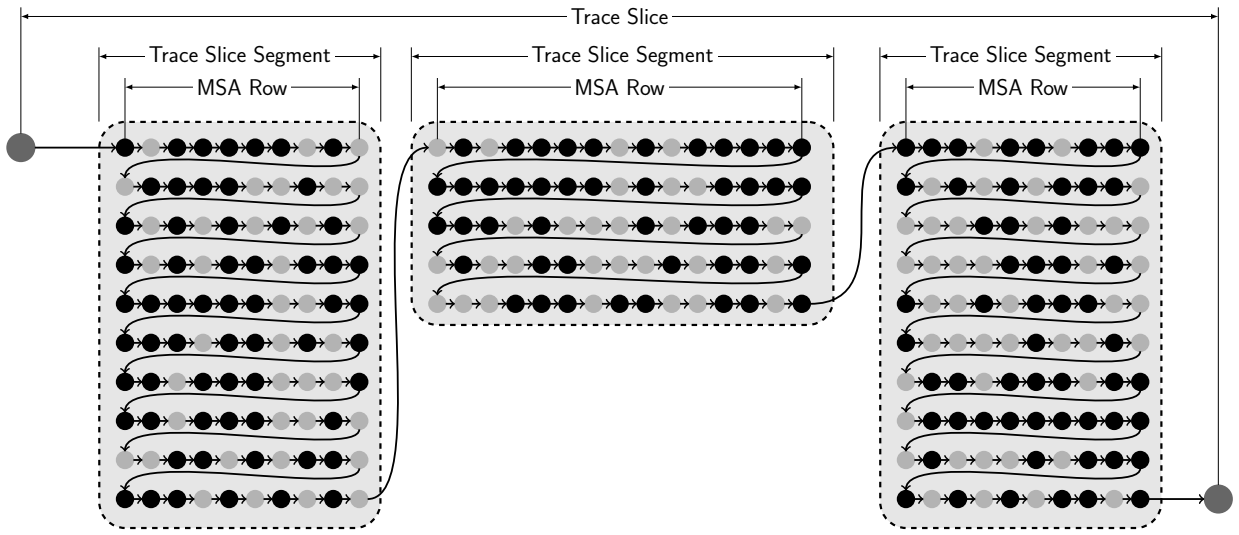


Figure 5.3: Foldable trace segments are used isolate and extract superstates of a program by conducting MSAs independently within each phase.

5.4 Role of Sequence Alignment

5.4.1 Initial State Uncertainty.

Classical automata-learning algorithms [12, 25], that infer finite-state automata from their observable input-output behavior, assume the existence of a means to *reset* the automaton being learned to a fixed start state. This has been identified as a serious limitation in [148].[15] To obviate the use of resets in Angluin's learning algorithm and handle the **initial-state uncertainty problem** [154], *homing sequences*, introduced in [134], can be inferred and applied so that the learning algorithm *homes* in on a system state that can be uniquely determined from the observable output [148]. But, unlike a reset, the final state reached cannot be known beforehand, since it depends on the current *unknown* state. A *synchronizing sequence* [154] is a sequence of inputs that leads to a unique final state independently of the starting or initial state. But an automaton need not have a synchronizing sequence and, if it does, homing sequences are usually shorter [154]. However, the use of resets or homing

[15]This reset can be used to simulate backtracking by restarting and stopping at the desired point.

sequences [148] is only meaningful for *active learning* of automata, where it is feasible to proactively apply a reset or a homing sequence and execute queries. In this thesis, only *passive learning* using nonintrusive observation to construct tentative automata is of interest. Therefore, we do not impose any assumptions or special prior structure (such as seed methods or a maximum number of manipulated objects as in [10]) on the software behavior yet to be discovered. Moreover, since a superstate may start anywhere within a trace and may have multiple entry points, it desired to learn the internal structure of a super state *without knowing the initial state*.

Only a few manual interventions are occasionally required from a user, such as selection of initial states which happens after a relatively small SR-NFA has already been mined. This enables users to make more informed decisions about the form of inferred specifications.

5.4.2 Multiple Sequence Alignment

Now, we explain how multiple sequence alignment (MSA), widely used for biological sequence analysis [59], holds the answer to the above *initial-state uncertainty problem* [154] in passive learning contexts.[16] For every sequence $\boldsymbol{\sigma} = (\sigma_0, \sigma_1, \sigma_2, \ldots)$ of parametric events, there is an implicit *unknown* labeling or *path* $\mathscr{L} : \mathbb{N} \to S \times \Lambda$ that maps each event to a *hidden state* reached at that event in transition system $TS' = TS \bigotimes \mathcal{A}$. Conceptually, a path $\mathscr{L} = (s_0, \lambda_1), (s_1, \lambda_2), \ldots$ represents two *synchronous* and *parallel* runs $s_0, s_1, \ldots$ and $\lambda_1, \lambda_2, \ldots$ of TS and $\mathcal{A}$, respectively, where there is $\lambda_0 \in \Lambda$ such that $\forall i \geq 0 : \ \sigma_i = L(s_i)$ and:

$$\lambda_0 \xrightarrow{\sigma_0} \lambda_1 \xrightarrow{\sigma_1} \lambda_2 \xrightarrow{\sigma_2} \lambda_3 \ldots \quad \| \quad s_0 \longrightarrow s_1 \longrightarrow s_2 \longrightarrow s_3 \ldots$$

Given N sequences $\{\boldsymbol{\sigma}^1, \ldots, \boldsymbol{\sigma}^N\}$ of parametric events, assume, for the sake of argument, that a corresponding set of hidden-state paths $\{\mathscr{L}_1, \ldots, \mathscr{L}_N\}$ is also given. Then the (possibly nondeterministic) transition structure of $S \times \Lambda$ (or a small part thereof) can be reconstructed

[16] As opposed to active learning algorithms [12, 25, 148].

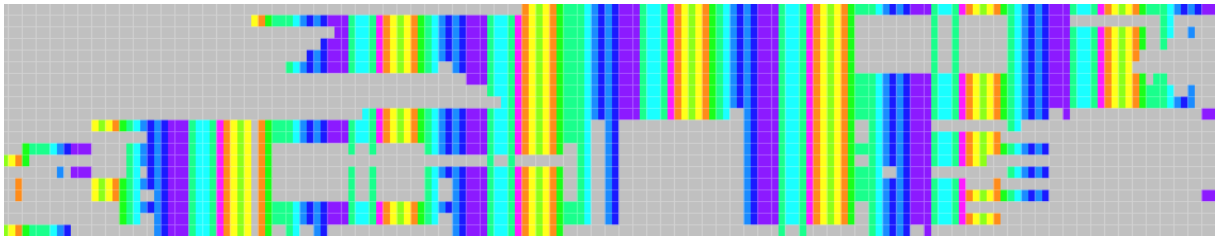


Figure 5.4: A section of an example MSA of 10 syscall trace slices from the `proftp` benchmark. Each row is a slice, gaps are gray and every event is depicted by a different color.

from these paths by noting that for all $1 \leq k \leq N$, we have:

$$\left.\begin{array}{l} \mathscr{L}_k(\sigma^k_{i-1}) = (s, \lambda) \\ \mathscr{L}_k(\sigma^k_i) \quad = (s', \lambda') \end{array}\right\} \Longrightarrow \left\{\begin{array}{c} s \longrightarrow s', \lambda \xrightarrow{\sigma^k_i} \lambda' \\ L(s') = \sigma^k_i \end{array}\right.$$

An alternative viewpoint, that will later help to recover the set $\{\mathscr{L}_1, \ldots, \mathscr{L}_N\}$ of paths traversed by a set of (known or observable) event sequences $\{\boldsymbol{\sigma}^1, \ldots, \boldsymbol{\sigma}^N\}$ in an unknown state space $TS \bigotimes \mathcal{A}$, posits that the set $\{\mathscr{L}_1, \ldots, \mathscr{L}_N\}$ induces an alignment on these event sequences, where any two events σ^k_i and σ^l_j from $\boldsymbol{\sigma}^k$ and $\boldsymbol{\sigma}^l$, respectively, can be aligned iff $\mathscr{L}_k(\sigma^k_i) = \mathscr{L}_l(\sigma^l_j)$ even if $\sigma^k_i \neq \sigma^l_j$. In a MSA, as shown in Figure 6.5 and Figure 5.5, two or more event sequences are arranged as rows of a 2-dimensional matrix so that, in each column, events from one or more sequences are aligned if they have identical labels in $S \times \Lambda$. Due to the intrinsic variability of program behaviors (which are represented by languages over finite or countable alphabets), different event sequences cannot typically be perfectly superimposed or aligned. Non-alignable positions are filled with gap symbols. Moreover, there is not a unique MSA induced by every set of paths $\{\mathscr{L}_1, \ldots, \mathscr{L}_N\}$. It is only possible to find a MSA that maximizes a given optimality criterion (or scoring scheme) that ranks different MSAs.

The main premise of ParaMiner is that, conversely, hidden-state paths $\{\mathscr{L}_1, \ldots, \mathscr{L}_N\}$ can be recovered (*up to an equivalence relation* over $S \times \Lambda$) if an appropriate MSA of the event sequences can be established, which can then be used to reconstruct an abstract version of TS or $\mathcal{A}$ or their product $TS \bigotimes \mathcal{A}$.

However, there will still be many variants or alternatives that are nearly as good or as likely as an optimal alignment [59, Chapter 4]. This brings forth the ill-posed nature of the sequence alignment problem in specification mining, which necessitates the use of incorporating prior information (see Section 5.7.7) or regularization in the form of Occam's razor, where we seek the alignment that results in the smallest SR-NFA.

One complication not present in conventional MSA problems is that each parametric event in a trace slice has concrete argument values. So many ensemble states that label events in different trace slices will possess essentially the same structure except for the actual values they memorize. To leverage the similarity among different trace slices (as well as within a given trace slice), we need to factor out the dependence of ensemble states on the exact argument values and focus on their structure by defining an equivalence relation over the ensemble state space Λ. Two composite states (s_1, λ_1) and (s_2, λ_2) in $S \times \Lambda$ are equivalent (denoted $(s_1, \lambda_1) \equiv_E (s_2, \lambda_2)$) *only if*[17] there is a set of bijections (rearrangements or permutations) $f_s : |\mathsf{S}|_s \to |\mathsf{S}|_s$ for all $s \in \mathbf{S}$ such that $\lambda_1 \odot \mathbf{f}_\Xi = \lambda_2$ and $\mathbf{f}_\Sigma(L(s_1)) = L(s_2)$, where $\odot$ denotes function composition. Also, $\mathbf{f}_\Xi : \Xi \to \Xi$ and $\mathbf{f}_\Sigma : \Sigma \to \Sigma$ are the automorphisms over Ξ and Σ, respectively, induced by the set of maps $\{f_{s_1}, \ldots, f_{s_n}\}$. This symmetry under bijections formalizes the notion of *abstracting* away the concrete argument values of event predicates and replacing them with formal parameters. This can be formalized as follows. Let $\mathcal{FP}$ be a countable set of formal parameters. Also, let $\Xi(\mathcal{FP})$ be the set of all abstract variable valuations $\xi : \mathcal{X} \to \mathcal{FP}$, let $\Lambda(\mathcal{FP})$ be the set of all abstract ensemble states $\lambda : \Xi(\mathcal{FP}) \to Q$, and finally let $\Sigma(\mathcal{FP})$ be the set of all FOLe atoms of the form $p(x_1, \ldots, x_n)$ with each $x_i \in \mathcal{FP}$. Then we have this theorem:

Theorem 5.1. *For every $v \in (S \times \Lambda)/\equiv_E$, there is a FOLe atom $\sigma' \in \Sigma(\mathcal{FP})$ and there is $\lambda' \in \Lambda(\mathcal{FP})$ such that for every $(s, \lambda) \in v$, there is a set of maps (not necessarily bijections) $f_s : |\mathsf{S}|_s \to \mathcal{FP}$ for all $s \in \mathbf{S}$ with $\lambda \odot \mathbf{f}_\Xi = \lambda'$ and $\mathbf{f}_\Sigma(L(s)) = \sigma'$*

[17] Conversely, if those bijections exist, it is not necessary that $(s_1, \lambda_1) \equiv_E (s_2, \lambda_2)$. Thus, if $\equiv_E$ is such an equivalence relation over $S \times \Lambda$, any *finer* relation is also allowed.

Proof. This theorem follows directly from transitivity of the equivalence relation $\equiv_E$. $\square$

This way, events are abstracted into FOLe atoms of the form $p(x_1, \ldots, x_n)$ before MSA starts, where x_i belong to a set $\mathcal{FP}$ of formal parameters. In Section 5.9.2, we discuss how argument values in different trace slices to be aligned are replaced with *variable symbols* or *constant symbols* chosen in such a way that similarity among the trace slices is maximized so as to minimize the automaton ultimately inferred from the resulting MSA without unnecessarily losing any discriminative data patterns latent in those traces. Then MSA will be used to construct a *finite* abstract version $\mathbb{TS} = (\mathbb{S}, \twoheadrightarrow, \mathbb{I}, \Sigma(\mathcal{FP}), \mathbb{L})$ of $TS' = TS \otimes \mathcal{A}$ rather than TS' itself, where $\mathbb{S} = (S \times \Lambda)/\equiv_E \subseteq 2^{S \times \Lambda}$ is a quotient space consistent with the observed traces.

5.5 Ensemble States from MSAs.

Before discussing MSA algorithms, we first explain how to construct a probabilistic finite-state automaton (PFSA) [52, 145] $\mathcal{G}_{\mathbf{m}} = (\Lambda', \Sigma(\mathcal{FP}), Pr)$ that represents $\Lambda(\mathcal{FP})$ from a MSA $\mathbf{m}$, as shown in Figure 5.5, where $\Lambda' = \Lambda(\mathcal{FP})$ is a finite nonempty set of abstract ensemble states, $\Sigma(\mathcal{FP})$ is the set of FOLe atoms over $\mathcal{FP}$, and $Pr : \Lambda' \times \Sigma(\mathcal{FP}) \times \Lambda' \to [0, 1]$ is the transition probability function[18] such that:[19]

$$\forall v \in \Lambda' : \sum_{v' \in \Lambda'} \sum_{a \in \Sigma(\mathcal{FP})} Pr(v, a, v') = 1$$

A PFSA state $v \in \Lambda'$ is connected by a directed edge to another state v' if, for at least one of the aligned trace slices, a letter in column $\mathbf{m}_v$ directly follows a letter in column $\mathbf{m}_{v'}$,

[18] Transition edges having zero probability are not depicted.

[19] The standard definition of PFSA has:

$$\forall v \in \Lambda',\ \forall a \in \Sigma(\mathcal{FP}) : \sum_{v' \in \Lambda'} Pr(v, a, v') = 1$$

This standard definition only replaces nondeterminism with probability.

possibly with intervening gap symbols only. Thus, every trace slice is a path in $\mathcal{G}_{\mathbf{m}}$. Every PFSA edge is labeled with the alphabet symbol in the MSA column associated with its sink PFSA state. This graph representation has been called a partial-order graph in [111]. An edge in $\mathcal{G}_{\mathbf{m}}$ is annotated with transition probability according to how many trace slices follow that edge in $\mathbf{m}$. For these transition probabilities to be accurate, it is desired to align as many trace slices as possible (i.e., to have *deep alignments*).

It is typical in many PFSA construction algorithms [52, 145] to represent the set of input traces as a prefix tree, which is then followed by state merging. On the other hand, a MSA represented by a dag is much more compact than the corresponding prefix tree, since all letters in the same column are identified. Thus, MSA improves the runtime of PFSA inference algorithms.

A PFSA is constructed by noting that all identical atoms $a \in \Sigma(\mathcal{FP})$ in the same column $\mathbf{m}_i$ of $\mathbf{m}$ stand for a single *unknown* (abstract) hidden state $v \in (S \times \Lambda)/\equiv_E$. Since the structure of $(S \times \Lambda)/\equiv_E$ is unknown, we use a unique label v for every column of $\mathbf{m}$. In Section 5.8, we will identify (i.e., merge) PFSA states that are *approximately equivalent* in a probabilistic sense. The label $a \in \Sigma(\mathcal{FP})$ of a PFSA edge $(\lambda'_1, a, \lambda'_2) \in \Lambda' \times \Sigma(\mathcal{FP}) \times \Lambda'$ in $\mathcal{G}_{\mathbf{m}}$ can be inferred from Equation 5.1:

$$\forall v_1, v_2 \in \mathbb{S} \frac{v_1 \twoheadrightarrow v_2,\ \mathbb{L}(v_2) = a}{\exists \lambda'_1, \lambda'_2 \in \Lambda(\mathcal{FP}) : \lambda'_1 \xrightarrow{a} \lambda'_2} \tag{5.1}$$

In Section 5.8, reductions are applied to a PFSA $\mathcal{G}$ to reduce its size and improve its generalization performance.

Stated differently, $v_1, v_2 \in \mathbb{S}$, $v_1 \twoheadrightarrow v_2$ and $\mathbb{L}(v_2) = a \in \Sigma(\mathcal{FP})$ *implies* that there is $\lambda'_1, \lambda'_2 \in \Lambda(\mathcal{FP})$ such that for every $\lambda_1 \in \Lambda$ which is an instantiation of λ'_1, there is $\lambda_2 \in \Lambda$ which is an instantiation of λ'_2 such that $\lambda_1 \xrightarrow{\sigma} \lambda_2$ with $\sigma \in \Sigma$ being an instantiation of a.

5.5.1 Initial State Selection

As shown in Figure 5.5, a fictitious starting state is added as a source state for all the aligned sequences. In our experiments, this state proved to be valuable, since it usually allows monitoring to start anytime during execution and the extracted SR-NFA automatically detects the appropriate next state or states. Thus, users of ParaMiner may not need to specify an initial SR-NFA state.

5.6 From Ensemble States to SR-NFA States.

How to reconstruct a SR-NFA $\mathcal{A}$ consistent with the set of observed traces? MSA yields an *abstract* ensemble state space $\Lambda(\mathcal{FP})$, where each abstract state $\lambda'_1 \in \Lambda(\mathcal{FP})$ stands for a set of possible ensemble states. There is not a unique SR-NFA consistent with a given $\Lambda(\mathcal{FP})$. For example, if we consider each $\lambda'_1 \in \Lambda(\mathcal{FP})$ to contain only one ensemble state λ in which all replicas of $\mathcal{A}$ have the same state $q \in Q$ (i.e., $\forall \xi \in \Xi : \lambda(\xi) = q$), we get a SR-NFA which is a classical NFA with a state space Q identical to $\Lambda(\mathcal{FP})$, since it responds only according to event predicate symbols ignoring any data carried by event arguments. To avoid this degenerate extreme, we now devise an iterative data-flow algorithm that constructs

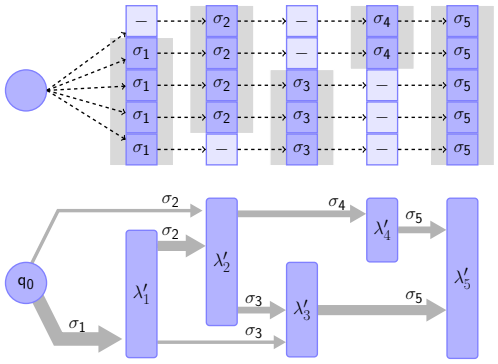


Figure 5.5: A MSA viewed as a PFSA. Each state corresponds to one MSA column. Each edge is annotated with σ of the MSA column associated with its sink PFSA state. Edge line width is proportional to transition probability.

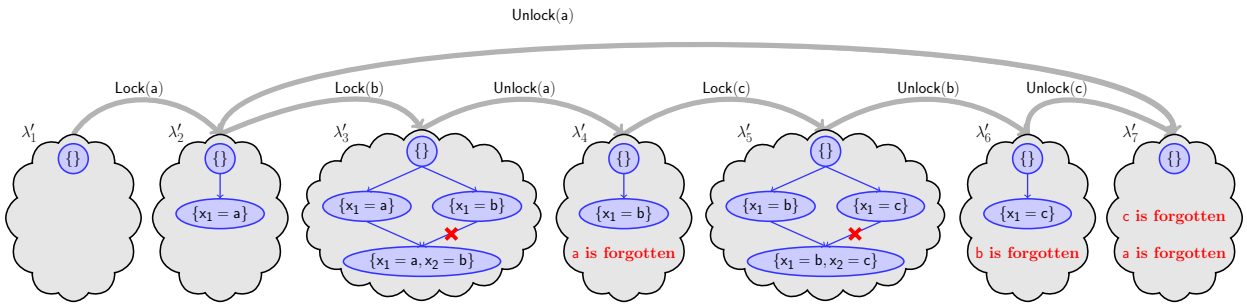


Figure 5.6: Ensemble-to-automata states.

the *finest* SR-NFA $\mathcal{A}$ consistent with a given $\Lambda(\mathcal{FP})$. It can later be reduced in size at the cost of less precision.

We now explain how a SR-NFA $\mathcal{A}$ can be recovered from a given MSA $\mathbf{m}$.

MSA yields an abstract ensemble state space $\Lambda(\mathcal{FP})$, as shown in Figure 5.6. Each abstract ensemble state $\lambda' \in \Lambda(\mathcal{FP})$ is an unknown function $\lambda' : \Xi(\mathcal{FP}) \to Q$ whose LBFG representation we now construct. Note that the set $\mathcal{FP}$ of *formal parameters* used during the MSA phase to label PFSA edges are not the same as the set $\mathcal{X}$ of FOL variables used by SR-NFAs. They are intended only to abstract the actual values observed among multiple traces so that they can be aligned and recurrent behavior extracted. This becomes clear when we note that one and the same event may cause bindings of different sets of variables according to the current state of the spawning replica. For example, in Figure 2.1, if a $\mathsf{lock}(1,1)$ event is observed, a replica at state q_1 will spawn a new replica binding $(t = 1, n_1 = 1)$, and a replica (already having $(t = 1)$) at q_2 will spawn a new replica binding $(n_2 = 1)$. Therefore, a single edge in the abstract ensemble state space $\Lambda(\mathcal{FP})$ might correspond to multiple edges in an underlying SR-NFA. Therefore, it is desired to construct one or more FOLe formulas $f \in \mathsf{Atoms}$ consistent with every edge label obtained from Equation 5.1.

To limit checker population size, it is necessary to note that an ensemble state is characterized by the set of data values (from the set $\mathcal{FP}$) it remembers from past events and the set of combinations of those values that are remembered as codified in their association with

variables. It is important to specify when a combination of values in an ensemble state is forgotten in any of its successor ensemble states. Aligning two events in a MSA effectively declares that, from this point on, their past histories are essentially equivalent or that their differences are irrelevant.[20] Hence, any data values that are not common among all converging histories can be dropped from that point on.

5.6.1 Data-Flow Analysis.

The process used to convert ensemble states to automaton states needs to determine which parts of the ensemble state space that each formal parameter value propagates to. This is similar to optimizing compilers, where a control-flow graph (CFG) is used to represent the set of all possible execution paths of a program where nodes represent basic blocks and edges represent jumps.

For every $\lambda' \in \Lambda(\mathcal{FP})$, if a formal parameter a appears along some, but not all, paths $\lambda'_0 \rightarrow \lambda'$, then this implicitly indicates that a is not an essential part of the history embodied by λ' and, hence, can be removed from λ' (forgotten by λ'). Moreover, an ensemble state needs to remember a value if it is used to decide future transitions. Thus, the formal parameters $\mathcal{FP}_{\lambda'}$ remembered by each ensemble state λ' is the intersection of two sets:

- $\mathsf{AVAIL}(\lambda')$: the set of formal parameters available at λ'.
- $\mathsf{LIVE}(\lambda')$: the set of formal parameters live at λ'.

The computation of $\mathsf{AVAIL}(\lambda')$ and $\mathsf{LIVE}(\lambda')$ for all ensemble states λ' combines the results from a *sequence* of two separate unidirectional analyses:

- A *forward-must analysis* or an available-values analysis, which starts at the initial state λ'_0 and proceeds forward by intersecting, at each ensemble state, sets of values observed

[20] This is an informal statement of Myhill-Nerode theorem [95] which characterizes regular languages in terms of such equivalence classes.

on *all* incoming paths.

- A *backward-may analysis* or *liveness analysis*, which starts at terminal states and proceeds backwards by joining, at each ensemble state, sets of values observed on at least one outgoing path. The liveness analysis ensures that a formal parameter a is not stored in ensemble state λ' if it is not live (i.e., if a is not used on any execution path originating from λ').

In this liveness analysis, using meaningful notions of a *generate set* $\mathsf{GEN}(\lambda')$ and a *kill set* $\mathsf{KILL}(\lambda')$ at each ensemble state $\lambda' \in \Lambda(\mathcal{FP})$ may be effective in reducing the sets of values that need to be remembered at each state. Moreover, the entry node required by the backward liveness analysis can be the same as the exit node. So starting the forward and backward data-flow analyses from the initial ensemble state λ'_0, no formal parameters are remembered at λ'_0: $\mathsf{AVAIL}(\lambda'_0) = \mathsf{LIVE}(\lambda'_0) = \emptyset$.

The data-flow equations at each ensemble state s are given by:

$$\begin{aligned}
\mathsf{AVAIL}(\lambda'_1) &= \bigcap \{ \quad \mathsf{AVAIL}(\lambda'_2) \quad \cup \{a_1, \ldots, a_n\} | \lambda'_2 \xrightarrow{p(a_1,\ldots,a_n)} \lambda'_1 \in \delta_G \} \\
\mathsf{LIVE}(\lambda'_1) &= \bigcup \{ \quad \mathsf{LIVE}(\lambda'_2) \quad \cup \{a_1, \ldots, a_n\} | \lambda'_1 \xrightarrow{p(a_1,\ldots,a_n)} \lambda'_2 \in \delta_G \}
\end{aligned}$$

where $\mathsf{AVAIL}(\lambda'_0) = \mathsf{LIVE}(\lambda'_0) = \emptyset$ is kept invariant. We iteratively solve these equations, as shown in Algorithm 5, until a fixed point is reached (possibly using a work list to improve efficiency). Note that convergence (i.e., reaching a fixed point) is guaranteed, since the partially ordered set $2^{\mathcal{FP}}$ is finite and both set union and intersection are monotonic operations over $2^{\mathcal{FP}}$.

State Assignment

In order to flesh out the internal details of ensemble states, keep in mind that every transition in the ensemble state space corresponds to multiple transitions in the associated automaton, each possibly containing a different set of FOL variables. To minimize the number of variables

ALGORITHM 5: ReachingValuesAnalysis($\mathcal{G}$)

Input: A PFSA $\mathcal{G} = (\Lambda', \delta_G \subseteq \Lambda' \times \Sigma(\mathcal{FP}) \times \Lambda', \lambda'_0)$, where Λ' is the set of abstract ensemble states, $\Sigma(\mathcal{FP})$ is the alphabet set of FOLe events over formal parameters $\mathcal{FP}$ and λ'_0 is the initial ensemble state.

Output: A labeling $\mathcal{R} : \Lambda' \to 2^{\mathcal{FP}}$ that maps every $\lambda' \in \Lambda'$ to the set of formal parameters observed along *all* paths $\lambda'_0 \to \lambda'$.

```
1  foreach (λ' ∈ Λ') do
2      R(λ') ← {};
3  WL ← Λ';
4  while (WL ≠ {}) do
5      λ'_2 ← pop WL;
6      X ← FP;
7      foreach (λ'_1, p(a_1, ..., a_n), λ'_2) ∈ δ_G do
8          X ← X ∩ (R(λ'_1) ∪ {a_1, ..., a_n});
9      if X ≠ R(λ'_2) then
10         R(λ'_2) = X;
11         foreach (λ'_2, p(a_1, ..., a_n), λ'_3) ∈ δ_G do
12             WL ← WL ∪ {λ'_3};
13 return R;
```

(i.e., *dimensionality*) of the mined automata, we follow the principle that *in each automaton replica,*[21] *a given value is bound only to at most one variable*. Thus, only values that are currently not part of a replica's memory will be bound to more variables. As a result, vertices of the LBFG representing each ensemble state λ' stand for subsets of $\mathcal{FP}$ and do not encode the order those formal parameters have been observed. Thus, each abstract ensemble state stands for a function whose domain is the set $2^{\mathcal{FP}}$.

Analogous to Myhill-Nerode theorem for regular languages [95], each ensemble state λ' stands for an equivalence class of finite words.[22] In order to recover SR-NFA states, we consider each unknown SR-NFA state to stand for a language to be constructed iteratively. *Each SR-NFA state stands for a set of subsequences of words in the languages associated with ensemble states it cuts through.*

Each ensemble state $\lambda' \in \Lambda(\mathcal{FP})$ stands for a function $f_{\lambda'} : 2^{\mathcal{FP}} \to 2^{\mathbf{P}^*}$, where $\mathbf{P}$ is the set of predicates in the underlying FOLe signature SIG. That is, every LBFG vertex v inside an

[21] Which is a vertex in the LBFG representation of each abstract ensemble state $\lambda' \in \Lambda(\mathcal{FP})$.

[22] Namely, the set of all words obtained by following all possible paths from the initial state λ'_0 to λ'.

ensemble state $\lambda' \in \Lambda(\mathcal{FP})$ is labeled with a language $f_{\lambda'}(v) \subseteq \mathbf{P}^*$. Note that automaton states do not represent languages over $\Sigma(\mathcal{FP})$, since each SR-NFA state is restricted to having at most one outgoing transition labeled by each predicate $p \in \mathbf{P}$. For example in Figure 5.6, both **Lock(a)** and **Lock(b)** trigger the same transition out of SR-NFA states q_0. Moreover, every transition edge $e = (\lambda'_1, p(a_1, \ldots, a_n), \lambda'_2) \in \delta_G$ is represented by a binary-valued function $f_e : 2^{\mathcal{FP}} \to 2^{\mathbf{P}^*}$ given by:

$$f_e(x) = \begin{cases} \{p\} & \text{if } x = \{a_1, \ldots, a_n\} \\ \{\varepsilon\} & \text{otherwise} \end{cases}$$

With abuse of notation, p stands both for a predicate and for a single-letter word over $\mathbf{P}$. The symbol ε stands for the empty string. Thus, the language at the output of a transition edge $e = (\lambda'_1, \sigma, \lambda'_2) \in \delta_G$ is given by the product (or concatenation) composition $f_{\lambda'_1} \otimes f_e$. Moreover, the language of an ensemble state $\lambda' \in \Lambda(\mathcal{FP})$ is given in terms of the languages of its predecessors as follows:

$$f_{\lambda'} = \left[(f_{\lambda'_1} \otimes f_{e_1}) \uplus \ldots \uplus (f_{\lambda'_n} \otimes f_{e_n}) \right] \cap \mathsf{AVAIL}(\lambda') \cap \mathsf{LIVE}(\lambda')$$

Note that given $\mathcal{L} \subseteq \mathbf{P}^*$, we have $\mathcal{L} \otimes \{\varepsilon\} = \mathcal{L}$. Hence, if two nodes in the LBFG representation of $f_{\lambda'_1}$ and $f_{\lambda'_2}$ are labeled with the same language $\mathcal{L} \subseteq \mathbf{P}^*$, then they stand for the same SR-NFA state. *To simplify the test of language equality, all languages that are labeling nodes at all ensemble states are represented as nodes in one graph having one unique initial state with each language given by one terminal state in that graph, where concatenation appends a new node (if necessary) to the terminal node of the source language and ε only extends the same terminal state to the destination.*

Variable Assignment

Once SR-NFA states are determined, transitions among these states are also easily determined from the ensemble state PFSA. It remains to assign variable labels to the arguments

of FOLe events that label these transitions. The baseline rule is that *all transitions emanating from the same SR-NFA (but possibly in different ensemble states) labeled with the same event predicate (possibly with different formal parameters) bind the same variables.* So the procedure goes as follows:

- Start a depth-first search (DFS) from the initial SR-NFA state q_0 and apply the baseline rule to all outgoing transitions at each state q.
- Assign new variable labels to arguments that do not yet have assigned variable names.
- Simultaneously propagate these assignments down to all successor LBFG vertices in all ensemble states where SR-NFA state q cuts through.

5.7 MSA Algorithms

MSA is performed within each phase or superstate of the observation traces.

5.7.1 Pairwise Sequence Alignment

In the Needleman-Wunsch (NW) global pairwise alignment algorithm, all possible pairs of *abstract letters* (or parametric events with concrete argument values substituted with variables) are represented in a two-dimensional matrix. The set of all possible alignments is in one-to-one correspondence with the set of all trajectories (consisting of horizontal, vertical and diagonal elementary steps) or paths starting at the top-left corner of that matrix and ending at the bottom-right corner of the same matrix. An optimal alignment of two sequence prefixes is recursively constructed in terms of optimal alignments of smaller prefixes of the same sequences. This recursive step is shown in Figure 5.7. Borrowing notation from [59, Chapter 2], $F(i, j)$ is the score of an optimal alignment of prefixes $\mathbf{x}[1 \ldots i]$ and $\mathbf{y}[1 \ldots j]$ of sequences $\mathbf{x}$ and $\mathbf{y}$, respectively. At the end of NW recursion (i.e., after calculating $F(m, n)$),

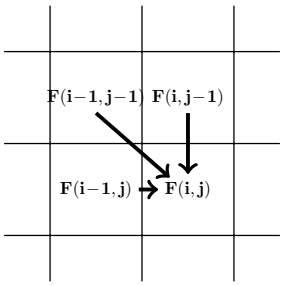


Figure 5.7: Needleman-Wunsch recursion step.

traceback reconstructs an optimal alignment starting at location (m, n) in the matrix and winding back to $(0, 0)$.

The NW global pairwise alignment algorithm can be easily generalized to MSA by constructing an n-dimensional distance matrix. The decision version of the MSA problem[23] that computes a globally optimum alignment of n sequences was shown to be NP-complete [171]. A widely used heuristic that does not guarantee a globally optimal alignment is the use of progressive MSA [93, 70] based on a binary *guide tree* that directs a hierarchical pairwise sequence alignment process [59, Chapter 7] until a final MSA is obtained. The progressive alignment heuristic aligns the most similar (and, hence, most reliably aligned) sequence pairs first. Leaves of the guide tree are the sequences and internal nodes are alignments of two or more sequences. The guide tree is traversed bottom up until all sequences are included in one MSA at the root. As we climb up the guide tree toward the root, the aligned sequence groups begin to look less similar. A *profile* is a MSA where each column is regarded as an alignable symbol [61].

5.7.2 Distance Matrix.

Given n sequences, a symmetric $n \times n$ distance matrix D is calculated by first scoring the $n(n-1)/2$ pairwise alignments constructed with NW algorithm and converting these scores

[23]Where the MSA score is computed as the sum-of-all-pairs of alphabet characters at all positions along the alignment (SP score).

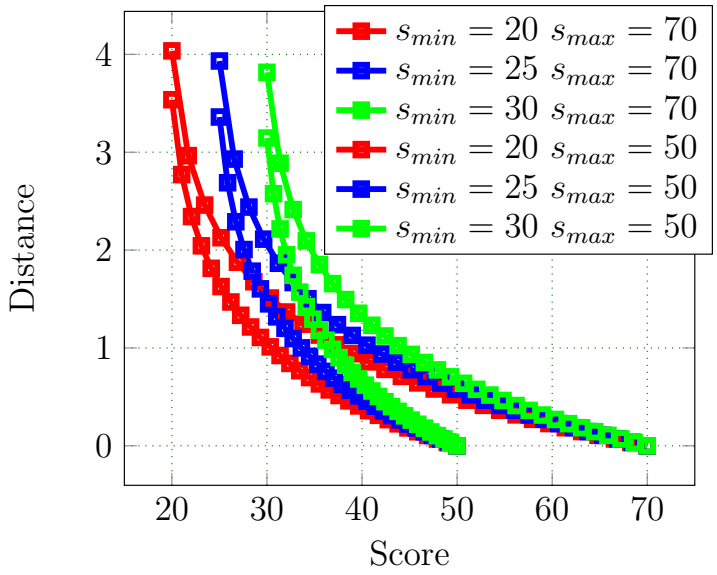


Figure 5.8: The score-to-distance function.

into distance-like metrics using the equation:

$$d_{ij} = -\log \frac{s_{ij} - s_{min} + \epsilon}{s_{max} - s_{min} + \epsilon}$$

where s_{max} and s_{min} are, respectively, the maximum and minimum scores in the score matrix. This conversion is different from the one in [70] and in [59, Chapter 7]. This transformation separates sequence pairs that score near s_{min} much more than sequence pairs that score near s_{max} during pairwise alignment. For any desired $d_{max} > 0$ and assuming $\epsilon \ll (s_{max} - s_{min})$, we have $\epsilon = (s_{max} - s_{min})e^{-d_{max}}$.

5.7.3 Guide Trees

Once we have a distance matrix d_{ij} summarizing the similarities between all pairs of the individual sequences to be aligned, a guide tree is built to basically represent a hierarchical clustering of these sequences. We can use either the UPGMA [159] method or neighbor joining [152] to construct a guide tree.

Guide Tree Pruning. As we will see in Section 5.7.9, given K trace slices, it is highly

desirable to keep the guide tree balanced. That is, keep guide tree depth a constant multiple of $\log_2 K$. Very deep subtrees of a guide tree indicate slices that are too similar and can be dropped from the MSA procedure, since they do not add much new information.

Guide Tree Subdivision. Even if the guide tree is balanced, the tree depth can still be so large as to cause too much growth in MSA lengths as the guide tree is traversed bottom up. Moreover, two subtrees of a guide tree can be very dissimilar which results in adding even more gaps when those subtrees are aligned, which accelerates hitting the complexity wall. Therefore, a large guide tree might be better subdivided into multiple more homogeneous subtrees, each resulting in one mined automaton, rather than merging them in one mega automaton.

5.7.4 Scalability - Trace Clustering

The most time consuming phase of specification mining is the construction of the distance matrix used in profile-based MSA. The distance matrix is crucial for the construction of a phylogenetic tree that captures pairwise similarity of trace slices and directs MSA in a bottom-up fashion. An efficient trace clustering technique is first applied to partition the entire training set of trace slices into more homogeneous disjoint subsets within which some similarity measure is maximized. Then distance matrix and profile-based MSA are then used within these groups and among them. As shown in Figure 5.9, an integer threshold $N > 0$ is used so that if the number of trace slices is greater than N, agglomerative hierarchical clustering is employed to construct a *dendrogram* that partitions the entire training set of trace slices into more homogeneous disjoint subsets of N traces each, within which some similarity measure is maximized. Each trace slice subset is aligned using a distance matrix (calculated with pairwise NW algorithm) and profile-based MSA. The process is repeated until we end up with only one MSA. In agglomerative clustering, every trace slice (with all

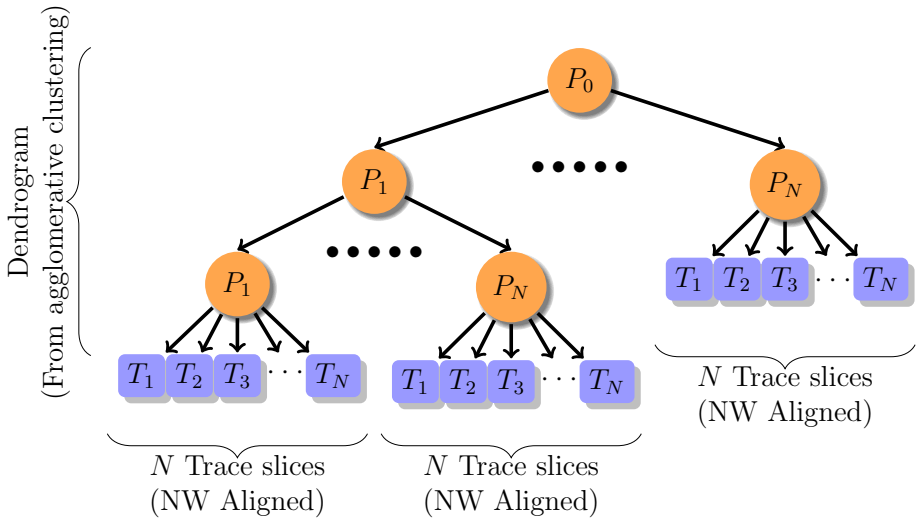


Figure 5.9: Using agglomerative clustering to partition large sets of trace slices into similar groups that are aligned with NW algorithm within each group and the resulting profiles are then aligned. N is a user-specified parameter.

call argument information stripped out) is characterized by a *vector profile* that captures *first-order statistics* (i.e., relative frequencies of all system calls) as well as *second-order statistics* (i.e., transition probabilities among system calls).

5.7.5 Profile Alignment

Given a guide tree T, a *profile* P_t is associated with every node t of T. A *profile* [59, Chapter 6] is a MSA where each column is treated as a single indivisible symbol or letter [61]. If t is a leaf, then P_t is a single sequence. If t is an internal node with child nodes r and s, then P_t is given by the pairwise alignment $P_r \parallel P_s$. When two profiles $\mathbf{x}$ and $\mathbf{y}$ are aligned, the NW pairwise alignment algorithm can still be used. At each cell F_{ij} of the dynamic programming matrix F, we have three cases:

- The i^{th} column $\mathbf{x}_i$ is aligned with the j^{th} column $\mathbf{y}_j$.
- The i^{th} column $\mathbf{x}_i$ is aligned with a column of gaps inserted at the j^{th} position of $\mathbf{y}$.
- The j^{th} column $\mathbf{y}_j$ is aligned with a column of gaps inserted at the i^{th} position of $\mathbf{x}$.

In all cases, columns of $\mathbf{x}$ and $\mathbf{y}$ are used as indivisible units and cannot be altered.

5.7.6 The Scoring Scheme.

The scoring scheme is key to distinguish true alignments from spurious alignments [59, Chapter 2]. As in [59, Chapter 7], a multiple sequence alignment $\mathbf{m} = \mathbf{x} \parallel \mathbf{y}$ is scored with a scoring function $S(\mathbf{m})$ given by:

$$S(\mathbf{m}) = \sum_k S(\mathbf{m}_k)$$

where $\mathbf{m}_k$ is the k^{th} column of $\mathbf{m}$. This function assumes columns are statistically independent. The column scoring function $S(\mathbf{m}_k)$ is the sum-of-pairs (SP) function:

$$S(\mathbf{m}_k) = \sum_{p<q} s(\mathbf{m}_k^p, \mathbf{m}_k^q) \tag{5.2}$$

where $\mathbf{m}_k^p$ is the event in the k^{th} column and p^{th} trace slice. The scores $s(a, b)$ are given by:

$$s(-, -) = 0 \tag{5.3}$$

$$s(a, -) = s(-, b) = \text{GOP or GEP} \tag{5.4}$$

$$s(a, b) = -\infty \text{ iff } a \neq b \tag{5.5}$$

$$s(a, a) = s_a \tag{5.6}$$

That is, a MSA column score is the sum of all pairwise-alignment scores in that column. It is important to note that it is prohibited to align different events (i.e., events induced by different FOLe predicates, or by the same FOLe predicate with different variables as arguments) in the same column, since $s(a, b) = -\infty$ iff $a \neq b$. The only degree of freedom left by this is *placement of gaps*.[24] As customary, an affine gap penalty model has been used based on a *gap opening penalty* (GOP) (an initial penalty for opening a gap) and a *gap extension penalty* (GEP) (incurred with each gap extension, so that the gap penalty

[24]This scheme is not susceptible to the problem observed in [59, Chapter 7] that the relative score difference between a correct alignment and an incorrect alignment diminishes as more sequences are aligned.

increases linearly with its length), where $GOP > GEP$.

In the resultant alignment $\mathbf{m} = \mathbf{x} \parallel \mathbf{y}$, the column scoring function $S(\mathbf{m}_k)$ in Equation 5.2 can be rewritten as [59, Chapter 6]:

$$S(\mathbf{m}_k) = \sum_{p<q\leq M+N} s(\mathbf{m}_k^p, \mathbf{m}_k^q) \tag{5.7}$$

$$= S(\mathbf{x}_i) + S(\mathbf{y}_j) + \sum_{p\leq M, q\leq N} s(\mathbf{x}_i^p, \mathbf{y}_j^q) \tag{5.8}$$

It is only the last *cross-sum* that needs to be optimized by NW algorithm, since the other two sums are independent of the ongoing alignment. This is called the Profile Sum-of-Pairs (PSP) function and is used by ClustalW [167] and MUSCLE [61]. In MSAs studied here, it is prohibited to align different symbols. Thus, except for gaps, MSA columns will be *monochromatic*, as shown in Figure 6.5. This makes profiles to take very simple and efficient representation. Specifically, at each position i along a profile, we need to maintain only three data items [61]:

- The alphabet symbol α_i observed at that position in some of the sequences in the given profile.
- The frequency FO_i of gap-open symbols.
- The frequency FE_i of gap-extend symbols.

Given these frequencies, the frequency FM_i of $\mathbf{x}_i$, or its *occupancy*, is given by $D^{\mathbf{x}} - FO_i - FE_i$. The quantity $FR_i = D^{\mathbf{x}} - FE_i = FM_i + FO_i$ is important and will be used later. This makes profile size completely independent of $D^{\mathbf{x}}$, the *depth* or number of sequences in the profile $\mathbf{x}$, as well as the alphabet size.[25] Also, given two child MSAs $\mathbf{x}$ and $\mathbf{y}$ and the NW traceback path of their profile alignment $\mathbf{z}$, it is possible to compute these frequencies for $\mathbf{z}$ [61] in time $O(L_{\mathbf{z}})$, where $L_{\mathbf{z}}$ is the length of $\mathbf{z}$. Thus, the final profile function is given

[25]Note that the number of sequences will still affect profile size indirectly because the profile length increases as the number of aligned sequences increases.

by:

$$PSP(\mathbf{x}_i, \mathbf{y}_j) = FM_i^{\mathbf{x}}\ FM_j^{\mathbf{y}}\ M \tag{5.9}$$

$$+ (FM_i^{\mathbf{x}}\ FO_j^{\mathbf{y}} + FM_j^{\mathbf{y}}\ FO_i^{\mathbf{x}})\ GOP \tag{5.10}$$

$$+ (FM_i^{\mathbf{x}}\ FE_j^{\mathbf{y}} + FM_j^{\mathbf{y}}\ FE_i^{\mathbf{x}})\ GEP \tag{5.11}$$

Because this is the most critical part of profile alignment, its optimization will have the highest payoff. The value of $PSP(\mathbf{x}_i, \mathbf{y}_j)$ can be rewritten as:

$$PSP(\mathbf{x}_i, \mathbf{y}_j) = FM_i^{\mathbf{x}}\ FM_j^{\mathbf{y}}\ M \tag{5.12}$$

$$+ FM_i^{\mathbf{x}}\ (FO_j^{\mathbf{y}}\ GOP + FE_j^{\mathbf{y}}\ GEP) \tag{5.13}$$

$$+ FM_j^{\mathbf{y}}\ (FO_i^{\mathbf{x}}\ GOP + FE_i^{\mathbf{x}}\ GEP) \tag{5.14}$$

The quantities $(FO_j^{\mathbf{y}}\ GOP + FE_j^{\mathbf{y}}\ GEP)$ and $(FO_i^{\mathbf{x}}\ GOP + FE_i^{\mathbf{x}}\ GEP)$ can be pre-calculated along each of the profiles $\mathbf{x}$ and $\mathbf{y}$ and reused throughout the profile alignment process. Finally, to align the i^{th} location $\mathbf{x}_i$ of $\mathbf{x}$ with a gap following $\mathbf{y}_j$, we need to pre-calculate the following gap penalties:

$$PSP(-_i, \mathbf{y}_j) = FM_j^{\mathbf{y}}\ (FR_i^{\mathbf{x}}\ GOP + FE_i^{\mathbf{x}}\ GEP) \tag{5.15}$$

$$PSP(\mathbf{x}_i, -_j) = FM_i^{\mathbf{x}}\ (FR_j^{\mathbf{y}}\ GOP + FE_j^{\mathbf{y}}\ GEP) \tag{5.16}$$

The best assignment of gap penalties is determined empirically by experimenting with different values for GEP and GOP.

5.7.7 Language-Based Alignment

During MSA, only identical symbols (event predicates and formal parameters) can be aligned. Although different symbols may possess the same hidden ensemble-state label, this can be handled later using PFSA state reduction (where state merging decisions are based on outgoing transitions rather than incoming transitions as in MSA). However, not all identical symbols should be alignable. Stipulating that symbols be identical to be alignable is not

sufficiently restrictive. MSA needs to be constrained in such a way that identical symbols can be aligned only if their outgoing languages are approximately equivalent. A MSA can be represented as a PFSA and, hence, the NW matrix used to align two MSAs needs to take into account languages of symbols associated with each MSA column. Two MSA columns are alignable iff they contain two identical symbols and their PFSA states are sk-equivalent as determined by two parameters $P > 0$ and $K > 1$.

5.7.8 Iterative Refinement

To counteract the deficiency of profile alignment that freezes columns of its inputs, the resultant progressive alignment outcome $\mathbf{x} = \mathbf{x}_1 \parallel \mathbf{x}_2 \parallel \ldots \parallel \mathbf{x}_n$ can be iteratively refined. Let $\mathbf{x} \setminus \mathbf{x}_k$ be the profile obtained from $\mathbf{x}$ by removing $\mathbf{x}_k$ and removing columns that become entirely composed of gap symbols. The procedure goes by repeating the following step for a fixed number of iterations or until the alignment score converges [59, Chapter 6]: For every k in a permutation of the set $\{1, \ldots, n\}$: extract a trace slice $\mathbf{x}_k$ and realign it to $\mathbf{x} \setminus \mathbf{x}_k$. Alternatively, the profile $\mathbf{x}$ can be partitioned into two disjoint profiles $\mathbf{x}_a$ and $\mathbf{x}_b$ and realigned. The result can be kept if a given scoring function is improved [91, 61].[26]

5.7.9 Complexity.

Pairwise profile alignment has time and space complexity $O(n_i n_j)$, where n_i and n_j are the lengths of the two profiles.[27,28] Therefore, it is desirable to keep slice length short enough to keep MSA run-time within reasonable limits. This requirement conflicts with the desire to make slice length as large as possible to ensure that every slice contains complete episodes

[26]The result can be kept with small nonzero probability if the scoring function is reduced to ameliorate the problem of local maxima.

[27]Space complexity can be reduced to $O(n_i + n_j)$ [59, Chapter 2].

[28]Note that profile lengths increase as the guide tree is traversed bottom-up.

of interesting behaviors. Another important contributor to overall complexity is guide tree construction. If the guiding tree is constructed using a distance matrix based on pairwise alignment scores, time and space complexity will also be $O(k^2n^2)$, where k is the number of trace slices and n is the slice length. To reduce the $O(n^2)$ factor, in Section 5.7.4, we use agglomerative clustering of trace slices based on feature vectors comprising the first-order and second-order statistics of each slice. Alternatively, distance can be based on the number of common two-letter substrings.

5.8 State-Space Reduction

The PFSA constructed from a MSA obtained by folding and aligning multiple long traces within multiple regions is acyclic (except for the *back edges* within each super-state) and, hence, it does not possess any recurrent behavior[29] and cannot generalize beyond the training set of traces. When abstract ensemble states are converted to SR-NFA states, many short cycles will form. However, large-scale recurrent behavior can be introduced into a SR-NFA only by merging closely related states that are likely to stand for *similar* hidden system state [19, Chapter 7]. Therefore, we now study state-space reduction methods that can be applied to the PFSA representing abstract ensemble states or the SR-NFA obtained from it.

Conventionally, two states q_1 and q_2 of an automaton $\mathcal{A}$ are considered equivalent, denoted $q_1 \sim q_2$, if their languages $\mathcal{L}(q_1)$ and $\mathcal{L}(q_2)$ are equal. By merging all such language-equivalent states, we obtain the quotient automaton $\mathcal{A}/\sim$, and it is guaranteed that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}/\sim)$. The sk-strings method [145], on the other hand, constructs an *over-approximation* $\mathcal{A}'$ of a SR-NFA $\mathcal{A}$ by associating a two-parameter set of languages $\mathcal{L}_K^P(q)$, parameterized by a depth $K > 0$ and a probability $0 \leq P \leq 1$, with every SR-NFA state q. $\mathcal{L}_K^P(q)$ is the set of most likely length-K words whose total probability is P.[30] $\mathcal{L}_K^P(q)$ is defined in terms

[29] Such PFSA becomes truly acyclic when each strongly connected component is collapsed.

[30] Words are of length up to $K > 0$, since we may have words of length less than K if a terminal state is

of $\mathcal{L}_K(q) = \{w \in \Sigma^* \text{ such that } |w| = K\}$ and $|w|$ is the length of w. The language $\mathcal{L}_K(q)$ is finite and, hence, K-words can be sorted in descending order according to their probabilities. Then $\mathcal{L}_K^P(q) \subseteq \mathcal{L}_K(q)$ is the set of all words in that order with total probability $\geq P$.

A SR-NFA *run* π is an alternating sequence $q_0 \xrightarrow{\sigma_0} q_1 \xrightarrow{\sigma_1} q_2 \ldots \xrightarrow{\sigma_{K-1}} q_K$. Then the probability of a word $\mathbf{w} = (w_1, w_2, \ldots, w_K) \in \mathcal{L}_K^P(q)$ starting from state $q \in Q$ is equal to:

$$p_K(\mathbf{w}) = \sum_{\textcircled{\textcircled{q}}_K} \prod_{i=1}^{K} Pr(q_{i-1}, w_i, q_i),$$

where $q_0 = q$ and $\textcircled{\textcircled{q}}_K$ is the set of paths of length K starting at q. Since a SR-NFA can be nondeterministic, there can be many runs on the same word.

Two states q_1 and q_2 are sk-equivalent up to depth K and with probability P if $\mathcal{L}_K^P(q_1) \subseteq \mathcal{L}_\infty^1(q_2)$ and $\mathcal{L}_K^P(q_2) \subseteq \mathcal{L}_\infty^1(q_1)$. By merging two states q_1 and q_2 that are sk-equivalent, it is guaranteed that $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A}')$.

Precision-Generalization Trade-off. The degree of over-approximation is controlled by P and K which helps to control the trade off between SR-NFA precision and conciseness.

The use of large value for K and P helps sk-equivalence reach more precise (and, hence, more predictive) SR-NFAs so that it develops a higher resolving power of anomalous behavior. However, this comes with poor generalization beyond the training set, since the SR-NFA effectively memorizes the training set. This increased model precision also comes at the expense of a larger model size. The trade-off between model precision and generalization implies a parallel trade-off between resistance to mimicry attacks and the false-positives rate.

encountered.

State Clustering

Since PFSA reduction is an iterative process where each iteration can be quadratic in the size of the given PFSA, it is crucial to improve the average-case complexity in order to handle large PFSAs. This is achieved by clustering (i.e., partitioning the PFSA state-space into sets of similar states) and checking sk-equivalence within each cluster. Each PFSA is described by the vector of its outgoing transition probabilities on each possible atom $w \in \mathsf{Atoms}$. Then we use k-means clustering to partition PFSA states into disjoint groups which are highly likely to be sk-equivalent and merge. The value of k can be chosen to be approximately the expected number of reduced PFSA states or to tune the run-time of the algorithm. Care is taken to guide the k-means algorithm to produce clusters of almost equal sizes to reap the benefits of clustering.

5.8.1 State Recurrence

A strongly connected component (SCC) of a graph is *nontrivial* if it contains two or more vertices. A state $q \in Q$ of a SR-NFA $\mathcal{N}$ is *recurrent* if it resides in a nontrivial SCC of $\mathcal{N}$ or if a nontrivial SCC of $\mathcal{N}$ is reachable from q. Otherwise, q is *nonrecurrent* or *transient*. A nonrecurrent state q can only generate a finite number of traces of finite length. A SR-NFA $\mathcal{N}$ can be simplified before determinization by the following optimization: For every recurrent state s, remove all nonrecurrent successors whose traces can be generated by other (not necessarily recurrent) successors of s.

5.8.2 SR-NFA Determinization

Once a PFSA is reduced, it is turned into a SR-NFA simply by keeping any transition with nonzero probability and eliminating all other zero-probability transitions. Since the

initial states are specified by the interactive user after a SR-NFA has been extracted, it is important to understand the impact of user's choices on performance. The user needs to be frugal about which states are initial, since that tends to increase the number of current states of the SR-NFA during the determinization procedure, and hence blows up the size of the resulting SR-DFA and its transition structure.

The usual power-set construction [95] can be applied to a mined SR-NFA (with initial states carefully selected) to obtain the corresponding SR-DFA. Before determinization, edges of a SR-NFA are annotated with atomic FOLe formulas containing symbolic variables. After determinization, a SR-DFA has its transitions labeled with FOLe formulas.

5.8.3 SR-DFA Completion

The SR-DFA resulting from determinizing a mined SR-NFA is complete. That is, every state has a transition for every possible event. This is accomplished by adding a failure (or accepting) state to the constructed SR-DFA and extending a *failure edge* to it from every other SR-DFA state. A failure edge is annotated with a FOLe formula that is complementary to all other edges of its source state.

5.8.4 SR-DFA Minimization

Parametric DFAs produced by ParaMiner usually contain redundant states that can be reduced by using a DFA minimization procedure. In this thesis, we use Brzozowski's algorithm [34] because it easily generalizes to SR-DFAs. Brzozowski's algorithm has exponential worst-case complexity but typically works very well in practice.

5.8.5 Validation

Once a SR-DFA has been constructed, we can use the system whose behavior is being modeled as a source of counterexamples, by running the system long enough under various operating conditions to reveal any violations of the conjectured SR-DFA specification and refine it.[31] A set of traces deemed representative of all correct system behaviors is divided into two sets: a *training set*, used in the specification mining process, and a *validation set* used to validate the mined SR-DFA. After completely mining a SR-NFA, determinizing it and adding the failure edges, there might still be spurious failures on the validation set of traces. That might be indicative of inadequate training set coverage of program behavioral space.

5.9 Variable Equivalence

A recurring problem, when dealing with parameterized specifications, is the problem of identifying variable names between two parameterized objects (e.g., symbolic variables used by two MSAs, symbolic variables used by a PFSA with the goal of making two PFSA states equivalent, or symbolic variables used by two separate SR-DFAs with the goal of quantifying the distance metric between them).

5.9.1 PFSA Variable Matching

Two states q_1 and q_2 of a PFSA[32] $\mathcal{A}$ can have closely related languages $\mathcal{L}(q_1)$ and $\mathcal{L}(q_2)$ up to an equivalence relation on the set of formal parameters $\mathcal{FP}$ used by the parametric events

[31] Using the terms of [12], only *equivalence queries* are allowed, whereas *membership queries* are not.

[32] We consider equivalencies over $\mathcal{FP}$ associated with a PFSA before converting abstract ensemble states to SR-NFA states. The same arguments apply to SR-NFAs with equivalencies defined over $\mathcal{X}$.

annotating edges of $\mathcal{A}$. That is, there is an equivalence relation $\sim \subseteq \mathcal{FP} \times \mathcal{FP}$ such that q_1 and q_2 become equivalent in the quotient automaton $\mathcal{A}/\sim$. Thus, we can increase the number of sk-equivalent states by identifying formal parameters as follows: We construct a compatibility graph $\mathscr{C} = (\mathscr{V}, \mathscr{E})$, where $\mathscr{V} = Q \times Q$. For every pair of states $(q_1, q_2) \in Q \times Q$, we construct an equality-logic (EL) formula[33] $\mathbb{B}(q_1, q_2)$ that encodes all possible equivalence relations on $\mathcal{FP}$ (i.e., all possible ways formal parameters can be identified) so that q_1 and q_2 become equivalent. In the compatibility graph $\mathscr{C}$, there is an edge from (q_1, q_2) to (q_1', q_2') iff $\mathbb{B}(q_1, q_2) \wedge \mathbb{B}(q_1', q_2')$ is satisfiable.[34] We then find a maximum clique $\{(q_1^1, q_2^1), \ldots, (q_1^n, q_2^n)\}$ in $\mathscr{C}$, which represents a maximally compatible set of identifiable state pairs. Then using a SAT solver [109], we can find a truth assignment to the formula $\mathbb{B}(q_1, q_2) \wedge \ldots \wedge \mathbb{B}(q_1^n, q_2^n)$ which can be easily turned into an equivalence relation $\sim \subseteq \mathcal{FP} \times \mathcal{FP}$ that materializes all the equivalences $\{(q_1^1, q_2^1), \ldots, (q_1^n, q_2^n)\}$.

Atoms of an EL formula $\mathbb{B}(q_1, q_2)$ are of the form $(x_1 = x_2)$ where $x_1, x_2 \in \mathcal{FP}$. Moreover, an EL formula $\mathbb{B}(q_1, q_2)$ is a conjunction of two parts:

$$\mathbb{B}(q_1, q_2) = \mathbb{B}(q_1|q_2) \wedge \mathbb{B}(q_2|q_1)$$

where $\mathbb{B}(q_1|q_2)$ is an EL formula for all equivalence relations on $\mathcal{FP}$ such that $\mathcal{L}_K^P(q_1) \subseteq \mathcal{L}_\infty^1(q_2)$ and similarly $\mathbb{B}(q_2|q_1)$ is an EL formula for all equivalence relations on $\mathcal{FP}$ such that $\mathcal{L}_K^P(q_2) \subseteq \mathcal{L}_\infty^1(q_1)$. Also, $\mathbb{B}(q_1|q_2)$ is defined to be:

$$\mathbb{B}(q_1|q_2) = \wedge_{\tau_1 \in \mathcal{L}_K^P(q_1)} \mathbb{B}(\tau_1|q_2)$$

In general, $\mathbb{B}(\tau|q)$ is an EL formula expressing the set of all equivalence relations on $\mathcal{FP}$ such that $\tau \in \mathcal{L}_\infty^1(q)$ where $\tau \in \Sigma(\mathcal{FP})^*$. The set of all runs of a PFSA $\mathcal{A}$ on a trace $\tau \in \Sigma(\mathcal{FP})^*$ starting at state $q \in Q$ is a rooted tree with vertices $Q^\tau \subseteq Q \times \mathbb{N}$. With every vertex $u \in Q^\tau$, we associate an EL formula $f : Q^\tau \to EL$ defined recursively as:

$$f(q, 0) = \mathsf{true}$$

[33] See [104, Chapter 3]. Equality logic is equivalent to Boolean logic.
[34] We use a SAT solver [109] to test for satisfiability.

Given two atoms $p(x_1, \ldots, x_n)$ and $p(y_1, \ldots, y_n)$, the corresponding EL formula is given by:

$$(x_{i_1} = y_{j_1}) \wedge \ldots \wedge (x_{i_k} = y_{j_k}),\ x_{i_1}, y_{j_1}, \ldots \in \mathcal{FP}$$

Unless the equivalence depth parameter K is large enough, the number of state-pairs that can be equivalent will be too large, in the sense that the graph of these state-pairs that will go through the clique finding algorithm will be too large. Moreover, making K too large hits another complexity wall, namely the size of state languages which grows exponentially with K in the worst case. If the size of a PFSA is n, the number of state pairs is $O(n^2)$ and, hence, the number of edges in the compatibility graph (that captures the compatibility of state pairs) is $O(n^4)$, which is too large except for small n. A much faster, but less precise, variable identification heuristic iteratively examines all state pairs one by one in sequence. For each pair $(q_1, q_2) \in Q \times Q$, it constructs $\mathbb{B}(q_1, q_2)$ and uses a SAT solver [109] to find an equivalence relation $\sim \subseteq \mathcal{FP} \times \mathcal{FP}$ and applies it immediately so that (q_1, q_2) become equivalent and can be merged. This step is repeated until no more state pairs can be merged. Note that equivalencies among states now depend upon the order in which (probabilistically) equivalent states are merged.

5.9.2 MSA Variable Matching

One complication not present in conventional MSA problems is that each parametric event in a trace slice has concrete argument values and they need to be *abstracted* (i.e., each argument value is substituted with a symbolic formal-parameter name) before use in MSA. This way, we effectively determine which parametric events, possibly having different argument values, correspond to equivalent ensemble states. Abstraction of argument values for multiple sequences needs to be chosen in such a way that similarity among the trace slices or sequences is maximized so as to minimize the SR-NFA ultimately inferred from the resulting MSA *without blurring or obscuring the data usage patterns latent in those sequences.*

Starting at the leaves of the guide tree before profile-based MSA starts, each unique or distinct argument value in a trace is either replaced with a *constant symbol* (from the *execution context*) or with a *unique formal-parameter symbol* before use in MSA. For each program, a set of user-defined constant symbols may be used to reserve special status for particular data values (e.g., to distinguish the ID of the main process of **proftpd**). The same constant may have different values (i.e., interpretations) in each execution trace of the same program (e.g., the leader process ID may be different in each run of the program). The set of values ascribed to constant symbols by a given trace form an *execution context* for that trace.

At every pairwise alignment step of the profile-based MSA guide tree, we first perform variable matching between the two incoming profiles (where concrete argument values are abstracted and identified) and then perform conventional NW alignment, since each FOL atom can be encoded by a distinct integer. Given two sequences $\mathbf{x} = (x_1, x_2, \ldots, x_m)$ and $\mathbf{y} = (y_1, y_2, \ldots, y_n)$, a *weighted bipartite graph* $G = ((U, V), E)$, where $U \cap V = \emptyset$ and $E \subseteq U \times V$, is constructed as follows:

- For each pair (x_i, y_j), if $x_i = p(u_1, u_2, \ldots)$ and $y_j = p(v_1, v_2, \ldots)$ for some flexible predicate $p \in \mathbf{P}$, we add:

$$U \longleftarrow U \cup \{u_1, u_2, \ldots\}$$
$$V \longleftarrow V \cup \{v_1, v_2, \ldots\}$$
$$E \longleftarrow E \cup \{(u_1, v_1), (u_2, v_2), \ldots\}$$

- For each new edge (u_k, v_k), we assign a weight $w(u_k, v_k)$ equal to the score s_{ij} of aligning x_i with y_j, as detailed in Section 5.7.6:

$$w(u_k, v_k) = w(u_k, v_k) + s_{ij}$$

The graph G can be completed by adding edges of zero weight.[35] Then the *Hungarian*

[35] A bipartite graph $G = ((U, V), E)$ is complete if $E = U \times V$.

algorithm [35] is used to find a maximal set of variable pairs (u, v) that can be consistently identified between $\mathbf{x}$ and $\mathbf{y}$ to maximize similarity between the sequences.[36] A *matching* in G is a subset $M \subseteq E$ such that $\forall v \in U \cup V$, there is at most one edge in M incident on v (i.e., $\exists u \in U \cup V$ such that either $(u, v) \in M$ or $(v, u) \in M$). In a weighted bipartite graph G, the weight of a matching M is the given by $w(M) = \sum_{(u,\ v) \in M} w(u,\ v)$. Given a matching $M \subseteq E$, every variable in one sequence can be identified with at most one variable in the other sequence, thus missing many other identification opportunities. The weighted bipartite graph constructed above contains more information about variable relations than *pairwise matchings* can capture. Therefore, we instead use a higher-order generalization of pairwise matchings, called *bicliques* [56]. A *biclique* of a bipartite graph $G = ((U, V), E)$ is a complete bipartite subgraph $((A, B), A \times B)$ of G such that $A \subseteq U$, $B \subseteq V$ and $A \times B \subseteq E$. Basically, we want to construct a *maximu-weight biclique partition* of G which in a sense generalizes what a maximum-weight matching is. A biclique has weight equal to the sum of weights of all edges in that biclique. So first, we list all maximal bicliques in G. There are many efficient algorithms to list all maximal bicliques in a bipartite graph. We use the one in [55]. Some of these bicliques overlap. Therefore, to construct a biclique partition (i.e., a set of disjoint bicliques), we construct another graph $\mathscr{G} = (\mathscr{V}, \mathscr{E})$ whose vertices $\mathscr{V}$ are the maximal bicliques of G. Two vertices in $\mathscr{G}$ are connected by an edge if their corresponding bicliques in G are disjoint. Then every maximum-weight biclique partition of G corresponds to a maximum-weight clique in $\mathscr{G}$, which can be found by listing all cliques of $\mathscr{G}$ and selecting the one with maximum weight. However, the clique decision problem is NP-complete [100]. It is not even efficiently approximable [14]. However, there are many heuristic algorithms that find "good" solutions [29]. We use tabu search as described in [78] with parameterized tabu list length. Since the tabu search metaheuristic may not be able to visit all vertices of graph $\mathscr{G}$ before it stops with a (supposedly good) solution, that graph is only constructed *on the fly* as tabu search demands or queries about edges from $\mathscr{E}$.

[36] The Hungarian algorithm solves the assignment problem, or the matching problem in weighted bipartite graphs.

5.9.3 MSA Variable Matching: An Alternative

The variable identification algorithm in Section 5.9.2 needs to discover and preserve interesting data usage patterns. For example, the following trace segment is taken from the `Tomcat` Java benchmark:

1 : java.io.PushbackInputStream.read(20, 56, 0, 30) = 30

1 : java.io.PushbackInputStream.read(20, 10, 0, 04) = 04

1 : java.io.PushbackInputStream.read(20, 56, 0, 30) = 30

1 : java.io.PushbackInputStream.read(20, 10, 0, 05) = 05

1 : java.io.PushbackInputStream.read(20, 56, 0, 30) = 30

1 : java.io.PushbackInputStream.read(20, 10, 0, 04) = 04

1 : java.io.PushbackInputStream.read(20, 56, 0, 30) = 30

In this trace slice, thread 1 of `Tomcat` reads from the same `PushbackInputStream` object (with identifier 20) alternately into two different `byte` arrays (with identifiers 10 and 56). If we try to align two MSA rows identical to the above trace, it is beneficial to assign the two `byte` arrays different variable symbols in order to preserve this alternation. It is noteworthy that even without this variable identification, both traces would perfectly align. This suggests that variable identification opportunities should be evaluated based on the gain they achieve in terms of alignment quality (measured by alignment score). A trade-off between model precision and conciseness is controlled by a threshold on that gain.

Within the k^{th} foldable trace segment, MSA constructs a score matrix $\mathbf{s}^k = [s^k_{ij}]$ where s^k_{ij} is the score of aligning the i^{th} row and the j^{th} row in the k^{th} segment. The average pairwise (inter-sequence) score $\widehat{\mathbf{s}}$ is defined by:[37]

$$\widehat{\mathbf{s}} = \frac{1}{\sum_k N_k^2} \sum_k \sum_i \sum_j s^k_{ij}$$

The gain achieved by identifying variables is given by $\widehat{\mathbf{s}}_+ - \widehat{\mathbf{s}}_-$, where $\widehat{\mathbf{s}}_+$ (and $\widehat{\mathbf{s}}_-$, resp.) is the average pairwise score after (before, resp.) variable identification.

Let $EL(\mathcal{FP})$ be equality logic over the set $\mathcal{FP}$ of MSA formal parameters. $EL(\mathcal{FP})$ consists

[37] The diameter $S^k = \max_k s^k_{ij}$ may not reflect the increase in alignment quality as variables are identified.

of Boolean combinations of atoms of the form $(x = y)$ where $x, y \in \mathcal{FP}$. Note that every $EL(\mathcal{FP})$ cube gives rise to a (reflexive, transitive and symmetric) relation $\mathcal{R} \subseteq \mathcal{FP} \times \mathcal{FP}$ over the set of formal parameters as follows: Each atom $(x = y)$ can be visualized as an edge in a graph with vertex set given by $\mathcal{FP}$ and, hence, every cube is logically equivalent to the transitive closure of all of its edges. The set $\mathscr{C}$ of all such cubes is, thus, a lattice. Identifying variables can only improve the score of aligning any pair of traces, since NW algorithm used in pairwise alignment computes an optimal alignment. Therefore, the average MSA score is a *monotonic* real-valued function $\widehat{\mathbf{s}} : \mathscr{C} \to \mathbb{R}$ over the lattice $\mathscr{C}$ and can be represented as a LBFG constructed symbolically from more elementary LBFGs as follows:

Every entry s_{ij} of the NW matrix used in a pairwise alignment is itself a function $s_{ij} : \mathscr{C} \to \mathbb{R}$ obtained from adjacent entries by two symbolic operations: maximum ($\vee$) and addition ($+$):

$$s_{ij} = h_{ij} \vee d_{ij} \vee v_{ij}$$

where h_{ij}, d_{ij} and v_{ij} are, respectively, the horizontal, diagonal and vertical score functions given by:

$$\begin{aligned} h_{ij} &= s_{i,j-1} &&+HGP_{ij} \\ d_{ij} &= s_{i-1,j-1} &&+M_{ij} \\ v_{ij} &= s_{i-1,j} &&+VGP_{ij} \end{aligned}$$

Also, HGP_{ij} and VGP_{ij} are the horizontal and vertical gap penalties, respectively. The horizontal gap penalty $HGP_{ij} = GEP$ if $s_{i,j-1}$ also implies a horizontal gap, and $HGP_{ij} = GOP$ otherwise. Similarly, the vertical gap penalty $VGP_{ij} = GEP$ if $s_{i-1,j}$ also implies a horizontal gap, and $VGP_{ij} = GOP$ otherwise. Thus, HGP_{ij} and VGP_{ij} are given by:

$$HGP_{ij} = r_{ij}GEP + (1 - r_{ij})GOP$$

$$VGP_{ij} = c_{ij}GEP + (1 - c_{ij})GOP$$

where $r_{ij} : \mathscr{C} \to \mathbb{R}$ is an indicator function that indicates whether the symbol at position $(i, j-1)$ is a gap, and $c_{ij} : \mathscr{C} \to \mathbb{R}$ is an indicator function that indicates whether the symbol

at position $(i-1, j)$ is a gap. Thus, we have:

$$r_{ij} = \mathsf{Clip}(1 + (h_{i,j-1} - s_{i,j-1}))$$

$$c_{ij} = \mathsf{Clip}(1 + (v_{i-1,j} - s_{i-1,j}))$$

The unary function Clip clips the negative part of its input function. Note that the expression $\mathsf{Clip}(1 + (h_{i,j-1} - s_{i,j-1}))$ will be 1 when $h_{i,j-1} = s_{i,j-1}$, since we always have $h_{i,j-1} \leq s_{i,j-1}$, and similarly for the expression $\mathsf{Clip}(1 + (v_{i-1,j} - s_{i-1,j}))$. Note that the graph for HGP_{ij} is the same as the graph for r_{ij} where all nodes labeled with 1 are relabeled with GEP and all nodes labeled with 0 are relabeled with GOP, and similarly for VGP_{ij}.

For each pair (x_i, y_j), if $x_i = p(u_1, \ldots, u_n)$ and $y_j = p(v_1, \ldots, v_n)$, the function $M_{ij} : \mathscr{C} \to \mathbb{R}$, for every $c \in \mathscr{C}$, is given by:

$$M_{ij}(c) = \begin{cases} -\infty \text{ if } x_i \text{ and } y_j \text{ are not alignable} \\ M \text{ if } c = (u_1 = v_1) \wedge \ldots \wedge (u_n = v_n) \end{cases}$$

By considering $\widehat{\mathbf{s}}$ as a function $\widehat{\mathbf{s}} : \mathscr{C} \to \mathbb{R}$, the optimum MSA depends on which different argument values in the input parametric traces are identified. Different groupings of argument values may result in structurally different SR-NFAs. We can now study the effect of various groupings of argument values on the optimum MSA, quantified in terms of the average pairwise alignment score $\widehat{\mathbf{s}}$ as a function of these groupings of argument values. $\widehat{\mathbf{s}}$ is an indicator of the expected quality of the resulting MSA and assists in determining which groupings of argument values result in appreciable gains in overall MSA quality and which groupings cause negligible gains at the expense of considerable loss in precision. This is useful in *sensitivity analysis* needed to assess the robustness of an optimum alignment against perturbations in argument value groupings.

5.9.4 Abstraction

To allow users to trade off precision of a mined SR-DFA $\mathcal{A} = (Q, \mathsf{SIG}, \delta, Q_0, F)$ for more conciseness, equivalence relations over Q with varying granularities are needed. Given an equivalence relation $\sim \subseteq Q \times Q$, the state space Q can be reduced by taking the quotient $\mathcal{A}/\sim$. A first step toward an equivalence relation is a *simulation relation* [19]. A simulation preorder $\mathcal{R} \subseteq Q \times Q$ is a relation such that for all $(p, q) \in \mathcal{R}$ and $\sigma \in \Sigma$, if $p \xrightarrow{\sigma} p'$, then there is $q' \in Q$ such that $q \xrightarrow{\sigma} q'$ and $(p', q') \in \mathcal{R}$. If a simulation relation $\mathcal{R}$ exists over $\mathcal{A}$ and $(p, q) \in \mathcal{R}$, it is said that state q *simulates* state p, denoted by $p \preceq q$. ParaMiner uses algorithms that compute simulation preorders in time $\mathcal{O}(|\delta|.|Q|)$ [19]. A simulation relation $\preceq$ over Q can give rise to equivalence relations, such as the symmetric closure $\sim = \preceq \cup \preceq^{-1}$ and the symmetric kernel $\cong = \preceq \cap \preceq^{-1}$. Between these two extremes, it is left to the users of ParaMiner to identify genuine state equivalences that preserve classes of properties important to them.

5.10 Experimental Validation

5.10.1 Benchmarking

To demonstrate the ability to extract specifications for different types of APIs and applications, we used both Java as well as Linux benchmarks, listed in Table 5.10. These include the DaCapo-9.12-bach benchmark suite of Java apps [27], `proftpd` and `dropbox` running on Linux. Because Java intrinsically supports reflection and byte-code instrumentation, we created an instrumentation agent that, with the help of Javassist-3.19 [40], instruments all (`public`, non-`static`) method calls on standard Java SE and EE packages made by DaCapo programs. Instrumented packages include:

- Java SE:
  - `java.util`
  - `java.net`
  - `java.io`
  - `java.nio`
  - `java.rmi`
  - `java.security`
  - `java.util.concurrent`
  - `java.sql`
- Java EE:
  - `javax.mail`
  - `javax.servlet.*`
  - `javax.transaction`

For every Java method call, the dumped traces include calling thread, method full name, argument values (e.g., object references) and return values. In addition to this, we also

	Benchmark	File size (MB)
	Avrora	659
	Batik	12
	Eclipse	112
	Fop	58
	H2	1292
	Jython	4200
	LuIndex	21
	LuSearch	630
	Pmd	310
	Sunflow	0.5
	Tomcat	3
	TradeBeans	330
	TradeSoap	330
	Xalan	35
	Proftpd	134
Drop box	Indexing	36
	Uploading	301
	Up-to-date	16

Figure 5.10: Benchmark traces.

used ParaMiner to mine properties of parametric system-call traces dumped by the Linux `strace` utility. The `proftpd` is instrumented with `strace` while generating random 10000 ftp requests. An example SR-DFA extracted from `proftpd` traces is shown in Figure 5.11. Experimentation is currently still ongoing will all benchmarks, including 3000 Linux malware samples from VirusTotal and VirusShare.

5.10.2 Quality of Results (QoR) Metrics

Mined specifications are evaluated according to their utility/effectiveness in serving their intended application [149, 157]. For example, unlike specifications intended for anomaly detection, specifications intended for program understanding must be understandable by humans. In the former application, specification precision almost always entails large complex automata, whereas in the latter application understandability may come at the expense

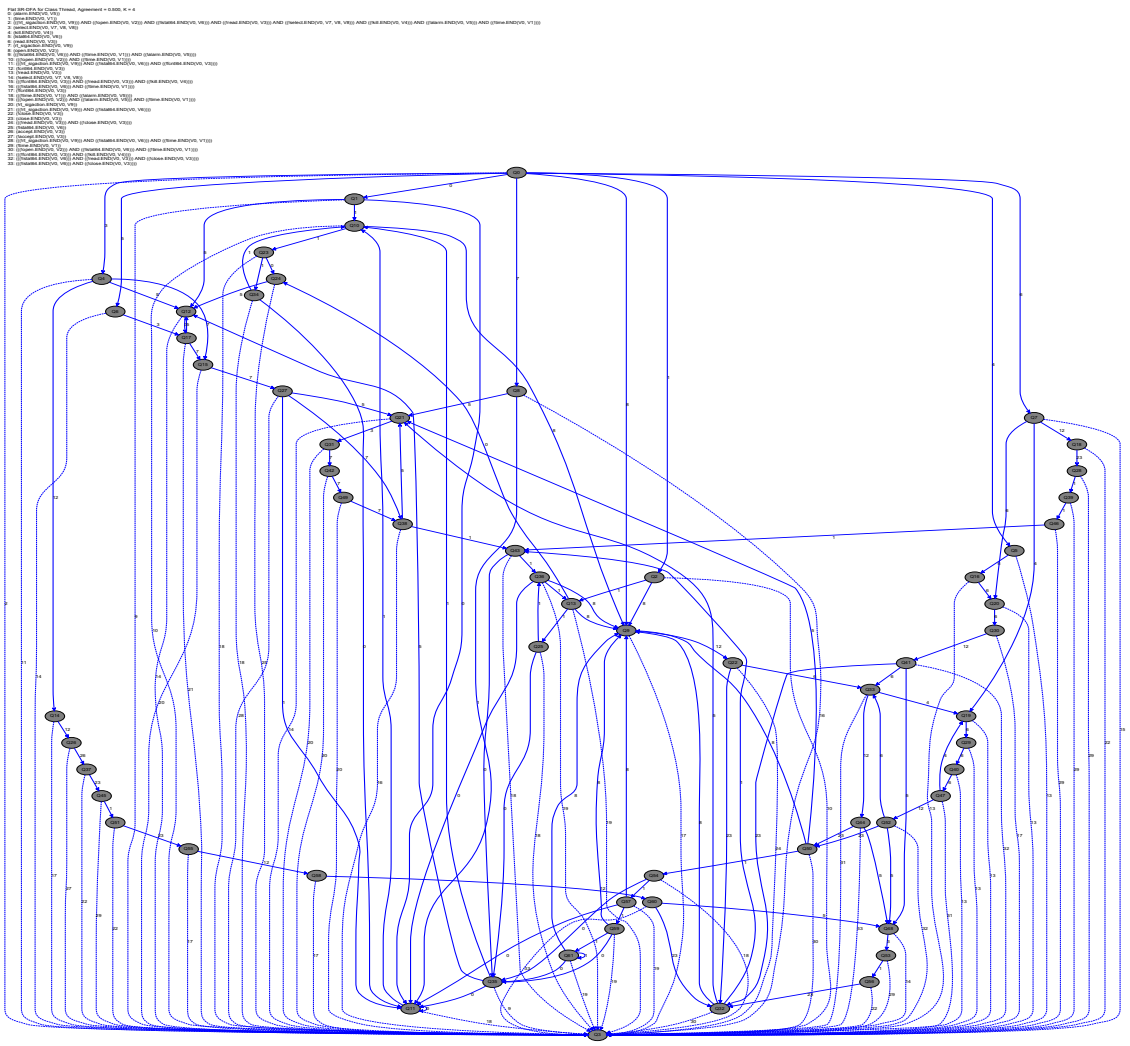


Figure 5.11: Example of a SR-DFA extracted by ParaMiner from `proftpd` traces.

of less precision.

In specification mining tools with automata as the target formalism, quality of mined specifications is usually evaluated in terms of *conciseness or size metrics* (e.g., the total number of states and edges), *precision metrics* (e.g., the average branching factor, also called the density, which is the average outgoing degree of all states) and *accuracy metrics* (e.g., false-positives rate or statistical significance). Moreover, it is important to relate the size metrics to the size of input traces using a *compression ratio*.

The branching factor (out-degree) quantifies how much predictive power each state of the mined automaton has. If the average branching factor is too high, states do not uniquely predict what events should come next (the states are too permissive).

Note that the single-state automaton with an all-encompassing self-loop transition is the most imprecise model and it has the lowest average branching factor and best compression ratio. Therefore, MSA statistical significance needs to be incorporated in the overall evaluation.

5.10.3 Statistical Significance

One of the main challenges that specification mining tools are facing is false positives [80]; that is inferring *spurious hypotheses* (i.e., patterns that do not codify genuine properties of the system). Therefore, a core requirement is to ensure that a MSA carries statistical significance (i.e., *probably* captures statistical regularities actually present and shared among the observed trace slices) rather than being a product of chance or statistical noise. Equivalently, a MSA must be qualified with a measure of how likely the aligned sequences are completely unrelated (*p-value*) and, hence, the optimal MSA is meaningless. In this thesis, MSA naturally adopts a scoring scheme that can then be used to quantify the statistical significance of inferred formal specifications and reject those properties scoring too low to

be significant (i.e., scoring too low to be indistinguishable from noise or scores achievable by aligning sequences drawn by chance). We can construct a statistical distribution of the MSA score random variable for independent random sequences[38]. Given a significance level (or a *false-positives rate*) α, a MSA is considered statistically significant if the *p-value* p of the alignment is less than α, where p is the probability of having an alignment of independent random sequences scoring more than the observed score. Of course, we are left with two problems:

- Generating and aligning enough independent random sequences is tedious (if not impractical) and must be done for every new scoring scheme.
- Figuring out the probability distribution to fit the sampled score distribution. This is important because statistical significance depends heavily on the tail shape of the fitting distribution.[39]

A p-value that is too large can be an indication that the set of aligned trace slices are not closely related, probably because of one or more divergent traces that should be removed from the alignment. The score distribution of global multiple sequence alignments depends on lengths, composition and number of aligned sequences as well as the scoring scheme. No theoretical results on the score distribution of global multiple sequence alignments are known [133].

The score distribution will depend on how much information about the aligned traces we retain in the randomly generated traces used to estimate the score distribution for random sequences. Sequence shuffling (or permutation) [7, 72, 99] has already been used as a tool to construct the background models (or null hypotheses) necessary to quantify/assess the statistical significance of biological sequence alignments. The first-order shuffling method[40]

[38]This is the *null hypothesis* to be tested.

[39]The Extreme Value Distribution (EVD) applies only to local alignments where the score can be modeled as the maximum of N Gaussian random variables [59, Chapter 2].

[40]Implemented, for example, by the Java `Collections.shuffle()` method.

preserves only event frequencies (i.e., first-order statistics). In many cases, there can be logical units of activity larger than single events (or *singlets* [7]) that need to be preserved in order for permutations to be meaningful (i.e., *legal*). Otherwise, meaningless or illegal permutations will overwhelm and skew statistical significance estimates to be too optimistic and favor noisy alignments. As in [7], assessing statistical significance relative to a noise background of randomly permuted sequences evaluates whether or not similarity among the unpermuted sequences can be attributed to their *composition* (i.e., event frequencies) alone without regard to event orderings inside each sequence. Any other *natural* or *invariant* structure (in addition to composition) *common to the entire population of execution traces* and that we preserve in the generated permutations will be factored out from any statistical significance value ascribed to the aligned sequences. This is important because otherwise unrelated sequences will be declared similar due to these natural causes [7]. This can also be understood as follows: *permutations used to construct the null model need to reflect characteristics of the population of all possible/legal execution traces so that the inferred score distribution faithfully reflects the relationship among sequences drawn randomly from that population.*

In the context of specification mining, a simple example is preserving consecutive call-start/call-end pairs. Another example is preserving second-order (i.e., transition) statistics (or *doublet* frequencies). A more difficult structure, that is not present in biological sequences, to preserve in shuffled execution traces is recursive method calls, which appear in traces similarly to a perfectly matched set of nested parentheses. But recursive structure is destroyed by random permutations as well as by automata learning algorithms, since finite-state automata cannot capture/express recursion. Therefore, recursive structure is precluded from our consideration.

We use an algorithm from [7, 99] to generate all doublet-preserving permutations with equal probability, and use an algorithm from [173] as a subroutine to generate uniformly random

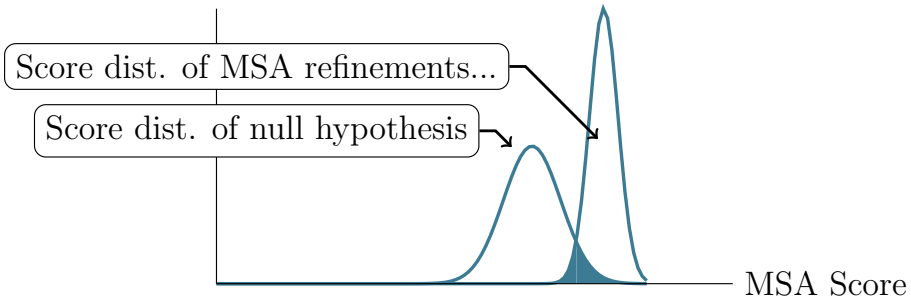


Figure 5.12: Statistical significance.

spanning trees. Basing statistical significance tests on shuffling and scoring the aligned sequences only may introduce some bias (or overfitting) [133]. Therefore, given a large set $\mathcal{S}$ of sequences containing the set $\mathcal{L} \subseteq \mathcal{S}$ of sequences being aligned, we construct the score distribution by shuffling, aligning and scoring H subsets $\{\mathcal{S}_1, \ldots, \mathcal{S}_H\}$, with $\mathcal{S}_k \subseteq \mathcal{S}$, each drawn randomly from $\mathcal{S}$.

As shown in Figure 5.13, the alignment score distribution of randomly shuffled sequences is a mixture of two components, a normal distribution and a Gamma distribution. However, we could not prove this decomposition consistently for all sequence alignments. Therefore, no assumptions about the form of score distributions are made and the p-value for a given alignment with score S is calculated as $p = M/N$, where N is the number of doublet-preserving permutations aligned and scored, and M is the number of those experiments scoring equal to or higher than S.

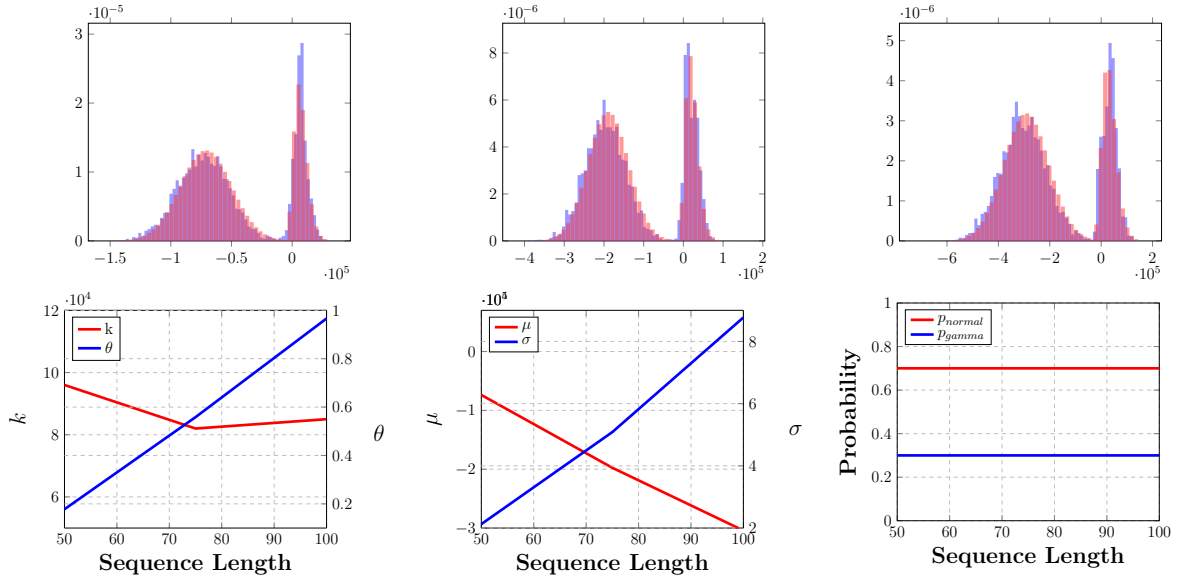


Figure 5.13: Alignment score distribution generated by conducting 5000 MSAs of 20 sequences of length 50/75/100 sampled randomly (**with replacement**) from 20 trace slices, and shuffled randomly each time. Benchmark is **Dropbox** (in indexing/uploading state) and the monitored data type is **Thread**. Doublet-preserving permutation is used. Blue bars depict the empirical PDF and the red bars depict the MLE-fitted PDF.

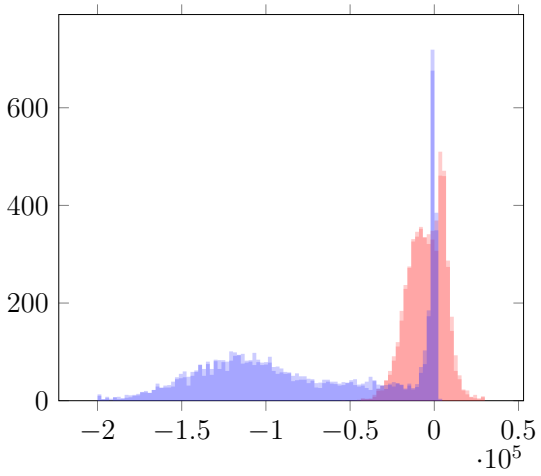


Figure 5.14: Alignment score distribution generated by conducting 5000 MSAs of 20 sequences of length 50 sampled randomly (**with replacement**) from 100 trace slices, and shuffled randomly each time. Benchmark is **Dropbox** (in indexing state) and the monitored data type is **Thread**. Effect of doublet-preserving permutation on score distribution is clearly demonstrated.

Part III

Applications and Future Research

Chapter 6

Mining Specifications for Digital Logic Designs

In this chapter, we embark on a novel and rigorous mining methodology (data preparation, mining algorithms, selection criteria, etc.) for finite-state automata checkers from large simulation traces of digital logic design using an iterative and interactive mining tool, called Topaz.[1] Topaz is evaluated using an open-source 32-bit RISC CPU design as a case study to demonstrate extraction of local (block-level) as well as global, core-level complex temporal properties cross-cutting through all CPU pipeline stages, guided by the CPU instruction set specification.

6.1 Introduction

Complex computing systems may possess behaviors never conceived of by their designers. Thorough functional verification helps to reduce design re-spins due to functional bugs and

[1] Topaz is a mineral gemstone.

enables first-pass silicon success. All verification techniques, static and dynamic, decide correctness of an implementation against some notion of a specification that describes *what* a computing system does or does not do (in the form of *properties*, golden models, etc.) independently of *how* the system will be implemented. Unfortunately, formal specifications[2] are not always used, mainly because of their high development and maintenance cost and complexity, since they require substantial expertise in formal specification languages and their decision procedures, as well as abstraction techniques. This difficulty hinders formal specifications from coping with agile, fast-paced development environments that foster rapidly evolving systems. Additionally, specifications of *legacy* RTL designs are needed to verify *new* designs integrated with them [176]. Here, a distinction can be made between standard interface-specific properties whose development cost is amortized over a large number of designs, and (2) the properties relevant only to one particular proprietary design. Unless the prohibitive specification development cost is reduced, design-specific properties will remain unjustified except for critical control blocks of a design.

A key insight is that specification of a design module can be implicit or hidden in how it is being used by, and reacting to, high-quality client code [176]. Therefore, *specification mining* [10**?**] emerged as an automated technique used to discover formal specifications of systems from examples or samples of their behavior executions, source code, change logs or any associated artifact. From Table 6.1, inferred properties can take many forms, such as rules and value invariants, finite-state machines (FSMs), and temporal properties. Inferred specifications and can then be examined, refined, abstracted or corrected by designers and verification specialists. Once validated, these specifications will drive functional verification, regression testing, or serve as invariants to be preserved as the design evolves, or can be published as part of documentation. Specification mining for digital hardware designs has been gaining much traction recently [37, 62, 71, 84, 97, 113, 150, 168]. Many of these tools

[2]Industrial specification languages include the Property specification language (PSL) [2] and SystemVerilog Assertions (SVAs) [3] with Open Verification Library (OVL) [4] and IBM Sugar as their precursors.

are of great practical value but most of them still suffer from serious limitations detailed in Section 6.2. From Table 6.1, notable examples of these limitations include the use of inadequate design state abstractions, limited expressive power (e.g., using a predefined repertoire of candidate property templates and a restricted set of operators to combine them), and using a finite time window which can easily miss long-range temporal relations.

In this chapter, we present a tool, called *Topaz*, that discovers design properties of arbitrary, tunable complexity and precision from large simulation traces. Due to their wide scope and intuitiveness, the target specification formalism of Topaz is deterministic finite automata (DFAs) that detect violations of automatically learned design properties. Topaz relies on a novel specification mining, and language learning, technique that is the first (to the best of our knowledge) to use *multiple sequence alignment* (MSA) [59] in order to obviate many limiting assumptions made by prior tools and resolve the long-standing *initial-state uncertainty* problem in *offline* specification mining [154]. Sound theoretical underpinnings of using MSA as a language learning tool are presented here. Using MSA, Topaz can reconstruct properties with *abstract* state spaces which do not merely duplicate the hidden design state space and whose sizes are dictated solely by the complexity of observed simulation traces. Each *abstract* state of the constructed DFA may stand for many *concrete* design states and can capture temporal relations among widely separated logic events. More abstraction can then be selectively applied among *simulation-equivalent* states [19] to trade off precision for conciseness or understandability. MSA also naturally adopts a scoring scheme that can be used to quantify the statistical significance of inferred specifications and reject spurious properties. Topaz also enables controlling abstraction levels of mined properties through the use of simulation preorder and equivalence [19] and by taking advantage of user-defined logic events, that can enable extracting transaction-level properties from RTL designs.

The rest of the chapter reviews the prior work on specification mining, and then develops terminology and notation for subsequent discussions. Next, we explain the Topaz mining

flow, followed by a case study on the open-source Amber CPU design. Some experimental results from the Amber case study are also scattered throughout the chapter. Finally, we conclude with a summary of results and venues for future work.

6.2 Related Work

A brief survey (which is by no means comprehensive) of specification mining tools for digital hardware designs is shown in Table 6.1. Dynamic analysis tools, such as Topaz, can be unsound. That is, they can return specifications not satisfied by the analyzed designs in all situations. *Inferno* [97] scours simulation traces for *transaction diagrams* depicting design behavior at a high level and generates Verilog checkers for them. Inferno focuses on control signals and abstracts away data buses (by using a bus-width threshold). This is justified by contrasting to PropGen [71], where the infrequent data values will obscure repetitive control sequences. However, a serious limitation of Inferno is that it identifies the internal design state with its output control signals. Completely ignoring internal design state (other than its outputs) producing FSMs that might have modest predictive power (i.e., modest precision in constraining observable design behavior) when used as an anomaly detector. Alternatively, control signal values should be used as an alphabet set Σ annotating *transitions* (rather than *states*) of a DFA whose states abstract the design internal state space while preserving as much information as possible.[3]

Dianosis [150] analyzes simulation traces and builds property candidates over all combinations of signals given a set of parameterized basic property templates, such as OVL checkers [4]. Extracted basic properties are then recursively combined into higher-level transactions until no more combination is possible. Combining lower-level properties relies on using three types of inter-property relations: mutual exclusion (or disjointness), coincidence

[3]This is the typical usage model in formal verification where transitions of a Büchi automaton derived from a LTL formula are triggered by the labels of the DUV transition system states.

(or sequence conjunction in the SVA sense [3]), and ordering. However, these relations are not sufficient to capture all of LTL/SERE semantics. For example, they cannot express consecutive and non-consecutive Klein closure (open-ended repetition). Moreover, being template-based, every new property must be explicitly added, which restricts expressiveness (i.e., the properties that can be mined using Dianosis).

In [37], extracted specifications express sequential relations among *events* which are unique combinations of signal values. Signals contributing to events are user-defined. Extracted assertions take the implication form $\mathbf{A} \Rightarrow \mathbf{C}$, where $\mathbf{A}$ is the antecedent and $\mathbf{C}$ is the consequent, and both $\mathbf{A}$ and $\mathbf{C}$ are *episodes* of events. An episode is a partially ordered (multi-)set of events and, hence, can be represented with a DAG. To reduce computational complexity, only relations among episodes within a preset sliding window are considered. Another window-based tool is [62], where a simulation trace is divided into fixed-size windows. Therefore, a sequential data miner can only detect patterns shorter than the given window size. Extracted assertions also take the implication form $\mathbf{A} \Rightarrow \mathbf{C}$, where $\mathbf{A}$ and $\mathbf{C}$ are frequent event sequences. Yet another window-based tool is PropGen [71], where extracted properties are low-level Boolean formulas over the inputs, state variables and outputs within a moving finite window W. Such formulas might not give as much insight as higher-level or more abstract properties.

All window-based tools have severely limited ability to discover long-range temporal relations among events[4], which are quite common in many pipelined designs. For example, in a processor pipeline, cache misses, branch prediction and resource conflicts can introduce wild latency variations. IODINE [84] extracts dynamic invariants (e.g., `req-ack` pairs, mutual exclusion, FSMs, scoreboards, etc.) from logic simulation traces. The FSMs mined by IODINE are based on *explicit* state vectors (i.e., bit-vectors that make selective transitions among a limited set of values). However, this lacks any abstraction or generalization, and

[4]In order to reduce search space, the window size must be kept small.

is thus not scalable and may duplicate design bugs in the extracted specification. In this chapter, we construct DFAs based on an *implicit* or *hidden* state space whose size is dictated by the complexity of simulation traces, as shown in Fig. 6.1. This is expected to abstract away many details pertinent only to the implementation and, hence, hopefully extracts bug-free more understandable specifications.

In [113], recurring temporal behaviors matching a set of pattern templates in a trace are mined and synthesized into more complex patterns by using inference rules, and can be translated into linear temporal logic (LTL) formulas or regular expressions. Specifications mined from correct as well as erroneous traces are used to localize or diagnose errors. Learning property templates from traces rather than learning a single FSM is justified by the fact that learning an automaton from samples of its behavior is NP-hard [79]. However, this misses the point of specification mining as a learning technique that is supposed to *generalize* beyond a limited set of observed traces rather than memorize them in a minimal FSM. Moreover, the assumption that multiple smaller properties can be learned and then composed into more complex specifications is also untenable since the synchronous product of two distinct properties extracted separately from the traces is not equivalent to (and is less precise than) a single *joint property* extracted directly from the traces.

Finally, GoldMine [168] uses data mining to automatically generate LTL assertions of the form $\mathbf{G}(A \Rightarrow C)$, where A and C can be propositional or temporal formulas involving operators **X**, **U** or **F**. However, the GoldMine methodology is also applicable within a bounded time window and "*cannot generate unbounded safety or liveness properties*".

In this chapter, we allow the salient design behaviors speak for themselves. Parameters are provided to control the trade-off between precision and mining time/space complexity, without constraining the *forms* of specifications inferred or capturing temporal relations within a bounded time window.

Table 6.1: Classification of specification mining tools.

Tool		Analysis			Target Formalism			Prior Info.	Requires Source Code[b]	Statistical Metric	Mining Techniques	Limitations
		Static	Dynamic	Hybrid	FSMs	Assertions	Rules					
Topaz		✖	✔	✖	✔	✖	✖	Interface signals	No	Statistical significance	Multiple sequence alignment	(Only safety)[a]
Dianosis	[150]	✖	✔	✖	✖	✔	✖	OVL templates	No	None	N.A.	I[a]
Chang et al.	[37]	✖	✔	✖	✖	✖	✔	Mining window size	No	Support/Confidence	Sequential data mining	I, II[a]
Mandouh et al.	[62]	✖	✖	✔	✖	✔	✖	Mining window size	Yes	Support/Confidence	Sequential data mining	I, II[a]
GoldMine	[168]	✖	✖	✔	✖	✔	✖	Mining window size	Yes	Support/Confidence	Decision Tree Learning	II[a]
IODINE	[84]	✖	✔	✖	✔	✔	✖	Templates/Analyzers	No	Confidence	Template-specific analysis	I[a]
PropGen	[71]	✖	✔	✖	✖	✔	✖	Mining window size	Yes	None	Bit-vector pattern search	I, III[a]
Inferno	[97]	✖	✔	✖	✔	✖	✖	Interface signals	No	None	Likely transact. boundary search	III[a]
Li et al.	[113]	✖	✔	✖	✖	✔	✖	Property templates	Only hierarchy	Frequency, variance	Pattern matching	I[a]

[a](I) Limited expressiveness. (II) Finite time span. (III) Lack of state-space abstraction. [b]Tools not relying on source code can be used in reverse engineering.

6.2.1 The Need for Abstraction

The number of embedded processor cores and IPs crammed on a single SoC quadrupled in the period from 2006 to 2014 [73] concomitant with many HW/SW *cross-cutting layers* of interacting functional, security and power domains. As a result, bug-triggering conditions became more complex and hard to stumble upon simply by manipulating bit-level design interfaces. That is why the most widely used verification methodologies (e.g., UVM, OVM, VMM, etc.) are centered around the *transaction* concept which raises the level of abstraction.

Specifications are abstract by definition. This means that specifications are detached from the inundating implementation details which refine those specifications. Therefore, a specification mining tool must be good at finding/crafting/discovering abstractions from very detailed logic simulation traces. This viewpoint is illustrated in Fig. 6.1, where Topaz extracts or constructs a DFA model for a large digital design hidden behind its simulation traces. Each (*abstract*) state of the constructed DFA model may stand for many (*concrete*) states of the modeled design. In many cases, the protocol of an interface is known or standard and it is required to reconstruct or *reverse-engineer* a higher-level protocol built on top of that interface. For example, a cache coherence protocol can operate on top of AMBA AXI bus protocol.

Topaz does not require access to the RTL source code and only works on simulation traces

dumped from comprehensive test-case runs.

6.3 Background

Functional verification of digital designs relies on various complementary technologies to efficiently implement verification plans and facilitate pre-silicon debugging. Among these technologies are constrained random testing (CRT), coverage-driven verification (CDV), and assertion-based verification (ABV). CRT drives a *design under verification* (DUV) into regions of the state space never conceived of by designers, thus finding bugs that would otherwise be hard to stumble upon using directed tests alone. CDV closes the loop by continuously measuring verification progress and flagging the need to steer CRT stimulus generation towards sparsely covered states. Using assertions in logic simulations add valuable insights into the verification process by virtue of the increased visibility and *precision* of bug detection (i.e., closeness of assertion firing to the problem root causes).

Usually, the control (i.e., sequencing) path of a digital design presents most of the correctness challenge and the hiding place for most bugs, while the data (or computing) path is less likely to hide any bugs and more amenable to analytic evaluation techniques.

6.4 Preliminaries

In this section, we develop the terminology and notation used throughout the chapter. In a synchronous digital design, a global clock orchestrates the propagation of signal values among all flip-flops.[5] A formal specification consists of one or more *properties*, where each property establishes a relation between signal values spanning multiple, not necessarily con-

[5]Nowadays, a typical digital design contains multiple, possibly asynchronous, clock domains.

secutive, clock cycles. The most widely used specification formalism is *propositional linear temporal logic* (PTL) [19], usually augmented with sequential extended regular expressions (SEREs). Binary signals taking part in a property specification are formally represented by a set $\mathbf{AP}$ of atomic propositions. These signals can themselves be explicit or (combinationally or sequentially) derived from design signals (such as the modeling-layer signals in PSL [2]), and temporal operators are used to express temporal relations among signals. Typical temporal operators include `next`, `eventually`, `always`, `until` in addition to (non-)consecutive repetition operators.

The design under verification (DUV), which is a synchronous digital logic design, can be described by a labeled transition system TS [19], given by a tuple $(S, \rightarrow, I, \mathbf{AP}, L)$, where S is the set of states, $\rightarrow \subseteq S \times S$ is a (possibly nondeterministic) transition relation, $I \subseteq S$ is a set of initial states, $\mathbf{AP}$ is a set of atomic propositions, and $L : S \rightarrow 2^{\mathbf{AP}}$ is a labeling function, where $2^{\mathbf{AP}}$ is the power set of $\mathbf{AP}$. If the set $\mathbf{AP}$ has n bits, then $2^{\mathbf{AP}}$ is isomorphic to the set $\{0,1\}^n$ of all n-bit vectors. A state typically represents the current value of all design registers *and* binary inputs. The transition relation $\rightarrow$ models the change of these registers and output bits in response to input changes at clock edges. Nondeterminism of $\rightarrow$ might be due to lack of constraints on the inputs. The set $\mathbf{AP}$ is chosen depending on the properties to be verified, and summarizes the *observable* facts of interest about states of TS. Each proposition characterizes exactly those states in which the proposition holds. Clock cycle time is modeled by the set $\mathbb{N}$ of integers.

A *path* in TS is a (possibly infinite) sequence of states $s_0, s_1, s_2, \ldots$ such that $s_0 \in I$ and $s_i \rightarrow s_{i+1}$ for $i \geq 0$. For every path $s_0, s_1, s_2, \ldots$, there is a *trace* $L(s_0), L(s_1), L(s_2), \ldots$. The set $Traces(TS)$ is the set of all traces of TS starting from an initial state. Let $(2^{\mathbf{AP}})^\omega$ be the set of infinite words over $2^{\mathbf{AP}}$. A linear-time (LT) property φ over $\mathbf{AP}$ is a subset of $(2^{\mathbf{AP}})^\omega$. Transition system TS satisfies LT property φ, written as $TS \models \varphi$, iff $Traces(TS) \subseteq \varphi$. A LT property φ is a *safety property* if every violating trace $\tau \in (2^{\mathbf{AP}})^\omega$ (i.e., $\tau \notin \varphi$) has a *finite bad*

prefix (i.e., a prefix all of whose infinite extensions also violate φ). A regular safety property is a safety property whose bad prefixes constitute a regular language and, hence, can be recognized by a finite automaton [19]. A finite automaton is a tuple $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$, where Q is a finite nonempty set of states, $Q_0 \subseteq Q$ is the set of *initial* states, $F \subseteq Q$ is the set of *accepting* (or *final*) states, Σ is a nonempty finite alphabet set.[6] Finally, $\delta : Q \times \Sigma \to 2^Q$ is the transition function. An automaton $\mathcal{A}$ is *deterministic* (a DFA) if $|Q_0| \leq 1$ and for every $q \in Q$ and $\sigma \in \Sigma$ we have $|\delta(q, \sigma)| \leq 1$.[7] It is *nondeterministic* (a NFA) otherwise. The extended transition function $\delta^* : Q \times \Sigma^* \to 2^Q$ is inductively defined as:

$$w, w' \in \Sigma^*, \sigma \in \Sigma, w = \sigma w' \Longrightarrow \delta^*(q, w) = \delta^*(\delta(q, \sigma), w')$$

A finite word $w \in \Sigma^*$ is accepted by $\mathcal{A}$ iff $\delta^*(Q_0, w) \cap F \neq \emptyset$. The language $\mathcal{L}(\mathcal{A})$ of automaton $\mathcal{A}$ is the set of all accepted finite words. For any $p, q \in Q$, if $q \in \delta(p, \sigma)$, this is abbreviated as $p \xrightarrow{\sigma} q$. If $\sim \subseteq Q \times Q$ is an equivalence relation, then each equivalence class is an abstract state [42] and $\mathcal{A}/\!\sim$ is the *quotient automaton*, which can be nondeterministic even if $\mathcal{A}$ is deterministic. It then holds that $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A}/\!\sim)$.[8]

Alphabet Set - Logic Events. Given **AP**, the set of logic signals used in specification mining, the alphabet set Σ of an extracted DFA is a *partition* over $2^{\mathbf{AP}}$. That is:

$$\bigcup_{\sigma \in \Sigma} \sigma = 2^{\mathbf{AP}} \qquad \text{and} \qquad \sigma_1 \neq \sigma_2 \Rightarrow \sigma_1 \cap \sigma_2 = \emptyset \tag{6.1}$$

Users of Topaz specify the alphabet symbols of Σ as (mutually exclusive) Boolean formulas over **AP**, since the set of Boolean formulas over **AP** is isomorphic to the power set of $2^{\mathbf{AP}}$. We can establish a *trace abstraction function* $\mathfrak{A} : Traces(TS) \to \Sigma^\omega$ in such a way that a logic trace $\tau : \mathbb{N} \to 2^{\mathbf{AP}}$ of transition system TS is translated to an *event trace* $\mathbf{e} : \mathbb{N} \to \Sigma$ by noting that for all $n \geq 0$, if $\sigma \in \Sigma$ and $\tau_n \in \sigma$, then $e_n = \sigma$.

[6] The set Σ will be related to $2^{\mathbf{AP}}$ later.

[7] In this case, δ becomes a (total) function $\delta : Q \times \Sigma \to Q$.

[8] This holds if the quotienting operation uses *existential abstraction* [42], which over-approximates the original automaton $\mathcal{A}$.

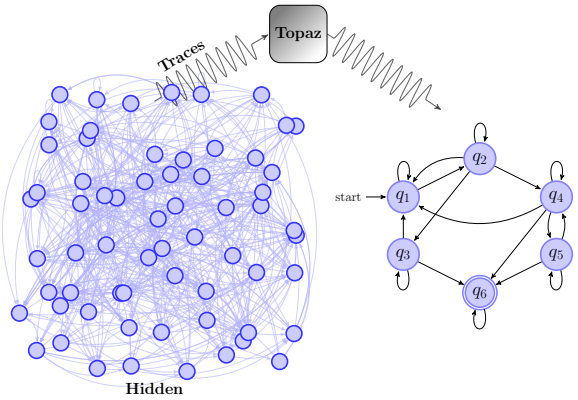


Figure 6.1: Mining FSMs infers a set of abstract states that capture the essence of design behavior, rather than duplicate its implementation internals.

Topaz extracts DFAs from logic simulation traces to capture regular safety properties, which can then be verified formally or by simulation. For a DFA $\mathcal{A}$, let $\mathcal{L}^\dagger(\mathcal{A}) = \mathcal{L}(\mathcal{A}).\Sigma^\omega$ be the set of all infinite extensions of words from the language of $\mathcal{A}$. Then the complementary set $\overline{\mathcal{L}^\dagger(\mathcal{A})}$ is the set of all event traces satisfying the regular safety property φ expressed by $\mathcal{A}$ and, in turn, $\mathfrak{A}^{-1}(\overline{\mathcal{L}^\dagger(\mathcal{A})})$ is the set of all logic traces satisfying φ. An extracted DFA $\mathcal{A}$ is intended to *over-approximate* the set $Sim \subseteq Traces(TS)$ of logic simulation traces used in the mining process. If Sim is adequate, it is then hoped that $\mathcal{A}$ also over-approximates the set $Traces(TS)$ of all possible observable behaviors of TS. That is, $Traces(TS) \subseteq \mathfrak{A}^{-1}(\overline{\mathcal{L}^\dagger(\mathcal{A})})$.

Formal Verification. A transition system TS can be verified to satisfy a regular safety property φ described by a NFA $\mathcal{A}^\varphi = (Q, \Sigma, \delta, Q_0, F)$, where $\Sigma = 2^{\mathbf{AP}}$, with the help of the product $TS \bigotimes \mathcal{A}^\varphi = (S', \rightarrow', I', \mathbf{AP}', L')$, defined in [19] as:

$$S' = S \times Q, I' = \{(s_0, q) | s_0 \in I, \delta^{-1}(q, L(s_0)) \cap Q_0 \neq \emptyset\} \tag{6.2}$$

$$\forall s, t \in S, \forall p, q \in Q : \frac{s \rightarrow t,\ p \xrightarrow{L(t)} q}{(s, p) \rightarrow' (t, q)} \tag{6.3}$$

To verify that $TS \models \varphi$, it is sufficient to check that for all $(s, q) \in S'$ reachable from I', we have $q \notin F$, since accepting states in F indicate violation of φ.

6.5 Specification Mining Flow

Topaz specification mining flow is shown in Fig. 6.2.

Trace Recording. Topaz needs a database of simulation traces (e.g., VCD files) providing sufficient coverage of system behavior in order to increase model precision. These traces typically originate in the testbench environment.

Event Extraction and Coalescing. Topaz helps users build the event alphabet by extracting the set of unique combined values of a user-specified list of signals. These values can then be coalesced into coarser (i.e., more abstract) events by the user. If the alphabet set Σ specified by the user violates the first condition of Equation 6.1, Topaz *completes* Σ by adding an event `NONE` that is complementary to all user-defined events.

Trace Profiling. Trace profiles ensure quality of results from the subsequent trace slicing and alignment stages. Simulated designs typically exhibit phase transitions, as shown in the spectrograms in Figure 6.4 produced by Topaz for `Amber` [1], an open-source design studied later here. Moreover, given an alphabet set Σ, a logic simulation trace T (or parts thereof) can be irrelevant for specification mining with Σ if T does not show sufficiently diverse and rich activity over Σ. To filter irrelevant simulation traces, Topaz can measure event (and event-doublet) frequencies over a large set of traces to help users identify testcases with maximum information content. In Figure 6.3, Topaz profiled 66 VCD files of Amber testcases over 5 alphabet sets. Relevant events are those specified by each profiled alphabet (i.e., all events other than `NONE`).

Trace Slicing. Trace slicing merely extracts the event traces (given a description of Σ) from raw logic simulation traces and divides each trace into multiple *trace slices* for subsequent MSA. It is crucial to select slice lengths and boundaries properly so as to ensure that slices contain complete episodes of the behaviors of interest. This helps to improve the quality of

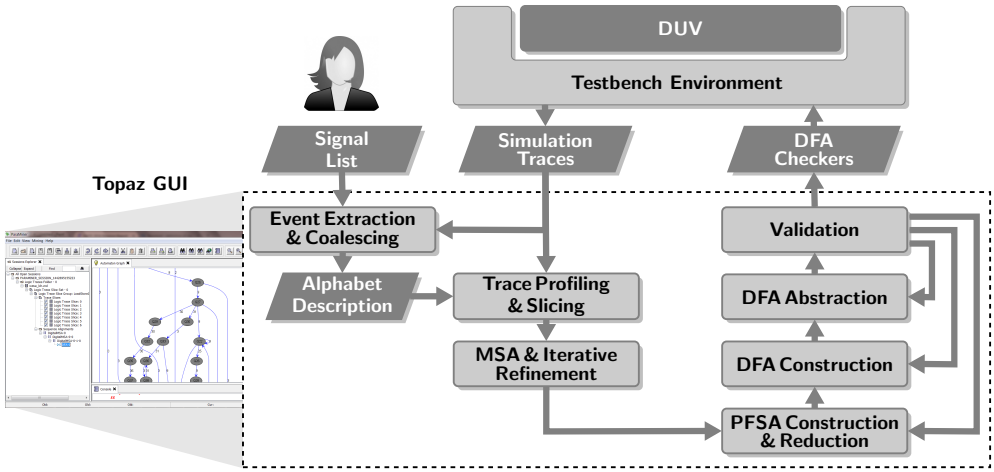


Figure 6.2: Topaz specification mining flow. Topaz is a Java application with 55,000 lines of code.

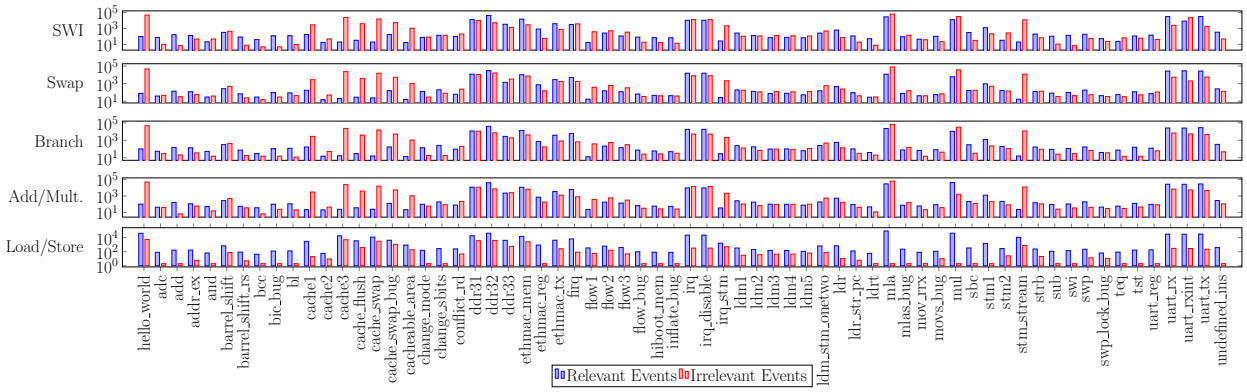


Figure 6.3: Testcase alphabet profiles. The x-axis is a set of 66 testcases from the Amber test suite. The y-axis is a set of 5 alphabet sets used in Section 6.6.

subsequent MSA and the final outcome.

Sequence Alignment. Most automata-learning algorithms [12, 25] assume the existence of a means to *reset* the automaton being learned to a fixed start state, either by using a reset signal or by applying *homing sequences* [148]. However, these techniques are only meaningful for *online* learning of automata. In this chapter, only *offline* learning using passive observation of simulation traces is used. To obviate the use of resets[9], we now

[9]Even though resets are ubiquitous in almost all digital designs, a specification mining algorithm *must* avoid them completely. Resets are useful in learning finite automata which are, by definition, terminating systems, whereas digital designs are intended to be *reactive* or non-terminating [19]. A single sufficiently long trace should be enough for Topaz to infer system properties.

show that MSA, which revolutionized biological sequence analysis [59], holds the answer to the *initial-state uncertainty problem* [154] in passive or offline learning contexts. Moreover, MSA enables modeling the observed system behavior with a NFA $\mathcal{A}$ whose abstract state space Q is not set in advance. In a MSA, as shown in Figure 6.5, two or more traces are arranged as rows of a 2-dimensional matrix so that *identical* logic events common to one or more traces are aligned in the same column. Due to the intrinsic variability of computing systems behaviors, different traces cannot typically be perfectly superimposed or aligned. Non-alignable positions are filled with *gap* symbols. Unlike biological sequences alignments [59], it is prohibited to align different symbols (i.e., $\sigma_1, \sigma_2 \in \Sigma$ with $\sigma_1 \neq \sigma_2$) in the same column. The only degree of freedom left by this is *placement of gaps*. How can MSA be useful for specification mining? For every *observation sequence* $\boldsymbol{\sigma} = (\sigma_0, \sigma_1, \sigma_2, \ldots) \in \Sigma^\omega$ of logic events, there is an implicit *unknown* labeling $\mathscr{L} : \mathbb{N} \to S \times Q$ of each event with a *hidden state* of transition system $TS' = TS \bigotimes \mathcal{A}$. Conceptually, a labeling $\mathscr{L} = (s_0, q_1), (s_1, q_2), \ldots$ represents two *synchronous* and *parallel* runs $s_0, s_1, \ldots$ and $q_1, q_2, \ldots$ of TS and $\mathcal{A}$, respectively, where there is $q_0 \in Q_0$ such that $\forall i \geq 0 : \ \sigma_i = L(s_i)$ and:

$$q_0 \xrightarrow{\sigma_0} q_1 \xrightarrow{\sigma_1} q_2 \xrightarrow{\sigma_2} q_3 \ldots \quad \| \quad s_0 \longrightarrow s_1 \longrightarrow s_2 \longrightarrow s_3 \ldots$$

Therefore, given n different observation sequences (or trace slices) $\{\boldsymbol{\sigma}^1, \ldots, \boldsymbol{\sigma}^N\}$ and a corresponding set of hidden-state labelings $\{\mathscr{L}_1, \ldots, \mathscr{L}_N\}$, we can impose an alignment on these sequences, where any two events σ_i^m and σ_j^n from $\boldsymbol{\sigma}^m$ and $\boldsymbol{\sigma}^n$, respectively, can be aligned

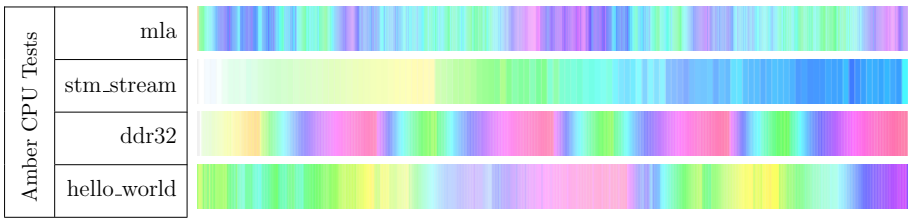


Figure 6.4: Examples of distinct design operation phases vs. cycle time as captured by the largest two principal components (displayed as color hue and saturation, resp.) of the feature-vector histogram of the **design bit-vector** (DBV) within a moving window of size 100 clock cycles. The DBV combines values of all logic signals in one giant bit vector. DBV features are counts of (overlapping) binary strings from "`0`" to "`111`".

only if $\mathscr{L}_m(\sigma_i^m) = \mathscr{L}_n(\sigma_j^n)$. Conversely, hidden-state labelings can be recovered if an appropriate alignment of the observation sequences can be established, which can then be used to reconstruct an abstract version of TS or $\mathcal{A}$ or their product $TS \bigotimes \mathcal{A}$. However, there is not a unique alignment induced by every labeling. It is only possible to find an alignment that maximizes a given optimality criterion (or *scoring scheme*) that ranks different alignments. Moreover, there is not a unique hidden-state labeling for every observation sequence due to nondeterminism of both TS and $\mathcal{A}$. By using enough simulation traces, a sufficiently faithful image of all nondeterministic choices in $TS \bigotimes \mathcal{A}$ can be reconstructed. Despite some peculiarities of specification mining in Topaz, the details of MSA are largely standard [59, 61, 167] and are omitted for space constraints.

Complexity. Heuristic MSA algorithms [59] rely on pairwise sequence alignment, which has complexity $\mathcal{O}(mn)$, n and m being the lengths of the two sequences. Moreover, a distance matrix is constructed based on pairwise alignment scores. Therefore, time and space complexity will be $\mathcal{O}(k^2nm)$, where k is the number of trace slices.

Graph Representation of MSAs. We now explain how a NFA $\mathcal{A}$ can be recovered from a given MSA $\mathbf{m}$. As a first step, we construct a probabilistic finite-state automaton (PFSA) [52, 145] $\mathcal{G}_{\mathbf{m}} = (V, \Sigma, p)$, as shown in Figure 6.6, where V is a finite nonempty set of states, Σ is the alphabet set (which is a partition over $2^{\mathbf{AP}}$), and $p : V \times \Sigma \times V \to [0, 1]$

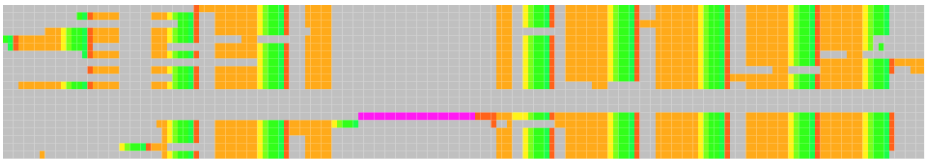


Figure 6.5: A section of an example MSA of 20 trace slices from the Amber `stm_stream` testcase. Gaps are gray and every logic event is depicted by a different color.

is the transition probability function such that:[10]

$$\forall v \in V : \sum_{v' \in V} \sum_{\sigma \in \Sigma} p(v, \sigma, v') = 1$$

For now, there are no initial or final states in $\mathcal{G}_{\mathbf{m}}$. The PFSA $\mathcal{G}_{\mathbf{m}}$ is a directed acyclic graph (DAG) constructed by noting that all identical letters $\sigma \in \Sigma$ in the same column $\mathbf{m}_i$ of $\mathbf{m}$ stand for a single *unknown* hidden state (s, q) of $TS \bigotimes \mathcal{A}$. Since the state spaces S and Q of TS and $\mathcal{A}$, respectively, are unknown, we use a unique pair of labels (s, q) for every column of $\mathbf{m}$.[11] Later, we will identify (i.e., merge) PFSA states that look sufficiently similar. A PFSA state $v \in V$ is connected by a directed edge to another state v' if, for at least one of the aligned trace slices, a letter in column $\mathbf{m}_v$ directly follows a letter in column $\mathbf{m}_{v'}$, possibly with intervening gap symbols only. Thus, every trace slice is a path in $\mathcal{G}_{\mathbf{m}}$. An edge in $\mathcal{G}_{\mathbf{m}}$ is annotated with transition probability according to how many trace slices follow that edge in $\mathbf{m}$. For these transition probabilities to be accurate, it is desired to align as many trace slices as possible (i.e., to have *deep alignments*). The label $\sigma \in \Sigma$ of a PFSA edge $(u, \sigma, v) \in V \times \Sigma \times V$ in $\mathcal{G}_{\mathbf{m}}$ can be inferred by reversing Equation 6.3:

$$\forall s, t \in S, \forall p, q \in Q : \frac{(s, p) \rightarrow' (t, q)}{p \xrightarrow{L(t)} q}$$

So every PFSA edge is labeled with the alphabet symbol in the MSA column associated with its sink PFSA state.

PFSA Reduction Until now, the PFSA constructed from a MSA is acyclic and, hence, it does not possess any recurrent behavior and cannot generalize beyond the training set of traces. Recurrent behavior can be introduced into a PFSA by merging closely related states that are likely to stand for *similar* hidden system state [19, Chapter 7]. The sk-strings method [145] constructs an *over-approximation* $\mathcal{G}'$ of a PFSA $\mathcal{G}$ by associating a

[10] The standard definition of PFSA has:

$$\forall v \in V,\ \forall \sigma \in \Sigma : \sum_{v' \in V} p(v, \sigma, v') = 1$$

This standard definition only replaces nondeterminism with probability.

[11] This yields an unfolded version of $TS \bigotimes \mathcal{A}$.

set of languages $\mathcal{L}_K^P(v)$, parameterized by a depth $K > 0$ and a probability $0 \leq P \leq 1$, with every PFSA state v. $\mathcal{L}_K^P(v)$ is the set of most likely length-K words whose total probability is P. Two states v_1 and v_2 are sk-equivalent up to depth K and with probability P if $\mathcal{L}_K^P(v_1) \subseteq \mathcal{L}_\infty^1(v_2)$ and $\mathcal{L}_K^P(v_2) \subseteq \mathcal{L}_\infty^1(v_1)$. By merging two states v_1 and v_2 that are sk-equivalent, it is guaranteed that $\mathcal{L}(\mathcal{G}) \subseteq \mathcal{L}(\mathcal{G}')$. The degree of over-approximation is controlled by P and K which helps to trade off PFSA precision for conciseness.

Determinization and Minimization. As a first step toward determinization and minimization, a PFSA is turned into a NFA simply by dropping transition probabilities and selecting one or more meaningful initial states. Selection of initial states happens after a relatively small NFA has already been mined, which enables users to make more informed decisions.

Topaz applies power-set construction [95] to a mined NFA to obtain the corresponding *complete* DFA, where every state has a transition on every logic event. Therefore, a failure (or accepting) state is added to the constructed DFA and a *failure edge* is extended to it from every other DFA state. A failure edge is annotated with an event (i.e., a Boolean formula) that is complementary to all other edges of its source state.

Abstraction. To allow users to trade off precision of a mined DFA $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ for

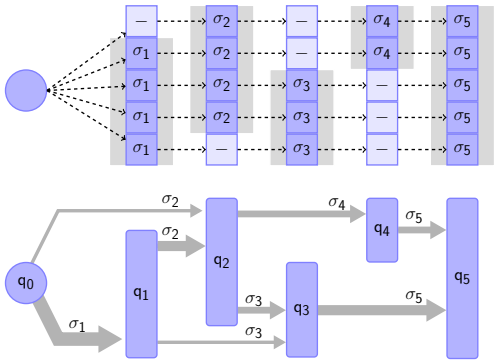


Figure 6.6: A MSA viewed as a PFSA. Each state corresponds to one MSA column. Each edge is annotated with σ of the MSA column associated with its sink PFSA state. Edge line width is proportional to transition probability.

more conciseness, equivalence relations over Q with varying granularities are needed. Given an equivalence relation $\sim \subseteq Q \times Q$, the state space Q can be reduced by taking the quotient $\mathcal{A}/\sim$. A first step toward an equivalence relation is a *simulation relation* [19]. A simulation preorder $\mathcal{R} \subseteq Q \times Q$ is a relation such that for all $(p, q) \in \mathcal{R}$ and $\sigma \in \Sigma$, if $p \xrightarrow{\sigma} p'$, then there is $q' \in Q$ such that $q \xrightarrow{\sigma} q'$ and $(p', q') \in \mathcal{R}$. If a simulation relation $\mathcal{R}$ exists over $\mathcal{A}$ and $(p, q) \in \mathcal{R}$, it is said that state q *simulates* state p, denoted by $p \preceq q$. Topaz uses algorithms that compute simulation preorders in time $\mathcal{O}(|\delta|.|Q|)$ [19]. A simulation relation $\preceq$ over Q can give rise to equivalence relations, such as the symmetric closure $\sim = \preceq \cup \preceq^{-1}$ and the symmetric kernel $\cong = \preceq \cap \preceq^{-1}$. Between these two extremes, it is left to the users of Topaz to identify genuine state equivalences that preserve classes of properties important to them.

Statistical Significance. One of the main challenges that specification mining tools are facing is false positives [80]; that is inferring spurious patterns that do not codify genuine properties of the system. MSAs naturally adopt a scoring scheme that can then be used to quantify the statistical significance (the *p-value*) of inferred properties. To estimate the p-value for a given MSA $\mathbf{m}$, Topaz generates random sequences from the sequences of $\mathbf{m}$ by randomly shuffling order of their events [7, 72, 99]. The p-value for a given alignment with score X_0 is calculated as $p = M/N$, where N is the number of permutations aligned and scored, and M is the number of those experiments scoring $\geq X_0$.

Validation. Once a DFA has been constructed, we can use the DUV as a source of counterexamples, by running the system long enough under various operating conditions to reveal any violations of the conjectured DFA and refine it.[12] Alternatively, a set of traces deemed representative of all correct system behaviors is divided into two sets: a *training set*, used in the specification mining process, and a *validation set* used to validate the mined DFA.

[12]Using the terms of [12], only *equivalence queries* are allowed, whereas *membership queries* are not.

6.6 Amber Case Study

This section details one case study of Topaz on a fairly rich CPU design, Amber [1], which is an open-source 32-bit RISC processor core implementing ARMv2a instruction-set architecture (ISA). The Amber-25 has a five-stage pipeline, separate data and instruction caches, and a Wishbone interface, but does not contain a memory management unit (MMU).

Target Signal Groups. Given a digital logic design TS, the set **AP** of logic signals at the focus of specification mining and verification can be selected in many different ways. For each set Sim of logic simulation traces, Topaz produces a DFA for each set **AP** of logic signals. We select sets of signals that span multiple pipeline stages and, hence, their temporal relations may span many clock cycles. The organizing principle here is that for each ARMv2a instruction, there is a set of signals that may exhibit some activity during the processing of that instruction in the pipeline. That activity typically depends on preceding and succeeding instructions (e.g., branches, back-to-back loads or stores), the cache state, etc. This is a treasure trove for specification miners.

Simulation Traces. Topaz is driven by VCD simulation traces dumped by Amber's test-cases. Most of them are directed test-cases that exercise a particular instruction sequence or a bug-triggering scenario.

Extracted Properties. Table 6.2 shows the inputs and parameters used by Topaz in 5 mining sessions with 5 different alphabet sets corresponding to 5 instruction families. It also shows some descriptors of the mining outcomes. The number of DFA states reported in each case is before any simulation equivalence is applied. Moreover, the number of simulation-preordered pairs of states is an indication of the room for abstraction present in a mined DFA.

Topaz Mining Run-time. Topaz, as well as RTL simulations, were run on a quad-core

Table 6.2: Amber specification mining parameters and results for 5 different alphabet sets associated with 5 instruction types.

Alphabet Sets ▶	**Load/Store**	**Add/Mult.**	**Branch**	**Swap**	**SWI**
Alphabet size	18	45	17	30	34
Modeled Signals	10	28	15	23	20
K	4	4	4	4	4
P	0.5	0.5	0.5	0.5	0.5
Trace Slices	225	152	14	158	100
Slice Size	200	100	200	200	200
Alignment time (min)	< 1	< 1	< 1	< 1	< 1
p-value	0%	0%	0%	0%	0%
DFA States	57	172	127	84	137
Sim-preordered pairs	763	277	850	934	281

Intel i5, 2.5GHz CPU with 8GB of RAM and 64-bit operating systems (Windows for Topaz, and Centos-6 for RTL simulations). The most time-consuming stage in Topaz mining flow is MSA. In all 5 cases shown in Table 6.2, it completed in less than a minute, which is on par with state-of-the-art MSA tools [61, 167].

SystemVerilog Checkers. Topaz automatically generates a synthesizable SystemVerilog module for every extracted DFA. These checker modules can be bound to instances of design modules using the SystemVerilog `bind` command without changing design source code. A checker module has one output signal that is set to logic `1` once a minimal bad prefix has been observed and stays high forever.

6.7 Summary and Conclusions

The feasibility of mining meaningful and diverse checker automata for a realistic CPU design, with minimal input from users, has been demonstrated. Designers and verification engineers can easily target different points on the precision-vs.-conciseness trade-off frontier. They may also defer deciding initial states until a relatively small NFA has already been mined and, hence, more informed decisions can be made. We are now working on integrating Topaz with a formal verification tool to decide validity of mined properties. Finally, there is still

room for improving automated ranking of simulation-preordered pairs, alphabet selection and abstraction.

Chapter 7

Epilogue

In conclusion, we summarize the main contributions made in this thesis and comment on the currently ongoing and future research efforts.

We introduced the expressive and efficiently monitorable specification language of self-replicating automata, making use of first-order logic to empower classical finite-state automata to ensure proper handling of data along the time dimension. The graph representation of self-replicating automata offered a precise, dynamic and elegant picture of the underlying sea or ensemble of automaton replicas teeming with life as program execution unfolds. It also proved to be an efficient and shareable distributed data structure suitable for implementation on a scalable, distributed and low-overhead RV architecture, NUVA. That graphical representation eventually emerged as an independent contribution in its own right, and was applied to the most ubiquitous class of functions, Boolean functions. An essential piece of the puzzle is ParaMiner, our specification mining and behavioral modeling flow that automates learning of software behavior, without prior assumptions about its form, and helps to extract specifications that truly abstract away implementation details.

With many offerings on the market, and even more to come, of multiprocessor chips fitted

out with on-chip FPGA fabric, NUVA has a clear path to adoption in enterprise, cloud and embedded computing systems putting a premium on visibility, safety and security.

We are currently working on many challenging problems including:

- *Using RV based on SR-DFAs for anomaly-based intrusion detection by monitoring system calls made by a running program*: The ability of SR-DFAs to digest, not only the sequences of system calls, but also their argument values offers an opportunity to craft more precise specifications of normal behavior. This is beneficial for two reasons: first, precise behavioral models or specifications hinder attackers ability to mount mimicry attacks intended to avoid detection. Second, precise models help to reduce the false-positives rate which has been the main inhibitor of widespread adoption of anomaly-based intrusion detection.
- *Malware classification*: We are experimenting with exploiting the enhanced precision levels of SR-DFA models of computing systems in summarizing system-call traces (with argument values included) of many malware samples (we collected north of 3000 samples) and using them in rapid classification of these samples and inferring their malicious intent. This is instrumental in forensic analysis in enterprise environments where there is an explosion in the rate of attacks and intrusion attempts.
- *Using lattice-based function graphs (LBFGs) as a basis for approximation algorithm design*: we are studying representation of combinatorial optimization problems defined on partially ordered sets and how to approximate their solutions by symbolically manipulating LBFGs.
- *Runtime verification of transaction-level models* (TLMs): We are experimenting with SR-DFAs ability to express many high-level properties of embedded computing systems and digital logic designs and offer a very good match to transaction-level modeling.

Bibliography

[1] OpenCores. `http://opencores.org/`. Accessed: 2015-09-18.

[2] *IEEE standard 1850-2005 for property specification language (PSL)*. IEEE Standards Association, 2005.

[3] *IEEE Standard for SystemVerilog–unified Hardware Design, Specification, and Verification Language*. IEEE Standards Association, 2010.

[4] Accellera. *OVL V2 Standard Library Reference Manual*. 2010.

[5] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining API Patterns As Partial Orders from Source Code: From Usage Scenarios to Specifications. ESEC/FSE, pages 25–34, 2007.

[6] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. OOPSLA, 2005.

[7] S. F. Altschul and B. W. Erickson. Significance of nucleotide sequence alignments: a method for random sequence permutation that preserves dinucleotide and codon usage. *Molec. Bio. and Evol.*, 1985.

[8] R. Alur, P. Černý, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. POPL, pages 98–109, 2005.

[9] L. Amarú, P.-E. Gaillardon, and G. De Micheli. Biconditional bdd: A novel canonical bdd for logic synthesis targeting xor-rich circuits. DATE, 2013.

[10] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. POPL, 2002.

[11] D. Angluin. On the complexity of minimum inference of regular sets. *Information and Control*, 39(3):337 – 350, 1978.

[12] D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, Nov. 1987.

[13] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha. Secure embedded processing through hardware-assisted run-time monitoring. In *DATE*, 2005.

[14] S. Arora and S. Safra. Probabilistic checking of proofs: A new characterization of np. *J. ACM*, 45(1):70–122, Jan. 1998.

[15] T. M. Austin. Diva: A reliable substrate for deep submicron microarchitecture design. MICRO 32, 1999.

[16] S. Axelsson. Intrusion detection systems: A survey and taxonomy. Technical report, Chalmers University of Technology, 2000.

[17] M. M. Bach, M. Charney, R. Cohn, E. Demikhovsky, T. Devor, K. Hazelwood, A. Jaleel, C.-K. Luk, G. Lyons, H. Patil, and A. Tal. Analyzing parallel programs with pin. *Computer*, 2010.

[18] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. ICCAD, 1993.

[19] C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[20] H. Barringer, Y. Falcone, K. Havelund, G. Reger, and D. E. Rydeheard. Quantified event automata: Towards expressive and efficient runtime monitors. In *FM*, 2012.

[21] A. Bauer, J.-C. Küster, and G. Vegliach. From propositional to first-order monitoring. In *RV*, 2013.

[22] A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.*, 2011.

[23] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton, NJ, USA, 2011.

[24] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. TACAS, 1999.

[25] A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Trans. Comput.*, 21(6):592–597, June 1972.

[26] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The Gem5 simulator. *SIGARCH Comput. Archit. News*, 2011.

[27] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06*, pages 169–190, Oct. 2006.

[28] E. Bodden, L. Hendren, and O. Lhoták. A staged static program analysis to improve the performance of runtime monitoring. ECOOP, 2007.

[29] I. M. Bomze, M. Budinich, P. M. Pardalos, and M. Pelillo. The maximum clique problem. In *Handbook of Combinatorial Optimization*, pages 1–74. Kluwer Academic Publishers, 1999.

[30] D. Brand. Verification of large synthesized designs. ICCAD, 1993.

[31] L. C. Briand, Y. Labiche, and J. Leduc. Toward the reverse engineering of uml sequence diagrams for distributed java software. *IEEE Trans. Softw. Eng.*, 32(9):642–663, Sept. 2006.

[32] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 1986.

[33] R. E. Bryant. Binary decision diagrams and beyond: enabling technologies for formal verification. In *ICCAD*, 1995.

[34] J. Brzozowski. Canonical regular expressions and minimal state graphs for definite events. *Mathematical Theory of Automata*, 1962.

[35] R. Burkard, M. Dell'Amico, and S. Martello. *Assignment Problems*. Society for Industrial and Applied Mathematics, 2012.

[36] S. Butt, V. Ganapathy, A. Baliga, and M. Christodorescu. Monitoring data structures using hardware transactional memory. RV, 2012.

[37] P.-H. Chang and L. C. Wang. Automatic assertion extraction via sequential data mining of simulation traces. ASP-DAC, 2010.

[38] F. Chen and G. Roşu. Mop: An efficient and generic runtime verification framework. OOPSLA, 2007.

[39] F. Chen and G. Roşu. Parametric trace slicing and monitoring. TACAS, 2009.

[40] S. Chiba. Javassist – a reflection-based programming wizard for Java. In *Proceedings of the ACM OOPSLA Workshop on Reflective Programming in C++ and Java*, 1998.

[41] D. G. Chinnery and K. Keutzer. Closing the power gap between asic and custom: An asic perspective. In *Proceedings of the 42nd Annual Design Automation Conference*, DAC'05. ACM, 2005.

[42] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. CAV, pages 154–169, 2000.

[43] J. E. Cook and A. L. Wolf. Automating process discovery through event-data analysis. ICSE, pages 73–82, 1995.

[44] S. A. Cook. The complexity of theorem-proving procedures. STOC, 1971.

[45] F. Corno, M. S. Reorda, and G. Squillero. RT-level ITC'99 benchmarks and first ATPG results. *IEEE Design & Test of Computers*, 17(3):44–53, 2000.

[46] O. Coudert. Two-level logic minimization: An overview. *Integr. VLSI J.*, 17(2):97–140, Oct. 1994.

[47] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller. Mining object behavior with adabu. WODA, pages 17–24, 2006.

[48] M. d'Amorim and G. Roşu. Efficient monitoring of ω-languages. CAV, 2005.

[49] A. Darwiche. SDD: A new canonical representation of propositional knowledge bases. IJCAI, 2011.

[50] C. de la Higuera. A bibliographical study of grammatical inference. *Pattern Recogn.*, 38(9):1332–1348, Sept. 2005.

[51] C. de la Higuera. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, New York, NY, USA, 2010.

[52] C. de la Higuera. Learning finite state machines. FSMNLP, 2010.

[53] R. Dedekind. Über Zerlegungen von Zählen durch ihre größten gemeinsamen Teiler. In *Festschrift Hoch Braunschweig Ges. Werke. vII*, pages 103–148, 1897.

[54] J. N. Departarnento and P. Garcia. Identifying regular languages in polynomial time. In *Advances in Structural and Syntactic Pattern Recognition, volume 5 of Series in Machine Perception and Artificial Intelligence*, pages 99–108. World Scientific, 1992.

[55] V. M. F. Dias, C. M. H. de Figueiredo, and J. L. Szwarcfiter. On the generation of bicliques of a graph. pages 109–113, 2004.

[56] R. Diestel. *Graph Theory*. Springer-Verlag Berlin Heidelberg, 2005.

[57] A. T. Do, C. Yin, K. Velayudhan, Z. C. Lee, K. S. Yeo, and T.-H. Kim. 0.77 fJ/bit/search Content Addressable Memory Using Small Match Line Swing and Automated Background Checking Scheme for Variation Tolerance. *IEEE JSSC*, 2014.

[58] C. M. Dobson. Protein folding and misfolding. *Nature*, 426:884–890, 2003.

[59] R. Durbin. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge university press, 1998.

[60] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. ICSE, 1999.

[61] R. C. Edgar. MUSCLE: multiple sequence alignment with high accuracy and high throughput. *Nucleic Acids Res*, 32(5):1792–7, 2004.

[62] E. El Mandouh and A. G. Wassal. Automatic generation of hardware design properties from simulation traces. ISCAS, 2012.

[63] M. El-Ramly, E. Stroulia, and P. Sorenson. From Run-time Behavior to Usage Scenarios: An Interaction-pattern Mining Approach. KDD, pages 315–324, 2002.

[64] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs As Deviant Behavior: A General Approach to Inferring Errors in Systems Code. SOSP, pages 57–72, 2001.

[65] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 213–224, 1999.

[66] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Softw. Eng.*, 27(2):99–123, 2001.

[67] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, 2007.

[68] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. ISCA, 2011.

[69] Y. Falcone. You should better enforce than verify. In *RV*, 2010.

[70] D.-F. Feng and R. F. Doolittle. Progressive sequence alignment as a prerequisite to correct phylogenetic trees. *Journal of Molecular Evolution*, 25(4):351–360, 1987.

[71] G. Fey and R. Drechsler. Improving simulation-based verification by means of formal methods. ASP-DAC, 2004.

[72] W. M. Fitch. Random sequences. *J. of Molec. Bio.*, 1983.

[73] H. Foster. The perfect storm: Trends in functional verification. DAC, 2015.

[74] M. Frantzen and M. Shuey. Stackghost: Hardware facilitated stack protection. SSYM, 2001.

[75] M. Fujita, P. McGeer, and J.-Y. Yang. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. *Formal Methods in System Design*, 1997.

[76] A. Garg, A. Di Cara, I. Xenarios, L. Mendoza, and G. De Micheli. Synchronous versus asynchronous modeling of gene regulatory networks. *Bioinformatics*, 2008.

[77] G. Geeraerts, G. Kalyon, T. L. Gall, N. Maquet, and J.-F. Raskin. Lattice-valued binary decision diagrams. In *ATVA*, 2010.

[78] M. Gendreau, P. Soriano, and L. Salvail. Solving the maximum clique problem using a tabu search approach. *Ann. Oper. Res.*, 41(1-4):385–403, May 1993.

[79] E. M. Gold. Complexity of automaton identification from given data. *Information and Control*, 37(3):302–320, June 1978.

[80] C. Goues and W. Weimer. Specification mining with few false positives. TACAS, pages 292–306, 2009.

[81] R. Grigore, D. Distefano, R. L. Petersen, and N. Tzevelekos. Runtime verification based on register automata. TACAS, 2013.

[82] G. D. Hachtel and F. Somenzi. *Logic Synthesis and Verification Algorithms*. Springer, 2013.

[83] S. Hallé and R. Villemaire. Runtime enforcement of web service message contracts with data. *IEEE Trans. Serv. Comput.*, 2011.

[84] S. Hangal, N. Chandra, S. Narayanan, and S. Chakravorty. IODINE: A Tool to Automatically Infer Dynamic Invariants for Hardware Designs. DAC, 2005.

[85] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. ICSE, pages 291–301, 2002.

[86] D. Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274, 1987.

[87] K. Havelund. Monitoring with data automata. ISoLA, 2014.

[88] T. A. Henzinger, R. Jhala, and R. Majumdar. Permissive interfaces. *SIGSOFT Softw. Eng. Notes*, 30(5):31–40, Sept. 2005.

[89] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. POPL, 2002.

[90] C. D. L. Higuera and J.-C. Janodet. Inference of omega-Languages from Prefixes. ALT, pages 364–378, 2001.

[91] M. Hirosawa, Y. Totoki, M. Hoshida, and M. Ishikawa. Comprehensive study on iterative algorithms of multiple sequence alignment. *Computer Applications in the Biosciences*, 11(1):13–18, 1995.

[92] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *J. Comput. Secur.*, 1998.

[93] P. Hogeweg and B. Hesper. The alignment of sets of sequences and the construction of phyletic trees: an integrated method. *Journal of molecular evolution*, 20(2):175–186, 1984.

[94] G. J. Holzmann. The Model Checker SPIN. *IEEE Trans. Softw. Eng.*, 1997.

[95] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2006.

[96] P.-T. Huang and W. Hwang. A 65nm 0.165 fJ/Bit/Search 256x144 TCAM Macro Design for IPv6 Lookup Tables. *IEEE JSSC*, 2011.

[97] B. Isaksen and V. Bertacco. Verification through the principle of least astonishment. ICCAD, 2006.

[98] D. Jin, P. O. Meredith, D. Griffith, and G. Rosu. Garbage collection for monitoring parametric properties. PLDI, 2011.

[99] D. Kandel, Y. Matias, R. Unger, and P. Winkler. Shuffling biological sequences. *Discrete Appl. Math.*, 71(1-3):171–185, Dec. 1996.

[100] R. M. Karp. *Reducibility among combinatorial problems*. Springer, 1972.

[101] M. J. Kearns and L. G. Valiant. Cryptographic limitations on learning boolean formulae and finite automata. In D. S. Johnson, editor, *STOC*, pages 433–444. ACM, 1989.

[102] C. Kern and M. R. Greenstreet. Formal verification in hardware design: A survey. *ACM Trans. Des. Autom. Electron. Syst.*, 4(2):123–193, Apr. 1999.

[103] D. E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams*. Addison-Wesley Professional, 12th edition, 2009.

[104] D. Kroening and O. Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer Publishing Company, Inc., 2008.

[105] F. Kröger and S. Merz. *Temporal Logic and State Systems*. Springer Publishing Company, Inc., 2008.

[106] M. Kulkarni, M. Burtscher, K. Pingali, and C. Cascaval. Lonestar: A suite of parallel irregular programs. ISPASS, 2009.

[107] A. v. Lamsweerde. Formal specification: A roadmap. ICSE, 2000.

[108] K. J. Lang. Random dfa's can be approximately learned from sparse uniform examples. In D. Haussler, editor, *COLT*, pages 45–52. ACM, 1992.

[109] D. Le Berre and A. Parrain. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, 2010.

[110] C. Lee, F. Chen, and G. Roşu. Mining parametric specifications. ICSE, 2011.

[111] C. Lee, C. Grasso, and M. F. Sharlow. Multiple sequence alignment using partial order graphs. *Bioinformatics*, 18(3):452–464, 2002.

[112] M. Leucker and C. Schallhart. A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, 2009.

[113] W. Li, A. Forin, and S. A. Seshia. Scalable specification mining for verification and diagnosis. DAC, 2010.

[114] Z. Li and Y. Zhou. PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code. ESEC/FSE, pages 306–315, 2005.

[115] B. Livshits and T. Zimmermann. DynaMine: Finding Common Error Patterns by Mining Software Revision Histories. ESEC/FSE, pages 296–305, 2005.

[116] D. Lo and S.-C. Khoo. Smartic: Towards building an accurate, robust and scalable specification miner. FSE, pages 265–275, 2006.

[117] D. Lo, S.-C. Khoo, J. Han, and C. Liu, editors. *Mining Software Specifications: Methodologies and Applications*. Chapman & Hall/CRC Data Mining and Knowledge Discovery Series, 2011.

[118] D. Lo, S.-C. Khoo, and C. Liu. Efficient mining of iterative patterns for software specification discovery. KDD, pages 460–469, 2007.

[119] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. ICSE, pages 501–510, 2008.

[120] H. Lu and A. Forin. Automatic processor customization for zero-overhead online software verification. *IEEE Trans. VLSI Syst.*, 2008.

[121] S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: Detecting atomicity violations via access interleaving invariants. ASPLOS, 2006.

[122] Q. Luo, Y. Zhang, C. Lee, D. Jin, P. O. Meredith, T. F. Serbanuta, and G. Rosu. Rv-monitor: Efficient parametric runtime verification with simultaneous properties. RV, 2014.

[123] F. Maggi, M. Matteucci, and S. Zanero. Detecting intrusions through system call sequence and argument analysis. *IEEE Trans. Dependable Secur. Comput.*, 2010.

[124] S. Malik. Runtime verification: A computer architecture perspective. In *RV*, 2011.

[125] H. Mannila, H. Toivonen, and A. Inkeri Verkamo. Discovery of frequent episodes in event sequences. *Data Min. Knowl. Discov.*, 1(3):259–289, Jan. 1997.

[126] J. a. P. Marques-Silva and K. A. Sakallah. Boolean satisfiability in electronic design automation. DAC, 2000.

[127] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[128] C. Meinel and T. Theobald. *Algorithms and Data Structures in VLSI Design*. Springer-Verlag New York, Inc., 1998.

[129] A. Meixner, M. E. Bauer, and D. Sorin. Argus: Low-cost, comprehensive error detection in simple cores. MICRO 40, 2007.

[130] A. Meixner and D. J. Sorin. Dynamic verification of memory consistency in cache-coherent multithreaded computer architectures. *IEEE Trans. Dependable Secur. Comput.*, 2009.

[131] C. C. Michael and A. Ghosh. Simple, state-based approaches to program-based anomaly detection. *ACM Trans. Inf. Syst. Secur.*, 5(3):203–237, Aug. 2002.

[132] S.-i. Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. DAC, 1993.

[133] A. Y. Mitrophanov and M. Borodovsky. Statistical significance in biological sequence analysis. *Briefings in Bioinformatics*.

[134] E. F. Moore. Gedanken-experiments on sequential machines. In C. Shannon and J. McCarthy, editors, *Automata Studies*, pages 129–153. Princeton University Press, 1956.

[135] A. Nassar and F. J. Kurdahi. Lattice-based boolean diagrams. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 468–473, 2016.

[136] A. Nassar, F. J. Kurdahi, and W. Elsharkasy. Nuva: Architectural support for runtime verification of parametric specifications over multicores. In *Compilers, Architecture and Synthesis for Embedded Systems (CASES), 2015 International Conference on*, CASES, pages 137–146, 2015.

[137] A. Nassar, F. J. Kurdahi, and S. R. Zantout. Topaz: Mining high-level safety properties from logic simulation traces. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1473–1476, 2016.

[138] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., 1999.

[139] J. W. Nimmer and M. D. Ernst. Automatic generation of program specifications. ISSTA, pages 229–239, 2002.

[140] P. Perona and J. Malik. Scale-space and edge detection using anisotropic diffusion. *IEEE Trans. Pattern Anal. Mach. Intell.*, 12(7):629–639, July 1990.

[141] L. Pike, A. Goodloe, R. Morisset, and S. Niller. Copilot: A hard real-time runtime monitor. RV, 2010.

[142] M. Prvulovic. Cord: Cost-effective (and nearly overhead-free) order-recording and data race detection. HPCA, 2006.

[143] M. Prvulovic and J. Torrellas. Reenact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. ISCA, 2003.

[144] M. Prvulovic, Z. Zhang, and J. Torrellas. Revive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors. ISCA, 2002.

[145] A. Raman, J. Patrick, and P. North. The sk-strings method for inferring PFSA. ICML, 1997.

[146] S. Rehman, M. Shafique, F. Kriebel, and J. Henkel. Reliable software for unreliable hardware: Embedded code generation aiming at reliability. CODES+ISSS, 2011.

[147] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. Swift: Software implemented fault tolerance. CGO, 2005.

[148] R. L. Rivest and R. E. Schapire. Inference of finite automata using homing sequences. STOC, 1989.

[149] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. Automated API Property Inference Techniques. *IEEE Trans. Softw. Eng.*, 39(5):613–637, May 2013.

[150] F. Rogin, T. Klotz, G. Fey, R. Drechsler, and S. Rülke. Automatic generation of complex properties for hardware designs. DATE, 2008.

[151] S. K. Sahoo, M.-L. Li, P. Ramachandran, S. V. Adve, V. S. Adve, and Y. Zhou. Using likely program invariants to detect hardware errors. In *DSN*, 2008.

[152] N. Saitou and M. Nei. The neighbor-joining method: a new method for reconstructing phylogenetic trees. *Molecular biology and evolution*, 1987.

[153] M. B. Salem, S. Hershkop, and S. J. Stolfo. A survey of insider attack detection research. In *Insider Attack and Cyber Security*, pages 69–90. Springer, 2008.

[154] S. Sandberg. Homing and synchronizing sequences. In M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors, *Model-Based Testing of Reactive Systems*, 2004.

[155] S. R. Sarangi, A. Tiwari, and J. Torrellas. Phoenix: Detecting and recovering from permanent processor design bugs with programmable hardware. MICRO 39, 2006.

[156] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 1997.

[157] S. Shoham, E. Yahav, S. Fink, and M. Pistoia. Static specification mining using automata-based abstractions. ISSTA, pages 174–184, 2007.

[158] J. H. A. Shyamkumar Thoziyoor, Naveen Muralimanohar and N. P. Jouppi. Cacti 5.1. In *HP Labs, Tech. Rep. HPL-2008-20*, April 2008.

[159] R. R. Sokal and C. D. Michener. A statistical method for evaluating systematic relationships. *University of Kansas Scientific Bulletin*, 28:1409–1438, 1958.

[160] F. Somenzi. *CUDD: CU Decision Diagram Package, Release 2.4.0.* University of Colorado at Boulder, 2005.

[161] D. J. Sorin, M. D. Hill, and D. A. Wood. *A Primer on Memory Consistency and Cache Coherence.* Morgan & Claypool, 2011.

[162] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. Safetynet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery. ISCA, 2002.

[163] A. Srinivasan, T. Ham, S. Malik, and R. K. Brayton. Algorithms for discrete function manipulation. In *Computer-Aided Design, 1990. ICCAD-90. Digest of Technical Papers., 1990 IEEE International Conference on*, pages 92–95, Nov 1990.

[164] V. Stolz. Temporal assertions with parametrized propositions. *J. Log. and Comput.*, 2010.

[165] V. Stolz and E. Bodden. Temporal assertions using AspectJ. *Electron. Notes Theor. Comput. Sci.*, 2006.

[166] M. B. Taylor. Is dark silicon useful?: Harnessing the four horsemen of the coming dark silicon apocalypse. DAC, 2012.

[167] J. Thompson, D. Higgins, and T. Gibson. CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic Acids Research*, 22:4673–4680, 1994.

[168] S. Vasudevan, D. Sheridan, S. Patel, D. Tcheng, B. Tuohy, and D. Johnson. Goldmine: Automatic assertion generation using data mining and static analysis. DATE, 2010.

[169] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. CCS, 2002.

[170] I. Wagner, V. Bertacco, and T. Austin. Shielding against design flaws with field repairable control logic. DAC, 2006.

[171] L. Wang and T. Jiang. On the complexity of multiple sequence alignment. *Journal of Computational Biology*, 1(4):337–348, 1994.

[172] W. Weimer and G. C. Necula. Mining temporal specifications for error detection. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'05, pages 461–476, Berlin, Heidelberg, 2005. Springer-Verlag.

[173] D. B. Wilson. Generating random spanning trees more quickly than the cover time. STOC, 1996.

[174] M. Xu, R. Bodík, and M. D. Hill. A serializability violation detector for shared-memory server programs. PLDI, 2005.

[175] S. Yang. Logic synthesis and optimization benchmarks user guide version 3.0, 1991.

[176] A. Zeller. Mining Specifications: A Roadmap. In S. Nanz, editor, *The Future of Software Engineering*, pages 173–182. Springer, 2011.

[177] P. Zhou, R. Teodorescu, and Y. Zhou. Hard: Hardware-assisted lockset-based race detection. In *HPCA*, 2007.

[178] Y. Zhou, P. Zhou, F. Qin, W. Liu, and J. Torrellas. Efficient and flexible architectural support for dynamic monitoring. *ACM TACO.*, 2005.