Inferring Programmer Intent and Related Errors from Software

By

MARK GREGORY GABEL

B.S. (California Polytechnic State University, San Luis Obispo) 2006
M.S. (California Polytechnic State University, San Luis Obispo) 2006

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Zhendong Su, Chair

Premkumar Devanbu

Todd Green

Committee in Charge

2011

Contents

Abstract

Software is difficult to write and maintain. Much of the challenge in developing a program lies in specifying it—understanding precisely what it should be doing. Both human-oriented tasks (like fixing a bug) and automated tasks (like mechanical verification) require knowledge of a program's intended behavior. For the vast majority of software projects, though, complete and well-documented specifications simply do not exist. Writing specifications—discovering and codifying intent—is a time-consuming and largely manual process.

This dissertation presents research into easing this process through automation. The work has focused on the problem of automatically "inferring" specifications directly from programs by analyzing their behavior. This dissertation presents a family of related algorithms, frameworks, and tools for reverse engineering a specific (but common) class of specification: temporal safety properties. It also includes a presentation of OCD, a software tool that leverages these "inference" techniques to both learn specifications and find bugs—simultaneously and fully automatically. Each presented algorithm and tool is practical, finding useful specifications and previously unknown bugs in large, widely-used software projects like Eclipse.

This dissertation concludes with a discussion of future work on this topic, including an outline of my vision of a new research area I am calling Intelligent Program Analysis.

Acknowledgments

First and foremost, I would like to thank my partner of eight years, Marisa Nolasco. Her love and support made this work possible. I would also like to thank my friends, family, and coworkers for their support as well.

I have had the incredibly fortunate experience of having had excellent managers (or “bosses”) throughout my entire career. These individuals have motivated me, nurtured my strengths, and helped me develop my weaknesses. I would like to thank them all for laying the path that has led me to where I am today:

Erik Vesneski, of Epicentric;

Mark McGarvey, of Epicentric;

Ken Larson, of Quintron;

Mark Johnson, of SRI International;

Michael Haungs, my M.S. advisor at California Polytechnic;

and, finally,

Zhendong Su, my patient, supportive, inspiring, and brilliant doctoral advisor at UC Davis.

1 Introduction

A software program is an implementation of ideas. In order to perform virtually any software engineering task—indeed, to do anything nontrivial with software—a developer must understand those ideas. This knowledge might be codified as, say, documents containing formal requirements or source code comments detailing the code's design. While superficially quite different, a general idea underlies these examples: they are both forms of *specifications*.

Unfortunately, specifications are often missing, incomplete, or (perhaps worst of all) incorrect. This raises a natural question: how, then, does software get built at all? The answer appears to be "slowly, and full of mistakes." Development and quality assurance tasks are crippled without a clear idea of a program's intent, with maintenance on larger systems approaching more art than engineering. And the problem restricts research as well: program verification tools, which are becoming eminently more practical of late, are useful only if we can feed them important specifications to verify.

This problem—the lack of specifications in software—is the focus of this dissertation. This work presents four distinct algorithms, frameworks, and tools for automatically reverse engineering, or *mining* specifications directly from software projects.

This dissertation is organized into chapters, each of which forms a distinct set of contributions. The material in chapters 2–4 has been peer-reviewed and previously published at ACM-sponsored software engineering conferences, while the newest material in Chapter 5 is yet unpublished.

Chapters 2 and 3: Javert The next two chapters (Chapters 2 and 3) describe the components of the Javert specification mining system. Javert is unique in that it requires very little actual input from the user, whereas other approaches depend on daunting, error-prone user input like *templates* to match ("specifications in my system should take this form") and *elements* to focus on (“specifications in my system should contain only x, y, and z”). In other words, Javert is completely automated.

The heart of the Javert (Chapter 3) is a completely new way of looking at the problem, one more mathematical in nature. Javert first mines small specification *components* we call “micropatterns.” With these axiomatic components in place, Javert then uses deductive inference rules to *assemble* them into larger, more complete specifications—effectively allowing the expressive forms of the specifications to emerge on their own.

Javert is highly practical, finding rich, accurate specifications in several large real-world software applications. It owes a great deal of its scalability to the underlying technique used to mine “micropatterns.” Existing algorithms lacked the scalability and expressiveness to mine these foundational patterns, so we developed a highly scalable symbolic mining algorithm—the first of its kind—and implemented it using Binary Decision Diagrams (BDDs) (Chapter 2).

Its practical effectiveness aside, Javert is proving to be a strong conceptual contribution as well. A recent reimplementation and extension of our work by a group at UC Berkeley and Microsoft Research [Li et al., 2010] successfully applied the Javert technique to an entirely different domain: mining specifications in hardware designs.

Chapter 4: OCD The most visible effects of software development problems are bugs. Bugs—errors in software—affect all software projects at some point, and they are difficult and costly to find and fix. There is an opportunity here for automated tools to markedly improve the quality of software and save countless hours of effort.

In order to find errors in our software, we must have an idea of what it *should* be doing. There are a few things we know of all programs: they should not crash, for example. But many (perhaps most) software bugs involve more domain-specific problems, like, say, a web browser rendering a page incorrectly.

Finding these “semantic” bugs requires knowledge of an oft-missing specification. But—fortunately—a specification is something our tools can provide. This is what led us to build OCD, a tool that finds bugs in software by simultaneously mining *and* checking specifications (Chapter 4). The tool is quite practical: it found previously-unknown defects in well-known open source software projects.

OCD works online and completely autonomously, learning about a program and self-tuning its own parameters on the fly. The ability to maintain this level of automation while finding "semantic" bugs is OCD's most exciting property: it blurs the line between program verification and artificial intelligence. we believe it to be the first step in what we are calling *Intelligent Program Analysis*: a family of techniques and tools that actively *comprehend* software as they work, automatically improving it in ways previously thought to be possible only by an experienced *human* developer.

Chapter 5: DSI The final contribution of this dissertation is a new conceptual framework for specification inference called Deductive Specification Inference (DSI). Nearly all other work on specification inference—including the work presented in Chapters 2–4—is based on *inductive reasoning*. Induction is the process of inferring general knowledge from examples. Tools like Javert and OCD make general inferences about specifications based on *observations* (of software) and *beliefs* (about software). Those tools are effective, but they both must grapple with the issue of precision. A mined specification can satisfy all our prior "beliefs" about what a good specification should be, including being concise, frequent, and satisfied, and still be completely wrong.

DSI is a new way of approaching the problem that is designed around the fundamental essence of a specification: if a program violates a specification, then that program should be "incorrect." An instance of DSI starts with a given *potential* specification—perhaps one mined by Javert—and continues to explicitly define and build a *design space* of potential programs to which the specification may apply. Within that design space, DSI systematically tests the hypothesis "a violation of this specification leads to a failure."

We implemented DSI for the domain of temporal specifications (the domain covered by the earlier chapters). In our implementation, we generate an entire *family* of programs around each specification through *automated program transformations*. We then perform our hypothesis testing using traditional software testing. DSI is not only effective—it provides a much deeper and more complete understanding of what it means to "infer" specifications from programs.

2 Symbolic Inference of Temporal Specifications

Program specifications are important in many phases of the software development process, but they are often omitted or incomplete. An important class of specifications takes the form of temporal properties that prescribe proper usage of components of a software system. Recent work has focused on the automated inference of temporal specifications from the static or runtime behavior of programs. Many techniques match a specification pattern (represented by a finite state automaton) to all possible combinations of program components and enumerate the possible matches. Such approaches suffer from high space complexity and have not scaled beyond simple, two-letter alternating patterns (*e.g.* $(ab)^*$). In this chapter, we precisely define this form of specification mining and show that its general form is NP-complete.

We observe a great deal of regularity in the representation and tracking of all possible combinations of system components. This motivates us to introduce a symbolic algorithm, based on binary decision diagrams (BDDs), that exploits this regularity. Our results show that this symbolic approach expands the tractability of this problem by orders of magnitude in both time and space. It enables us to mine more complex specifications, such as the common three-letter resource acquisition, usage, and release pattern ($(ab^+c)^*$). We have implemented our algorithm in a practical tool and used it to find significant specifications in real systems, including Apache Ant and Hibernate. We then used these specifications to find previously unknown bugs.

2.1 Introduction

Accurate specifications are invaluable assets for the software development process. These specifications often take the form of finite automata that describe *temporal properties*. These properties formally specify the legal interaction patterns with specific components of a system. Canonical examples include alternation properties, like lock acquisition and release, and resource ownership properties, like the requirement that calls to `read(fd)` must all exist between calls to `open(fd)` and `close(fd)`.

When used prescriptively, temporal specifications can help to inexpensively prevent future bugs. When used retroactively, these specifications are candidates for verification by model checkers and static verifiers [Ball and Rajamani, 2001, Das et al., 2002, Flanagan et al., 2002, Xie and Aiken, 2005], lighter weight static analysis techniques [Chen and Wagner, 2002, Hovemeyer and Pugh, 2004], or manual inspection. These properties can also be verified through software testing, where they can be verified as system invariants. The characteristics of these properties can aid in the generation of test cases.

These specifications are often missing, incorrect, or otherwise incomplete. Researchers have recognized this problem and have sought to develop methods for retroactively *inferring* specifications from systems based on their common behavior. Current techniques can be categorized by the inputs and outputs of their respective algorithms. Most algorithms take as input some type of program or program trace and produce one or more temporal specifications as finite automata. More specifically, some techniques [Alur et al., 2005, Ammons et al., 2002, Whaley et al., 2002] seek to produce a single, arbitrarily complex automaton based on a known alphabet. For example, one might wish to learn the interaction patterns of a language's socket API. The user provides the algorithm with the API functions and a program representation (either the source code or dynamic traces), and a finite automaton describing the probable correct behavior is returned.

Other techniques [Engler et al., 2001, Ramanathan et al., 2007a, Weimer and Necula, 2005, Yang et al., 2006] take as input a specification *pattern* in the form of an automaton. The user provides the algorithm with a larger set of input symbols, often representing every function in the system or event in the trace, and a program representation. The algorithm then enumerates the set of assignments of the input symbols into the automaton's alphabet such that the automaton's interaction pattern holds over the program. Most previous work has focused on *alternating* patterns. For example, the user can provide an alternating template representing the regular expression $(ab)^*$, a program, and an alphabet of possible assignments. The algorithm returns a set of properties that match this expression, including properties like lock acquisition and release.

One possible reason for this limitation to alternating patterns is the expense of tracking all possible assignments into the input automaton's alphabet. Basic algorithms track the state of every possible assignment over the program's execution. This approach has a large space requirement: for n input symbols and a pattern alphabet of size k, these algorithms require $O(n^k)$ space. Because there are only two participants in an alternating pattern, the computation can be performed in quadratic space.

This work provides a novel symbolic technique for this type of specification inference that is efficiently implemented using binary decision diagrams (BDDs) [Bryant, 1986]. Our technique reduces both the space and time requirements by using the regularity of the problem space to form an efficient symbolic representation.

Under empirical evaluation on real traces from large systems, we observe that our algorithm is able to extend this approach to specification patterns with alphabets of at least three letters. More importantly, this allows the inference of the previously mentioned common resource acquisition/use/release protocol, which is described by the pattern $(ab^+c)^*$. We have applied our algorithm to traces from real systems and used the discovered properties to find previously unknown bugs.

This chapter makes the following technical contributions:

1. We formalize the pattern-based specification mining problem and show that its general form is NP-Complete.

2. We provide a symbolic algorithm based on *Binary Decision Diagrams* (BDDs) that greatly expands the computational tractability of this problem.

3. We provide practical modifications to our algorithm to allow further scaling and tolerance of imperfect input data.

4. We implement a specification mining tool based on our algorithm and evaluate its performance. We use this tool to mine several interesting specifications, which we use to locate previously unknown bugs in real systems.

The rest of this chapter is structured as follows. We begin with a formal definition of the pattern-based specification mining problem and an analysis of its computational complexity (Section 2.2). We continue with the presentation of our symbolic algorithm (Section 2.3), which includes the necessary background information on BDDs. We then discuss our implementation and present the results of our empirical evaluation (Section 2.4). Finally, we discuss related work (Section 2.5) and conclude with ideas for future work (Section 2.6).

2.2 Formal Analysis

In this section, we formalize the pattern-based specification mining problem and analyzes its computational complexity. We also analyze its expressiveness to motivate our work.

2.2.1 The Specification Mining Problem

Our work focuses on mining properties from dynamic traces and is most related to the work of Yang et al. [2006]. We begin with a precise definition of the problem.

Definition 1 (Specification Mining). Assume we have a specification template represented by an automaton A over an alphabet Σ and an execution trace T over a *disjoint* alphabet Σ', *i.e.*, $\Sigma \cap \Sigma' = \emptyset$. The *pattern-based specification mining* problem is defined as the enumeration of all satisfying total and one-to-one mappings $\rho : \Sigma \to \Sigma'$ such that the projected trace of T over the range of ρ is accepted by $\rho(A)$, which is the same as A except each transition $\delta(s,a) = t$ is replaced by $\delta(s,\rho(a)) = t$.

Consider this example trace:

open, use, use, close, dispose, get, get, open, use, close, dispose

Suppose we mine the *alternating* automaton pattern in Figure 2.1 over this trace. In this example, the assignments (a=open, b=close), (a=open, b=dispose), and (a=close, b=dispose) are satisfied over the trace.

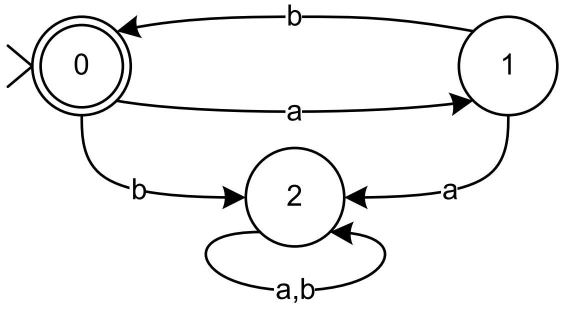


Figure 2.1: An automaton representing the alternating pattern.

To demonstrate the complexity of this problem, we show that its general, decision problem counterpart is NP-Complete.

Definition 2 (SpecMine)**.** Given a specification template represented as an FSA A over an alphabet Σ and an execution trace T over a *disjoint* alphabet Σ', *i.e.*, $\Sigma \cap \Sigma' = \emptyset$. The *specification mining question (SpecMine)* asks whether there exists a total and one-to-one mapping $\rho : \Sigma \to \Sigma'$ such that the projected trace of T over the range of ρ is accepted by $\rho(A)$, which is the same as A except each transition $\delta(s,a) = t$ is replaced by $\delta(s,\rho(a)) = t$.

Note that for this to be of interest, it is clear that $|\Sigma| \leq |\Sigma'|$. Our reduction is from the Hamiltonian Path Problem. First let us recall the definition of the problem.

Definition 3 (HamPath)**.** We consider the directed version of the problem: Given a directed graph $G = (V,E)$, does G have a path that visits each vertex $v \in V$ exactly once?

The problem is well-known to be NP-complete [Karp, 1972]. We will give a polynomial-time reduction from HamPath to SpecMine to show that SpecMine is NP-hard. Consider an instance of HamPath $G = (V,E)$. We construct a nondeterministic automaton $A = (\Sigma, S, s_0, \delta, F)$ as follows:

- $\Sigma = V$;
- $S = \bigcup_{v \in V} \{s_v\} \cup \{s_0, s^*\}$;
- s_0 is the start state;
- $\forall (u,v) \in E.\ \delta(s_u, u) = s_v$;
- $\forall u \in V.\ \delta(s_0, \epsilon) = s_u$;

- $\forall u \in V.\ \delta(s_u, u) = s^*$;
- $F = \{s^*\}$.

Let $|V| = n$. Let Σ' be an arbitrary alphabet such that Σ' and Σ have the same cardinality (*i.e.*, $|\Sigma'| = |\Sigma|$) and are disjoint (*i.e.*, $\Sigma \cap \Sigma' = \emptyset$). We then construct a trace $T = T_1 \ldots T_n$ over Σ' such that $T_i \neq T_j$ for all $1 \leq i \neq j \leq n$.

Lemma 1. *$G = (V, E)$ has a Hamiltonian path if and only if $SpecMine(A, T)$.*

Proof. ($\Rightarrow$) Assume $G = (V, E)$ has a Hamiltonian path p. We construct a mapping $\rho : \Sigma \rightarrow \Sigma'$ as follows:

$$\rho(p[i]) = T[i]$$

where $p[i]$ denotes the i-th vertex on the path p and $T[i]$ denotes the i-th letter on the trace T ($1 \leq i \leq n$). This mapping shows that T is accepted by A by considering the following path p through A:

$$s_0, s_{p[1]}, s_{p[2]}, \ldots, s_{p[n]}, s^*$$

The path p causes A to accept the string

$$\begin{aligned} & \epsilon.\rho(p[1]).\rho(p[2]).\ldots.\rho(p[n]) \\ = \ & \rho(p[1]).\rho(p[2]).\ldots.\rho(p[n]) \\ = \ & T[1].T[2].\ldots.T[n] \\ = \ & T \end{aligned}$$

($\Leftarrow$) Assume $SpecMine(A, T)$. Then there exists a mapping $\rho : \Sigma \rightarrow \Sigma'$ such that A has a path p that accepts T. Since ρ is invertible, the path p must have the form:

$$s_0, s_{\psi(T[1])}, \ldots, s_{\psi(T[n])}, s^*$$

where ψ denotes ρ^{-1}. Thus $\psi(T[1]), \ldots, \psi(T[n])$ is a path of G and it is Hamiltonian. □

Theorem 1 (NP-completeness)**.** *SpecMine is NP-complete.*

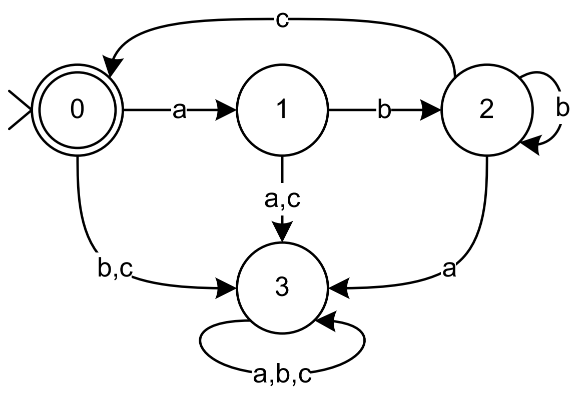


Figure 2.2: An automaton representing the resource usage pattern.

Proof. We first show that SpecMine is in NP. Its NP-completeness follows because it is clear that the reduction from HamPath to SpecMine is polynomial. SpecMine is in NP because we can generate in polynomial time a mapping ρ and check, also in polynomial time, whether the projected trace is accepted by $\rho(A)$. $\square$

2.2.2 The Need for Larger Automata

Consider the automaton in Figure 2.2. It represents the pattern $(ab^+c)^*$, which describes common programming tasks that involve acquiring a resource, using it, and properly releasing it. It is desirable to mine this specification pattern from programs, but it is either costly (for small examples) or impossible (for moderately sized examples) with current techniques.

In some cases, it is possible to exploit the transitive relationships between smaller properties to form larger properties. In our earlier example, the algorithm discovers the properties $(\mathsf{open},\mathsf{close})^*$, $(\mathsf{open},\mathsf{dispose})^*$, and $(\mathsf{close},\mathsf{dispose})^*$. From these three 2-letter properties and their transitive relationships, we may infer a 3-letter property: $(\mathsf{open},\mathsf{close},\mathsf{dispose})^*$. The intersection of the *expansion* of three discovered languages over their combined alphabet is precisely the inferred property. As further motivation for our work, we show that the automaton in Figure 2.2 has no two-letter decomposition.

Definition 4 (Expansion)**.** The *expansion* of an automaton A with alphabet Σ over an arbitrary alphabet Σ', $E_{\Sigma'}(A)$, is an automaton, A', that is equal to A with a looping transition $\delta(q,a)=q$ added to each state q for each letter $a \in \Sigma' \setminus \Sigma$.

The expansion operator is essentially the inverse of projection. For example, the regular expression corresponding to $E_{\{a,b,c\}}((ab)^*)$ is $c^*(ac^*bc^*)^*$. In the context of specification mining, the expansion of a property over the alphabet of all possible trace events yields a language that describes all possible program traces in which the property holds.

Definition 5 (Decomposition)**.** A set of automata $\{A'_1,\ldots,A'_n\}$ with alphabets $\{\Sigma'_1,\ldots,\Sigma'_n\}$ is a *decomposition* of a finite automaton A with alphabet Σ iff:

$$\bigcup_{i=1}^{n} \Sigma'_i = \Sigma \quad \bigwedge \quad \bigcap_{i=1}^{n} \mathscr{L}\left(E_\Sigma(A'_i)\right) = \mathscr{L}(A)$$

Theorem 2. *The resource usage automaton in Figure 2.2, R, represented by the regular expression* $(ab^+c)^*$*, has no two-letter decomposition.*

Proof. Let Σ be the alphabet of R ($\{a,b,c\}$). We can assume without loss of generality that the decomposition would consist of exactly $\binom{3}{2} = 3$ automata, as any automata over the same subset of the alphabet can be unioned. We refer to these three automata as R_1, R_2, and R_3 with respective alphabets $\Sigma_1 = \{a,b\}$, $\Sigma_2 = \{b,c\}$, $\Sigma_3 = \{a,c\}$.
It is clear from the definition of a decomposition that:

$$\mathscr{L}(E_\Sigma(R_1))\bigcap\mathscr{L}(E_\Sigma(R_2))\bigcap\mathscr{L}(E_\Sigma(R_3)) = \mathscr{L}(R)$$

It then follows that:

$$\mathscr{L}(R) \subseteq \mathscr{L}(E_\Sigma(R_1))$$
$$\mathscr{L}(R) \subseteq \mathscr{L}(E_\Sigma(R_2))$$
$$\mathscr{L}(R) \subseteq \mathscr{L}(E_\Sigma(R_3))$$

Applying the projection operator to both sides yields:

$$\mathscr{L}(R|_{\Sigma_1}) \subseteq \mathscr{L}(R_1)$$
$$\mathscr{L}(R|_{\Sigma_2}) \subseteq \mathscr{L}(R_2)$$
$$\mathscr{L}(R|_{\Sigma_3}) \subseteq \mathscr{L}(R_3)$$

We then convert these projections into their associated regular expressions.

$$(ab^+)^* \subseteq \mathscr{L}(R_1)$$
$$(b^+c)^* \subseteq \mathscr{L}(R_2)$$
$$(ac)^* \subseteq \mathscr{L}(R_3)$$

The language of every two-letter decomposition of R must be included in the intersection of the expansion of these three languages. However, the following trace is accepted by this intersection but *not* by R.

$$a, b, b, c, b, b, a, b, b, c$$

It follows that R has no two-letter decomposition. $\square$

2.2.3 Current Approaches

Yang *et al.* describe a basic algorithm, Perracotta, for solving the specification mining problem for input automata with alphabets of size two. The algorithm creates an n by n matrix that contains the state of every possible assignment of the trace alphabet into the automaton alphabet, where n is the number of unique input symbols. Each (i, j) entry in the matrix is initially set to the starting state of the automaton A.

The algorithm then iterates through the input trace. Each time an input symbol s is seen, its index i is retrieved. It then iterates through each row (corresponding to all $(s, \_)$ assignments) and column (corresponding to all $(\_, s)$ assignments) and updates the state based on the transition function of A. After each symbol in the trace has been processed, it iterates through the possible assignments and outputs those that are in the final state. The space requirement for this algorithm is clearly $O(n^2)$, and the time complexity is $O(nL)$, where L is the length of the input trace.

Consider a hypothetical extension of this algorithm to three dimensions. In order to track all the assignments, $O(n^3)$ space is now needed. In addition, the scanning of an input symbol s now requires updating all assignments for $(s,\_,\_)$, $(\_,s,\_)$, and $(\_,\_,s)$, of which there are a total of $O(n^2)$. This raises the time complexity to $O(n^2L)$. In general, this approach uses $O(n^k)$ space and $O(n^{k-1}L)$ time, where k is the size of the alphabet of the template automaton. This is prohibitively costly for alphabets with greater than two letters (Section 2.4).

2.3 Algorithm Description

In this section, we describe our algorithm in detail. We first give the generic, high-level version (Section 2.3.1). We then provide some background on binary decision diagrams (Section 2.3.2), a graph-based representation of Boolean functions that we then use to effectively implement the algorithm (Section 2.3.3).

2.3.1 High-level Algorithm

Consider an automaton (Q,Σ,δ,q_0,F) and a trace T over the larger, disjoint trace alphabet Σ'. At a high level, our algorithm constructs a symbolic formula $\varphi : (\Sigma \rightarrow \Sigma') \rightarrow Q$ that represents the current automaton state of each assignment of trace symbols into the pattern alphabet. It then scans each letter in the input trace and symbolically applies the transition function of the automaton to the original formula.

The formula representing the initial configuration maps each possible assignment to the initial automaton state q_0.

$$\varphi_0 \equiv \left(\bigwedge_{x\in\Sigma} \left(\bigvee_{a\in\Sigma'} x = a \right) \right) \wedge s = q_0$$

We refer to this symbolic formula as a *state configuration*. In our algorithm, we only consider assignments in $\Sigma \rightarrow \Sigma'$ that are one to one; assignments of the same trace letter into two distinct letters of the automaton do not represent interesting specifications.

To apply the transition function δ on a trace letter $a \in \Sigma'$, we update the state of all affected assignments. Consider an arbitrary state configuration φ. We define a transition function, $trans(\varphi, a) = \varphi'$, that applies δ to φ on the input letter a. It is given by the following symbolic formula:

$$trans(\varphi,a) \equiv \bigvee_{(q,x)\to q' \in \delta} \Big((\varphi \land \neg(x = a \land s = q)) \quad \lor \quad ((\exists s.\, \varphi \land (x = a \land s = q)) \land s = q') \Big)$$

The first term of the disjunction, $\varphi \land \neg(x = a \land s = q)$, captures the state mapping of those assignments that are unaffected by a transition $(q, x) \to q'$. The second term, $(\exists s.\, \varphi \land (x = a \land s = q)) \land s = q'$, captures the state mapping of affected assignments. It first projects the affected assignments with an existential quantification and updates their corresponding state to q'.

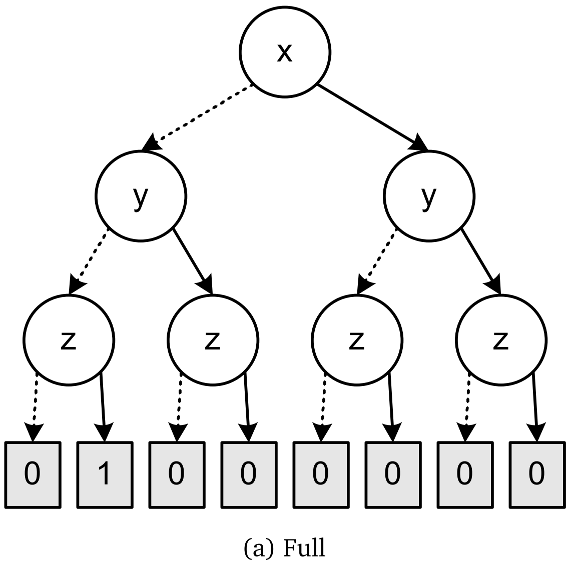
To implement this algorithm, we now need a suitable representation for the symbolic formulas. We use binary decision diagrams (BDDs) for this because they can compactly represent a large, regular state space. Before presenting our BDD-based algorithm, we first introduce some basic background on BDDs.

2.3.2 BDD Background

A Binary Decision Diagram (BDD) [Bryant, 1986] is a decision tree representation of a Boolean function over a finite set of ordered Boolean variables. Each non-terminal BDD node is associated with a Boolean variable and has exactly two outgoing edges: a **true** edge and a **false** edge. Every BDD has exactly two terminal nodes: a **zero** node and a **one** node.

Given an arbitrary truth assignment into these variables, we can determine if the Boolean function is satisfied by performing a traversal of the decision tree. Starting at the root node, we follow the **true** and **false** edges, depending on our assignment into the associated Boolean variable, until we arrive at a terminal node. If the final node is the **one** node, the function accepts the truth assignment. Similarly, if the final node is the **zero** node, the function rejects the assignment.

BDDs are used in *reduced* form, sometimes referred to as ROBDDs (reduced ordered BDDs). Nodes that represent the same variable and have the same successors can be merged, and nodes with both successor edges pointing to the same node can be removed entirely. ROBDDs are



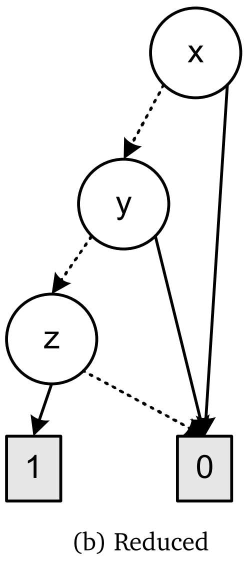


Figure 2.3: BDD example for $(x \Rightarrow y) \wedge \neg(z \Rightarrow y)$. True edges are solid; false edges are dotted.

canonical: given a specific variable ordering, there is exactly one minimal BDD for a given Boolean formula.

It is because of this that BDDs are often adept at representing extremely large sets, as long as they show a certain degree of regularity. In the programming languages and software engineering fields, BDDs have been used to create scalable pointer analyses [Berndl et al., 2003, Whaley and Lam, 2004] and represent dynamic traces [Zhang et al., 2004]. Given an efficient variable ordering, the BDD can be sufficiently compressed. However, finding an optimal variable ordering is problem dependent and NP-complete in general [Bollig and Wegener, 1996].

Efficient algorithms exist for performing basic Boolean operations on BDDs, and they run in time proportional to the number of nodes. More importantly, the BDDs are maintained in reduced form throughout the execution of the algorithms.

Consider the Boolean formula $(x \Rightarrow y) \wedge \neg(z \Rightarrow y)$. The decision tree representation of this formula (with variable ordering xyz) appears in Figure 2.3a.

Note that there are several redundant nodes: the graph only requires one terminal node of each type, and several of the z level nodes are identical and can be combined. When combined, several tests become redundant (characterized by identical **true** and **false** edges) and can be eliminated entirely. When all of these transformations have been applied, we are left with the canonical reduced form in Figure 2.3b.

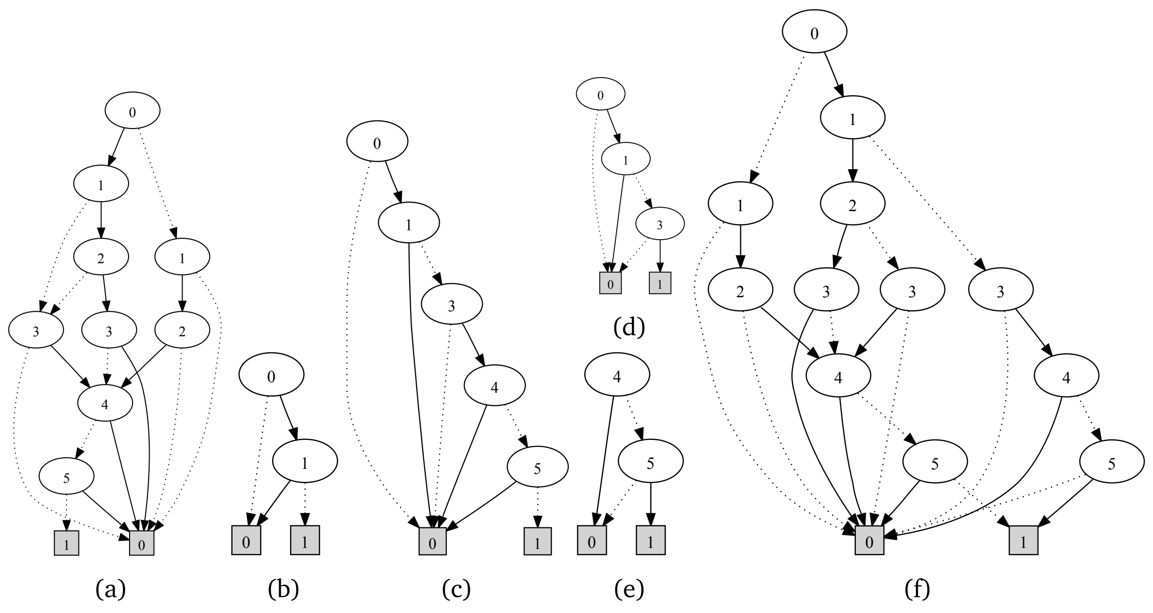


Figure 2.4: State-tracking BDD example. Nodes are labeled with their corresponding variable numbers. Solid edges are true edges and dotted edges are false edges.

2.3.3 BDD-based Algorithm

We implement the general symbolic approach described in Section 2.3.1 using BDDs. This section describes the implementation of our algorithm in terms of BDD operations.

Running Example

We first give an example to illustrate the core steps of the BDD-based algorithm. Suppose we wish to track the state of six possible assignments of the alphabet {a, b, c} into the two-letter, three-state automaton in Figure 2.1. We can encode each letter and the state in two bits each for a total of six Boolean variables. Using 01 for **a**, 10 for **b**, and 11 for **c**, Figure 2.4a depicts a BDD representing the assignments in their initial state: {(a,b,0), (b,a,0), (a,c,0), (c,a,0), (b,c,0), (c,b,0)}.

Suppose we wish to update all assignments that contain an **a** in the first position (a,_,_) to state 2, *i.e.*, we apply the transition[1] $\delta(a,*)=2$. We can perform the update on our set as a series of Boolean operations. Figure 2.4b is the BDD representing all assignments with an **a** in the first position. Note the omission of the other variables: this indicates a “don’t care” assignment. To

[1]To simplify the example, we are not considering the current state.

extract the set of affected assignments, we intersect this BDD with our state BDD, yielding the BDD in Figure 2.4c.

To remove this extracted set from our main set, we intersect the *complement* of the BDD in 2.4b with the state. The complement operation involves trading the inbound edges on the terminal **zero** and **one** nodes.

We now wish to update the state on our affected set. We first remove the current state assignment in Figure 2.4c through *existential quantification* over the state variables (Figure 2.4d). We then intersect the BDD corresponding to the new state, 2 (Figure 2.4e), and union this updated set into our main set. The final result, which represents the set {(a,b,2), (b,a,0), (a,c,2), (c,a,0), (b,c,0), (c,b,0)}, is shown in Figure 2.4f.

Basic Algorithm

Our BDD-based algorithm appears as Algorithm 2.1. The inputs to our algorithm are 1) a finite trace of symbols, 2) a total and one-to-one mapping of the trace's unique symbols to an index, and 3) an automaton pattern to mine.

On line 5, we iterate through the alphabet of the input automaton[2] and allocate enough Boolean variables to encode the unique symbols of the trace (line 6). On line 7, we build up the Cartesian product of all possible assignments.

The function `bddOf`($vars$,$values$) returns a BDD corresponding to the encoding of $values$ on variables $vars$. All other Boolean variables are considered to be in a "don't care" state. On line 10, we place every assignment in our main set in the start state of the automaton.

In our implemented version, we add one additional step. Because assignments with duplicate symbols are not interesting, we build up BDDs corresponding to the $\binom{|\Sigma|}{2}$ possibly equal variable sets, *i.e.*, the BDDs in which the two variable sets are equal. We intersect the negation of each of these with our main set.

We then iterate through the trace and repeatedly apply the transition function. Line 18 builds a mask consisting of 1) the input symbol in the appropriate alphabet position and 2) the appropriate current state. We intersect this mask with the global set to retrieve the set of

[2]For simplicity, we treat the states and alphabet of the automaton as zero-based integers.

Algorithm 2.1 Pattern-based Specification Mining

```
 1: function SpecMine($T$:Trace, $\mu$:Symbol $\to m$, $A$:$(Q,\Sigma,\delta,q_0,F)$)
 2:     $n \leftarrow |\mu|$
 3:     asgnVars $\leftarrow$ []
 4:     set $\leftarrow$ True
 5:     for all $i$ in $[0..(|\Sigma|-1])]$ do
 6:         asgnVars$[i] \leftarrow$ bVars($n$)
 7:         set $\leftarrow$ set $\bigcap$ bddOf(asgnVars$[i]$, 0..$(n-1)$)
 8:     end for
 9:     stateVars $\leftarrow$ bVars($|Q|$)
10:     set $\leftarrow$ set $\bigcap$ bddOf(stateVars, $q_0$)
11:
12:     for all symbol $t$ in $T$ do
13:         $i \leftarrow \mu(t)$
14:         old $\leftarrow \emptyset$
15:         new $\leftarrow \emptyset$
16:
17:         for all transition $\delta(q,a) = q'$ do
18:             affected $\leftarrow$ bddOf(asgnVars$[a]$, $i$) $\bigcap$ bddOf(stateVars, $q$)
19:             affected $\leftarrow$ affected $\bigcap$ set
20:             old $\leftarrow$ old $\bigcup$ affected
21:
22:             affected $\leftarrow \exists$ stateVars . affected
23:             affected $\leftarrow$ affected $\bigcap$ bddOf(stateVars, $q'$)
24:             new $\leftarrow$ new $\bigcup$ affected
25:         end for
26:
27:         set $\leftarrow$ set $\bigcap \overline{\text{old}}$
28:         set $\leftarrow$ set $\bigcup$ new
29:     end for
30:     return set
31: end function
```

all assignments that are affected by this transition. We union this set with the set “old” for later removal.

We then delete the old state through existential quantification (line 22) and intersect in the new state (line 23). We union this updated set with “new” for later addition.

After all relevant transitions have been applied, we subtract the previous assignments (line 27) from the main set and add the updated assignments (line 28).

After execution of the algorithm, the main set contains the state of every possible assignment at the end of the trace. At this point, we intersect this set with the BDD corresponding to the final state and iterate the satisfying assignments to retrieve the potential specifications.

The running state of this algorithm exhibits a high degree of regularity: the algorithm consists of shuffling the numerous assignments into comparatively few partitions defined by the states of the automaton. In practice, BDDs proved to be an appropriate choice for implementation (Section 2.4).

Practical Improvements

Algorithm 2.1 precisely and effectively solves the pattern-based specification mining problem as defined in Section 2.2.1. However, there are several practical issues that must be addressed to make the results more applicable to real systems.

Problem Size Although the use of BDDs greatly improves the tractability of this problem, some traces may still have too many unique events to consider all possible assignments. In our evaluation, the largest trace has 7000 unique events, and our precise algorithm completed its scan in a reasonable amount of time. However, it is conceivable that there exist problems that are too large to solve with this approach.

To reduce the state space, we can partition the original problem into a set of smaller problems. Even a very coarse, conservative partitioning can significantly improve execution speed, and in practice, partitioning is especially effective when using BDDs. Because we process the partitions simultaneously, the BDDs can cache and share nodes between the otherwise unrelated state sets.

Large Solution Sets and Imperfect Traces In general, mining specifications through pattern matching produces a large result set. Previous work [Engler et al., 2001, Ramanathan et al., 2007a, Yang et al., 2006] on mining *alternating* specifications has largely focused on developing efficient ranking and selection mechanisms. Weimer and Necula [2005] focus their search on exceptional control flow paths, and they experiment with several ranking statistics. Yang et al. [2006] develop a suite of heuristic filters that greatly reduce and refine the solution set.

Yang *et al.* also tailor their analysis to handle potentially *imperfect* traces. Imperfection in traces can originate in flaws in the trace generation mechanism, the lack of context information like threading or object identity, or simply from bugs. To address this, their tool ranks candidate specifications based on the ratio of the number of times a specific assignment passes the final state to the number of times it passes the error state (a failed assignment is reset to the start state on error). This allows the discovery of frequently occurring specifications from potentially buggy or otherwise imperfect traces.

Unfortunately, we cannot directly apply this technique to our analysis. Storing a raw satisfaction count or floating point ratio in the BDD is prohibitively expensive: the set quickly becomes heterogeneous and the memory usage and computation time escalate, causing the computation to diverge.

We address both of these issues with extensions to our basic algorithm. Algorithm 2.2 differs from Algorithm 2.1 in that it allows the input traces to contain a certain amount of imprecision and it only returns specifications that occur frequently.

We add an additional set of Boolean variables to our global state. These variables encode a *satisfaction count* for each assignment that can rise and fall as the trace is processed. The declaration of these variables occurs on line 11 of the modified algorithm.

The count is initialized to a specified value that is greater than zero (line 14). If it reaches zero, the specification is dropped (line 35). If it reaches our threshold, we declare it to be satisfied and set it aside (line 24). We no longer execute the transition function on specifications that have either fully failed or have been fully accepted.

Each time an assignment enters an error state (line 33), we decrement this count. Each time it enters a final state, we increment the count nondeterministically with a configurable probability. In both cases, we reset the assignment's state to the start state of the automaton so that execution can continue.

We found that using a threshold of 7 (3 bits), a starting point of 2, and a probability of 0.5 worked well. The nondeterminism at line 29 effectively "amplifies" the count: with a probability of 0.5, we can expect that an accepted specification reaches a final state twice as many times as it would have if the probability was set to 1.

More importantly, we are able to enforce a stricter requirement–a larger satisfaction threshold–without adding additional Boolean variables. We experimented with deterministic thresholds of 16, 32, and 64, and found that even the first extra bit affected the scaling of this approach significantly. The nondeterminism allows us to locate more frequently occurring specifications without incurring this overhead.

Algorithm 2.2 Approximate Pattern-based Specification Mining

```
 1: function SpecMine($T$:Trace, $\mu$:Symbol $\rightarrow m$, $A$:$(Q,\Sigma,\delta,q_0,F)$)
 2:     $n \leftarrow |\mu|$
 3:     asgnVars $\leftarrow$ []
 4:     set $\leftarrow$ True
 5:     for all $i$ in $[0..(|\Sigma|-1])]$ do
 6:         asgnVars[$i$] $\leftarrow$ bVars($n$)
 7:         set $\leftarrow$ set $\bigcap$ bddOf(asgnVars[$i$], $0..(n-1)$)
 8:     end for
 9:     stateVars $\leftarrow$ bVars($|Q|$)
10:     set $\leftarrow$ set $\bigcap$ bddOf(stateVars, $q_0$)
11:     countVars $\leftarrow$ bVars(THRESHOLD+1)
12:     NOTFAILED $\leftarrow$ bddOf(countVars, 1..THRESHOLD)
13:     NOTSATISFIED $\leftarrow$ bddOf(countVars, 0..(THRESHOLD $-$ 1))
14:     set $\leftarrow$ set $\bigcap$ bddOf(countVars, STARTVAL)
15:
16:     for all symbol $t$ in $T$ do
17:         $i \leftarrow \mu(t)$
18:         old $\leftarrow \emptyset$
19:         new $\leftarrow \emptyset$
20:
21:         for all transition $\delta(q,a) = q'$ do
22:             affected $\leftarrow$ bddOf(asgnVars[$a$], $i$) $\bigcap$ bddOf(stateVars, $q$)
23:             affected $\leftarrow$ affected $\bigcap$ set
24:             affected $\leftarrow$ affected $\bigcap$ NOTSATISFIED
25:             old $\leftarrow$ old $\bigcup$ affected
26:
27:             affected $\leftarrow \exists$ stateVars . affected
28:             if $q' \in F$ then
29:                 if nondet ( ADVPROB ) then
30:                     increment( affected, countVars )
31:                 end if
32:                 affected $\leftarrow$ affected $\bigcap$ bddOf(stateVars, $q_0$)
33:             else if isError($q'$) then
34:                 decrement( affected, countVars )
35:                 affected $\leftarrow$ affected $\bigcap$ NOTFAILED
36:                 affected $\leftarrow$ affected $\bigcap$ bddOf(stateVars, $q_0$)
37:             else
38:                 affected $\leftarrow$ affected $\bigcap$ bddOf(stateVars, $q'$)
39:             end if
40:             new $\leftarrow$ new $\bigcup$ affected
41:         end for
42:
43:         set $\leftarrow$ set $\bigcap \overline{\text{old}}$
44:         set $\leftarrow$ set $\bigcup$ new
45:     end for
46:     return set $\bigcap$ bddOf(countVars, THRESHOLD)
47: end function
```

Note that we have abstracted away the increment and decrement operators on lines 30 and 34, respectively. Performing arithmetic operations on encoded values in a BDD is non-trivial. To increment the count, we actually need a second set of Boolean variables as temporary storage. We build an "adder" BDD across the two sets that consists of the relation $(n, n+1)$ ($(n, n-1)$ for subtraction) for all n in $0..\texttt{THRESHOLD}-1$. Adding then becomes a sequence of Boolean operations: we intersect the value to add with the "adder" BDD and remove the original value by existential quantification. We then substitute the new value (now in the temporary values) for the original with the BDD `replace` operation.

In practice, this modification scales well. We use this variant of the technique to locate interesting specifications and find bugs.

2.4 Implementation and Results

We implemented both versions of our algorithm as part of a practical specification mining tool. This section describes our implementation and presents the results of our empirical evaluation.

2.4.1 Implementation and Experimental Setup

Our specification miner is a Java application that uses the JavaBDD[3] library to provide BDD functionality. The JavaBDD library is a common Java-based abstraction layer for several common BDD packages. We chose to configure the library to use the well-tested BuDDy library.[4]

We implemented the precise symbolic algorithm (Algorithm 2.1), the approximate symbolic algorithm (Algorithm 2.2), and the explicit state tracking algorithm (Section 2.2.3) for comparison. As described in Section 2.3.3, we also implemented variants that partition the data for scalability.

Our tool is wrapped in a front end that prescans the trace and identifies the unique symbols. It then uses a hash function to build a mapping on to the natural numbers. For convenience, this front end tracks the average execution time of a single element of the trace and provides a progress meter and regular estimates of the time to completion.

Other BDD-based algorithms [Berndl et al., 2003, Whaley and Lam, 2004] can show great variability in performance depending on the ordering of the Boolean variables within the BDDs.

[3]`http://javabdd.sourceforge.net`
[4]`http://www.itu.dk/research/buddy`

Compared to the sets represented in these algorithms, our sets are far more saturated and homogeneous: we build the Cartesian product of every possible value. We found that a simple interleaving of all assignment variables yielded the best general performance.

If the shape of the main set changes drastically at some point in the execution of the algorithm, we allow BuDDy's dynamic variable reordering to execute and shuffle the assignment variables using a built in sliding window algorithm. This reordering is triggered with a pathological operation causes a dramatic increase in the number of BDD nodes. In practice, we find that this occurs relatively infrequently.

We collected traces for two large Java projects, Hibernate[5] and Apache Ant[6]. Each trace consists of a sequence of logged method invocations with their fully qualified class names. We built a trace collection tool using Java bytecode instrumentation provided by the JRat[7] toolkit. Our tool takes as input a Jar (Java archive) or directory and recursively instruments all method calls with hooks into our logging code. We also slipstream our logging classes into the instrumented Jar files to avoid the task of changing the projects' runtime profiles.

Our traces are generated from both normal usage of each project and extensive testing using each project's respective system test suite.

2.4.2 Results for the Precise Algorithm

We first tested the scalability of our precise algorithm on four small to medium sized traces. For comparison, we also tested the basic algorithm (Section 2.2.3) that maintains the state of every assignment explicitly in a matrix. Figure 2.5a contains the results of mining the alternating pattern found in Figure 2.1. On these smaller traces, the basic algorithm outperforms our symbolic algorithm. However, the gap narrows as the number of unique events increase.

Figure 2.5b shows the results of mining the 3-letter resource usage pattern found in Figure 2.2. When the extra dimension is added, the symbolic algorithm is orders of magnitude more scalable. The basic algorithm quickly diverges in terms of memory usage and is three orders of magnitude slower.

[5]An object-relational mapping tool.

[6]A `make`-like build tool for Java.

[7]`http://jrat.sourceforge.net`

Trace		**Basic Algorithm**		**Symbolic**	
Uniq. Evts.	Length	Time	Max Mem.	Time	Max Mem.
338	24,470	0.17s	18 MB	0.84s	52 MB
591	94,789	0.79s	29 MB	2.7s	52 MB
664	227,595	3.0s	30 MB	5.6s	64 MB
981	1,435,613	18.4s	37 MB	28.8s	66 MB

(a) The alternating ($|\Sigma| = 2$) pattern.

Trace		**Basic Algorithm**		**Symbolic**	
Uniq. Evts.	Length	Time	Max Mem.	Time	Max Mem.
338	24,470	3m 31s	196 MB	10.4s	54 MB
591	94,789	42m 3s	942 MB	34.5s	54 MB
664	227,595	2h 8m	1.2 GB	68.4s	64 MB
981	1,435,613	27h 59m	4.0 GB	2m 53s	66 MB

(b) The resource usage ($|\Sigma| = 3$) pattern.

Figure 2.5: Results of mining patterns using the precise algorithm on four small traces.

Trace		**Basic Algorithm**		**Symbolic**	
Uniq. Evts.	Length	Time	Max Mem.	Time	Max Mem.
3821	21,055,090	22m	91 MB	7m 30s	68 MB
7000	49,779,538	3h 19m	238 MB	23m	119 MB

(a) The alternating pattern.

Trace		**Basic Algorithm**		**Symbolic**	
Uniq. Evts.	Length	Time	Max Mem.	Time	Max Mem.
3821	21,055,090	–	–	5h 17m	302 MB
7000	49,779,538	–	–	13h 34m	589 MB

(b) The resource usage pattern.

Figure 2.6: Execution of the precise algorithm on large traces.

We then tested the algorithms on our two largest traces. Figure 2.6 contains the results of these tests. When mining the 2-letter pattern, the symbolic algorithm exploits the regularity of the solution set and both executes faster and in less space than the basic algorithm.

We were naturally unable to mine the 3-letter pattern using the basic algorithm, as this would involve individually tracking the states of over 50 billion and 300 billion potential assignments for the two traces, respectively. The symbolic algorithm was able to complete this computation in a reasonable amount of time and with low memory usage.

Overall, the symbolic approach greatly extends the tractability of this problem.

Trace Data			
Source	Unique Evts.	Partitions	Max Partition Size
Ant	3821	32	1164 Events
Hibernate	7000	68	703 Events

(a) Partitioning.

	Exact		
Trace	Time	Max Mem.	Count
Ant	21m	90 MB	1,383,532
Hibernate	41m	117 MB	55,374

(b) Exact, resource usage pattern.

Approximate		
Time	Max Mem.	Count
27m	122 MB	7,724
48m	146 MB	13,608

(c) Approx, resource usage pattern.

Figure 2.7: Results of running the precise and approximate versions on partitioned versions of the larger traces.

2.4.3 Results for the Approximate Algorithm

Our approximate algorithm yields more practically relevant solution sets. In this section, we evaluate its performance and illustrate previously unknown bugs that we located using specifications derived from this analysis.

We used the same two large traces, which were generated from executions of Apache Ant and Hibernate, respectively. We evaluated both the precise and practical algorithms using a package-level partitioning: we only considered assignments in which the involved methods were defined in classes of the same package. Figure 2.7a describes the partitioning of the traces. Code is seldom uniformly distributed into packages, so we have included the maximum partition size as well. This value is closely related to scalability: the number of potential assignments is cubic in the size of any given partition.

Performance

Figures 2.7b and 2.7c contain the results of executing our symbolic algorithms on these traces using the package-level partitioning. Both algorithms execute in similar amounts of space and time.

Executing the exact algorithm on the Apache Ant trace, however, produces an enormous amount of potential specifications. This is likely because Ant's overall build process emulates the resource usage pattern at a higher level: builds are characterized by several sub-builds surrounded by a start and end. This causes the recognition of virtually every build process method as a "usage" method between the build start and end methods.

The use of the approximate algorithm substantially prunes this result set. Although we potentially admit more specifications by allowing failed specifications additional chances to accepted, we restrict the set by admitting only frequently occurring specifications.

Result Quality

For the Hibernate project, we inferred several properties of the `Session` object of the form:

```
a: SessionFactoryImpl.openSession(Connection,...)
b: AbstractSessionImpl.createQuery(String)
c: AbstractSessionImpl.setClosed()
```

where the a) and c) assignments involved all combinations of the various signatures of openSession and the various close/setClosed methods, and the b) assignment consisted of frequently executed methods of the Session class. These specifications are documented and have been targets for previous automated specification tools [Weimer and Necula, 2005].

Most spurious specifications were either redundant or artifacts of control flow: we use flat traces, so the execution of one method within the body of another appears as subsequent invocations. To mitigate this issue, we can use a filtering step, like that of Yang et al. [2006], that prunes assignments with a direct control flow relationship.

The Ant build system interacts with many types of resources that fit our pattern. These include various archive formats, flat files, and resources defined by URLs. We inferred several properties of the form:

```
a: ZipResource.getInputStream()
b: InflaterInputStream.read()
c: InflaterInputStream.close()
```

We inspected our results and the source code, and we discovered that each of these resources inherits from a common class, `Resource`. Analyzing the usages of this class and specifically searching for resource allocation bugs, we discovered the flaw in Figure 2.8.

```
 1 public static void copyResource(Resource source, Resource dest,
 2     String inputEncoding, String outputEncoding,
 3     Project project) {
 4   BufferedReader in = null;
 5   try {
 6     InputStreamReader isr = null;
 7     if (inputEncoding == null) {
 8       isr = new InputStreamReader(source.getInputStream());
 9     } else {
10       isr = new InputStreamReader(source.getInputStream(), inputEncoding);
11     }
12     in = new BufferedReader(isr);
13     // getToken() indirectly calls the read() method
14     String line = lineTokenizer.getToken(in);
15     while (line != null) {
16       line = lineTokenizer.getToken(in);
17     }
18   }
19   finally {
20     FileUtils.close(in); // null-safe close()
21   }
22 }
```

Figure 2.8: A previously unknown bug in Apache Ant 1.7.0. Only relevant lines are shown.

The parameter **inputEncoding** is an unsantitized user input string. Note that the allocation and release of the resource are properly enclosed in a try/finally block. However, line 10 contains a problem: the creation of an **InputStreamReader** with a user-supplied character set string can fail, while the parameterless creation of Readers on lines 8 and 12 cannot. If the allocation on line 10 does fail, the call to **getInputStream** will be eagerly evaluated and the resource will be allocated. The variable **in** will never be assigned, and the null-safe **close** call on line 20 will not close the resource, causing a leak.

Both of the listed patterns are alike in the sense that all participants are not necessarily of the same type. This is a key advantage of the pattern-based approach in general: the scope of the analysis is not limited to a small set of suspected methods that must be decided by a user.

An interesting example that illustrates this advantage occurs in the Ant project:

```
a: MailMessage(String,int)
b: MailPrintStream.write(byte[],int,int)
c: MailMessage.sendAndClose()
```

A programmer utilizing the public interface of this class might be confused[8] by the method naming and assume that all socket connections are performed in the final **sendAndClose** method.

The frequency of the discovered pattern suggests that the methods of this class actually follow an allocation pattern. On inspection of the source code, we noted that the first constructor call does in fact allocate and connect a socket, the second **MailPrintStream** object uses it, and the final **sendAndClose** closes it.

Figure 2.9 illustrates a usage of this class in the Ant source code. The proper allocation and use of the **MailMessage** resource should be enclosed in a try/finally block, but all exceptions are thrown to the caller. This could potentially cause resource leaks.

While evaluating the solution sets produced by our algorithms, we noted several instances where transitive relationships might be used to chain smaller specifications into larger ones. As discussed in Section 2.2.2, previous work has shown this to be useful for alternating properties. Our work enables other new fundamental building blocks, including three-letter patterns with loops. We leave for future work a study of these constructed properties.

2.5 Related Work

The literature on specification mining is varied, but many tools and techniques share similar characterizations of properties: finite automata that describe correct behavior. Ammons et al. [2002] develop a specification miner, Strauss, that mines specification by learning a probabilistic finite state automaton. Unlike our approach, Strauss requires the input alphabet of the automaton to be specified, but it does have the potential to find more complex specifications.

Whaley et al. [2002] present an alternative method: the user specifies the input alphabet of the automaton (a Java API), and the algorithm identifies and prunes illegal transitions based

[8] For fairness, we point out that this proper usage pattern is documented in the code comments. It does not mention the resource allocation issues, though.

```
private void sendMail(String mailhost, int port, String from, String
    replyToList, String toList, String subject, String message) throws
    IOException {
  MailMessage mailMessage = new MailMessage(mailhost, port);
  mailMessage.setHeader("Date", DateUtils.getDateForHeader());

  mailMessage.from(from);
  if (!replyToList.equals("")) {
    StringTokenizer t = new StringTokenizer(replyToList, ",␣", false);
    while (t.hasMoreTokens()) {
      mailMessage.replyto(t.nextToken());
    }
  }
  StringTokenizer t = new StringTokenizer(toList, ",␣", false);
  while (t.hasMoreTokens()) {
    mailMessage.to(t.nextToken());
  }

  mailMessage.setSubject(subject);

  PrintStream ps = mailMessage.getPrintStream();
  ps.println(message);

  mailMessage.sendAndClose();
}
```

Figure 2.9: Another previously unknown bug in Apache Ant 1.7.0.

on the existence state-checking code that throws exceptions on errors. This technique relies on defensively-programmed components that imply correct usage through error checking.

More recently, Shoham et al. [2007a] present a technique based on abstract interpretation that is capable of learning arbitrarily complex specifications. The abstract value is the automaton itself, and it is constructed as the program is interpreted. This technique is accurate, but like the previously mentioned tools, the alphabet of the automaton must be known before the analysis begins.

Ramanathan et al. [2007a,b] create a static analysis that uses predicate mining to infer *function precedence protocols*. These rules take take the form of "when b is called, a must have been called at least once." When viewed as a specification pattern, these rules take the form of the regular expression (a^+b). Our work is capable of locating more complex properties. In addition, as we show in Section 2.2.2, it is generally impossible to assemble two letter properties like this to form our more complex properties.

Kremenek et al. [2006] use *annotation factor graphs* to probabilistically assign annotations to functions and form specifications. This technique allows the user to incorporate other domain-specific information into the analysis, like a belief in the overall ratio of allocators to deallocators, as components in the factor graph. The user must still specify the set of annotations that can be inferred, which is similar to the specification of the potential alphabet of an automaton in the sense that it limits the results of the analysis.

One tool that does not follow this pattern of deriving automata-described temporal properties is PR-Miner [Li and Zhou, 2005]. This tool, presented by Li and Zhou, mines *association* properties from programs by correlating related function calls using *frequent itemset mining*. This work, like ours, is highly scalable. However unlike our tool, PR-Miner requires no input from the user other than the program (our technique requires a specification template). However, these association properties are not as strict as temporal properties: the members of the properties are only related by frequent association and not by the order or frequency of invocation.

There has been recent work on dynamic analysis as well. Ernst et al. [2000] have developed Daikon (later refined for object oriented systems [Csallner and Smaragdakis, 2006]) and Hangal and Lam [2002] have developed DIDUCE. These tools locate program invariants by monitoring the runtime state of a program and attempting to match invariant templates to expressions. These tools are able to infer simple properties, like $a \neq 0$, but at present do not infer temporal properties. When viewed as an online algorithm, our work can be thought of as a Daikon-like tool for temporal properties: we begin by setting up a large set of potentially valid properties and prune them as execution continues.

2.6 Conclusion

In this chapter, we have precisely defined the pattern based specification mining problem and shown that its general form is NP-complete. We have provided a novel symbolic algorithm that greatly expands the computational tractability of this problem. Our algorithm exploits the regularity of the running state and solution set of current approaches to form a compact, efficient symbolic approach. We have implemented our algorithm as a practical tool using binary decision diagrams and have used it to find meaningful specifications in real systems. These specifications led us to discover previously unknown bugs in large, real-world systems. For future work, we plan to investigate the synthesis of larger specifications from our inferred properties, and we are also interested in adapting our approach to locate other frequently-occurring specification patterns.

3 Mining Richer Specifications Through Composition

Program specifications are important for many tasks during software design, development, and maintenance. Among these, temporal specifications are particularly useful. They express formal correctness requirements of an application's ordering of specific actions and events during execution, such as the strict alternation of acquisition and release of locks. Despite their importance, temporal specifications are often missing, incomplete, or described only informally. Many techniques have been proposed that *mine* such specifications from execution traces or program source code. However, existing techniques mine only simple patterns, or they mine a single complex pattern that is restricted to a particular set of manually selected events. There is no practical, automatic technique that can mine general temporal properties from execution traces.

In this chapter, we present Javert, the first general specification mining framework that can learn, fully automatically, complex temporal properties from execution traces. The key insight behind Javert is that real, complex specifications can be formed by composing instances of small generic patterns, such as the alternating pattern ($(ab)^*$) and the resource usage pattern ($(ab^*c)^*$). In particular, Javert learns simple generic patterns and composes them using *sound* rules to construct large, complex specifications. We have implemented the algorithm in a practical tool and conducted an extensive empirical evaluation on several open source software projects. Our results are promising; they show that Javert is scalable, general, and precise. It discovered many interesting, non-trivial specifications in real-world code that are beyond the reach of existing automatic techniques.

3.1 Introduction

Temporal specifications of software systems describe requirements on the ordering of specific actions or events. These specifications are often used to formally specify legal function call sequences over module APIs. Temporal API specifications are useful for a number of reasons: they can shorten development time by guiding the production of correct code; they can be used as input to static analysis tools [Ball and Rajamani, 2001, Das et al., 2002, Flanagan et al., 2002, Xie and Aiken, 2005] to find bugs automatically; and they can facilitate software maintenance tasks by aiding program comprehension.

Despite these desirable characteristics, precise temporal specifications are often missing, incomplete, or only informally stated. Recognizing this problem, researchers have developed techniques that allow the automated reverse engineering—mining—of temporal specifications from programs. Recent work has recognized that API usage patterns can be specified as regular languages [Ammons et al., 2002]. This allows the compact representation of specifications as regular expressions or finite state automata, and it allows the characterization of the specification mining problem as a language learning problem.

Current approaches are fundamentally similar: each takes as input a static program or a dynamic trace or profile and produces one or more compact regular languages that specify temporal properties. However, the individual solutions differ in key ways.

Some techniques learn a single specification over a specific alphabet [Ammons et al., 2002, Shoham et al., 2007a, Whaley et al., 2002]. For example, one might be aware that some relationship occurs between the elements of a programming language's relational database query API. The specification miner would take as input a program and the elements of this API and return a minimal finite automaton that represents the probable set of correct usages. One particular advantage of these approaches is the ability to learn arbitrarily complex patterns; the miner has no prior knowledge of the structure of the specification.

Unfortunately, these techniques suffer from scaling and precision problems. Finding a minimal finite automaton for a set of input strings is NP-hard and cannot be approximated [Pitt and Warmuth, 1989], and precision suffers from the inability to learn from negative examples: a program is assumed to include entirely or mostly correct usages. Learned specifications must strike a careful balance between levels of generality. If a specification is too general, it can capture dangerous behavior. If it is too restrictive, it merely encodes a particular usage instance, not a prescriptive specification. In addition, the requirement of specifying the alphabet *a priori* is limiting: it prevents the discovery of latent relationships between components that the user does not anticipate.

Other algorithms learn multiple specifications over an arbitrary alphabet [Engler et al., 2001, Gabel and Su, 2008a, Weimer and Necula, 2005, Yang et al., 2006]. For example, many of these miners are capable of enumerating all pairs of events in a system that consistently alternate, like the opening and closing of a file descriptor. These techniques are scalable, and the user is not forced to select a small subset of the program's events to consider: all events in the system are considered simultaneously.

However, the structure of the specifications must be defined in advance as templates, which makes learning an arbitrarily complex specification impossible. The specifications must also be restricted in both alphabet size and number of states to maintain scalability: pattern matching of a specification is NP-hard in general [Gabel and Su, 2008a]. Current miners can locate instances of alternating patterns over event pairs [Engler et al., 2001, Weimer and Necula, 2005, Yang et al., 2006], resource usage patterns over event triples [Gabel and Su, 2008a], and precedence protocols [Ramanathan et al., 2007a,b] and partial orders [Acharya et al., 2007] over pairs of function calls. These approaches suffer from precision issues as well: although one can soundly enumerate instances of these small template patterns, it is often difficult to distinguish between true and coincidental relationships in the voluminous result sets.

In this chapter, we present a new general approach to temporal specification mining that addresses several of the limitations of current techniques. Our insight is twofold. First, we recognize that instances of smaller specification template patterns can be composed into larger specifications of arbitrary size. Second, we observe that the composition of a specific set of pattern templates sufficiently captures most temporal specifications published in the literature.

We then leverage this insight to create the first scalable temporal specification mining algorithm that mines specifications of arbitrary size over arbitrary alphabets. Unlike all previous approaches, our algorithm requires no input beyond the program representation: neither a pattern nor an alphabet must be specified. In short, we use our first insight to provide a general technique that increases the scope and power of pattern-based specification mining. We then create an instance of this general technique that leverages general domain knowledge of software to mine general temporal properties. We have implemented our algorithm as a practical tool and have demonstrated it to be general, scalable, and accurate.

Specifically, this chapter makes the following contributions:

1. We introduce a new general technique for mining temporal specifications. Our technique combines the generality of language learning-based approaches with the scalability of pattern matching-based approaches by assembling instances of smaller patterns into arbitrarily large specifications using sound inference rules.

2. We provide an instance of this general technique, consisting of specific sets of patterns and rules, and demonstrate that its domain, a restricted class of regular languages, captures most temporal specification instances in the literature. This instance thus defines an algorithm for mining general temporal properties that requires no input beyond the program representation.

3. We implement this algorithm in a practical tool, Javert[1] and perform an empirical evaluation on several open source software projects. Our evaluation demonstrates that our technique is scalable, general, and precise.

The following section (Section 3.2) illustrates our high-level technique through a motivating example. Section 3.3 formalizes our general technique and describes our specific property mining algorithm. It then argues that our algorithm is capable of finding a large body of temporal properties. Section 3.4 provides details about Javert's implementation. It then describes our empirical evaluation and results. Finally, Sections 3.5 and 3.6 survey and compare related work and conclude.

3.2 Motivating Example

In this section, we provide a motivating example and use it to describe the intuition behind our technique. This example was discovered by our practical tool, Javert, during our experiments.

[1] *Fr.*, Pronounced Jah·ver′, the relentless and obsessive inspector from Victor Hugo's *Les Misérables.*

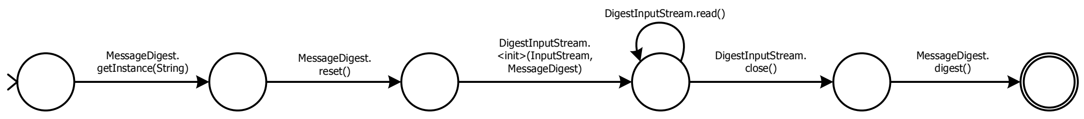


Figure 3.1: Usage specification for Java's MessageDigest class.

Consider the specification automaton in Figure 3.1. It describes the correct usage of Java's MessageDigest API, which is used to generate digests (*e.g.* MD5) of binary data. Assume that we have one or more full program method traces that contain several instances of this pattern. As in a typical program trace, the individual method calls that form this pattern may be interleaved with several unrelated calls, and we do not know *a priori* that a pattern necessarily holds over these method calls.

This pattern would be difficult to learn using a general language learning miner: as we do not know the alphabet of the specification, we would have to consider many projections over different subsets of interesting events of the trace or attempt to discover an interesting projection by tracing the flow of data (*e.g.* "scenario extraction" in [Ammons et al., 2002]). The pattern is also difficult to learn using a pattern matching approach: because this pattern has an alphabet of size six, there are $O(n^6)$ potential specifications in a trace with n unique events—a potentially intractable number to consider for even modestly diverse traces.

Despite this inability to mine the pattern directly, we can still mine the trace for instances of smaller patterns. Current pattern-based miners [Engler et al., 2001, Weimer and Necula, 2005, Yang et al., 2006] are capable of locating all instances of alternating events; that is, ones that fall into the regular pattern $(ab)^*$. Recent advances [Gabel and Su, 2008a] have leveraged the use of symbolic techniques to allow the mining of larger patterns, including ones with looping transitions (*e.g.* $(ab^*c)^*$). Figure 3.2 contains a subset of these smaller patterns, which we call *micropatterns*, that hold if this pattern exists.

Notice that several patterns appear to have intuitive transitive relationships. For example, we know that all calls to `MessageDigest.reset` and `DigestInputStream.<init>` strictly alternate, and we also know that there exists an alternating relationship between `Message-Digest.reset` and `DigestInputStream.close`. From this, we can deduce that the first three

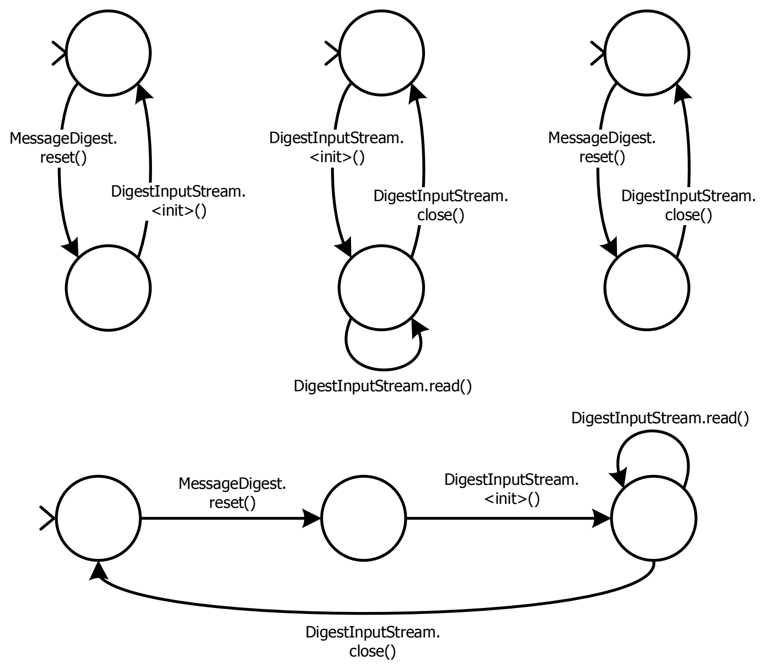


Figure 3.2: Three micropatterns related to the MessageDigest API and the larger pattern yielded by composition.

micropatterns in Figure 3.2 imply the existence of the fourth. In the following sections, we formalize this notion.

This composition of patterns is the essence of our approach. In general, if we can infer enough information from a given set of micropatterns, we can use them as building blocks for larger temporal properties. This allows us to leverage the advantages of pattern-based approaches—namely the ability to scalably enumerate all micropatterns for all possible combinations of trace events—while still maintaining a form of generality. Figure 3.3 depicts our high level approach. We use a pattern-based specification miner to mine an interesting set of templates. We then take the discovered patterns and compose them into larger specifications. Finally, we optionally perform filtering or ranking on the composed specifications. Note that the user provides no templates or alphabet sets: we consider all possible combinations of trace events for micropattern mining, and we compose arbitrarily large patterns without higher level templates.

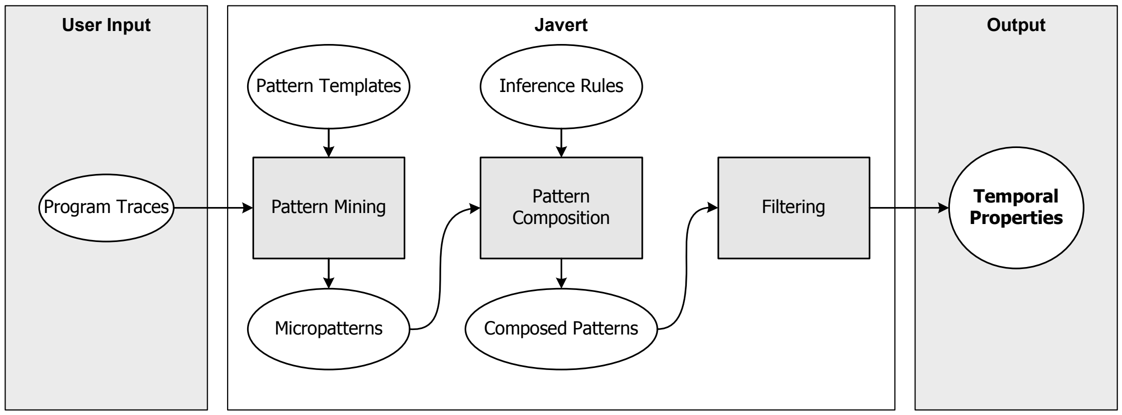


Figure 3.3: High-level architecture of Javert.

3.3 Technical Approach

In this section, we discuss the realization of our technique. In Section 3.3.1, we formalize the idea of pattern composition. In Section 3.3.2, we present the specific patterns and composition rules used in Javert. Section 3.3.3 argues that these rules and patterns are sufficient to locate a large number of real specifications in software systems.

3.3.1 General Framework

In Section 3.2, we introduced the intuitive idea of pattern composition. We now present formal definitions to more clearly illustrate this idea.

Definition 6 (Projection)**.** The *projection* π of a string s over an alphabet Σ, $\pi_\Sigma(s)$, is defined as s with all letters not in Σ deleted. The projection of a language L over Σ is defined as $\pi_\Sigma(L) = \{\pi_\Sigma(s) | s \in L\}$.

Definition 7 (Specification Pattern)**.** A specification pattern is a finite state automaton $A = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is a set of input symbols, $\delta : Q \times \Sigma \mapsto Q$ is the transition function, q_0 is the single starting state, and F is a set of final states. A pattern is *satisfied* over a trace T with alphabet $\Sigma' \supseteq \Sigma$ if $\pi_\Sigma(T) \in \mathscr{L}(A)$.

A pattern-based specification miner takes as input one or more traces and one or more templates of specification patterns. The alphabets of these templates contain abstract symbols in place of concrete trace characters. The miner produces as output a set of *satisfied instances* of the templates; that is, it produces a set of concrete specification patterns with alphabets over subsets of the trace alphabet.

Suppose two pattern instances, A_1 and A_2, are satisfied over a trace T. A_1 and A_2 may describe different elements of the system. For example, A_1 might describe an alternating property over `tryLock` and `lock` methods, while A_2 might describe the same alternating property over `lock` and `unlock`. To compose these patterns into a single pattern over the same alphabet, we must recognize that these patterns hold over *projections* of T on to their respective alphabets, and thus any interleaving of other trace letters may occur between state transitions. To account for this, we define the expansion operator, E, which widens a regular language with respect to a larger alphabet.

Definition 8 (Expansion). [Gabel and Su [2008a, § 2.6]] Assume a regular language defined by a finite state automaton $A = (Q, \Sigma, \delta, q_0, F)$. The *expansion* of $\mathscr{L}(A)$ over an arbitrary alphabet Σ', written $E_{\Sigma'}(\mathscr{L}(A))$, is the *maximal* language over $\Sigma \cup \Sigma'$ whose projection over Σ is $\mathscr{L}(A)$.

An automaton accepting $E_{\Sigma'}(\mathscr{L}(A))$ can be constructed by first duplicating A and then adding a looping transition $\delta(q, a) = q$ to each state q for each letter $a \in \Sigma' \setminus \Sigma$. For the remainder of this chapter, we will overload E to denote this construction when applied to an automaton rather than a language.

Expansion can be thought of as the maximal inverse of projection. For example, an expression corresponding to $E_{\{a,b,c\}}((ab)^*)$ is $c^*(ac^*bc^*)^*$. Note that projecting this new language over $\{a, b\}$ yields the original language, $(ab)^*$.

The composition of two patterns is defined as follows:

Definition 9 (Composition). The *composition* of two specification patterns A_1 and A_2 is the intersection of the expansion of each pattern over their combined alphabets, *i.e.*,

$$E_{\Sigma_2}(A_1) \cap E_{\Sigma_1}(A_2)$$

Intuitively, the composition of two patterns defines a language of traces in which both patterns hold.

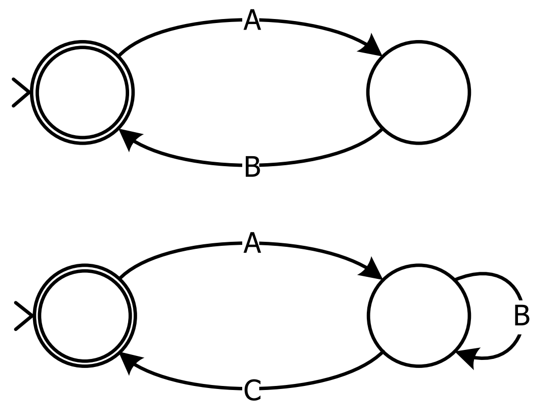


Figure 3.4: Micropatterns mined by Javert.

We could use this general definition to arbitrarily compose patterns by using standard algorithms for finite state automaton manipulation. However, in general, performing these pairwise compositions directly is undesirable. Given a reasonably large set of patterns, the finite state expansion, intersection, and minimization operations become more expensive as the automata grow. More importantly, we are interested in compact, concise specifications, and performing arbitrary language intersections is not likely to maintain a solution set with those characteristics.

To address this, we recognize special cases of composition in which the result of the composition is compact and intuitive. We then formulate these cases as inference rules, which leads to straightforward implementations in which composition is a constant time operation.

3.3.2 Javert

This section describes the specific micropatterns mined by Javert and the inference rules used to compose them.

Javert mines two micropatterns: basic *alternation* and *resource ownership*. These patterns correspond to the regular expressions $(ab)^*$ and $(ab^*c)^*$, respectively, and their representations as finite automata appear in Figure 3.4.

Branching Rule: The first rule describes the composition of two patterns with identical "end-points," *i.e.*, the first and last letters of a single iteration of the pattern.

$$\frac{(a\mathscr{L}_1{}^*b)^* \qquad (a\mathscr{L}_2{}^*b)^*}{(a\left(\mathscr{L}_1|\mathscr{L}_2\right)^*b)^*}\ [\textsc{Branch}]$$

The preceding rule holds if $\mathscr{L}_1$ and $\mathscr{L}_2$ have disjoint alphabets. Note that either $\mathscr{L}_1$ or $\mathscr{L}_2$ may represent the empty language.

Proposition 1 (Correctness of Branching). *Defining* Σ' *as* $\{a,b\}\cup\Sigma_{\mathscr{L}_1}\cup\Sigma_{\mathscr{L}_2}$*, the correctness of the* Branching Rule *follows from the following fact:*

$$E_{\Sigma'}(a\mathscr{L}_1{}^*b)^*\cap E_{\Sigma'}(a\mathscr{L}_2{}^*b)^* = (a\left(\mathscr{L}_1|\mathscr{L}_2\right)^*b)^*$$

This rule performs the composition of two patterns that describe legal operations at the same logical state. For example, from the patterns:

```
[open read* close]*
[open seek* close]*
```

we can infer a third pattern:

```
[open (read|seek)* close]*
```

Sequencing Rule: The second rule describes the sequencing of two patterns with compatible endpoints.

$$\frac{(a\mathscr{L}_1b)^* \qquad (b\mathscr{L}_2c)^* \qquad (ac)^*}{(a\mathscr{L}_1b\mathscr{L}_2c)^*}\ [\textsc{Sequence}]$$

As with the previous rule, $\mathscr{L}_1$ and $\mathscr{L}_2$ must have disjoint alphabets, which must in turn be disjoint from $\{a,b,c\}$.

Proposition 2 (Correctness of Sequencing). *Redefining* Σ' *as* $\{a,b,c\}\cup\Sigma_{\mathscr{L}_1}\cup\Sigma_{\mathscr{L}_2}$*, the correctness of the* Sequencing Rule *follows from the following fact:*

$$E_{\Sigma'}(a\mathscr{L}_1b)^*\cap E_{\Sigma'}(b\mathscr{L}_2c)^*\cap E_{\Sigma'}(ac)^* = (a\mathscr{L}_1b\mathscr{L}_2c)^*$$

Continuing the earlier example, from the patterns:

```
[open (read|seek)* close]*
[connect open]*
[connect close]*
```

we can infer a fourth pattern:

```
[connect open (read|seek)* close]*
```

Both of these rules are general; they apply to both micropatterns or any intermediate assembly thereof. Using these rules, Javert calculates the *pattern closure* for a given set of micropatterns; it repeatedly applies the above rules until they are no longer applicable.

3.3.3 Generality of Javert's Patterns and Rules

In this section, we argue that the two patterns and two inference rules presented in the previous section can sufficiently capture a large class of temporal API relationships. In Section 3.4, we present several examples of true, complex specifications that Javert finds in real software systems.

Our general temporal properties have similar structural characteristics: each consists of a linear sequence of state changing operations. In each state, there are a number (possibly zero) of legal operations that do not change the state. We believe that this generally models the *phasic* behavior of most module interfaces. For example, resource APIs, usually related to input and output, go through an initialization phase. At this point, a number of operations become legal—usually operations with environmental side effects. Finally, the interface moves through one or more state changing sequences; these often consist of finalization, deallocation, or other forms of cleanup.

We believe that most well-defined software interfaces follow similarly structured temporal patterns. Note, though, that the addition of other constraints, such as values on variables (*e.g.* no reading from an empty stack) or context free behavior (*e.g.* three calls to push can be followed by at most three calls to pop) are not captured by our technique. However, these specifications lie outside the scope of general temporal properties; they are in fact not modeled by any regular language.

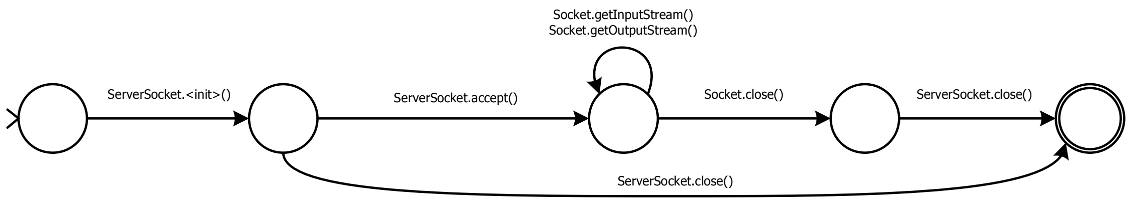


Figure 3.5: Socket API, Ammons et al. [2002]

We now demonstrate the expressiveness of our characterization by showing that it sufficiently captures many complex examples of temporal properties in the recent literature. We naturally omitted small patterns that are sufficiently captured by the micropatterns themselves. The following examples, presented in chronological order, are from systems that are capable of learning arbitrarily complex temporal properties.

A Socket API, Ammons et al.

Strauss, a tool developed by Ammons et al. [2002], mines arbitrarily complex specifications from dynamic traces. Consider the socket API in Figure 3.5. This figure has been reproduced from the original paper [Ammons et al., 2002] and translated to Java. Our approach is capable of fully composing this specification from micropatterns.
The subpattern:

```
[ ServerSocket.accept()
   ( Socket.getInputStream() |
     Socket.getOutputStream() )*
  Socket.close() ]*
```

is formed by an application of our first (branching) rule.

The sequencing of all related events is handled by repeated applications of the sequencing rule that make use of the pairwise alternating patterns that exist between all non-repeating method calls.

Finally, the branching that occurs at the `accept` call is constructed through an additional application of the branching rule, making use of the empty language.

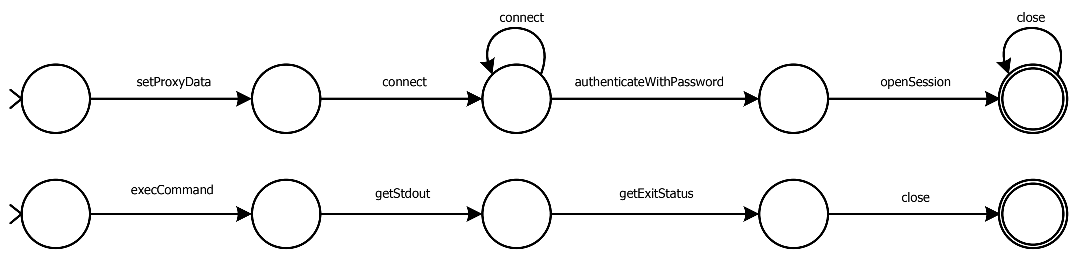


Figure 3.6: Ganymed APIs, Shoham et al. [2007a,b]

Ganymed APIs, Shoham et al.

More recently, Shoham *et al.* have developed a static specification miner that uses abstract interpretation and regular language learning. The two examples (`Connection` and `Session`) in Figure 3.6 are reproduced from their paper [Shoham et al., 2007a] and their online supplement [Shoham et al., 2007b], respectively.

The first API, `Session`, is nearly completely composable: Javert is capable of capturing all but the final repetition of `close`. Our inference rules operate on patterns with distinct, closed bounds; neither of our micropatterns captures open-ended repetition. It it likely, however, that our version of the specification (with a single call to `close`) is only slightly more restrictive and not violated in the common case.

The second API, `Connection`, is clearly composable by our technique: it involves a linear sequence of events. Javert would compute the pattern closure over the pairwise alternating relationships and yield the larger, sequenced API.

3.4 Implementation and Results

In this section, we describe Javert's implementation and empirical evaluation.

3.4.1 Implementation

Pattern Mining We implemented Javert in the Java programming language. The first phase of Javert's execution, which consists of mining the micropatterns, is performed by an existing symbolic specification mining algorithm [Gabel and Su, 2008a]. This algorithm leverages Binary

Decision Diagrams [Bryant, 1986] to maintain a compact state throughout its execution, despite simultaneously tracking up to billions of potential micropatterns. This algorithm is currently the most scalable pattern-based approach, and it is the only algorithm capable of scalably mining micropatterns with alphabets of size three. This is critical for our current approach: without this ability, we would be unable to mine our looping micropattern and introduce loops into our composed specifications.

Pattern Composition Javert's second phase is implemented in standard imperative Java. The rules are applied in a simple iterative approach until no longer applicable. After composition, Javert emits a dominating set of the composed specifications; that is, it refines the solution set so that no returned specification is contained within another. We also include the ability to filter the mined specifications by either alphabet size (*e.g.*, emit only specifications with at least four participants) or structural characteristics (*e.g.*, emit only those specifications with at least one loop). Note that while Javert is a Java application, it takes as input any finite sequence of symbols. There are no requirements on the form of the input traces; they need not be sequences of Java method calls.

Trace Collection To collect dynamic traces, we implemented a new method trace collection tool for Java programs. This tool uses the ASM[2] bytecode engineering framework along with Java's built in instrumentation capability to dynamically add tracing code to classes as they are loaded. This tool can easily instrument any Java program; its invocation is performed via an option to the parent virtual machine rather than the hosted application, obviating any need to change application configurations. Our trace collection framework has two significant advantages.

First, we log all method invocations *before* object-oriented dispatch is performed; that is, we log the compile-time targets of method calls, not the run-time method body that is eventually executed. We expect specifications to be most useful when they describe sequences of calls made by the programmer, not the implementer of an abstract interface. For example, a specification over Java's `Socket` type is likely to be more general than one over `SocketImpl`.

[2]`http://asm.objectweb.org`

Second, we log the static context (the calling function) of each method invocation. This allows us to project our traces over interesting source and destination sets. For example, assume we have a client application that uses several libraries in addition to the Java standard library. With contextual information, we can project the trace to include only *outbound* calls, *i.e.*, calls that originate in client classes and call into any non-client class. Projected traces of this form exhibit useful properties: the client can only escape its own classes through public interfaces—the targets of specification miners. Using this simple approach, we are able to log all public API calls without necessarily knowing what they are. The traces are also free of excessively "noisy" methods, like private methods within either the client or one of the libraries.

In its current form, our trace collection framework does not log object identities or values of primitive values. This adds a level of imperfection to the trace: nothing explicitly states that two calls to the same type were necessarily made using related data values. There are a number of justifications for this design decision.

First and foremost, we sought to avoid false negatives. Consider a hypothetical extension to our trace collector in which we log the receiver object of each non-static method call. We could then use this information to project our traces over all operations performed on a specific object, or equivalently, treat (call, instance) pairs as our trace alphabet. This extension would render Javert highly precise, but it would limit the discovered patterns to a single type. This would be severely limiting: note that every example specification in this chapter describes temporal relationships between two or more types. Attempting to address this by considering more dataflow is non-trivial [see Ammons et al., 2002, Scenario Extraction], as the bounds of a particular computation are unclear. If we greedily expanded our projected traces based on dataflow between objects, we could easily converge on the entire trace.

Second, we wished to design a technique that was not intrinsically dependent on information outside of the ordering of the trace events. This increases Javert's generality: it can learn patterns over unwieldy legacy traces, and it is more adaptable to environments where the trace collection mechanism is fixed or otherwise limited, possibly by architectural or performance constraints. Previous work on pattern-based specification mining has recognized this problem; we rely on those techniques to generate a coherent set of statistically significant micropatterns from imperfect traces.

Project	Description	Traces	Trace Events	Execution Time Pattern Mining	Execution Time Composition
Lucene	Document Search Engine	1	63,755	3.6s	0.03s
JGnash	Personal Finance Application	1	70,572	21.4s	25.7s
JEdit	Source Code Editor	1	973,230	103.4s	0.6s
Columba	Email Client	3	8,673,448	717.0s	87.4s
Findbugs	Static Analysis	1	14,072,862	1151.1s	51.3s
Ant	Build System	198	15,582,468	1295.9s	349.6s
Hibernate	Object Persistence API	184	26,588,144	2151.0s	1.5s

Figure 3.7: Trace data and analysis times.

Note, though, that although Javert can handle buggy or imprecise traces, it would certainly thrive with more accurate input. Techniques like SMArTIC [Lo and Khoo, 2006] perform preprocessing and clustering on traces to isolate and remove false behavior, reducing the incidence of false positives. Dynamic slicing [Agrawal and Horgan, 1990] over traces could also serve to this end by removing unrelated flows of data. Any technique for improving the accuracy of Javert's input is compatible and likely to improve results, but we sought to design for the common case.

3.4.2 Empirical Evaluation

To evaluate Javert, we collected traces from seven client applications and executed our analysis on each. For examples with more than one trace, we scanned the traces sequentially and kept the running union of the mined micropatterns from each. Javert then composed this larger set of micropatterns. We performed our experiments on a 2.66 GHz Core 2 Duo workstation equipped with Fedora Linux and the official Sun 1.6.0_04 64-bit server JVM.

Figure 3.7 lists our seven target projects, quantitative data about their respective representative traces, and Javert's execution time. Each trace consists of all outbound calls made by the client; these include calls to the Java standard library and other third party libraries. For Ant and Hibernate, we were afforded the luxury of complete test suites that automatically exercised many parts of the client applications. This allowed us to generate close to 200 traces for each. In each case, Javert was able to complete both phases of the analysis in a reasonable amount of time; the overall execution time is usually dominated by the first phase and is roughly linear in the size of the traces. The largest example completed in less than an hour.

	$\lvert\Sigma\rvert \geq 4$			$\lvert\Sigma\rvert \geq 7$			$\lvert\Sigma\rvert \geq 10$		
Project	Total	Real	False	Total	Real	False	Total	Real	False
Lucene Indexer	4	2	2	0	0	0	0	0	0
JGnash	35	5	30	28	5	23	22	5	17
JEdit	13	4	9	4	3	1	2	2	0
Columba	12	2	10	7	2	5	4	0	4
Findbugs	29	10	19	23	10	13	13	9	4
Ant	46	34	12	27	16	11	4	3	1
Hibernate	18	13	5	10	8	2	5	4	1

Figure 3.8: Quantitative results.

Our quantitative results are displayed in Figure 3.8. For each project, we display the total number of composed specifications, the number of "real" specifications, and the number of false positives. This information is presented for three minimum size thresholds, $\lvert\Sigma\rvert \geq 4$, 7, and 10.

Categorizing the results is a complex task: the definition of a real specification can be a subject of opinion. Temporal specifications, when expressed as regular languages, are by nature an overapproximation of the correct behavior of the system. In addition, the very motivation for our work—the lack of well documented specifications—makes validating our findings difficult and subjective.

If we assume a high level of precision of Javert's first phase, pattern mining, the soundness of our composition rules implies that we infer no "false" properties; every pattern we discover does in fact describe a frequently occurring sequence of events in the input trace. Our quantitative evaluation thus seeks to differentiate between artifacts of control flow—correlated method calls that form an incidental temporal property—and true temporal properties with a dataflow relationship. To this end, we settled on the following mechanical definition of a "real" temporal property.

Definition 10 (Real Temporal Property)**.** A temporal property is real if it can be traced back to a sequence of calls in the source code that are chained by a dataflow relationship.

In this definition, we traced the flow of data through parameters, return values, fields, and static variables. This definition, in effect, evaluates the *idea* of frequent pattern composition as a solution to the specification mining problem.

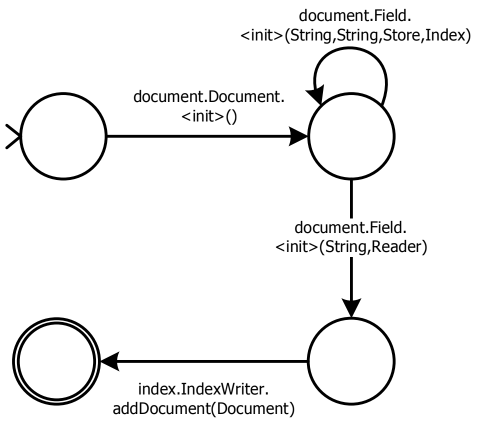


Figure 3.9: Specification of the use of Lucene's indexer.

We performed our first experiment on a trace from Apache's Lucene indexer, a component of the Lucene document search engine. We collected all calls made by the demo application into any non-demo class, *i.e.*, calls into the Lucene indexing library. From this, we discovered the specification in Figure 3.9. This represents the structure that must be followed in order to add a document to the search index: the `Document` object must first be created, and several instances of metadata fields can be added. Next, the body of the document is filled in by instantiating a particular `Field` instance with a `Reader` parameter. Finally, this `Document` object is used as the parameter of the `addDocument` method of indexer.

The next three examples, JGnash, JEdit, and Columba, are user applications with graphical interfaces. With these applications, we experienced a significant number of false positives. In all three cases, the majority of false positives consisted of large aggregations of code that performed the initialization of user interface elements. With Java's Swing GUI interface, one creates a large object model that conceptually mirrors the display. The creation of this model involves a large amount of boilerplate code[3] with a consistent (but not necessarily required) structure.

It is interesting to note, however, that Javert discovered large, distinct "clumps" of this code, which both simplified their identification as false positives and possibly reduced the overall number of false patterns. Figure 3.10 presents one particularly interesting specification from Javert's JGnash solution set. JGnash is a personal finance package; when saving bank account

[3]In fact, this code is often automatically generated using third party tools.

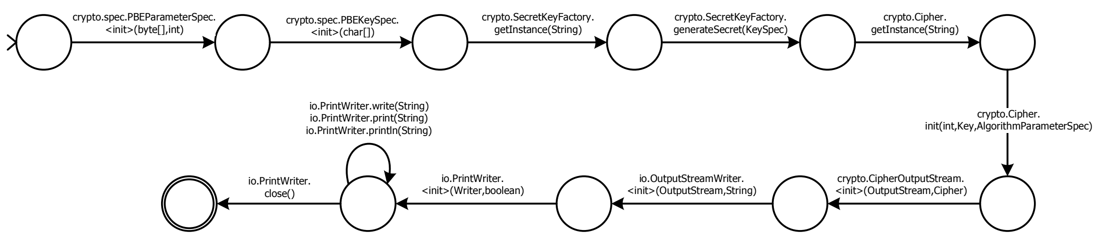


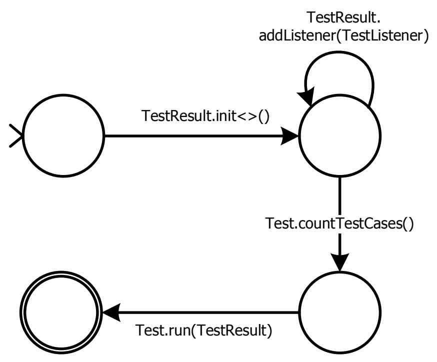
Figure 3.10: A specification for the use of a symmetric cipher, extracted from JGnash.

data on a local hard drive, the application uses a symmetric cipher. Java's cryptography API is quite complex: each of several required objects must be accessed through separate factory interfaces. The specification discovered by Javert correctly describes a general approach to using a symmetric cipher to encrypt a stream of character data.

Executing the Apache Ant and Hibernate test suites yielded a wealth of trace data. The Ant build system can interact with many external libraries as part of a project's build process, and Hibernate frequently uses structured APIs, including Java's SQL API for interacting with relational databases and various bytecode engineering frameworks for generating dynamic proxy classes. On these examples, Javert discovered a number of interesting specifications with few false positives. Figure 3.11 displays two compact properties discovered from Ant, including the server-side TCP socket API. Figures 3.12 and 3.13 display three properties mined from the Hibernate traces.

Note the size of the first: it describes the use of the `javaassist` library to perform an entire transformation of a Java class. This involves reading it in as a byte stream, building the object model, transforming the various objects, and rewriting it as a byte stream. Note, though, that it is somewhat wide (not as restrictive as it could be) during the point at which the code of a method is traversed: although our micropatterns capture the temporal relationships between `hasNext` and `next`, our rules were unable to compose this embedded subpattern. As future work, we are investigating other forms of inference rules to account for this.

Overall, the results of our analysis were precise, usable, and interesting. Although we experienced a number of false positives on each example, the ratio of false positives to real specifications remains reasonable, often below one. This is in sharp contrast to earlier work that uses pattern matching in isolation: the ratio of false positives to significant specifications is often



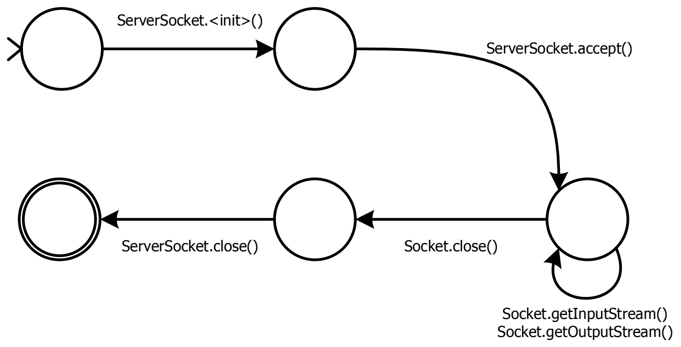


Figure 3.11: Two properties extracted from executions of Apache Ant.

orders of magnitude greater than Javert's. For example, large portions of papers [Weimer and Necula, 2005, Yang et al., 2006] have been dedicated to filtering interesting sets of the large number of simple alternating patterns in programs.

We believe that Javert's precision is partly due to the way we compose patterns through inference rules. Intuitively, a micropattern in isolation has a low probability of representing a significant temporal property. However, if several *related* micropatterns exist that can be composed or chained, the probability of significance increases. Evidence of this hypothesis exists in Figure 3.8: note the decrease in the ratio of false positives to true specifications as we raise the minimum size threshold. Our composition process is, in spirit, a form of natural selection: unless a pattern can connect with one or more others, it does not survive the selection process.

Absent from this evaluation is a direct comparison with other specification miners. The primary reason for this is that no other tool solves exactly the same *problem*: some tools enumerate many small micropatterns, while others learn a single language. To our knowledge, Javert is the only tool that can mine several complex, significant specifications from a trace in a single

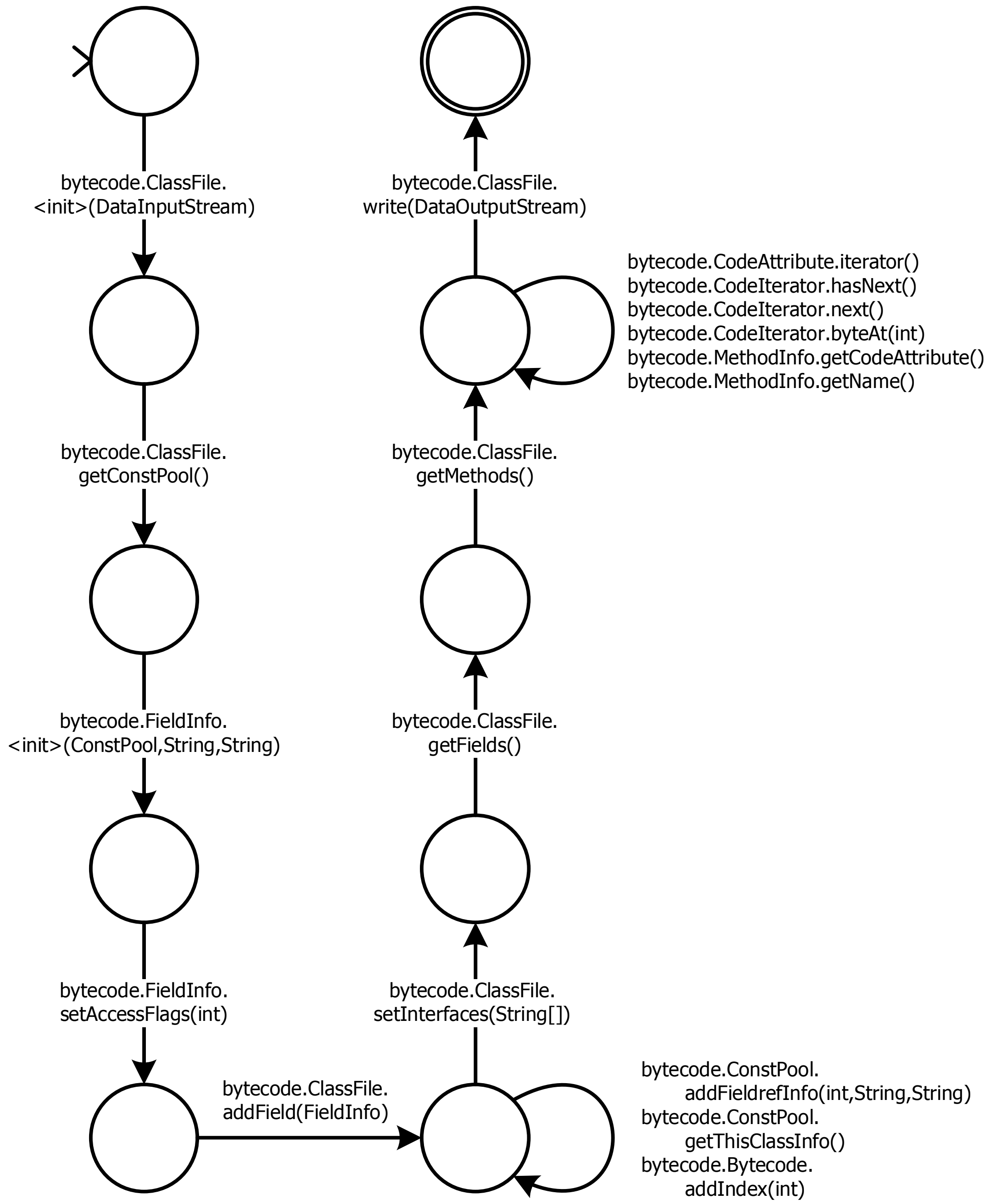


Figure 3.12: A larger temporal property of the JavaAssist bytecode framework, extracted from its use by Hibernate.

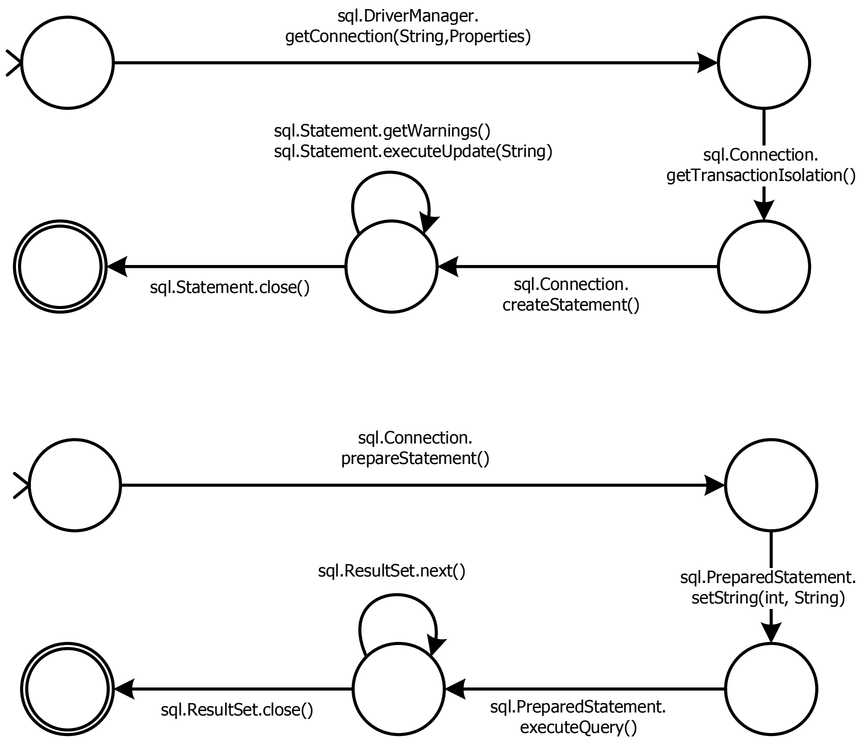


Figure 3.13: Two specifications extracted from Hibernate.

pass—with no required input outside of the trace. Thus, the methodology for comparing Javert with a regular language learner is not clear: for a trace with n unique events, we would have to run 2^n projected traces (over each subset of the trace alphabet) through the regular language learner to allow it to consider all possibilities.

Javert is a dynamic analysis, and it carries with it most typical advantages and disadvantages of all dynamic analyses. Javert's precision is limited by the precision of the input traces, and its recall is limited by the variety of data available in the input traces. However, Nimmer and Ernst's findings [Nimmer and Ernst, 2002] suggest that this may not be a major problem: they found that a surprisingly small number of test runs are sufficient to capture most of a program's static behavior. It is not likely, however, that Javert misses an important specification that *does* appear in an input trace. Javert's pattern mining front end considers all possible instances of micropatterns, and its composition engine computes the full pattern closure: every legal composition is considered.

3.5 Related Work

In this discussion, we present related work in the software specification mining and dynamic analysis areas.

3.5.1 Temporal Specification Mining

Ammons et al. [2002] first characterized the inference of temporal specifications as a language learning problem. In this work, the authors used a probabilistic finite automaton learner to extract likely specifications. A key challenge with their approach was simplifying specifications to an acceptable level of precision. The authors addressed this challenge in a later paper [Ammons et al., 2003] by applying concept analysis to debug the learned specifications. Shoham *et al.* recently presented a static analysis with the same general goal [Shoham et al., 2007a]. Both of these techniques are limited in that they require the alphabet of the mined specification to be known. Javert is capable of finding specifications of similar complexity in a much more scalable manner.

Various static analyses [Alur et al., 2005, Henzinger et al., 2005, Whaley et al., 2002] take as input a type and produce as output an automaton that encodes legal call sequences of operations on that type. Call sequences are considered legal if they do not lead to an assertion failure or another exceptional control path. These techniques are limited in that they find a specification over a single type, and they may be too permissive: if the implementation of the type is not programmed defensively, it may have illegal call sequences that lead to an inconsistent state but do not throw an exception or violate an assertion. In contrast, Javert operates on the assumption that common *usage* likely reflects the true specification.

Engler *et al.* first introduced the idea of matching an alternating pattern over a program to produce possible specification candidates [Engler et al., 2001]. This approach suffered from imprecision, so the authors used statistical methods to rank the possible properties. Weimer and Necula [2005] built on this idea by restricting their search to alternating patterns that traverse exceptional control flow paths. While this improved the precision of the approach, the patterns were still fundamentally limited to simple two-letter alternating sequences. We later introduced a highly scalable symbolic technique that extends this approach to patterns of size three [Gabel and Su, 2008a] and greater.

Yang et al. [2006] adopted a similar approach for locating alternating events, Perracotta, that introduced novel methods for handling imperfect traces. Sources of imperfection include interleaved concurrent executions, omitted information (like memory addresses), or bugs. In this work, the authors briefly describe a heuristic for combining simple alternating patterns, but the approach is limited to finding simple sequencing patterns. Our work provides a more general mechanism for inferring a class of general temporal properties.

ADABU [Dallmeier et al., 2006] is similar to Javert in that it dynamically learns temporal specifications from Java programs. The temporal specifications have a similar structure: transitions are labeled by method calls. However, the authors take a different approach to the problem: the tool *directly observes* state changes by calling inspector methods, like `isEmpty()` on a collection type, rather than inferring them from ordering information. In addition, like many other miners, the tool is limited to finding specifications over a single type with a known alphabet.

Ramanathan *et al.* have developed a static analysis [Ramanathan et al., 2007a] for detecting "function precedence protocols." These specifications are of the form "function x is called on all paths leading to an invocation of function y." The authors later generalized this technique to include other predicates like constraints on variables [Ramanathan et al., 2007b]. These specifications are of limited expressiveness; they correspond to the simple pattern (b^+a). Our analysis is capable of finding more complex specifications of arbitrary size.

Acharya *et al.* recently introduced a static analysis that mines partial orderings on function invocation sequences [Acharya et al., 2007]. The authors use a data mining technique, frequent closed partial order mining, to enumerate possible specifications. These specifications are of limited form—simple chains—and are not as strict; the partial orderings represent a "may" requirement, not a "must" requirement like a strict temporal specification.

A similar technique, developed by Wasylkowski et al. [2007], uses a static analysis to extract simple partial orders on function call sequences. The tool uses frequent itemset mining to both recognize frequently occurring patterns *and* detect violations. When compared with Javert, the mined patterns are more limited in complexity. However, a similar extension—combining verification with mining—would be a valuable addition to Javert's functionality.

3.5.2 Other Specification Miners

Li and Zhou constructed an analysis, PR-Miner, that makes use of frequent itemset mining to find highly correlated function calls [Li and Zhou, 2005]. They then used these correlated sets as specifications; they analyzed the source code for instances of sets of function invocations that omit a commonly correlated call. Lu *et al.* later extended this technique to find highly correlated variable accesses, which they use to locate concurrency bugs [Lu et al., 2007]. These techniques are orthogonal to our own: the correlated sets returned by these analyses do not contain any temporal relationships.

Kremenek *et al.* have developed a general approach to specification mining that uses probabilistic models called annotation factor graphs [Kremenek et al., 2006]. This analysis probabilistically assigns annotations that denote a role to functions with a program. These flexible models allow the user to add additional domain-specific information to the analysis. Unlike our analysis, this technique locates instances of a very restricted type of property.

3.5.3 Dynamic Analysis

The Daikon project [Csallner and Smaragdakis, 2006, Ernst et al., 2000] is a dynamic technique that is similar in spirit to our own analysis. Daikon locates invariants on the values of variables, while we locate invariants on the sequencing of function invocations. DIDUCE is a similar technique that also locates potential violations of invariants [Hangal and Lam, 2002].

Combining the ideas of invariant detection and temporal property mining, Lorenzoli *et al.* have developed a dynamic analysis algorithm for extracting software behavioral models [Lorenzoli et al., 2008]. The algorithm, GK-tail, builds an *Extended Finite State Machine* from a set of dynamic traces. The transitions in these extended models include both a called function or method and a set of constraints on the parameters or environment. For future work, we are interested in investigating the compatibility of these extended models and our general approach.

3.6 Conclusion

In this chapter, we have presented a general specification mining framework, Javert, that can fully automatically mine complex, real-world temporal specifications. It is based on the observation that software often operates in phases and that complex temporal specifications can be constructed from smaller generic patterns. The framework is general; it is independent of the method used to mine these smaller patterns, and any set of inference rules can be used to compose them. We have introduced two intuitive, sound rules that are general enough to learn most of the temporal specifications in the literature. We have implemented our framework as a practical tool, and our empirical evaluation of it on several open source projects demonstrates that Javert is scalable, general, and precise.

There are a few interesting directions for future work. First, we have considered two specific rules for pattern composition in this chapter. It would be interesting to investigate whether there are other suitable choices of composition rules for specification mining. Second, we plan to investigate the effectiveness of incorporating additional dataflow information, such as the information provided by program slicing, into our analysis. Third, we would like to investigate how to adapt our technique and develop a static specification mining algorithm that operates directly on source code. Pattern composition may help reduce the number of false positives by composing many small patterns (as we have observed in this work). Finally, we have focused on temporal specifications, and it would be interesting to consider more expressive properties and investigate whether there are useful instantiations of our framework in these more general settings.

4 Directly Inferring Temporal Safety Errors

The interfaces of software components are often paired with specifications or protocols that prescribe correct and safe usage. An important class of these specifications consists of *temporal safety properties* over function or method call sequences. Because violations of these properties can lead to program crashes or subtly inconsistent program state, these properties are frequently the target of runtime monitoring techniques. However, the properties must be specified in advance, a time-consuming process. Recognizing this problem, researchers have proposed various specification inference techniques, but they suffer from imprecision and require a significant investment in developer time.

This work presents the first fully automatic dynamic technique for simultaneously learning *and* enforcing general temporal properties over method call sequences. Our technique is an online algorithm that operates over a short, finite execution history. This limited view works well in practice due to the inherent temporal locality in sequential method calls on Java objects, a property we validate empirically. We have implemented our algorithm in a practical tool for Java, OCD, that operates with a high degree of precision and finds new defects and code smells in well-tested applications.

4.1 Introduction

The interfaces of software components are often paired with specifications or protocols that prescribe correct and safe usage. If violated, software systems may crash or—perhaps worse—be placed in an inconsistent state and behave nondeterministically. One important type of these specifications is the class of *temporal safety properties* over function or method call sequences. Common examples include locking disciplines, in which locking functions (*e.g.* `lock`, `unlock`) must be called in a strictly alternating fashion, and resource usage, in which all resource-like objects (*e.g.* files, sockets) must be eventually closed or disposed and cannot be used thereafter.

Formalized by researchers as the *typestate* [Strom and Yemini, 1986] concept, these properties capture a broad category of software defects and have inspired a diverse body of research. Many static formal verification algorithms (in particular software model checkers [Das et al., 2002]) either specifically target these specifications [Fink et al., 2008] or use them as their primary example. Similarly, dynamic tools, such as runtime monitoring frameworks [Chen and Rosu, 2007], often operate over these temporal properties as well. These tools and techniques have advanced significantly in recent years, particularly in the areas of scalability and automation, but they still must be supplied with temporal specifications to verify—generally a manual and time-consuming task.

This dearth of enforceable properties has led in part to the development of automated specification *mining* or *inference* techniques. These tools observe a system's source code or its runtime behavior and produce one or more temporal specifications as a result. Most of these tools leverage potentially imprecise parameters, such as the frequency of a specification's occurrence in the source code or the number of times it was satisfied in a dynamic trace. Similar to data mining (in fact, many specification mining tools directly use data mining algorithms), these inexact parameters lead to a precision/recall tradeoff: a precise tool may fail to infer important properties, while a more liberal tool may produce many false properties, requiring a large time investment by the software developer.

In this chapter, we present a novel technique and a practical tool, OCD, for *simultaneously* learning and enforcing general temporal properties over function or method call sequences. Both tasks are tightly integrated and form a symbiotic relationship: the verifier benefits from the abundance of inferred properties, and the learning algorithm benefits from the results of continuous verification to learn and refine properties. Most importantly, the software developer—our intended user—benefits from being removed from the center of the process: he or she can use OCD as a turn-key dynamic online bug finding tool that requires no input beyond the program to analyze.

OCD is a dynamic trace processor for Java programs: it analyzes Java method calls online through load-time instrumentation. At a high level, our algorithm functions by using a predefined set of specification *templates*—two-letter regular expressions that represent components of larger, more general temporal properties—and attempting to enforce them in a brute-force manner over all possible combinations of method calls. Our experience with the Javert specification miner [Gabel and Su, 2008b] provides evidence that the inference of these small properties can yield a surprisingly complete and general class of temporal specifications, and we show in this chapter that *enforcing* these smaller patterns is a safe approximation of enforcing the larger, general properties. Our work is enabled by two key observations:

Temporal Locality From a scalability perspective, this brute-force approach would ordinarily be intractable in both time and space. We solve this problem by operating over a relatively small finite window of trace events, which greatly constrains the number of property instances that we learn and enforce. Though we demonstrate that the *verification* of properties over a finite window is a safe approximation of verification over a complete trace, a finite window may greatly reduce the effectiveness of any *learning* algorithm: we may be unable to sufficiently speculate if our view is too short-sighted. This effect is greatly mitigated—sometimes even completely—by the fact that method calls in Java programs exhibit a high degree of *temporal locality*; that is, operations on particular objects tend to be tightly clustered in time. We have stated this observation anecdotally in previous work; we now evaluate it empirically in Section 4.3 and find it to be true for a diverse set of commonly used Java programs.

Verification of Redundant Properties Dynamic specification miners attempt to synthesize specifications by generalizing a program's observed behavior. Unfortunately, "false" specifications often result from the inference of true properties of the trace (perhaps inferred from coincidentally common behavior caused by control flow artifacts, for example) that are not considered by the developer to be true specifications. While this poses a major precision problem for specification miners, it affords us an interesting opportunity. As the goal of our technique is to locate defects—not to produce human-usable specifications—the properties we infer are seen by a human developer only if they are *violated*. Rather than applying coarse-grained filtering heuristics (as is commonly done [Yang et al., 2006]) and likely losing many important specifications, we can simply attempt to verify *all* learned properties without human validation.

The vast majority of the "false" properties are verified and produce no output, thus trading inexpensive CPU time for valuable human developer time.

We evaluated OCD on a set of commonly used Java programs and found that it learns and fully verifies a large set of temporal properties with acceptable overhead. On a subset of our evaluated programs, our tool revealed previously unknown defects and code smells. In all experiments, OCD maintained a high degree of precision.

We make the following specific contributions:

1. The first online algorithm that simultaneously learns and enforces general temporal properties of software systems. Our algorithm is an online trace processor that operates over a short-sighted, finite window of trace events.

2. A practical tool for Java, OCD, which we use to demonstrate the effectiveness of our algorithm. OCD learns and verifies a large number of properties with acceptable overhead and high precision, and it finds previously unknown defects.

3. A demonstration of the generality of our work. In particular, we show that our tool can be configured to discover and enforce function precedence protocols [Ramanathan et al., 2007a] as well as temporal association rules of function calls and field accesses [Li and Zhou, 2005].

4. An empirical evaluation of the temporal locality of Java method accesses in practice, which we use to justify our use of a short-sighted trace window (as well as set its size).

This chapter is organized as follows. The following section (Section 4.2) describes our general approach and algorithm, while Section 4.3 discusses the realization of our algorithm as a practical defect detection system. Section 4.4 contains an empirical evaluation of our work, and Sections 4.5 and 4.6 discuss related work and our plans for continuing this research, respectively.

4.2 Approach

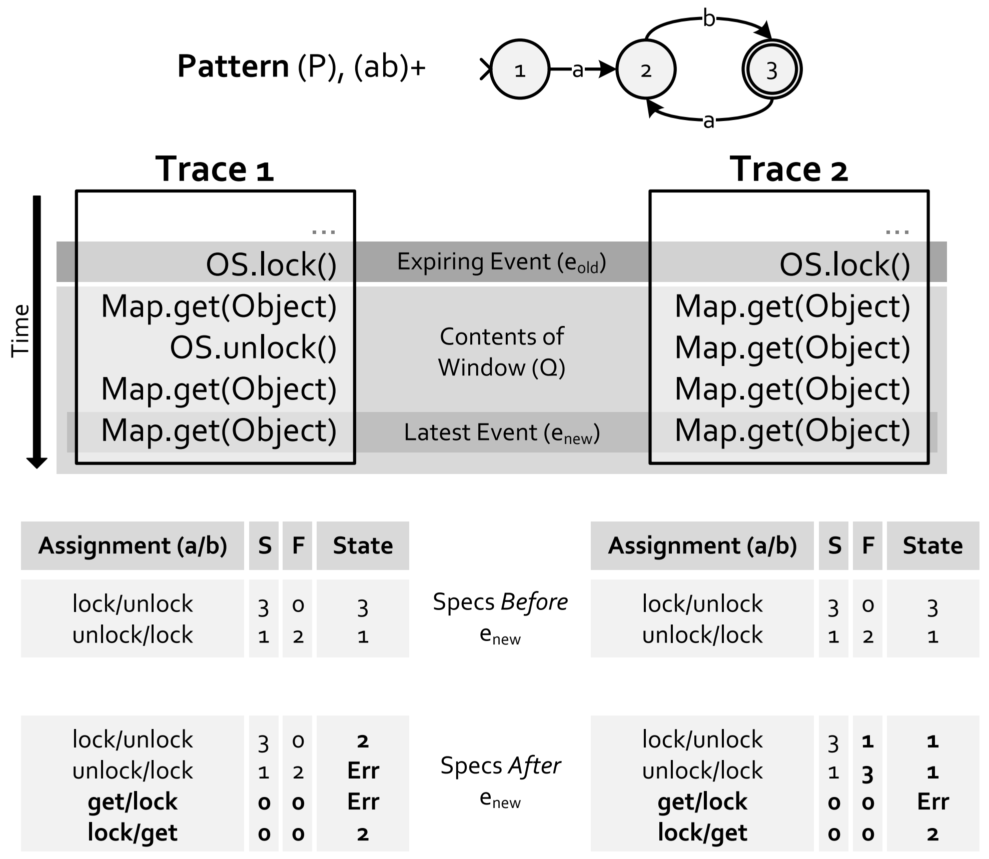
This section describes our basic approach. We first describe our algorithm in its simplest form (Section 4.2.1). We then expand on the basic definition with a series of generalizations (Sections 4.2.2–4.2.4) that form the final algorithm implemented in our tool, OCD.

Algorithm 4.1 Online inference and enforcement algorithm.

```
Constants: $P$ : Two-letter pattern automaton over $\{\mathtt{a}, \mathtt{b}\}$
           with states {INIT, ...}
Types: Asgn : $(a : \tau, b : \tau)$
       Spec : $(asgn : \text{Asgn}, sat : \text{int}, fail : \text{int}, st : \text{state of } P)$
State: $Q$ : Bounded Queue of $\tau$
       $specs$ : Asgn $\mapsto$ Spec
Require: $e_{new} : \tau$

 1: $Q \leftarrow$ ADD$(Q, e_{new})$
 2: $e_{old} \leftarrow$ REMOVE$(Q)$

 3: for all $e_{fut}$ in $Q$ do
 4:     if $(e_{old}, e_{fut}) \notin \text{domain}(specs)$ then
 5:         $specs(e_{old}, e_{fut}) \leftarrow \left(\left(e_{old}, e_{fut}\right), 0, 0, \text{INIT}\right)$
 6:         $specs(e_{fut}, e_{old}) \leftarrow \left(\left(e_{fut}, e_{old}\right), 0, 0, \text{INIT}\right)$
 7:     end if
 8: end for

 9: for all $spec$ in $specs(e_{old}, *) \cup specs(*, e_{old})$ do
10:     if $spec.asgn.a = e_{old}$ then
11:         $spec.st \leftarrow$ NEXT$(spec.st, \mathtt{a})$
12:     else
13:         $spec.st \leftarrow$ NEXT$(spec.st, \mathtt{b})$
14:     end if

15:     if $spec.asgn.a$ not in $Q$ and $spec.asgn.b$ not in $Q$ then
16:         if ISFINAL$(spec.st)$ then
17:             $spec.sat \leftarrow spec.sat + 1$
18:         else
19:             $spec.fail \leftarrow spec.fail + 1$
20:             if ISENFORCING$(spec.sat, spec.fail)$ then
21:                 REPORTANOMALY( )
22:             end if
23:         end if
24:         $spec.st \leftarrow$ INIT
25:     end if
26: end for
```

4.2.1 Basic Algorithm

Our algorithm, described in pseudocode as Algorithm 4.1, functions as an online trace processor that receives traced *events* from an instrumented application as they occur. It is configured with a *pattern template*—an abstract model of a specification—and produces as online output *anomalies*—specific instantiations of the templates that likely represent defects in the monitored system.



Assignment (a/b)	S	F	State		Assignment (a/b)	S	F	State
lock/unlock	3	0	3	Specs *Before* e_{new}	lock/unlock	3	0	3
unlock/lock	1	2	1		unlock/lock	1	2	1
lock/unlock	3	0	**2**	Specs *After* e_{new}	lock/unlock	3	**1**	**1**
unlock/lock	1	2	**Err**		unlock/lock	1	**3**	**1**
get/lock	**0**	**0**	**Err**		**get/lock**	**0**	**0**	**Err**
lock/get	**0**	**0**	**2**		**lock/get**	**0**	**0**	**2**

Figure 4.1: Execution of Algorithm 4.1 on two example traces.

Events In this basic incarnation of our algorithm, an *event* consists only of a type τ. When tracing Java method calls, for example, τ represents a method's fully qualified signature. (Sections 4.2.2 and 4.2.3 sections discuss generalizations that consider additional information, *e.g.* receiver objects.) Two example traces appear in Figure 4.1.

Pattern Templates A *pattern template* is a two-letter regular expression describing the general structure of specifications to infer. We refer to its alphabet as the *symbolic alphabet*, which for the remainder of the chapter we will assume without loss of generality to be exactly $\{a, b\}$. For this expository example, we will focus on the simple *alternating* pattern $(ab)+$, which describes the family of two-event specifications in which the events must strictly alternate. A minimal finite automaton that recognizes this pattern appears in Figure 4.1. A *concrete assignment* $\{a \mapsto \tau_1, b \mapsto \tau_2\}$ maps the symbolic alphabet to two (distinct) trace event types. In the first example trace of Figure 4.1, one possible concrete assignment into the alternating pattern is

$\{a \mapsto \mathsf{OS.lock}, b \mapsto \mathsf{OS.unlock}\}$, forming the potential specification $(\mathsf{OS.lock}\ \mathsf{OS.unlock})+$.

Finite Window Our algorithm operates over a finite *window*: a bounded view of a trace's history. The window is a standard FIFO queue; we add each new event to its head while simultaneously removing the oldest event from its tail, maintaining a fixed size. We formulate our algorithm in terms of this "expiring" event; the queue in effect provides a short-sighted view of the *future*. [1] Though omitted from this presentation for brevity, we populate the queue with null events on startup and drain it completely on shutdown.

Our algorithm aims to a) learn concrete assignments of the pattern (*i.e.*, specifications) that "should" be enforced and b) report violations as *anomalies*. Though conceptually distinct, our algorithm integrates the two processes such that they are indistinguishable. The following steps describe our algorithm's execution, and they serve to narrate the running example in Figure 4.1 and the pseudocode of Algorithm 4.1. As this is an online algorithm, we describe its execution in terms of the steps we perform on a single event.

State Our algorithm maintains a collection of 4-tuples, each of which contains a) a concrete assignment, defined earlier; b) a *satisfied count*, the number times the pattern was matched over a substring of the trace; c) a *failed count*, defined similarly; and d) a pattern automaton instance, which we encode as its current *state*.

Queue Maintenance (Lines 1-2) We add the newest event to the head of the queue and remove the oldest for processing. In our example (Figure 4.1), our queue is of length four and our newest and oldest events are the same for both traces: Map.get and OS.lock, respectively.

Lazy Instantiation (Lines 3-8) We observe the queue and identify any upcoming pairings—concrete assignments of the pattern—that we have not yet seen. We then instantiate two patterns, one for each symmetric assignment, in their initial state. In our example traces, Map.get and OS.lock have not yet occurred within a span of four (our window size) events, so they are

[1] It is straightforward to reformulate the algorithm in terms of the "newest" event, in which case the window provides the more standard view of the recent *past*. This formulation is more terse and intuitive, though.

absent from the initial specification table. After this step, two concrete assignments are added to the table.

Advancing Automata (Lines 9-14) We iterate through all specifications that our currently processed event (e_{old}) participates in (line 9) and advance their state machines (lines 10-14). The test at line 10 “dereferences” the concrete assignment to its symbolic letter, and the state updates on lines 11 and 13 access an external function NEXT, which is a simple accessor for the transition relation of the pattern P. To improve performance, our implementation incrementally maintains a mapped index from each seen trace element on to the set of all affected specifications. Trace 1 (left) of Figure 4.1 demonstrates this step: all four specifications (including the two instantiated in their initial state) are advanced according to the pattern.

Bookkeeping and Enforcement (Lines 15-26) Line 15 inspects the queue, determining if any forthcoming event is relevant to the current specification.[2] If not, we have reached the end of a time-clustered substring of the trace (with respect to the current specification) and we inspect the last state of the automaton instance. If the automaton was left in a final state (*i.e.*, this trace “scenario” matches the specification and is accepted), we increment the satisfied count. If not, the we increment the failing count.

Line 20 accesses ISENFORCING, an external function (predicate) that takes as input the *historical statistics* (*i.e.*, the sat and fail counts) and determines according to a predefined algorithm if the specification should be considered “real” and enforced. One simple implementation of ISENFORCING might be based on a ratio:

$$\textsc{IsEnforcing}(sat, fail) \equiv \frac{sat}{sat + fail} > \mathit{THRESHOLD}$$

We refer to such functions as *learning strategies*. The various implementations and the values of their constants/thresholds are of great importance to our system’s performance; we discuss them in detail in Section 4.3. Finally, in the event that ISENFORCING returns true, we report the current instance as an anomaly.

[2]For performance, our implementation maintains an incremental set view of the queue.

In Trace 2 (right) of our running example, both lock/unlock specifications must be counted and reset as neither lock nor unlock appear in the window. This results in a failure of both specifications, with the failure of the more intuitive of the two (lock/unlock) likely being flagged as an anomaly; that is, IsEnforcing returns true for lock/unlock and false for unlock/lock. Note that in this case our algorithm is conservative: it may be the case that an unlock event is forthcoming, but our window is not appropriately sized to recognize it. This may result in both unlearned properties (false negatives) and false anomalies (false positives), which highlights the importance of setting the window to an appropriate size.

4.2.2 Separating Event Instances

The most crucial omission from our basic algorithm is its lack of support for separating and tracking *multiple instances* of the learned specifications. When tracing Java method calls, for example, it is often desirable to separate trace events that are generated from different receiver objects; failing to do so can hurt both precision and recall. For example, if we consider a source program in which all operations require two *nested* locks, all traces would appear to fail due to the apparent "double locking." Even if we somehow learned the specification (or supplied it statically), we would generate false error reports.

We adapt our algorithm to accommodate differences in receiver objects—or, more generally, any form of different *instance*—by extending the type of events from a simple type τ to a pair: (τ, id), where τ is as defined previously and id is an integer identifier. The remaining changes are straightforward:

- Rather than a single state st, each specification tuple now contains a map: $instances : (id : \text{int}) \mapsto (st : \text{state of } P)$. Thus, the single-instance specification tuple becomes a specification "schema" that tracks multiple instances.

- The predicates "(**not**) **in** Q" on lines 3 and 15 now operate only over the relevant queue elements; *i.e.* those whose $id = e_{old}.id$.

- Matching and anomaly reporting (lines 9-26) occurs on the specific relevant instance.

- Lazy instantiation (lines 5-6) is extended to build specific instances, and the "reset" operation (line 24) is replaced with a full deletion from the specification's *instances* map to prevent unbounded memory usage. Note that our finite window allows a rather simple solution to this problem, while other runtime monitoring tools must interact with the target program's runtime (*e.g.* Java's garbage collector through weak references [Chen and Rosu, 2007]).

The concrete assignment and statistics (sat and failure counts) are shared between all instances.

4.2.3 Event Contexts and Multiple Patterns

The basic algorithm does not track any information about the static *source* of the events. For example, one may choose to represent the static source of a Java method call as its call site. This information is not critical to our algorithm's execution, but it does provide much more meaningful error reports. In addition, it allows for new, more rich implementations of IsEnforcing (our predicate that decides when a pattern is "learned"): we can now favor properties that are satisfied in multiple, distinct source locations of the target program. The details of this extension amount to straightforward bookkeeping and are omitted for brevity.

The final extension to our algorithm allows it to simultaneously learn and verify multiple specification pattern templates. This is also straightforward: it essentially amounts to running multiple copies of the algorithm, one for each of the pattern templates.

4.2.4 Additional Considerations

Caching Failing Instances In general, the question of *recall*—how many properties we enforce—is an empirical one. However, for all specifications that are *eventually* learned and chosen for enforcement, we do not miss the reporting of any anomalies. This is due to 1) our conservative, eager error reporting and 2) the fact that our implementation *caches* all failing instances for properties that are not (yet) enforced. If the targeted program exhibits a defect while the relevant property is still being "learned," we cache the failing instance and report it if or when the property reaches maturity.

Grouping and Ordering Patterns We observe that for any two event types (method calls), there is at most one "best" property that should be enforced. For example, consider our earlier example with methods OS.lock and OS.unlock, and the following simple trace:

lock, unlock, lock, unlock, lock, unlock, lock, unlock, lock, unlock

Using fuzzy criteria for learning (*i.e.*, an implementation of IsEnforcing that admits failing instances), it is likely that both alternating specifications (lock unlock)+ and (unlock lock)+ would be learned. To mitigate this effect, we group all specifications over the same trace letters—including those from multiple pattern templates, discussed above—and restrict anomaly reporting to the "best" *enforcing* specification. We find that using a simple "satisfied ratio" as a total ordering works well in practice as an implementation of "best."

4.3 System Design

This section presents the realization of our algorithm as a practical and effective defect detection tool, Ocd. We start with our methodology for selecting the default size for our finite window, arguably the most important parameter in our system (Section 4.3.1). Next, we discuss the various pattern templates we use (Section 4.3.2). We then discuss the design and implementation of *learning strategies* (Section 4.3.3), which have thus far been presented in terms of the predicate IsEnforcing. Finally, we discuss our automatic *multivariate self-tuner* (Section 4.3.4), which allows Ocd to function well on a wide variety of target programs without the danger of "overtraining" its various parameters.

4.3.1 Window Size

The length of the finite window is a critical parameter of our algorithm. If aggressively set to too low a value, we learn fewer properties and perhaps generate more false defect warnings. If conservatively set to too high a value, the algorithm may exhibit a prohibitive amount of time and/or space overhead. Our goal is to set a value that is as small as possible while still capturing a large number of important properties. We set our default window size based on an evaluation of the typical *temporal locality* of the method call sequences of several Java programs.

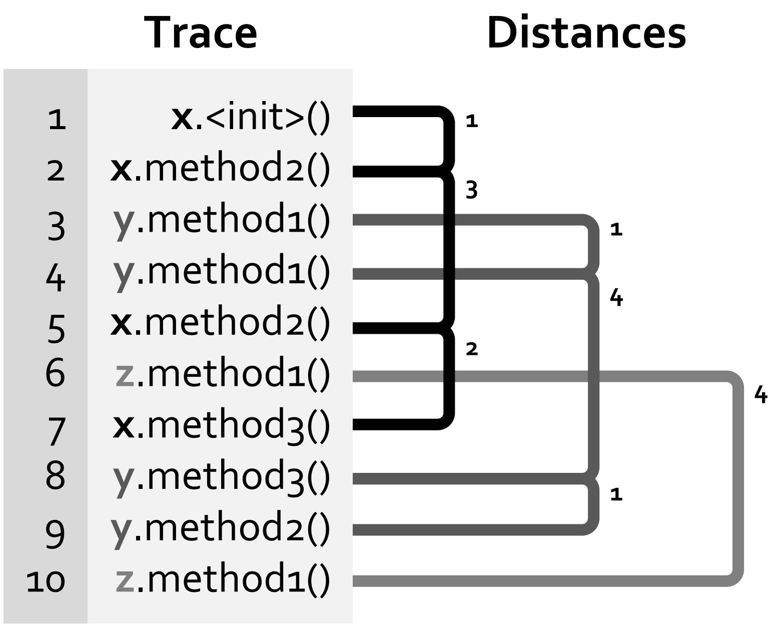


Figure 4.2: An example of our methodology for measuring temporal locality.

Our notion of temporal locality is based on a measure of the trace distance between successive method calls on individual objects; an example appears in Figure 4.2.

Our choice of trace distance as a metric (as opposed to an alternative measure of locality, such as real time) is practical and driven by our algorithm. Note, though, our restriction to pairs of *successive* method calls. It should not be immediately apparent that this is correct: if we consider a typical trace corresponding to the usage of a resource containing the methods open(), read(), and close(), for example, our definition omits any measure of distance between open() and close(), which sounds like an "alternating" property we might hope to learn. However, we *can* learn equally useful properties like "the string of read()s must occur after the call to open(), and call to close() must occur after the string of read()s." This is the essence of our reasoning: each pair of successive method calls represents a *transition* in a pattern automaton, and we can learn patterns over the most essential transitions by solely considering successive method calls.

We performed our study of temporal locality on the DaCapo workload [Blackburn et al., 2006], which includes a wide variety of production Java applications. For each benchmark, we evaluated the temporal locality of successive method calls with respect to two types of traces:

JDK A caller-side transformation that traces all calls originating in the benchmark and executing

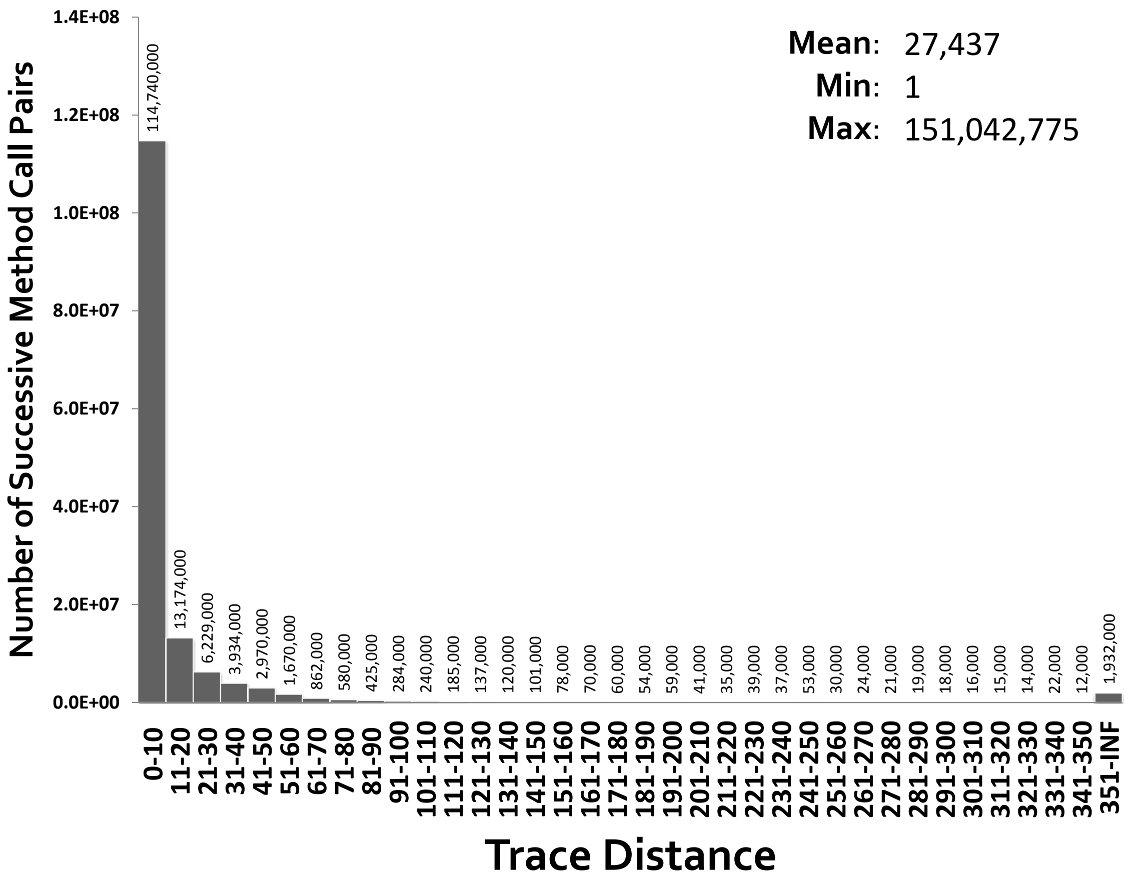


Figure 4.3: A histogram of the distances between successive method calls on the same object during the execution of Eclipse.

in the Java standard library. This family of traces represents the benchmarks' usage of multiple external APIs.

Project A callee-side transformation that traces all methods declared as public within the benchmark itself. We intend this family of traces to represent the manner in which a project uses its own APIs.

Figure 4.3 displays a histogram of the trace distances between successive method calls for the eclipse benchmark, the largest and longest-running of the suite, over the Project-typed traces. We omit detailed histograms for the remaining benchmarks for brevity, but we assert that the distribution is similar for all of the benchmarks and trace modes. Note that it is highly left-skewed: the vast majority of successive method calls are clustered within the trace, suggesting that we are justified in our use of a short-sighted window.

	Window Size									
Benchmark	5	10	15	20	**25**	30	35	40	45	50
antlr	95.0	96.5	97.1	97.5	**97.9**	98.2	98.4	98.6	98.7	98.8
bloat	97.9	98.0	98.1	98.2	**98.2**	98.2	98.2	98.3	98.3	98.3
chart	82.4	88.3	100	100	**100**	100	100	100	100	100
eclipse	96.5	97.5	97.6	97.7	**98.0**	98.2	98.2	98.3	98.4	98.4
fop	88.5	89.9	91.0	91.2	**91.3**	91.5	91.7	92.9	93.4	93.8
hsqldb	99.5	99.7	99.8	99.8	**100**	100	100	100	100	100
jython	97.0	98.7	98.9	99.0	**99.1**	99.1	99.4	99.4	99.4	99.5
luindex	88.4	92.3	94.9	96.2	**97.0**	97.5	97.9	98.1	98.3	98.5
lusearch	93.1	94.5	95.5	96.1	**96.3**	96.4	96.5	96.6	96.7	96.8
pmd	97.9	98.1	98.1	98.3	**98.4**	98.4	98.5	98.5	98.5	98.5
xalan	82.5	87.8	90.5	92.4	**93.7**	94.6	95.8	96.2	96.4	96.6

Table 4.1: Percentage of same-object call pairs whose trace distance is less than or equal to various potential window sizes. Traces consist of JDK method calls.

	Window Size									
Benchmark	5	10	15	20	**25**	30	35	40	45	50
antlr	87.4	89.0	90.3	93.5	**94.4**	95.5	96.9	97.1	97.3	97.6
bloat	96.4	96.9	97.2	97.3	**97.4**	97.4	97.5	97.5	97.5	97.6
chart	70.7	77.2	99.7	99.7	**99.8**	99.8	99.8	99.8	99.9	99.9
eclipse	52.9	77.3	82.3	86.2	**88.7**	90.4	92.0	93.1	94.3	95.1
fop	73.7	83.8	84.1	84.5	**85.4**	86.5	87.1	87.4	89.2	90.6
hsqldb	34.9	43.5	97.2	97.5	**97.7**	97.8	98.0	98.1	98.2	98.3
jython	65.9	88.5	90.3	93.3	**94.3**	94.8	95.3	96.1	96.4	96.4
luindex	70.1	75.8	81.8	85.8	**87.8**	88.7	89.0	89.2	89.4	89.5
lusearch	77.1	77.8	78.1	84.8	**97.7**	98.5	98.5	98.6	98.7	98.7
pmd	79.6	81.2	81.8	82.1	**82.5**	82.7	82.9	83.0	83.2	83.2
xalan	81.9	86.7	89.0	90.5	**91.7**	92.5	93.7	94.2	94.5	94.6

Table 4.2: Percentage of same-object call pairs whose trace distance is less than or equal to various potential window sizes. Traces consist of intra-project method calls.

Tables 4.1 and 4.2 contain evaluations of the effectiveness of various potential window sizes for the JDK and Project-typed traces, respectively. For a variety of sizes, we calculate the percentage of pairs of successive method calls that fall at or under the given window size. In other words, these data answer the question "If OCD is configured with the given window size, on what portion of a program's execution could we effectively operate?" Our chosen default window size (25) is emphasized.

These results are general and encouraging. However, they may underestimate OCD's potential. We conducted additional analysis on the distribution of the problematic method calls within the traces, but for space reasons, we elide a full presentation of these experiments in favor of short descriptions.

Application Phases The problematic method calls were *not* distributed uniformly throughout the execution trace: most were concentrated during the startup and shutdown phases of each benchmark. This suggests that our highly dynamic algorithm might perform much better in the common case as it adapts to the common "phase" of execution.

Fully Verifiable Types The problematic method calls were also *not* distributed uniformly throughout all types (Java classes): a majority of JDK types (and a sizable portion of project-specific types) were fully verifiable with a window size of 25; that is, every pair of method calls over these types occurred with fewer than 25 intervening trace events. In addition, the distribution of these fully verifiable types was skewed toward the most frequently used classes; that is, OCD has the potential to perform extremely well on those types whose method calls occurred most frequently in the dynamic traces.

Our sound enforcement of all inferred properties (*cf.* Section 4.2.4) implies that OCD's end-to-end recall—the proportion of all temporal safety violations it finds—rests largely on its ability to effectively learn temporal properties. The inference process is largely constrained by the finite window, but this demonstration of temporal locality suggests that OCD is capable of inferring a large subset of the relevant properties over any given execution.

Finally, note that we have specified a reasonable default window size based on this evaluation. However, it is entirely configurable—even online—and the temporal locality evaluation module is included within OCD itself for project-specific tuning.

4.3.2 Selecting Pattern Templates

In this section, we present OCD's rich suite of default pattern templates. These patterns demonstrate both our tool's power and its generality. We first present three patterns that contain enough expressive power to learn the *phasic specifications*, a general class of typestate specifications we defined while developing the Javert specification miner [Gabel and Su, 2008b]. We then present two patterns that form a dynamic version of *function precondition mining*, which we extend to its dual—operational postcondition mining—with two additional patterns. Two final patterns allow Ocd to be used as dynamic *association rule* miner.

Phasic Specifications In our previous work on the Javert specification miner [Gabel and Su, 2008b], we defined the set of *phasic* specifications and argued that it encompasses a large class of relevant temporal properties in real software projects. Briefly, these specifications can all be expressed as the *composition*—a generalized form of regular language intersection—of instances of the patterns (ab) and (ab^+c). For space reasons, we state without proof that the following patterns sufficiently form an over-approximation of this set:[3]

$$ab \qquad \text{Sequencing} \tag{4.1}$$

$$ab^+ \qquad \text{LoopBegin} \tag{4.2}$$

$$a^+b \qquad \text{LoopEnd} \tag{4.3}$$

Note that Ocd does not actually use these patterns to build larger specifications at runtime; it instead simply learns and enforces these smaller building blocks. Any error that manifests itself in any potentially composed specification also manifests itself as an error in at least one of these smaller specifications, rendering this process safe.

Pre and Postconditions Several recently developed tools have focused on mining *preconditions* in software systems and flagging violations as potential defects. In one particularly relevant example, Ramanathan et al. [2007a] mine "function precedence protocols," which are preconditions of the form "function x is always called before function y." We introduce the following patterns that extend this idea with the logical dual—postconditions, or function *sequence*

[3]We showed previously [Gabel and Su, 2008a] that it was generally impossible to precisely decompose three-letter patterns into a set of two-letter patterns. However, safe *approximations* are possible.

protocols—allowing OCD to function as a general, dynamic implementation of these tools.

$$ab? \qquad \text{PRECONDITION} \tag{4.4}$$

$$a?b \qquad \text{POSTCONDITION} \tag{4.5}$$

$$a^+b^* \qquad \text{GENERALIZED PRECOND.} \tag{4.6}$$

$$a^*b^+ \qquad \text{GENERALIZED POSTCOND.} \tag{4.7}$$

The first two patterns are straightforward, while the second two provide more generalized variants that allow strings of identical calls as preconditions and postconditions.

Association Rule Mining PR-Miner [Li and Zhou, 2005] is a tool that locates potential software defects by learning temporal *association rules* between function calls or variable accesses. An association rule miner infers instances of general temporal association—without a necessary ordering relationship. An example might include the pairing of the methods setHost and setPort on a socket: the two methods are always called together as a pair, but the calling *sequence* does not matter. The following patterns allow our system to learn and find violations of general association rules of method calls.

$$(ab|ba) \qquad \text{ASSOCIATION RULE} \tag{4.8}$$

$$(a^+b^+)|(b^+a^+) \qquad \text{GENERALIZED ASSOC. RULE} \tag{4.9}$$

We also add a generalized variant that allows for sequences of identical calls.

As with the configuration of the window size, the pattern suite is completely configurable: should this suite be insufficient for a particular specialized domain, a developer may add or remove patterns using the standard (academic) regular expression syntax.

4.3.3 Learning Strategies

Recall that a *learning strategy* is a function that decides if a given specification should be enforced. We have previously introduced this concept in terms of the ISENFORCING predicate, which operates over *historical statistics*, namely the raw counts of the number of times the given specification has been satisfied and has failed:

$$\textsc{IsEnforcing} : (\mathit{sat} : \mathrm{int}, \mathit{fail} : \mathrm{int}) \mapsto (\mathsf{true}|\mathsf{false})$$

OCD implements a slight generalization of this function:

$$\textsc{IsEnforcing} : (\mathit{sat} : \mathrm{int}, \mathit{fail} : \mathrm{int}) \mapsto (\textsc{Enforcing}|\textsc{Learning}|\textsc{Dead})$$

The previous values of true and false map to the new values of ENFORCING and LEARNING, respectively. The addition of the third value—DEAD—allows OCD to aggressively *remove* specifications that are showing strong evidence of being irrelevant. These stale specifications (*e.g.*, those that have failed a majority of the time) can cause a performance drain on the system and are generally safe to prune. Note that a given implementation of a learning strategy is not *required* to ever return DEAD; it can be conservatively omitted from the strategy's range, thus preventing any eager pruning.

We also allow learning strategies to be combined through a conservative join (OR) and a more aggressive meet (AND) operator, which closely resemble the AND and OR operations in ternary logic:

	E	**L**	**D**
E	E	E	E
L	E	L	L
D	E	L	D

Join

	E	**L**	**D**
E	E	L	D
L	L	L	D
D	D	D	D

Meet

OCD includes three basic strategies, and its default consists of the "meet" of all three.

Count This strategy operates directly on the satisfying and failing counts, returning ENFORCING if the satisfied count is above a threshold, DEAD if it is below a (different) threshold, and LEARNING otherwise.

Ratio This strategy closely resembles our example ISENFORCING predicate in Section 4.2.1: it calculates the ratio of satisfying instances to total instances and returns DEAD, LEARNING, or ENFORCING based on various constant thresholds.

Context This strategy considers the static *calling contexts* that have accumulated for the current specification. It returns ENFORCING if the specification has recorded at least a certain threshold number of unique calling contexts.

4.3.4 Self-Tuning

The inference of specifications directly from code is an inherently imprecise endeavor, and attempting to automatically *enforce* these specifications only compounds the problem. We have consolidated all of our "fuzzy" reasoning in our learning strategies, which operate implicitly with a multitude of constant "thresholds." Thus far, we have left the values of these constants conspicuously undefined.

These thresholds have a profound impact on our tool's output. When considering just a single constant—the minimum ratio in our RATIO strategy, for example—the extremal value of zero trivially produces an anomaly for *all* instances of *every* specification; the extremal value of one produces no anomalies whatsoever; and other values have the potential to produce any number in between.

A standard approach to setting these types of thresholds is to perform a series of exploratory experiments to find reasonable, apparently general values and evaluate them using a form of *cross-validation*. Unfortunately, we were unable to progress past the first step: seemingly reasonable values that produced a handful of anomalies on one workload would cause a flood of thousands on another.

Our solution to this problem is a *multivariate self tuning* module that allows OCD to actively tune itself to the current execution. The module takes as input an *objective function* and one or more *tunable variables*. The objective function is defined over the reals, and the "optimal" value is defined to be zero.

Objective Function Tools that learn specifications from code often make the assumption that code is *mostly correct*: common behavior represents correct behavior. With this in mind, we expect that an agnostic, dynamic fault detection tool like our own should not generally produce voluminous output. Our standard objective function is thus defined in terms of a user-defined "budget" of expected anomalies, which we currently set to a liberal default of 10:

$$\textsc{ObjectiveFunc} \triangleq \mathit{anomaly\_budget} - \mathit{anomaly\_count}$$

We build the set of tunable variables by programmatically collecting all thresholds and other constants accessed by the currently selected learning strategy.

The self tuner operates by conducting a sequence of experiments that simultaneously 1) attempt to minimize the objective function and 2) reveal the relative "power" of adjusting any given variable in terms of its observed effect on the objective function:

1. Pick a variable v from the set of tunable variables. Increment or decrement the variable's value according to its historical "power." Log this change and the current value of the objective function.
2. Wait for a specified interval or number of events to pass.
3. Observe the new value of the objective function and use the difference to refine our knowledge of v's "power." Repeat from Step 1.

The selection operation ("pick") of the first step is a randomized choice that favors the variables most likely to minimize the objective function. We do not assume the objective function to be stable: the self tuner calculates a variable's current "power" as the mean of its last three observed effects rather than an entire history. We also set the initial values of each constant to conservative values (*i.e.*, values that cause the learning strategy to admit large numbers of specifications).

This simple scheme works remarkably well in practice. It allows OCD to adapt well to programs of different types and sizes, and it greatly improves the tool's usability and general applicability.[4]

[4] We experimented with performance-tuning objective functions as well. As expected, OCD quickly learned that the best way to minimize overhead was to aggressively increase all failing thresholds and prune every single specification.

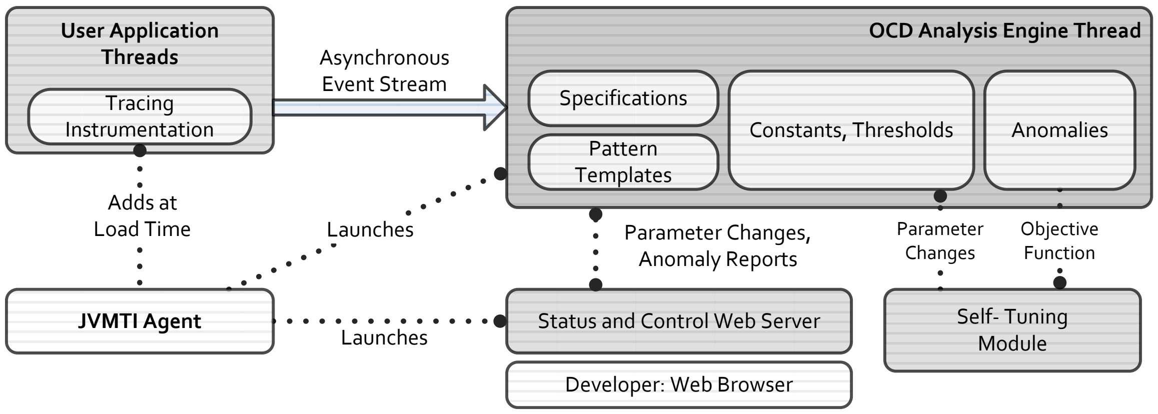


Figure 4.4: Implementation architecture.

4.4 Evaluation

This section describes the implementation and evaluation of OCD. We start with a brief description of OCD's architecture and continue with a brief description of a selection of its notable components. Next, we report on our tool's performance when run against the DaCapo workload, primarily in terms of precision and overhead. We conclude with a selection of experiments on other workloads that highlight OCD's practicality and effectiveness.

4.4.1 Implementation

OCD's high-level architecture is depicted in Figure 4.4. Our system is implemented as a pure-Java *agent* that is invoked by the Sun Java Virtual Machine just prior to the execution of the target application's entry point. At load time, OCD adds tracing instrumentation to the target application, which generates a stream of events. The analysis engine runs separately, decoupled from the target application. We briefly describe a selection of its components.

Tracing Instrumentation We have implemented a flexible tracing library using bytecode instrumentation. At load time, OCD transforms the target application's classes. Our framework is quite general: we have implemented a) both caller and callee tracing, b) the tracing of field accesses, c) the ability to trace static calling contexts for all types of tracing, and d) the ability to filter instrumentation points by signature and access.

Event Stream The tracing instrumentation is added directly to the target application's classes, revealing a potential thread safety issue for multithreaded targets. We solve this with a straightforward decoupling of OCD and the target application: we run the analysis engine in a separate thread that reads from an asynchronous event stream. This solution also allows for a modest amount of parallelism.

Status/Control Web Server For our primary evaluation, our usage of OCD is similar to that of most program analysis tools: we take a predefined workload, add our tool to its configuration, and collect a final report of the results. During development, however, we found more interactivity necessary. OCD embeds a lightweight web server within the target application that allows a) the viewing of the current collection of anomalies and specifications and b) the viewing and *online* mutation of any of its parameters. We expect this feature to become more useful as we explore more specialized uses of OCD, *e.g.* as a debugging tool for diagnosing known-failing test cases.

We performed our first evaluation on the DaCapo workload [Blackburn et al., 2006], a benchmarking suite consisting of several widely-used Java applications.[5] Adding OCD to the suite required no changes to the test harness, which conveniently verified that the benchmark suite continued to produce correct output while instrumented by OCD. We performed our experiments over two types of tracing: 1) tracing of all outgoing calls to Java's standard library and 2) tracing of all project-specific methods declared public. The results of these experiments appear in Figure 4.5.

4.4.2 The DaCapo Workload

In this evaluation, we expected OCD to be largely silent. As well-tested CPU and memory benchmarks with known inputs, we expected the executions to be relatively bug-free—at least on the common code paths that we are limited to as a dynamic analysis. Our goals for these experiments were to 1) verify that OCD effectively learns a wide variety of properties, 2) investigate the error reports, if any and 3) measure our typical overhead.

[5]We used DaCapo version 2006-10-MR2 on Sun's 64-bit Linux Server VM, version 1.6.0_16.

	Specifications			Overhead
Benchmark	Considered	Enforced	**Anomalies**	(factor)
antlr	304	31	0	2.9
bloat	1,632	12	0	5.8
chart	368	4	0	5.1
eclipse	3,272	118	2	3.0
fop	256	2	0	2.5
hsqldb	48	0	0	1.6
jython	960	23	1	2.9
luindex	472	6	0	2.1
lusearch	168	9	0	1.9
pmd	320	11	0	3.8
xalan	464	14	0	4.6

(a) JDK method tracing.

	Specifications			Overhead
Benchmark	Considered	Enforced	**Anomalies**	(factor)
antlr	23,280	380	0	282.5
bloat	50,560	156	1	52.7
chart	1,472	13	0	6.5
eclipse	145,256	898	3	14.5
fop	6,568	100	0	21.0
hsqldb	1,088	24	0	8.2
jython	46,344	81	0	89.8
luindex	2,528	104	0	143.1
lusearch	1,432	30	0	321.9
pmd	30,568	97	0	30.4
xalan	7,824	22	0	31.1

(b) Project-specific method tracing.

Figure 4.5: Results on the “known good” DaCapo suite.

Specifications Throughout the suite, OCD inferred and verified a large number of properties. These included many that were obviously relevant, a sampling of which we display in Table 4.3. In addition to these patterns, our system inferred and verified a significant number of properties that were *not* obviously relevant. This apparent waste of resources is a strength of our technique: we used OCD as an end-to-end anomaly detection tool and did not manually verify any of these properties before they were used. Because they produced no anomalous output, we effectively sidestepped the task of manual validation.

Java Type	**Pattern**	
Enumeration	hasMoreElements()	nextElement()?
Iterator	hasNext()	next()?
StringTokenizer	hasMoreTokens()	nextToken()?
Vector	size()	elementAt(int)?
BufferedReader	readLine()*	close()$^+$
BufferedWriter	write(String)*	close()$^+$
BufferedWriter	write(int)*	close()$^+$
ResultSet	next()*	close()$^+$
ListIterator	hasPrevious()	previous()?
Reader	read(char[],int,int)*	close()$^+$

Table 4.3: A small sampling of JDK-related patterns learned and verified over the DaCapo suite.

Anomalies Our expectations of few error reports notwithstanding, OCD did produce three JDK-related and four project-related anomalies. Despite originating in two different projects, the JDK-related anomalies were all derived from an identical pattern: the precondition relationship between Enumeration.hasMoreElements() and nextElement(). In two cases, the higher-level precondition—that the Enumeration has an element—was satisfied in a different way: by testing using the size method. In the third case, it was not immediately apparent that the enumerated collection contained at least one element on all possible code paths.

Of the four project-specific anomalies, none were either obviously defects or obviously false alarms. We did investigate the two highest-ranked anomalies reported in the Eclipse benchmark and found them to be quite interesting but benign inconsistencies. Both cases were within Eclipse's compiler internals. In the first case, a particular Statement-typed object was processed without first calling complainIfUnreachable. Our investigation revealed that the statement in question was a member of the statement list of the "increment" portion of a for loop. We consulted the Java Language Specification and found that these particular statements must be of type "Expression Statement" and do not need to be individually checked for reachability in this context. For brevity, we omit a detailed description of the second case; it was similar in scope and depth.

These results are encouraging: not only did OCD verify a large number of properties, it also produced very few false reports. The anomalies it *did* generate had intuitive causes, and—especially the project-specific reports—were worth investigating.

Overhead OCD incurs a significant amount of overhead, but it appears currently acceptable for a development-time bug finding tool—especially on the JDK-based experiments. The overhead on the project-specific experiments was much higher and highly variable, though still tractable for this workload, taking minutes instead of seconds per benchmark.[6] We investigated this phenomenon and noted that over the same workload, the project-specific tracing causes nearly an order of magnitude more events to be generated: it appears that the clearest path to significantly less overhead is to reduce the number of instrumentation points. As Java provides the ability to both add and *remove* instrumentation at runtime, something akin to Dwyer *et al.*'s Adaptive Online Program Analysis [Dwyer et al., 2007] would be desirable, though it is yet unclear how to adapt such techniques when the target analysis involves a *learning* component in addition to verification. Finally, we note that other runtime monitoring tools intended for production environments, with overheads in the tens of percents, do not instrument nearly as much of the target program and they do not infer properties.

4.4.3 Bug Finding: Eclipse and Ant

We then ran OCD on the full, latest versions of two production Java applications: Eclipse (a portion of which was already partially exercised by DaCapo) and Ant, a build system. Our goal in these experiments was to reveal defects by providing more variable workloads. We restricted our scope to JDK-based anomalies, as they do not generally require domain-specific knowledge to investigate.

Our test input consisted of performing common tasks with each tool, using our own code base as a dataset. For Eclipse, we a) launched the application and let the project build, b) performed several edits and a renaming refactoring, and c) closed the application. For Ant, we performed two invocations, one with our project's `clean` target and one with the `dist` target, which involved a full compile. We left the default anomaly "budget" (the number of anomalies the self tuner strives for through indirect manipulations of the learning parameters) at its default of 10. We sampled the set of anomalies after each operation.

[6]At present, the overhead of the system has substantially improved over these published figures due to optimizations, with the project-specific values reaching near parity with the JDK values.

Eclipse OCD produced a total of 10 anomalies, unioned across the three sampling points. Of these 10, only three were "false positives" in the truest sense:

1. Two consisted of exactly the same false errors that manifested themselves under the DaCapo Eclipse workload.
2. One was a violation of a clearly false property over two Collection methods. OCD learned it as a result of a common idiom used during Eclipse's initialization; it is likely that the property would have dropped out of the "Enforcing" state with additional input.
3. Three involved minor performance issues relating to the toArray(T[]) method on various Collection types. The violations involved calling this method with a freshly-allocated empty array, a waste of resources. The more efficient idiom—used throughout the majority of Eclipse's code base—is to freshly allocate an array of the appropriate size. (The specific property violated is that size() is a precondition for toArray(T[]).)
4. One was a certain resource leak, in which the contained InputStream of an InputStream-Reader was closed without closing the enclosing instance.
5. Three involved abuses of the InputStream type's interface in which the developers neglected to call close() on instances that they (apparently) knew would be of a concrete subtype whose close() method did nothing.

Ant OCD produced a total of five anomalies between the two sampling points. These consisted of:

1. Three harmless violations of the general has∗, next∗ type specifications.
2. A neglected call to hasMoreTokens() on a StringTokenizer on an unprocessed user string (though the runtime error is eventually handled through an uncaught exception handler, it is somewhat careless).
3. A resource that was closed *late*, by the finalizer thread. Our system reported a "false" error due to the lack of temporal locality in this situation. However, it is almost always preferable to close resources in a timely manner; Dillig *et al.*'s CLOSER project [Dillig et al., 2008], for example, aims to find and fix situations just like this one.

Both Eclipse and Ant were quite usable while under instrumentation. Eclipse was especially responsive: our decoupled design allowed "bursty" actions, like the opening of menus, to be processed on the second core of our dual core test system, which reduced interface lag.

None of the reported anomalies resulted in immediate program crashes: each defect-indicating anomaly either caused an inconsistent program state or hinted at different conditions—namely, other inputs—under which the anomaly would have resulted in a crash. However, crashing bugs are not outside OCD's scope. If a program crash is the result of a violation of a temporal property, then OCD will likely report its root cause.

4.4.4 Generality: Associated Field Accesses

Existing tools that search for inconsistent *field* accesses, *e.g.* MUVI [Lu et al., 2007], have demonstrated impressive results. As an exercise in the generality of our tool, we performed an informal experiment of our tool's ability to find these kinds of bugs. For this experiment, we used the FindBugs project as a workload[7] (not as an analysis tool) and set our tracing framework to log all field writes. To detect general inconsistent accesses, we used our "Association" and "Generalized Association" patterns (Section 4.3.2).

Our tool produced five anomalies, all of which were highly domain-specific. However, we were able to fully investigate one of them: the inconsistent updating of a size field in a data structure. OCD had detected an association between this field and another field that were always updated in tandem. However, in the clear() method, the fields were not updated consistently: the size field was *not* cleared to zero, leaving the structure in an inconsistent state. This defect has been confirmed and fixed.

4.5 Discussion and Related Work

Our algorithm is the first dynamic algorithm that simultaneously learns and verifies temporal properties. The most closely related work can be roughly categorized in three groups: specification inference, runtime monitoring, and the detection of software anomalies.

[7]Due to its complexity and ease of configuration with batch-mode inputs, we utilized FindBugs as our "benchmark" workload throughout OCD's entire development. To avoid an obvious threat to validity, we have omitted it from our standard evaluation but use it here for convenience—with a different form of tracing.

Specification Inference Ammons et al. [2002] produced the seminal work on specification mining. Their algorithm uses a language inference technique to learn a single, general specification over a known alphabet. OCD requires no input beyond the monitored program. Dallmeier *et al.*'s ADABU [Dallmeier et al., 2006] extracts specifications as finite automata with labeled states, which improves their usefulness. In our case, such improvements are not necessary: our properties are used mechanically without human validation. Acharya et al. [2007] present a static tool that extracts patterns as partial orders. Our precondition patterns capture the idea of a partial order, allowing our tool to learn and find violations of these patterns dynamically. Le Goues and Weimer [2009] present a specification miner that leverages a statistical model to drastically reduce the incidence of false specifications. In our experience, most dynamically-mined "false" specifications describe "true" but useless properties, which are not a problem for our fully automatic tool. However, integrating a technique like this could serve to reduce OCD's overhead.

More recently, Nguyen *et al.* present a new algorithm for mining specifications over *multiple* objects [Nguyen et al., 2009]. As configured, OCD learns patterns over single objects; however, it is not an inherent limitation: if the tracing framework could assign the same identifier to multiple related objects, OCD could possibly learn and enforce multi-object patterns without modification. We are investigating this line of improvement as ongoing work.

Thummalapenta and Xie present a technique for learning specialized instances of specifications for exceptional code paths [Thummalapenta and Xie, 2009], on which OCD's property *inference* performance is possibly poor. We are investigating ways to overcome this inherent limitation of dynamic analysis, including possibly leveraging additional information in the static code to augment our traces.

Runtime Monitoring Runtime monitoring frameworks, such as Chen and Roşu's MOP [Chen and Rosu, 2007], have seen dramatic improvements in performance in recent years. Often with the help of static information [Bodden et al., 2008, Dwyer and Purandare, 2007, Gopinathan and Rajamani, 2008], these tools can verify properties in production programs with overhead in the low tens of percents. Our problem domain is somewhat different: OCD must automatically infer properties as well as enforce them. Nonetheless, we are working toward leveraging these insights to improve Ocd's performance: we do, for example, have access to at least some static code when we perform instrumentation at load time.

Dwyer et al. [2008] improve the performance of runtime monitoring systems by breaking larger specifications into smaller "sub-alphabet" properties and monitoring a sampled subset to create an approximate verifier. This suggests an interesting avenue for investigation: an empirical evaluation of the end-to-end effectiveness of our tool when verifying only a subset of our smaller patterns, which resemble their "sub-alphabet" properties.

Detecting Anomalies The general idea of characterizing software bugs as anomalous program behavior was codified by Engler et al. [2001]. Hangal and Lam's DIDUCE [Hangal and Lam, 2002] hypothesizes and learns invariants over program values, much like Daikon[Ernst et al., 2000], and includes a component that checks the learned invariants as well. In some sense, our work is like DIDUCE, but our domain consists of *temporal* invariants. Chang *et al.* present a tool that mines program dependence graphs for neglected conditions [Chang et al., 2007], like missing null checks. Our tool is effective at finding neglected conditions that are sufficiently abstracted as method calls.

Elbaum *et al.* investigate the ability for anomalies in execution traces to predict field failures [Elbaum et al., 2007]. They measure the effectiveness of various anomaly detection algorithms, with their "sequence" variant appearing similar to our own—but at a much finer granularity. In our system, the anomalies are caused by violations of inferred temporal safety specifications, which themselves *are* a form of field failure.

The tool perhaps most related to our own is Wasylkowski *et al.*'s JADET [Wasylkowski et al., 2007], a static tool for finding general object usage anomalies. JADET uses concept analysis to infer properties that are *nearly* always satisfied and it reports the failures as anomalies. This technique is complementary to our own: Ocd learns more general properties with higher precision, but as a dynamic tool it has a limited view of the target program.

4.6 Conclusion and Future Work

We have presented the first online algorithm that simultaneously learns and enforces temporal properties. Our implementation, OCD, functions on production Java applications with acceptable overhead and is effective in learning and validating a large number of important properties.

Many of the properties we learn and verify are from standard, well-tested libraries. While convenient for validating our technique, these properties are effectively finite in number and perhaps not necessarily the best targets for a fully automatic technique: it is conceivable that they *could* be semi-automatically specified once, perfected, and shared for all tools to use. Instead, we believe that the greatest strength of our online tool is the learning and enforcement of *project specific* properties, which are likely being created—perhaps incidentally—faster than they can be specified. The primary obstacle to validating and improving our tool for this purpose, though, is that the time of domain experts is finite and expensive. To this end, we are working with our industrial partners to validate and improve OCD by evaluating it on commercial enterprise systems with full access to domain experts.

5 Deductive Specification Inference

Specifications are necessary for nearly every software engineering task, but they are often missing or incomplete. “Specification mining” is a line of research promising to solve this problem through automated tools that infer specifications directly from existing programs. The standard practice is one of inductive learning: tools make observations about software and inductively generalize them into specifications. Inductive reasoning is unsound, however, and existing tools commonly grapple with the problem of inferring “false” specifications.

In this work, we explore and define the conditions necessary for *deductive* inferences about specifications: a well-defined *design space* and a notion of *correctness*. We extract these concepts into a general framework, Deductive Specification Inference (DSI), which we implement in a practical tool that extracts temporal specifications from Java programs. Our tool treats a “fact” about a program as a *hypothetical specification* and attempts to prove its *necessity* and *importance* systematically. It explores a design space through automated program transformations, and it evaluates correctness through testing.

5.1 Introduction

Nearly all software engineering tasks require some form of a specification. Implementation, debugging, and testing, for example, all involve reconciling a software program’s specified and actual behaviors. Documentation and source code comments are standard sources of specifications, but they are often incomplete, incorrect, or missing entirely. Worse yet, time-saving software tools—our research focus—require formal, machine-readable specifications, which are even rarer.

Research in *specification inference* aims to solve this problem through tools that automatically reverse engineer specifications directly from programs. Although reasoning soundly about specifications from implementations is generally impossible, the intended behavior of a stable software project can be somewhat evident. Leveraging meta-heuristics like “frequently-observed behavior in a program is likely intended and important”, specification “mining” tools can make observations about software and generalize them into specifications, albeit with a degree of uncertainty.

That “degree of uncertainty” is a central issue in specification inference. A mining tool’s performance is always an empirical question and generally cannot be taken for granted. The programmer thus becomes responsible for validating each inferred specification manually, an expensive and error-prone process. The unreliability of specification inference also hinders the development of derivative tools, like a hypothetical “advanced debugger” that would both infer specifications and localize and fix related faults.

The fundamental cause of imprecision in specification inference is the standard *problem of induction*: generalizing from examples is unsound. Accordingly, current tools are at their most effective when they broaden their learning base by using as many examples as possible [Gruska et al., 2010] and, similarly, when they focus on widely-used specifications [Acharya et al., 2007]. In our own recent work, we have attempted to scale specification inference “down” and “out” to project-specific and highly “semantic” properties, which studies suggest are the largest contributors to software problems [Li et al., 2006]. Unfortunately, we have encountered what we are informally calling a “precision wall”: the deeper (and more domain-specific) an inference tool looks, the less evidence it tends to have to work with.

In this chapter, we present a new specification inference methodology that avoids many of the pitfalls of a purely inductive approach. Rather than generalizing a specification solely from an example program, our approach involves defining, exploring, and experimenting within a program’s *design space*. Specifications directly describe a design space, so—when performed carefully—this methodology can allow one to make sound, *deductive* inferences about specifications.

While evaluating our prior work on specification inference, we were often faced with the task of manually determining if an inductively-learned specification was “real and important” or a “false positive”. Deductive Specification Inference (DSI) is a formalization and automation of that process. Briefly, DSI is the process of treating a "fact" about a program as a *hypothetical* specification and experimentally testing whether or not it is is *necessary* for correct execution. In our implementation, we use a combination of automated program transformations and testing.

This chapter includes the following contributions:

1. A new specification inference methodology, Deductive Specification Inference (DSI).

2. An implementation of our framework for a well-studied domain: temporal specifications over the sequences of method calls in a program. This tool serves as a novel specification inference tool as well as a case study for the DSI method.

3. A case study demonstrating our tool's effectiveness on several open source Java programs.

The next section (Section 5.2) provides an overview of our approach through a set of examples. Section 5.3 then presents the Deductive Specification Inference framework in detail. We then present the design and implementation of a tool implementing DSI for the domain of temporal properties of method calls (Section 5.4). Our experimental results and related discussions follow in Section 5.5. In Section 5.6 we discuss DSI in the context of related work, and in Section 5.7 we conclude with a discussion of future work.

5.2 Overview

In this section, we provide an overview of our general approach through a series of examples. We begin with an introduction to our target domain, temporal specifications, and continue with a presentation of our general technique as well as our implementation.

5.2.1 Temporal Properties and Their Inference

The examples in this section are drawn from the domain of *temporal specifications over program elements*. Here, "temporal" refers to the span of runtime execution and "program elements" refers to executable code. A temporal specification extends traditional state assertions ("variable `x` is always positive") with the notion of time ("once `x` is positive, `y` will eventually become positive as well").

```java
CompilerTest test = ...
test.reset();
/* Set up 'prog' variable */
test.execute(prog, out, err);
```

a. "Call `CompilerTest.reset` at some point before calling `CompilerTest.execute`."

```java
ResourceAttributes attr = ...
/* Other setup */
attr.setArchive(true);
attr.setSymbolicLink(false);
```

b. "`ResourceAttributes.setArchive` and `ResourceAttributes.setSymbolicLink` must appear in sequence."

```java
GeneratorAdapter gen = ...
/* Set up 'type' and 'c'
    variables */
gen.loadThis();
/* Other 'gen' invocations */
gen.invokeConstructor(type, c);
```

c. "Call `GeneratorAdapter.loadThis` at some point before calling `GeneratorAdapter.invokeConstructor`."

```java
SaveManager sm = this;
/* Other state restoration actions */
try {
  sm.restoreMarkers(resource, true, p);
  sm.restoreSyncInfo(resource, true, p);
} catch (Exception e) { /* Ignore */ }
```

d. "`SaveManager.restoreMarkers` and `SaveManager.restoreSyncInfo` must appear in sequence."

Figure 5.1: Four observed temporal properties and a selection of the Java source code that generated them.

One commonly studied class of temporal properties involves ordering restrictions on function calls. Functions are building blocks of software projects, and the order in which they are composed is both critical and subtle—especially in imperative and object-oriented systems with side effects. Common examples include locking disciplines, in which a specification might state "calls to methods `lock` and `unlock` on each `Lock` object strictly alternate at runtime" and resource usage rules, in which a partial specification might state "one should call `close` on a file descriptor soon after its final use".

Temporal properties are often much more domain-specific and subtle than these canonical "locking" and "resource" examples, and they are rarely fully documented. Researchers have recognized this problem and developed automated software tools capable of "mining" temporal properties directly from programs. The predominant models are forms of *inductive learning*. Many tools operate similarly in two high-level steps: 1) *observing* (at runtime or statically approximating) the behavior of a program and 2) *generalizing* that behavior into a specification.

5.2.2 Validating Specifications

Figure 5.1 lists four examples of "potential" temporal specifications. They were synthesized from observations of real software projects, simplified excerpts of which are listed as well. Mining tools may report specifications like these for several reasons, including:

- The observed property is satisfied (or mostly so) by the observed program. This condition is often trivially true.
- The tool observes the property frequently, with examples occurring frequently at runtime or within the static source code. This encodes the belief that "common behavior is likely to be correct."
- Assorted heuristics. For example, the property listed in Figure 5.1a involves a method named `execute`, which may match a "function name filter" that identifies naming patterns that have often been important in the past.

Ultimately, a specification mining tool takes an inductive leap, essentially "lifting" observations into specifications based on prior beliefs.

"Potential" specifications may not be *true*, though, which is a natural consequence of inductive learning. When a programmer is presented with a mined specification, he or she must generally *validate* and/or debug it before it becomes useful. Approaches include:

- Code inspection. If the specification is not followed, would it lead to an obvious error?
- Reconciling with known requirements. Is the specification clearly (in)consistent with existing specifications?
- Consulting with experts and past software engineering data. Have the elements of this specification been involved in any prior issues?

Note the lack of a complete and algorithmic solution. This is precisely what makes specification inference difficult in practice and impossible in the limit. However, these validation techniques do follow a common theme: they involve using disparate sources of information to answer the following question as accurately as possible:

Given a potential specification φ, is φ necessary for my program's correct execution?

Our current work can be framed as a method for solving this problem as completely and automatically as possible.

```java
CompilerTest test = ...
//test.reset();
/* Set up 'prog' variable */
test.execute(prog, out, err);
test.reset();
```

a. "Call `CompilerTest.reset` at some point before calling `CompilerTest.execute`."

```java
ResourceAttributes attr = ...
/* Other setup */
//attr.setArchive(true);
attr.setSymbolicLink(false);
attr.setArchive(true);
```

b. "`ResourceAttributes.setArchive` and `ResourceAttributes.setSymbolicLink` must appear in sequence."

```java
GeneratorAdapter gen = ...
/* Set up 'type' and 'c'
     variables */
//gen.loadThis();
/* Other 'gen' invocations */
gen.invokeConstructor(type, c);
gen.loadThis();
```

c. "Call `GeneratorAdapter.loadThis` at some point before calling `GeneratorAdapter.invokeConstructor`."

```java
SaveManager sm = this;
/* Other state restoration actions */
try {
  //sm.restoreMarkers(resource, true, p);
  sm.restoreSyncInfo(resource, true, p);
  sm.restoreMarkers(resource, true, p);
} catch (Exception e) { /* Ignore */ }
```

d. "`SaveManager.restoreMarkers` and `SaveManager.restoreSyncInfo` must appear in sequence."

Figure 5.2: Transformed programs that should now be "wrong" if each specification is "real" or "necessary."

5.2.3 Automated Validation and Deductive Inference

Returning to the running examples, consider now the contrapositive of the "validation problem":

If I violate φ, will my program be incorrect?

Phrasing the question this way suggests an experimental solution. Figure 5.2 reprises the potential properties listed in Figure 5.1, but the code excerpts have now been transformed. For the domain of temporal properties, we have a strong idea of what it means to "violate" a specification, and in each case the code has been "minimally" and straightforwardly modified to violate each property. If each of the potential specifications is *true*, then each program in Figure 5.2 should now be *wrong*.

The problem now reduces to judging each "experiment" as "correct" or "wrong." If we were able to judge any as being correct—despite being transformed—we could say with some certainty that the associated specification is unnecessary for correct execution and thus *false*. Similarly, if one of those programs were now incorrect, we would obtain evidence that the associated specification is necessary and *true*. Note the lack of the word "certainty" in the latter case: it is rife with subtlety and will be discussed in more detail throughout this chapter.

Judging a program "correct" or "wrong" is generally impossible, of course, and to do so actually begs the question of a complete specification. However, correctness can often be approximated through testing and analysis, giving us the final component we need to automatically (but approximately) validate specifications. Our high-level technique is as follows:

1. Start with a proposed specification φ from a program P. For temporal function-call specifications, this might be of the form "calls to function `a` always precede calls to function `b`".
2. Create a space of *experimental programs* around P and φ, an explicit design space populated with programs "like" P but violating φ. We accomplish this through automatic program transformations. Continuing the earlier example, this space may consist of the family of programs in which calls to `a` and `b` are reordered.
3. *Test* the design space by testing the experimental programs. If φ is found to be unnecessary for correctness, then φ is not a specification.

On our example properties, this automated process is quite revealing.

- The experiment in Figure 5.2a crashes early: `reset` does in fact set up the precondition for `execute` to run; the specification is true.
- Experiment 5.2b passes: the order in which these two fields are set is irrelevant.
- Experiment 5.2c fails, but *not* with an immediate crash: it ultimately causes operations much later in the test suite to fail. `GeneratorAdapter` is a helper class within a Java bytecode library. Not following this specification will actually result in the generation of bytecode that violates the Java Bytecode Specification, which is what ultimately causes the later test failure.
- Experiment 5.2d passes, but perhaps surprisingly so: each operation contains a substantial amount of overlapping side effects. From a class-level perspective, though, the tests demonstrate that the observed ordering is irrelevant.

Transforming this specification *validation* procedure into a specification *inference* algorithm is straightforward, as validation and inference are fundamentally the same problem. In our case,

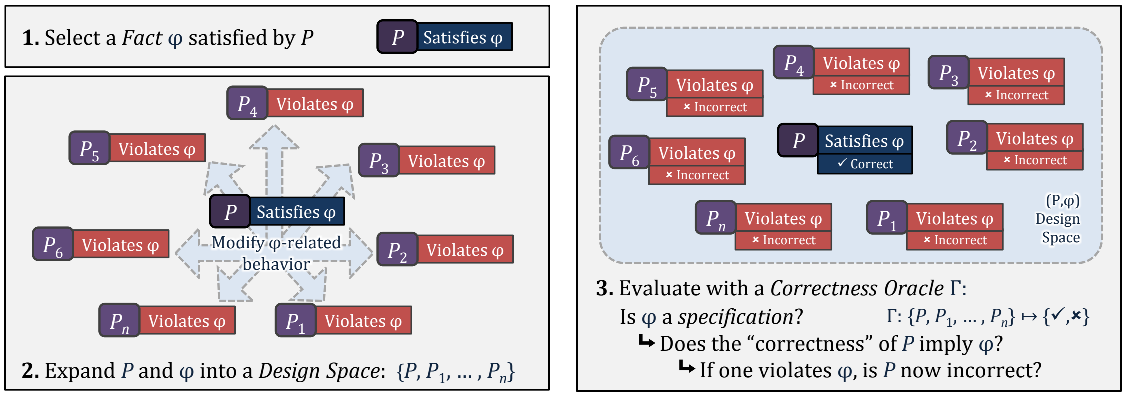

1. Select a *Fact* φ satisfied by *P*
P Satisfies φ
2. Expand *P* and φ into a *Design Space*: {*P*, *P*1, … , *P*n}
Modify φ-related behavior
P4 Violates φ
P5 Violates φ
P3 Violates φ
P6 Violates φ
P2 Violates φ
Pn Violates φ
P1 Violates φ
✓ Correct
× Incorrect
(P,φ) Design Space
3. Evaluate with a *Correctness Oracle* Γ:
Is φ a *specification*? Γ: {P, P1, … , Pn} ↦ {✓,×}
↳ Does the "correctness" of *P* imply φ?
↳ If one violates φ, is *P* now incorrect?

Figure 5.3: The Deductive Specification Inference process. Figure 5.4 presents the interpretation of the results.

we can define a domain of specifications and systematically enumerate and validate each one . For temporal specifications, we could simply enumerate every observed-true ordering of function calls and systematically validate each one. Alternatively, we could bootstrap the process with an inductive learning algorithm.

This section has provided an overview of what we mean by "Deductive Specification Inference" and how we implement it for the domain of temporal function-call properties. The following section presents the idea of Deductive Specification Inference more completely and domain-agnostically. It explores the most relevant question: how sound are these "deductions"? In other words, under what conditions do these "experiments" truly prove or disprove the existence of a specification?

5.3 Deductive Specification Inference

Deductive Specification Inference (DSI) is a framework for reasoning about specifications. It involves the use of a *correctness oracle* to reason about experimental, "alternate-world" programs within a well-defined *design space*. Figure 5.3 provides an overview of the DSI process, and in this section we present and discuss it in detail.

5.3.1 Problem Statement

DSI provides a method for inferring specifications from an *example program*. Our high-level problem statement is as follows:

> Given a program P and a fact φ satisfied by P, determine whether or not φ is a *specification*.

We purposefully leave the domain of φ unspecified. Note that P can be generalized to a finite set of programs, but we focus on the simpler case of one single example program.

A minimal criterion for a program fact to be considered a program specification is *necessity*. Introduced in the preceding section, a fact is necessary when any "violations" of it lead to an incorrect program. Systematically evaluating necessity thus requires two components: 1) a method of enumerating alternate, possibly "violating" programs and 2) a method of judging programs correct or incorrect. Within DSI, one accomplishes the former by explicitly defining a *design space* around a given program, which we discuss next.

5.3.2 The Design Space

A specification describes a constraint or requirement on a family of programs. Inductive specification inference reasons about that "program family" through generalization. If we wish to reason *deductively* about that family of programs (and thus the specification) we must define it explicitly.

A DSI instance must first define a design space.

Definition 11 (Design Space)**.** A *design space* is an enumerable set of programs derived directly from P that 1) varies the *behavior governed by* φ and 2) leaves *non-*φ *behavior* invariant.

This definition, though quite abstract, captures the idea of enumerating the "family" of φ-related programs that coincide with P's intended behavior. Evaluating the correctness of programs within this space constitutes *controlled experiments* that test the necessity of φ, and evaluating it *completely* can allow us to make deductions about specifications. This concept is illustrated in the left frame of Figure 5.3: a design space is "grown" around P by modifying P's φ-related behavior.

Remark. DSI inferences are sound with respect to a definition of a design space. The more faithfully a design space adheres to Definition 11, the more “valid” from a software engineering perspective the deductively inferred specifications become.

A perfect and exhaustive design space is challenging to define and implement in practice. We will discuss our experiences with temporal function-call properties in the next section (Section 5.4). Here, we present several higher-level conceptual requirements that follow from Definition 11.

Complete The design space should contain examples that disrupt φ-related behavior in every place it applies. For example, if φ describes a temporal restriction between two function calls, the design space should include programs that “break” each use of the two functions.

Intent-Preserving, or Controlled One must violate φ in a way that otherwise preserves the original intent of the program. In other words, the design space should focus expressly on *φ-related behavior*. If φ specifies, say, a temporal relationship between functions `a` and `b`, a sensible design space should not include examples in which the function calls are simply deleted: φ restricts the ordering of the function calls, not their presence or absence.

Limited in Power, or Realistic Experimental programs in the design space should reflect what a programmer might create. They should avoid, for example, modifications of the host language’s runtime or other artificial faults. We are ultimately trying to deduce specifications that programmers should care about, ones that they may violate in practice. If our reasoning is based on a design space that a programmer would never have encountered in practice, then the inferred “specifications” are irrelevant.

Beyond these requirements lies a great amount of flexibility. In our implementation, we explicitly generate a finite design space through automatic program transformations and evaluate it using traditional testing. However, nothing in this definition prevents one from implicitly defining an *infinite* design space and reasoning about it statically.

5.3.3 A Correctness Oracle

The design space provides us with a set of experimental programs in which our proposed fact φ is the independent variable. In order to *run* these experiments, we need a method for judging the correctness of programs. We factor this task into an unspecified *correctness oracle*.

Definition 12 (Correctness Oracle)**.** A *correctness oracle* Γ is a deterministic procedure for evaluating the correctness of a program P.

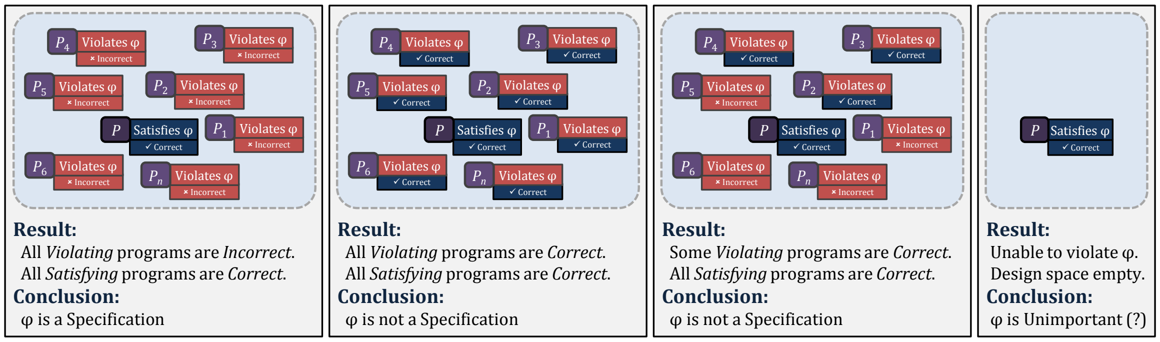
This requirement would superficially seem to be unrealistic and onerous. Reasoning about programs at this level is not mathematically possible, and even if it were, a true oracle would require a complete specification and thus negate any true usefulness of DSI.

In practice, incomplete and approximate oracles are effective. For one, an incomplete oracle is entirely effective as long as it correctly evaluates the portion of P explored in the design space, *i.e.* the φ-governed behavior. Other unsound or incomplete oracles may also be used, albeit with the addition of a (controllable) degree of uncertainty to the overall DSI process. And in our experiments we use an oracle as weak and incomplete as standard testing, and we are able to soundly *invalidate* thousands of potential specifications.

Remark. Recall from our overview (*cf.* Section 5.2) our discussion of the methods that human programmers use to validate specifications. In essence, they draw on numerous sources of existing software engineering knowledge to confirm or deny a specification. DSI emulates this process in an automated way, and the correctness oracle serves as a singular input point for all “existing knowledge”. DSI can be viewed, in a sense, as a method for “converting” software engineering knowledge embodied in one form into another. In our implementation, we “convert” the knowledge stored in standard test oracles into correct and useful temporal specifications.

5.3.4 The Complete Framework

Deductive Specification Inference is the process of determining if a fact φ that is true of a program P is a *specification*. At this point, we have defined 1) a design space around P and φ and 2) a means of evaluating it. The final step, which completes the framework, is the actual execution of these experiments, which is depicted in the right frame of Figure 5.3.


P_4 Violates φ × Incorrect
P_3 Violates φ × Incorrect
P_5 Violates φ × Incorrect
P_2 Violates φ × Incorrect
P Satisfies φ ✓ Correct
P_1 Violates φ × Incorrect
P_6 Violates φ × Incorrect
P_n Violates φ × Incorrect
P_4 Violates φ ✓ Correct
P_3 Violates φ ✓ Correct
P_5 Violates φ ✓ Correct
P_2 Violates φ ✓ Correct
P Satisfies φ ✓ Correct
P_1 Violates φ ✓ Correct
P_6 Violates φ ✓ Correct
P_n Violates φ ✓ Correct
P_4 Violates φ ✓ Correct
P_3 Violates φ ✓ Correct
P_5 Violates φ × Incorrect
P_2 Violates φ ✓ Correct
P Satisfies φ ✓ Correct
P_1 Violates φ × Incorrect
P_6 Violates φ × Incorrect
P_n Violates φ × Incorrect
P Satisfies φ ✓ Correct

Result:
All *Violating* programs are *Incorrect.*
All *Satisfying* programs are *Correct.*
Conclusion:
φ is a Specification

Result:
All *Violating* programs are *Correct.*
All *Satisfying* programs are *Correct.*
Conclusion:
φ is not a Specification

Result:
Some *Violating* programs are *Correct.*
All *Satisfying* programs are *Correct.*
Conclusion:
φ is not a Specification

Result:
Unable to violate φ.
Design space empty.
Conclusion:
φ is Unimportant (?)

Figure 5.4: Interpreting the results of a DSI run.

This process need not follow any specific algorithm. In our implementation, our design space consists of *explicit* programs; we simply enumerate and test each one. If one, say, *implicitly* defines a design space in some form like "the transitive closure of the following program transformations" and uses a static analysis as a correctness oracle, it is conceivable that one could evaluate entire groups of experiments simultaneously and efficiently.

Interpretation of the results is straightforward. Figure 5.4 enumerates the possible cases. In the first case (left) we have validated φ as a specification: the implication "$\textit{correct} \Rightarrow \varphi$" is satisfied. Every "violating" program in our design space is an incorrect version of P. (Note that the outcomes of the "satisfied" experiments are technically irrelevant to the implication. In practice, P itself is our only "satisfied" experiment and it is trivially correct.)

The inference of φ as a specification is technically sound—with respect to the design space, that is. From a *software engineering* perspective, however, problems with the design space may later manifest themselves as potentially defective specifications. These problems mirror the design principles we outlined earlier.

- An *incomplete* design space may result in a specification that lacks generality.
- An *uncontrolled* design space may cause truly false specifications to be inferred. This is the standard problem of determining causality through controlled experimentation, and in our setting it stems from our requirement that the design space explore *only* φ-related behavior. If φ is inextricably linked to some critical non-φ aspect of P, it may be impossible to create a controlled experiment that can precisely link φ and correctness.
- An *unrealistic* design space may result in technically true but unimportant specifications.

In the second case (Figure 5.4, second from left) we have completely invalidated φ as a specification: every program, "broken" or otherwise, is correct. We have a variation on the same outcome in the third case (Figure 5.4, second from right). Here, some but not *all* of our experimental "violating" programs are judged correct. This case highlights the fact that specifications are universal: to invalidate a proposition φ as a specification we need only locate as evidence *one* correct program that violates φ. This strict interpretation is intuitive, especially in our implementation. An experiment that shows an observation to be irrelevant to correctness is strong, programmer-comprehensible evidence.

That interpretation notwithstanding, there is an opportunity here for DSI to provide more information to the user. It is plausible that individual application domains may find approximate inferences useful. These could take the form "violating φ leads to failure approximately 50% of the time", for example.

The final case (Figure 5.4, right) is an anomaly that is surprisingly useful in practice. When a well-designed DSI instance—and thus a programmer—cannot violate a proposed specification in any reasonable way, that specification is likely to be unimportant *even if strictly true*.

An example will clarify this point. Our implementation works with the domain of temporal function-call specifications and explores the design space by (sensibly) reordering function calls. Consider the following code snippet:

```
public String getResult() {
  return this.calc.compute();
}
```

A simple inductive inference tool may observe that `getResult` and `compute` always occur in sequence at runtime and may present the relationship as a specification. Note, though, that the structure of the program prevents any sensible violation. A programmer could not violate it, and our DSI instance would produce an empty design space. A programmer would quickly dismiss this specification as false for the very same reason that DSI would "fail" to (even begin to) prove it true.

This problem of "false specifications caused by one function calling another" has been addressed in the literature through purpose-built heuristics, such as the "control-flow artifact filter" in Yang *et al.*'s Perracotta tool [Yang et al., 2006]. DSI's natural "failure to violate" elegantly generalizes and handles this and many other cases of "spurious specifications" without the need for heuristics.

In summary, Deductive Specification Inference provides a means for deductively reasoning about specifications. Its strength is drawn from explicit consideration of aspects of the specification inference problem that are traditionally defined only implicitly and in general terms: the meaning of a *design space* and the definition of *correctness*.

Finally, recall that DSI's purpose is to provide a decision procedure for determining whether or not a single fact is a specification. To transform this procedure into a full specification inference algorithm, we need only to combine it with some method of enumerating facts from programs.

5.4 Deductively Inferring Temporal Properties

We have implemented Deductive Specification Inference for the domain of *temporal function-call properties* of imperative and object-oriented systems. We realize a *design space* through the use of automated program transformations and we implement a *correctness oracle* using testing. We introduced the temporal function-call problem domain earlier (*cf.* Section 5.2.1) and continue here in more depth.

5.4.1 Temporal Function-Call Properties

We address a common class of specification: *ordering restrictions on function calls* within a software project. These specifications are common and error-prone, as they are not enforced by the type systems within standard compilers. When they are defined formally, however, advanced software tools can check them statically [Ball and Rajamani, 2001, Fink et al., 2008] or at runtime [Chen and Rosu, 2007], preventing and eliminating errors.

The formalism we use to represent specifications is *regular languages*. While the most general formalism for expressing these properties is some form of a temporal logic, many important properties can be expressed as simple regular languages. The earlier examples of "locking" and "resource disposal" are both regular: (`lock unlock`)* and (`read`* `close`), respectively. Each specification is quantified over a domain of possible "scenarios," which is a general way of capturing the notion that the properties only restrict *related* function calls, *e.g.* `lock` and `unlock` calls on the "same `Lock` object" or `read` and `close` calls on the "same file descriptor."

Our tool is implemented for programs written in the Java programming language. For simplicity we focus on temporal properties of function calls on a *single object*; that is, our domain of "scenarios" is the set of objects at runtime. Note that we use receiver objects solely as a convenient and reliable way of relating sets of method calls through data. No aspect of our implementation is fundamentally restricted to object-oriented systems.

Remark. A related concept is *typestate* [Strom and Yemini, 1986], the notion that individual types have a high-level "state" that dictates when certain operations (*e.g.* method calls) are "legal," *i.e.* when they do not violate an *internal* class invariant. Because we focus on single-object properties, it is tempting to view our implementation of DSI narrowly as a form of "typestate inference." Our technique certainly *will* infer typestate properties (the two examples above fall into this category) but it is far more general. By using "overall system correctness" as an oracle rather than "obeys a preexisting local class invariant", our tool can (and frequently does) infer interesting domain-specific properties like that shown in Figure 5.1c: the class does not crash when used incorrectly—no local class invariant is violated—but it eventually causes a violation of a higher-level *system* specification.

5.4.2 Temporal Function-Call DSI

Our temporal function-call implementation of Deductive Specification Inference takes as input:

1. a Java program, and
2. a regular temporal fact, or "hypothetical specification," over a set of function calls.

It returns as output:

1. "specification", or
2. "not specification" (along with supplemental details).

The high-level architecture mirrors that of the abstract DSI framework and appears in Figure 5.5.

Execution occurs in two phases: *Preamble* and *Experiment*. The first involves constructing a set of "experiments" by defining a *temporal property design space*. We implement this process

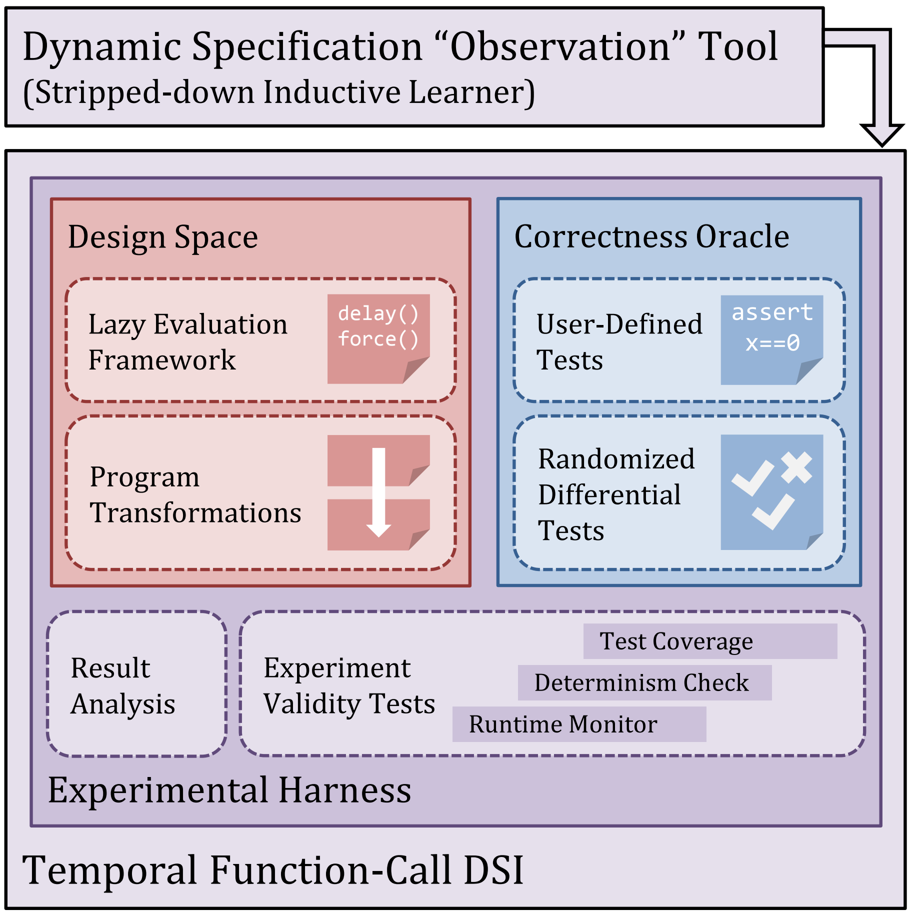

Dynamic Specification "Observation" Tool
(Stripped-down Inductive Learner)
Design Space
Lazy Evaluation Framework
delay()
force()
Program Transformations
Correctness Oracle
User-Defined Tests
assert x==0
Randomized Differential Tests
Result Analysis
Experiment Validity Tests
Test Coverage
Determinism Check
Runtime Monitor
Experimental Harness
Temporal Function-Call DSI

Figure 5.5: Implementation architecture.

using static program transformations that make use of a novel lazy evaluation framework for Java programs. The second phase is dynamic, executing and evaluating these experiments with respect to a *correctness oracle*, which we approximate through testing.

Generating a set of temporal facts to validate for a given program is straightforward (Figure 5.5, top). We make use of an existing dynamic, inductive specification inference tool [Gabel and Su, 2008a] that has been stripped of its "inductive" characteristics. This specification "observation" tool returns unfiltered results and uses no frequency data or other heuristics to limit its scope.

5.4.3 Preamble: The Temporal Property Design Space

Our temporal properties describe ordering relationships between sets of function calls. Our implementation of a *design space* consists of transforming the input program, in various ways, so that all relevant function calls are reordered in a way that violates the proposed fact.

```
GeneratorAdapter gen = ...
/* Set up 'type' and 'constr' variables */
gen.loadThis();
/* Other 'gen' invocations */
/* Possibly crossing procedure boundaries */
gen.invokeConstructor(type, constr);
```

a. Original source code.

```
Thunk t; // Global, known location
GeneratorAdapter gen = ...
/* Set up 'type' and 'constr' variables */
t = delay({gen.loadThis();});
/* Other 'gen' invocations */
/* Possibly crossing procedure boundaries */
gen.invokeConstructor(type, constr);
force(t)
```

b. Transformed source.

Figure 5.6: The essence of our transformation. Delaying the first invocation until the second has executed violates the hypothesized property.

Transforming Java Programs Reordering function calls in real software projects is problematic. Highly local cases are simple: if two function calls appear on subsequent lines, for example, their parameters tend to draw from the same variable scope, simplifying the actual transformation. High locality makes for more controlled experimentation as well. In our experience, though, actual projects tend to form a complex and rigid "scaffolding" that is difficult to modify—and the most important and subtle properties are likely to be those that are not confined to a pair of sequential lines of code.

We solve this problem by implementing a robust *lazy evaluation framework* for Java programs. It brings to Java the concept of *promises* in eager functional languages like Scheme. The entry point is analogous to the `delay()` primitive in Scheme, but slightly generalized: given an arbitrary sequence of (straight-line) Java bytecode, our framework 1) functionally abstracts it and 2) creates a closure with its (eagerly-bound) parameters, thus converting it into a *thunk*. This object may then be executed at any time, immediately or later, by an analogue of Scheme's `force()` primitive.

Lazy evaluation greatly simplifies the task of reordering function calls. Our transformation occurs at the bytecode level, but it maps conceptually well on to source code. An example

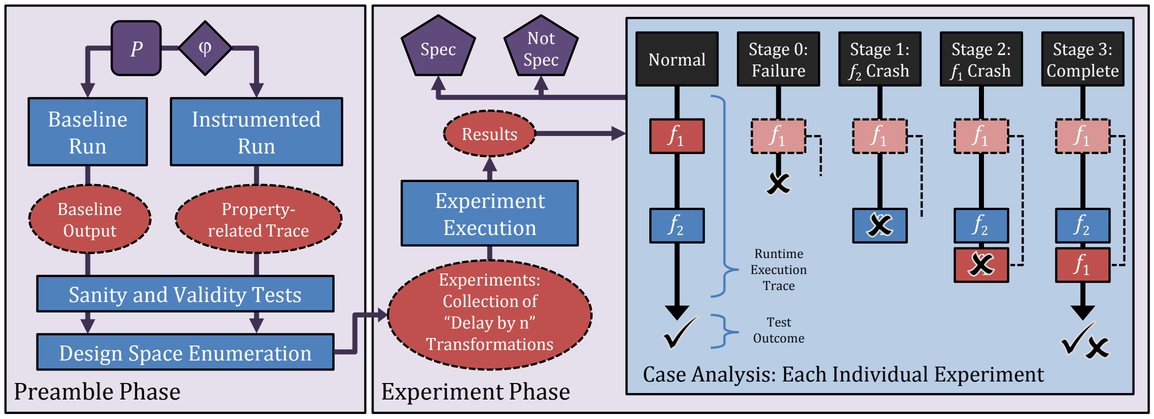


Figure 5.7: The complete DSI process for temporal function-call properties.

of the essence of our transformation appears in Figure 5.6. The higher level operation in the figure is "delay the first function call until point p". This operation is the basis of all of our transformations, and the remaining questions are *when* and *where* to apply it.

Selecting Experiments Recall two of our guidelines for an ideal design space: *completeness* and *control*. We respect completeness by generating test programs that violate each potential binding of the property at runtime, which in our case means violating each individual runtime object affected by the property. We maintain control by doing so in as minimally intrusive manner as possible. That is, for any given usage "scenario", we wish to delay the minimum number of function calls by the minimum amount of (execution) time necessary to violate the property.

We are able to generate this minimal set of transformations by proactively collecting information about the runtime behavior of the program. Figure 5.7 provides an overview of the complete DSI process; the left pane depicts this Preamble phase. Before creating the design space, we execute the program's test cases (the same tests that will be used during the Experiment phase) twice: once unmodified and once "instrumented", which collects a *property-related trace*. We use this trace along with a straightforward brute-force algorithm to generate our minimal set of transformations. In addition, these "pilot runs" allow us to perform various sanity checks, including:

- Is the property exercised in these test cases?

- Is the property satisfied by the program?
- Is the program's behavior deterministic enough to allow experimentation? Note that this requirement is not as strict as it sounds: we use a flexible form of execution indexing that tolerates a great amount of variation in program behavior.

The execution of these initial runs, tests, and experiment selection algorithms form the entirety of the Preamble phase. At its conclusion, we have produced a set of transformed experimental programs that are ready to be evaluated.

Remark. The primitive "delay a call until time p" is powerful: for example, *all* functions can be delayed and re-emitted in any order. In theory, though, this "delay primitive" is not powerful enough to "break" (*i.e.* transform to rejecting) every accepting string of any arbitrary (non-total) regular language. Fortunately, in practice, every valid trace of every regular *specification pattern* we have encountered in our work can be "broken" by delaying just a *single* event, albeit by varying amounts of time. We have performed a more thorough theoretical investigation of this problem, which we have omitted for brevity.

Other Implementation Notes One complication to our otherwise simple process is caused by the presence of explicit *return values* from the functions we "delay:" if a function is not evaluated, we cannot know what it will return. We solve this problem by implementing a model of what a *novice programmer* would do in this situation, inspired by our principle of "realistic" experiments. We implement a simple type analysis that allows us to replace the return value of a delayed call with the value of the "nearest-defined local variable of the appropriate type" (or the language-defined default value if one is not found). This simple process works well in practice. This general definition automatically captures many intuitive actions, including reusing the return value from a previous call (among others).

Multithreaded programs caused complications as well. Avoiding any single-threaded assumptions handled most issues, but our early experiments revealed several fundamental challenges. For one, we may "move" a function call to a program point at which an important lock is no longer held. To solve this issue, our lazy analysis framework makes note of the locks held when `delay()`ed and attempts to reacquire them, if necesary, when the thunk is `force()`ed. In another case, the delayed call was indirectly responsible for some event that, if omitted, would cause the second call to block indefinitely, creating a deadlock. In this case, we instituted a global "inactive timeout" on our experiments: if the subject program makes no forward progress after a period of time, we forcibly terminate the program.

Remark. Testing for "necessity for correctness" is superficially similar to testing for control or data *dependencies*, a heavily-studied subject. Lack of any control or data dependence is *sufficient* cause to invalidate a specification, but it is far from *neccesary*.

5.4.4 Experiment: Testing with a Correctness Oracle

The Experiment phase is conceptually simple: we evaluate each experiment in the suite of transformed programs and interpret the results.

Testing as an Oracle Our implementation of a correctness oracle is *testing*. This portion of our tool is pluggable to allow the use of user-defined tests and test oracles. In addition, we also provide a default implementation based on randomized regression testing. We first run the unmodified, assumed-correct program on random inputs. We record the input/output behavior on this inputs as a *behavioral profile*, which then becomes our test oracle. This process is similar to Differential Testing [Evans and Savoia, 2007], which uses automatically generated tests to test modified software for regressions.

Analysis of Results At a high-level, the only important output is the success or failure of each individual test. However, exactly *how* a test executes and fails can be useful knowledge. Regardless of the property—the precise relationship it defines or the number of functions it references—the experimental process reduces to delaying a single function call until a second call completes. This simplicity allows us to analyze a finite set of cases that may arise during execution; these cases are depicted in the right pane of Figure 5.7.

- *Normal:* This is the standard, unmodified case for reference. Function f_1 executes, followed by f_2, which is followed by normal execution.

- *Stage 0:* f_1 has been delayed, but execution failed before reaching f_2. The experiment failed at a very fundamental level: we could not violate this property using our standard program transformations. Interpretation of this case could fall in either direction, but there is a strong case here to be made for “not a specification” by way of the “empty design space” case discussed in the previous section (Section 5.3).

- *Stage 1:* Execution fails while executing f_2. This is indicative of a real specification, but the circumstances also suggest additional information: f_1, the delayed call, appears to directly or indirectly establish f_2’s *precondition*.

- *Stage 2:* Execution fails while forcing the execution of f_1. Once again, this is evidence of a real specification, but it also reveals that f_2 puts the program in a state in which f_1 cannot safely execute. An example might include f_1 involving the “use” of a resource and f_2 “closing” it.

- *Stage 3:* The experiment fully completes and execution continues as in the normal case. If the oracle decides the program has maintained its correctness, then we have strong evidence that the fact is “not a specification.” If the program (eventually) fails, then we have revealed what may be a particularly *subtle and important* true specification—one that if violated *silently* puts the system in an undefined error state.

Other Implementation Notes Software testing is naturally incomplete. In the cases in which we *invalidate* specifications, a weak test oracle (*i.e.* one that fails to identify incorrect programs) may cause us to draw an incorrect conclusion. More plainly, *any* experiment over a part of a program not covered by a test suite will lead to our tool concluding “not a specification.” We avoid the more egregious cases of this problem in practice by 1) inferring our observed facts from the *same* test suites we use for evaluation and 2) running our sanity checks during the Preamble phase to ensure the property-related behavior is exercised. Note, though, that these checks do *not* guarantee that the test suite is perfect.

Our implemented tool is robust, scalable, and general. In the following section, we present the results of a case study of our tool on real, widely-used Java projects.

5.5 Case Study: Specifying the DaCapo Suite

This section presents the results of a case study of our Deductive Specification Inference tool for Java programs. We have sought to answer the following research questions.

1. Is our tool robust? Does it function on complex, real-world software?
2. What are the characteristics of a deductively inferred specification "space" on a set of real-world Java programs?
3. What are the strengths and limitations of our tool, and how do they translate to the Deductive Specification Inference methodology as a whole?

We continue with a discussion of our experimental setup, which is followed by a presentation and analysis of our results.

5.5.1 Experimental Setup

Our test subjects are a set of Java programs drawn from the DaCapo benchmark suite (version 9.12-Bach). DaCapo differs from traditional "microbenchmarks" in that it is formed from real, widely-used applications to create realistic workloads. The benchmarks are listed in the left column of Figure 5.8 and comprise approximately 1.5 million lines of source code.

DSI requires "facts" to treat as hypothetical specifications. As we noted earlier, we modified an inductive specification learning tool [Gabel and Su, 2008a] to serve as a "fact observer." We configured it to find all instances of *simple sequences*: pairs of method calls that appear to be obeying a sequential ordering restriction between them. For brevity, we omit the implementation and experimental details of this process.

Remark. The "sequence" template is simple, almost to the point of being *simplistic*. In practice, it captures a surprisingly broad amount of specification behavior. Our previous work on the Javert tool [Gabel and Su, 2008b] has demonstrated that almost all temporal function-call specifications can be fundamentally decomposed into simple sequences and small loops. In addition, note that our current tool is not inherently limited to simple patterns, even at the current implementation level: it can work with any regular specification pattern over any number of distinct functions.

Benchmark	Facts	Specs	Non-Specs	Mean Time/Fact (s)
avrora	460	72	388	47.1
batik	2063	159	1904	69.4
eclipse	1426	145	1281	225.9
h2	497	57	440	102.8
jython	493	29	464	936.4
luindex	463	74	389	35.1
lusearch	167	23	144	91.4
pmd	911	50	861	79.1
sunflow	196	49	147	34.1
xalan	1172	79	1093	76.0
Total	7848	737	7111	

Figure 5.8: Summary of results on the DaCapo benchmark suite.

We ran all experiments in parallel on several 64-bit Linux servers, each configured with with Intel processors (Xeon and Core 2) and the 64-bit Oracle Java Virtual Machine, Server Edition, version 1.6.0_25.

5.5.2 Results

A summary of our results appears in Figure 5.8. Our tool individually analyzed 7,848 facts, systematically judging each as a *specification* or *non-specification*. In each case our tool performed robustly, both validating and invalidating specifications within large, complex software projects.

Performance Performance was acceptable: the majority of facts were analyzed in under two minutes. Our task is embarrassingly parallel as well, a fact we utilized fully in our study. Notable exceptions were the Jython and Eclipse benchmarks. Jython and Eclipse contain many "facts" whose violation hinders termination, forcing our system to often wait until a conservative timeout had expired (five minutes) before proceeding. In practice, this timeout can be reduced to a value more appropriate to a particular project.

Non-Specifications Figure 5.9 lists detailed results of our tool's inferences on the 7,848 input facts. The first group of four columns describes the facts our tool judged as *non-specifications*.

The first column ("Could Not Violate") counts facts that yielded a truly empty design space and were judged to be "unimportant". Many of these cases were a result of the "control-flow

	Non-Specifications				Specifications			
Bench-mark	Could Not Violate	Stage 0: Failure	Stage 3: Complete	Total	Stage 1: f_2 Crash	Stage 2: f_1 Crash	Stage 3: Complete	Total
avrora	16	91	281	388	18	2	52	72
batik	190	543	1171	1904	27	5	127	159
eclipse	357	297	627	1281	25	45	75	145
h2	19	96	325	440	18	2	37	57
jython	135	129	200	464	10	8	11	29
luindex	8	125	256	389	10	32	32	74
lusearch	16	55	73	144	3	6	14	23
pmd	116	152	593	861	12	1	37	50
sunflow	4	79	64	147	13	0	36	49
xalan	102	278	713	1093	18	6	55	79
Total	963	1845	4303	7111	154	107	476	737

Figure 5.9: Detailed results of our case study. Each entry is a count of the number of "facts" our tool judged to fall into the given column's category.

artifact" specifications described earlier, but there were several other cases as well that were elegantly captured by our tool's "cannot-violate implies non-importance" principle. For example, one case involved the type system preventing us from moving an object's constructor call after its first true method call, the more abstract principle here being "ordering restrictions involving constructors cannot be violated by programmers."

The second two columns correspond to "Stages" of execution described earlier and depicted in Figure 5.7. In the second column ("Stage 0"), every experiment in each fact's design space resulted in a program crash soon after we "delayed" the first function call. Stage 0 results generally fall into two categories:

1. The fact appears to be fully enforced at runtime and is thus impossible to violate on tested code.

2. There is a more specific and relevant fact we should be analyzing instead. For example, if functions `a`, `b`, and `c` all execute in sequence at runtime and a crucial relationship exists between `a` and `b`, we will be unable to run a successful experiment involving `a` and `c`. In our implementation, we test *every* possible fact and would eventually analyze the crucial `a/b` fact, so discarding the `a/c` as a "non-specification" loses no information.

The third column lists those facts whose experiments all fully completed, violating the property, but continued to be judged correct by the relevant tests. The experiments serve as a form of "certificate" demonstrating that the fact is truly unnecessary for correctness. Note the sheer volume of this category: 55% of the hypothesized facts were invalidated—every one of which had the potential to be called a "specification" by an inductive learning tool and to waste a programmer's time. We were surprised and encouraged by the fact that this level of interference and experimentation was possible in large and complex software projects.

Specifications The second group of columns in Figure 5.9 contains counts of the facts our tool judged to be *specifications*. Each column corresponds to a "stage" of execution described earlier and depicted in Figure 5.7. The first two columns of this group correspond to fairly standard "typestate" specifications:

1. Facts whose experiments all end in Stage 1 exhibit a *precondition* relationship: the first function call establishes (a part of) the precondition of the second, and delaying it causes the appropriate crash.

2. Facts whose experiments all end in Stage 2 exhibit a *state transition* relationship: the first function call was legal in its original context, but it becomes illegal after the second function executes (*e.g.* attempting to use a resource after it has been closed).

More interesting are the facts whose experiments all fully completed (Stage 3). In these cases, each experiment *silently* corrupted the program state and caused the tests to fail at a later—sometimes much later—time. A small, randomly sampled collection of these specifications follows:

- In Lucene, delaying a "commit" operation on an index until after it is closed corrupts the index but causes no overt failure.

- In H2, a connection information object will silently return a bogus password hash from a connection information if read before initialization is complete.

- Various XML parsers in the projects require handlers to be set before parsing begins and will not warn the user if none has been set (our sample included two of these).

- In Sunflow, the `SunflowAPI` object's initialization is idiomatic. Moving any part of it until after a call to `render` causes incorrect output but no obvious crash. Two other examples in our sample followed a similar "idiomatic initialization" pattern.

We are continuing to analyze these results in depth, but thus far this case study has suggested that this application of Deductive Specification Inference is a general and effective technique.

5.5.3 Discussion

Two salient points to discuss are the *generality* and *validity* of these deductively inferred specifications. Both issues relate to our definition of a design space for temporal specifications.

Generality concerns the applicability of the specifications to programs *outside* our (approximated) design space. If we infer a relationship between two functions, must every *future* use of those two functions necessarily follow it? If our design space is adequate then yes, the specification is certainly general. However, if it is not, our tool essentially reduces to an *inductive* tool (albeit one with a new kind of concrete evidence: a demonstration of necessity on the current program). Nonetheless, we do believe our temporal property design space to be effective in practice, especially for the more esoteric and project-specific specifications. As specifications become more targeted toward a single software project, the "universe" of possible usages likely shrinks as well. The entire "universe" of a project-specific property might simply be the current code base, and our design space covers programs "like" the current code base quite well.

Validity concerns our ability to establish causality between a fact and correctness. In practice, it can be hindered by a lack of *control* in our design space. If we violate a fact φ and the program fails, can we be sure that it was precisely the φ-specified behavior that caused the failure? In general, the level of control within a DSI instance is a continuum: specifications naturally overlap and no part of a program is truly isolated. However, we believe that the specific domain of temporal properties naturally lends itself toward controlled experimentation: the *ordering* between function calls can be modified independent of whether or not the functions execute *at all*. And this concern notwithstanding, the process of *invalidating* specifications is sound—at least with respect to the tests.

5.6 Related Work

This section discusses Deductive Specification Inference in terms of several lines of related work.

Inductive Specification Inference Inductive specification inference techniques have been developed for a variety of domains. Kremenek *et al.* have presented a framework [Kremenek et al., 2006] based on probabilistic inference that reflects the essence of the inductive process: leveraging beliefs about software to infer general specifications from specific examples of programs. Several targeted techniques infer temporal specifications, the subject of our own implementation of DSI. Both dynamic [Ammons et al., 2002, Dallmeier et al., 2006, Gabel and Su, 2008b, Yang et al., 2006] and static [Shoham et al., 2007a] techniques follow the same general approach: they observe temporal relationships in programs and inductively elevate them to specifications. Other successful application domains of inductive specification inference are assertions over program state [Ernst et al., 2000, Henkel and Diwan, 2003], determinism specifications for concurrent programs [Burnim and Sen, 2010], and function contracts [Ramanathan et al., 2007b]. Less formal approaches include lightweight "programming rules" [Chang et al., 2007, Li and Zhou, 2005], which have been particularly effective at revealing programming mistakes.

These techniques all confront one central issue: precision. Generalization is unsound, and the inductive leap from one program to a specification about an entire class of programs is essentially an educated guess. This issue is not merely theoretical: a recent study [Polikarpova et al., 2009] has shown that one third of inductively-learned code contracts are incorrect or irrelevant (and in our experience, the temporal property domain can be worse). As a result, specification inference research tends to contain an empirical component evaluating precision, and specific techniques have also been proposed to help programmers debug [Ammons et al., 2003] and filter [Le Goues and Weimer, 2009] erroneous mined specifications.

Deductive Specification Inference provides a new way of approaching the specification inference problem. Rather than *speculating* about an entire class of programs, our framework advocates defining it *explicitly* through our notion of a design space. When that is possible, one can make sound, deductive inferences about specifications. And even when a design space must be approximated for practical reasons, DSI provides a sound framework for *invalidating* mined specifications. Lastly, we note that DSI and inductive techniques are not fundamentally opposed: inductive techniques are ideal sources of "facts" that can be more thoroughly (albeit expensively) explored through DSI.

Non-Inductive Specification Inference Some specification inference techniques are non-inductive. Work on extracting *component interfaces* [Alur et al., 2005, Henzinger et al., 2005, Whaley et al., 2002] is superficially similar to inductive specification inference techniques, but the processes are more well-defined and involve no inductive generalization. These techniques require low-level specifications as input (or they are extracted from explicit assertions in the code). They then solve the formal problem of extracting a sound, higher-level "model" of component usage that avoids violating any of the given low-level specifications. In essence, DSI takes this process and lifts it to entire programs. In place of a model of a specific component, we define a design space for a complete program, and in place of low-level specifications, we leave a placeholder for any form of correctness oracle.

Combining Testing with Specification Inference Our implementation of DSI leverages testing to enhance specification inference, the general idea of which was originally proposed by Xie and Notkin [Xie and Notkin, 2004]. Dallmeier *et al.*'s Tautoko tool [Dallmeier et al., 2010] also uses testing in a similar way. Tautoko's problem setup is similar to that of the "component interface" tools described earlier: generate a component "model" that avoids errors. In Tautoko's case, the component is a Java class that is assumed to crash or otherwise raise an error if used incorrectly. Tautoko starts with an inductively inferred model, but it then enhances it in a feedback loop by generating targeted, exploratory tests. There is a parallel here to DSI: we start with an inductively learned specification and essentially "generate tests" to validate it.

Experimenting with Software and Evaluating Necessity Our implementation of DSI uses experimentation to infer properties of programs. A similar idea has been used by Renieris *et al.* in their study of *elided conditionals* [Renieris et al., 2004]. In their work, the authors generate experiments that test the whether or not the outcome of a conditional statement (*i.e.* a branch) affects the outcome of a test, much as we generate experiments that test the necessity of proposed specifications.

Automatic parallelization tools make use of *Commutativity Analysis* [Aleen and Clark, 2009, Rinard and Diniz, 1997], which evaluates the necessity of a given ordering of program statements. This is similar to how we test the "necessity" of a temporal ordering constraint. Commutativity analysis uses a far stricter criterion than DSI does in practice: that various orderings produce semantically *identical* results. In our implementation of DSI, the fact that two functions commute is only one of several reasons a specification might be invalidated, and the fact that two functions do *not* commute does not imply a given ordering is necessary to correctness.

Our work is also similar to Mutation Testing [DeMillo et al., 1978, Hamlet, 1977]. In mutation testing, modified programs ("mutants") are used to experimentally test the validity to *test suites*. In our work, we use programs in our design space, which are similar to mutants, to experimentally test the validity of potential *specifications*.

5.7 Future Work and Conclusion

This chapter has presented Deductive Specification Inference (DSI), a new methodology for inferring specifications from programs. DSI's novelty lies in the explicit definition of a *design space* of programs to which a potential specification might apply. Combining this notion with a *correctness oracle* allows deductive reasoning about specifications. We have implemented DSI for the domain of temporal function-call properties of Java programs. Our implementation creates a design space by making use of fully automated program transformations, and it approximates a correctness oracle through traditional software testing. In a case study, we demonstrated that our tool is effective on real-world programs.

Our most immediate future work involves implementing Deductive Specification Inference for other domains to further demonstrate the strengths of the method. Our early results in this area have been promising.

We are also interested in exploring other aspects of the definition of importance of a specification. We presently define "important" conservatively: if a programmer can feasibly violate a specification it is important. Collecting more data on what it means for a specification to be "important" might allow us to synthesize other effective and *testable* definitions like this "ability to violate" concept introduced here.

Finally, we believe that deductively mined specifications are potentially strong and precise enough in practice to be used directly by other software tools—without intervention by a human programmer. We are exploring the construction of vertically-integrated derivative tools that leverage this newly strengthened specification inference approach to automate other software engineering tasks.

6 Conclusion

Intelligent Program Analysis is a term we have coined to describe the work in this dissertation. "Intelligent" relates to a certain level of insight we believe all the tools and algorithms exhibit—especially Chapter 4's Ocd—and we believe this to be a fundamental difference that defines a new research area, one we will pursue aggressively as we continue our research.

Traditional program analysis tools extract what we call "first-level facts" about programs. For example, verification-oriented tools calculate the values a program "may" or "must" compute. As close to perfect as these tools may become, they are still just a part of a process dominated by expensive and potentially error-prone human labor. There is an analogy here to security research: flaws in the ways secure systems are *used* are much more common than flaws in the (well-researched) systems *themselves*, like, say, implementations of solid cryptography algorithms.

In the previous program verification example, a programmer is trying to accomplish a distinct *task*: to find (or verify the absence of) errors in a program. But the verification tool cannot work on its own: the programmer must 1) *feed it knowledge* (in this case, comprehension of the program in the form of specifications to verify) and 2) *act on its results* (in this case, to fix any bugs that may be blocking verification). Both of these steps are critical and arguably as important as the verification tool itself, but *automating* them is quite difficult, as they so frequently rely on the intuition and experience of developers—concepts that do not fit well in our framework of computability.

Our results demonstrate that it is possible to automate some developer-oriented tasks, however. The Javert (Chapters 2–3) and Ocd (Chapter 4) systems, for example, successfully automate specification inference (the "feeding of knowledge" step above) so adeptly that it becomes effectively invisible to the developer: when scanning a program, Ocd directly presents the programmer with *domain-specific bugs*. It is this behavior—this autonomy—that led us label it Intelligent.

If program analysis is about exploring and interpreting a program, then Intelligent Program Analysis is about exploring and interpreting a program and its design space—what it does and what it intends to do, a concept we explored in Chapter 5. Program analysis works to retrieve facts; an Intelligent Program Analysis retrieves facts and turns them into *knowledge*, while inferring intent and possibly *acting*.

Our current research results have led us to envision a number of concrete ideas, including—among several others—tools capable of finding *design bugs*; tools that leverage inferred designs to provide intuitive, root-cause-oriented *bug fixes*; and tools that recognize *emergent designs* within software before developers are even cognizant of them. This is just a sampling of what we believe will be a defining characteristic of our continuing research: unprecedented levels of automation in software development.

Bibliography

Mithun Acharya, Tao Xie, Jian Pei, and Jun Xu. Mining API Patterns as Partial Orders From Source Code: From Usage Scenarios to Specifications. In *ESEC-FSE '07: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 25–34, 2007. ISBN 978-1-59593-811-4. doi: {http://doi.acm.org/10.1145/1287624.1287630}. Cited on pages 34, 55, 85, and 89.

Hiralal Agrawal and Joseph R. Horgan. Dynamic Program Slicing. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, pages 246–256, 1990. ISBN 0-89791-364-7. doi: {http://doi.acm.org/10.1145/93542.93576}. Cited on page 47.

Farhana Aleen and Nathan Clark. Commutativity Analysis for Software Parallelization: Letting Program Transformations See The Big Picture. In *Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ASPLOS '09, pages 241–252, 2009. doi: {http://doi.acm.org/10.1145/1508244.1508273}. Cited on page 116.

Rajeev Alur, Pavol Cerný, P. Madhusudan, and Wonhong Nam. Synthesis of Interface Specifications for Java Classes. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 98–109, 2005. ISBN 1-58113-830-X. Cited on pages 5, 54, and 115.

Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining Specifications. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming*

Languages, pages 4–16, 2002. ISBN 1-58113-450-9. doi: {http://doi.acm.org/10.1145/503272.503275}. Cited on pages 5, 28, 33, 36, 43, 46, 54, 85, and 114.

Glenn Ammons, David Mandelin, Rastislav Bodík, and James R. Larus. Debugging Temporal Specifications with Concept Analysis. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming Language Design and Implementation*, pages 182–195, 2003. ISBN 1-58113-662-5. doi: {http://doi.acm.org/10.1145/781131.781152}. Cited on pages 54 and 114.

Thomas Ball and Sriram K. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. In *SPIN '01: Proceedings of the 8th International SPIN Workshop on Model Checking of Software*, pages 103–122, 2001. ISBN 3-540-42124-6. Cited on pages 5, 33, and 101.

Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to Analysis Using BDDs. In *Proceedings of PLDI*, pages 103–114, 2003. ISBN 1-58113-662-5. doi: {http://doi.acm.org/10.1145/781131.781144}. Cited on pages 15 and 22.

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 *Proceedings of OOPSLA '06*, oct 2006. Cited on pages 69 and 79.

Eric Bodden, Patrick Lam, and Laurie Hendren. Finding Programming Errors Earlier by Evaluating Runtime Monitors Ahead-of-time. In *Proceedings of SIGSOFT '08/FSE-16*, 2008. ISBN 978-1-59593-995-1. Cited on page 85.

Beate Bollig and Ingo Wegener. Improving the Variable Ordering of OBDDs Is NP-Complete. *IEEE Trans. Comput.*, 45(9):993–1002, 1996. ISSN 0018-9340. doi: {http://dx.doi.org/10.1109/12.537122}. Cited on page 15.

Randal E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986. ISSN 0018-9340. Cited on pages 6, 14, and 45.

Jacob Burnim and Koushik Sen. DETERMIN: Inferring Likely Deterministic Specifications of Multithreaded Programs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, ICSE '10, pages 415–424, 2010. doi: {http://doi.acm.org/10.1145/1806799.1806860}. Cited on page 114.

Ray-Yaung Chang, Andy Podgurski, and Jiong Yang. Finding What's Not There: A New Approach to Revealing Neglected Conditions in Software. In *Proceedings of ISSTA '07*, 2007. ISBN 978-1-59593-734-6. Cited on pages 86 and 114.

Feng Chen and Grigore Rosu. MOP: An Efficient and Generic Runtime Verification Framework. In *Proceedings of OOPSLA '07*, pages 569–588, 2007. Cited on pages 59, 67, 85, and 101.

Hao Chen and David Wagner. MOPS: An Infrastructure for Examining Security Properties of Software. In *CCS '02: Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 235–244, 2002. ISBN 1-58113-612-9. doi: {http://doi.acm.org/10.1145/586110.586142}. Cited on page 5.

Christoph Csallner and Yannis Smaragdakis. Dynamically Discovering Likely Interface Invariants. In *Proceedings of the International Conference on Software Engineering*, pages 861–864, 2006. ISBN 1-59593-375-1. doi: {http://doi.acm.org/10.1145/1134285.1134435}. Cited on pages 30 and 56.

Valentin Dallmeier, Christian Lindig, Andrzej Wasylkowski, and Andreas Zeller. Mining Object Behavior with ADABU. In *WODA '06: Proceedings of the 2006 International Workshop on Dynamic Systems Analysis*, pages 17–24, 2006. ISBN 1-59593-400-6. doi: {http://doi.acm.org/10.1145/1138912.1138918}. Cited on pages 55, 85, and 114.

Valentin Dallmeier, Nikolai Knopp, Christoph Mallon, Sebastian Hack, and Andreas Zeller. Generating Test Cases for Specification Mining. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 85–96, 2010. ISBN 978-1-60558-823-0. doi: {http://doi.acm.org/10.1145/1831708.1831719}. Cited on page 115.

Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: Path-sensitive Program Verification in Polynomial Time. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 57–68, 2002. ISBN 1-58113-463-0. doi: {http://doi.acm.org/10.1145/512529.512538}. Cited on pages 5, 33, and 59.

R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer*, 11(4):34–41, 1978. doi: {10.1109/C-M.1978.218136}. Cited on page 116.

Isil Dillig, Thomas Dillig, Eran Yahav, and Satish Chandra. The CLOSER: Automating Resource Management in Java. In *ISMM '08: Proceedings of the 7th International Symposium on Memory Management*, 2008. ISBN 978-1-60558-134-7. Cited on page 83.

Matthew B. Dwyer and Rahul Purandare. Residual Dynamic Typestate Analysis Exploiting Static Analysis: Results to Reformulate and Reduce the Cost of Dynamic Analysis. In *Proceedings of Automated Software Engineering (ASE)*, 2007. ISBN 978-1-59593-882-4. Cited on page 85.

Matthew B. Dwyer, Alex Kinneer, and Sebastian Elbaum. Adaptive Online Program Analysis. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, 2007. ISBN 0-7695-2828-7. Cited on page 82.

Matthew B. Dwyer, Madeline Diep, and Sebastian G. Elbaum. Reducing the Cost of Path Property Monitoring Through Sampling. In *ASE: Automated Software Engineering*, 2008. Cited on page 86.

Sebastian Elbaum, Satya Kanduri, and Anneliese Andrews. Trace Anomalies as Precursors of Field Failures: an Empirical Study. *Empirical Softw. Engg.*, 12(5), 2007. ISSN 1382-3256. Cited on page 86.

Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *SOSP '01: Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, pages 57–72, 2001. ISBN 1-58113-389-8. doi: {http://doi.acm.org/10.1145/502034.502041}. Cited on pages 5, 19, 34, 36, 54, and 86.

Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly Detecting Relevant Program Invariants. In *Proceedings of ICSE*, pages 449–458, 2000. ISBN 1-58113-206-9. doi: {http://doi.acm.org/10.1145/337180.337240}. Cited on pages 30, 56, 86, and 114.

Robert B. Evans and Alberto Savoia. Differential Testing: A New Approach to Change Detection. In *The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers*, pages 549–552, 2007. ISBN 978-1-59593-812-1. doi: {http://doi.acm.org/10.1145/1295014.1295038}. Cited on page 107.

Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective Typestate Verification in the Presence of Aliasing. *ACM Trans. Softw. Eng. Methodol.*, 17(2), 2008. ISSN 1049-331X. Cited on pages 59 and 101.

Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. In *Proceedings of PLDI*, pages 234–245, 2002. ISBN 1-58113-463-0. doi: {http://doi.acm.org/10.1145/512529.512558}. Cited on pages 5 and 33.

Mark Gabel and Zhendong Su. Symbolic Mining of Temporal Specifications. In *Proceedings of ICSE '08*, 2008a. Cited on pages 34, 36, 39, 44, 55, 73, 103, and 109.

Mark Gabel and Zhendong Su. Javert: Fully Automatic Mining of General Temporal Properties from Dynamic Traces. In *Proceedings of SIGSOFT '08/FSE-16*, 2008b. ISBN 978-1-59593-995-1. Cited on pages 60, 72, 73, 109, and 114.

Madhu Gopinathan and Sriram K. Rajamani. Enforcing Object Protocols by Combining Static and Runtime Analysis. In *Proceedings of OOPSLA '08*, 2008. ISBN 978-1-60558-215-3. Cited on page 85.

Natalie Gruska, Andrzej Wasylkowski, and Andreas Zeller. Learning from 6,000 Projects: Lightweight Cross-Project Anomaly Detection. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 119–130, 2010. ISBN 978-1-60558-823-0. doi: {http://doi.acm.org/10.1145/1831708.1831723}. Cited on page 89.

R. G. Hamlet. Testing Programs with the Aid of a Compiler. *IEEE Trans. Softw. Eng.*, 3(4): 279–290, 1977. doi: {10.1109/TSE.1977.231145}. Cited on page 116.

Sudheendra Hangal and Monica S. Lam. Tracking Down Software Bugs Using Automatic Anomaly Detection. In *Proceedings of ICSE*, pages 291–301, 2002. ISBN 1-58113-472-X. doi: {http://doi.acm.org/10.1145/581339.581377}. Cited on pages 30, 56, and 86.

Johannes Henkel and Amer Diwan. Discovering Algebraic Specifications from Java Classes. In *ECOOP 2003 - 17th European Conference on Object-Oriented Programming*, 2003. Cited on page 114.

Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Permissive Interfaces. In *ESEC/FSE-13: Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 31–40, 2005. ISBN 1-59593-014-0. doi: {http://doi.acm.org/10.1145/1081706.1081713}. Cited on pages 54 and 115.

David Hovemeyer and William Pugh. Finding Bugs is Easy. In *OOPSLA '04: Companion to the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, pages 132–136, 2004. ISBN 1-58113-833-4. doi: {http://doi.acm.org/10.1145/1028664.1028717}. Cited on page 5.

R. M. Karp. Reducibility among Combinatorial Problems. In *Complexity of Computer Computations*. 1972. Cited on page 8.

Ted Kremenek, Paul Twohey, Godmar Back, Andrew Ng, and Dawson Engler. From Uncertainty to Belief: Inferring the Specification Within. In *OSDI'06: Proceedings of the 7th conference on USENIX Symposium on Operating Systems Design and Implementation*, pages 12–12, 2006. Cited on pages 29, 56, and 114.

Claire Le Goues and Westley Weimer. Specification Mining with Few False Positives. In *TACAS '09: Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2009. ISBN 978-3-642-00767-5. doi: {http://dx.doi.org/10.1007/978-3-642-00768-2_26}. Cited on pages 85 and 114.

Wenchao Li, Alessandro Forin, and Sanjit A. Seshia. Scalable Specification Mining for Verification and Diagnosis. In *Proceedings of the 47th Design Automation Conference*, DAC '10, 2010. Cited on page 2.

Zhenmin Li and Yuanyuan Zhou. PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code. In *Proceedings of ESEC/FSE-13*, pages 306–315, 2005. ISBN 1-59593-014-0. doi: {http://doi.acm.org/10.1145/1081706.1081755}. Cited on pages 30, 56, 61, 74, and 114.

Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have Things Changed Now?: An Empirical Study of Bug Characteristics in Modern Open Source Software. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, ASID '06, pages 25–33, 2006. ISBN 1-59593-576-2. doi: {http://doi.acm.org/10.1145/1181309.1181314}. Cited on page 89.

David Lo and Siau-Cheng Khoo. SMArTIC: Towards Building an Accurate, Robust and Scalable Specification Miner. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 265–275, 2006. ISBN 1-59593-468-5. doi: {http://doi.acm.org/10.1145/1181775.1181808}. Cited on page 47.

Davide Lorenzoli, Leonardo Mariani, and Mauro Pezze. Automatic Generation of Software Behavioral Models. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, 2008. Cited on page 56.

Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A. Popa, and Yuanyuan Zhou. MUVI: Automatically Inferring Multi-variable Access Correlations and Detecting Related Semantic and Concurrency Bugs. In *SOSP '07: Proceedings of the Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, pages 103–116, 2007. ISBN

978-1-59593-591-5. doi: {http://doi.acm.org/10.1145/1294261.1294272}. Cited on pages 56 and 84.

Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. Graph-based Mining of Multiple Object Usage Patterns. In *Proceedings of ESEC/FSE '09*, 2009. ISBN 978-1-60558-001-2. Cited on page 85.

Jeremy W. Nimmer and Michael D. Ernst. Automatic Generation of Program Specifications. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 229–239, 2002. ISBN 1-58113-562-9. doi: {http://doi.acm.org/10.1145/566172.566213}. Cited on page 53.

L. Pitt and M. K. Warmuth. The Minimum Consistent DFA Problem Cannot Be Approximated Within a Polynomial. In *STOC '89: Proceedings of the Twenty-first Annual ACM Symposium on Theory of Computing*, pages 421–432, 1989. ISBN 0-89791-307-8. doi: {http://doi.acm.org/10.1145/73007.73048}. Cited on page 33.

Nadia Polikarpova, Ilinca Ciupa, and Bertrand Meyer. A Comparative Study of Programmer-Written and Automatically Inferred Contracts. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, pages 93–104, 2009. doi: {http://doi.acm.org/10.1145/1572272.1572284}. Cited on page 114.

Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. Path-Sensitive Inference of Function Precedence Protocols. In *Proceedings of ICSE*, pages 240–250, 2007a. ISBN 0-7695-2828-7. doi: {http://dx.doi.org/10.1109/ICSE.2007.63}. Cited on pages 5, 19, 29, 34, 55, 61, and 73.

Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. Static Specification Inference Using Predicate Mining. In *Proceedings of PLDI*, pages 123–134, 2007b. ISBN 978-1-59593-633-2. doi: {http://doi.acm.org/10.1145/1250734.1250749}. Cited on pages 29, 34, 55, and 114.

Manos Renieris, Sébastien Chan-Tin, and Steven P. Reiss. Elided Conditionals. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and*

Engineering, PASTE '04, pages 52–57, 2004. doi: {http://doi.acm.org/10.1145/996821.996839}. Cited on page 115.

Martin C. Rinard and Pedro C. Diniz. Commutativity Analysis: A New Analysis Technique for Parallelizing Compilers. *ACM Trans. Program. Lang. Syst.*, 19(6):942–991, 1997. doi: {http://doi.acm.org/10.1145/267959.269969}. Cited on page 116.

Sharon Shoham, Eran Yahav, Stephen Fink, and Marco Pistoia. Static Specification Mining Using Automata-based Abstractions. In *Proceedings of ISSTA*, pages 174–184, 2007a. ISBN 978-1-59593-734-6. doi: {http://doi.acm.org/10.1145/1273463.1273487}. Cited on pages 29, 33, 44, 54, and 114.

Sharon Shoham, Eran Yahav, Stephen Fink, and Marco Pistoia. Gallery of Mined Specifications, 2007b. Cited on page 44.

R. E. Strom and S. Yemini. Typestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Trans. Softw. Eng.*, 12(1), 1986. ISSN 0098-5589. Cited on pages 59 and 102.

Suresh Thummalapenta and Tao Xie. Mining Exception-handling Rules as Sequence Association Rules. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, 2009. ISBN 978-1-4244-3453-4. doi: {http://dx.doi.org/10.1109/ICSE.2009.5070548}. Cited on page 85.

Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. Detecting Object Usage Anomalies. In *ESEC-FSE '07: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 35–44, 2007. ISBN 978-1-59593-811-4. doi: {http://doi.acm.org/10.1145/1287624.1287632}. Cited on pages 55 and 86.

Westley Weimer and George Necula. Mining Temporal Specifications for Error Detection. In *Proceedings of TACAS*, pages 461–476, 2005. Cited on pages 5, 19, 26, 34, 36, 51, and 54.

John Whaley and Monica S. Lam. Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In *Proceedings of PLDI*, pages 131–144, 2004. ISBN 1-58113-807-5. doi: {http://doi.acm.org/10.1145/996841.996859}. Cited on pages 15 and 22.

John Whaley, Michael C. Martin, and Monica S. Lam. Automatic Extraction of Object-oriented Component Interfaces. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 218–228, 2002. ISBN 1-58113-562-9. doi: {http://doi.acm.org/10.1145/566172.566212}. Cited on pages 5, 28, 33, 54, and 115.

Tao Xie and David Notkin. Mutually Enhancing Test Generation and Specification Inference. In Alexandre Petrenko and Andreas Ulrich, editors, *Formal Approaches to Software Testing*, pages 1100–1101. 2004. Cited on page 115.

Yichen Xie and Alex Aiken. Scalable Error Detection Using Boolean Satisfiability. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 351–363, 2005. ISBN 1-58113-830-X. doi: {http://doi.acm.org/10.1145/1040305.1040334}. Cited on pages 5 and 33.

Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Perracotta: Mining Temporal API Rules from Imperfect Traces. In *Proceedings of ICSE*, pages 282–291, 2006. ISBN 1-59593-375-1. doi: {http://doi.acm.org/10.1145/1134285.1134325}. Cited on pages 5, 7, 19, 26, 34, 36, 51, 55, 60, 101, and 114.

Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. Efficient Forward Computation of Dynamic Slices Using Reduced Ordered Binary Decision Diagrams. In *Proceedings of ICSE*, 2004. ISBN 0-7695-2163-0. Cited on page 15.