Javert: Fully Automatic Mining of General Temporal Properties from Dynamic Traces*

Mark Gabel Zhendong Su
Department of Computer Science
University of California, Davis
{mggabel,su}@ucdavis.edu

ABSTRACT

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 paper, 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.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*; F.3.1 [**Logics and Meaning of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms

Languages, Algorithms, Verification

Keywords

Specification mining, dynamic analysis, formal specifications

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 [6, 10, 13, 31] 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 [4]. 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 [4, 27, 30]. 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

*This research was supported in part by NSF CAREER Grant No. 0546844, NSF CyberTrust Grant No. 0627749, NSF CCF Grant No. 0702622, US Air Force under grant FA9550-07-1-0532, and a generous gift from Intel. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

SIGSOFT 2008/FSE-16, November 9–15, Atlanta, Georgia, USA

Figure 1: Usage specification for Java's MessageDigest class.

strings is NP-hard and cannot be approximated [23], 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 [11,14,29,32]. 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 [14]. Current miners can locate instances of alternating patterns over event pairs [11, 29, 32], resource usage patterns over event triples [14], and precedence protocols [24, 25] and partial orders [1] 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 paper, 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 paper 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 2) illustrates our high-level technique through a motivating example. Section 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 4 provides details about Javert's implementation. It then describes our empirical evaluation and results. Finally, Sections 5 and 6 survey and compare related work and conclude.

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.

Consider the specification automaton in Figure 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

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

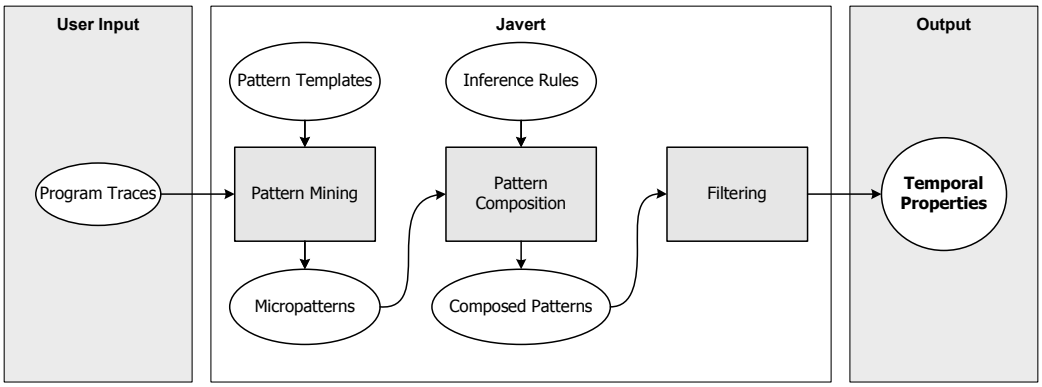

Figure 3: High-level architecture of Javert.

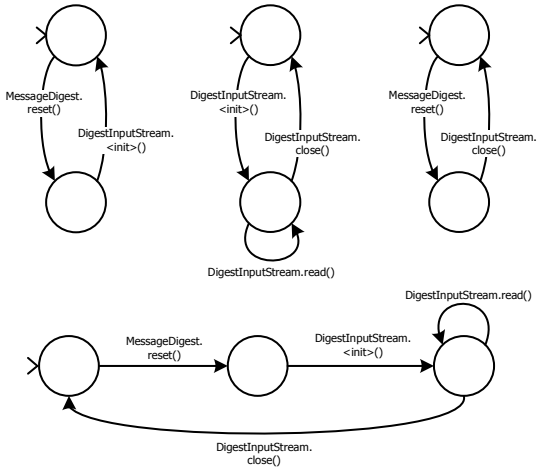

Figure 2: Three micropatterns related to the MessageDigest API and the larger pattern yielded by composition.

interesting projection by tracing the flow of data (*e.g.* "scenario extraction" in [4]). 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 [11, 29, 32] are capable of locating all instances of alternating events; that is, ones that fall into the regular pattern $(ab)^*$. Recent advances [14] have leveraged the use of symbolic techniques to allow the mining of larger patterns, including ones with looping transitions (*e.g.* $(ab^*c)^*$). Figure 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 `MessageDigest.reset` and `DigestInputStream.close`. From this, we can deduce that the first three micropatterns in Figure 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 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.

3. TECHNICAL APPROACH

In this section, we discuss the realization of our technique. In Section 3.1, we formalize the idea of pattern composition. In Section 3.2, we present the specific patterns and composition rules used in Javert. Section 3.3 argues that these rules and patterns are sufficient to locate a large number of real specifications in software systems.

3.1 General Framework

In Section 2, we introduced the intuitive idea of pattern composition. We now present formal definitions to more clearly illustrate this idea.

Definition 3.1 (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* $\pi_\Sigma(L) = \{\pi_\Sigma(s) \mid s \in L\}$.

Definition 3.2 (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* $\pi_\Sigma(T) \in \mathcal{L}(A)$.

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 3.3 (Expansion) *[Gabel and Su, [14] § 2.6] Assume a regular language defined by a finite state automaton $A = (Q, \Sigma, \delta, q_0, F)$. The* expansion *of $\mathcal{L}(A)$ over an arbitrary alphabet Σ', written $E_{\Sigma'}(\mathcal{L}(A))$, is the* maximal *language over $\Sigma \cup \Sigma'$ whose projection over Σ is $\mathcal{L}(A)$.*

An automaton accepting $E_{\Sigma'}(\mathcal{L}(A))$ 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 paper, 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 3.4 (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.*,*

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

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

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.

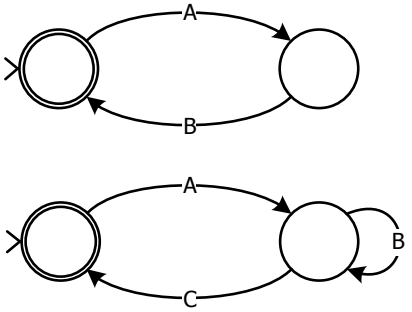

Figure 4: Micropatterns mined by Javert.

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 4.

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

$$\frac{(a{\mathcal{L}_1}^*b)^* \qquad (a{\mathcal{L}_2}^*b)^*}{(a\,(\mathcal{L}_1|\mathcal{L}_2)^*\,b)^*} \quad \text{[BRANCH]}$$

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

Proposition 3.5 (Correctness of Branching) *Defining Σ' as $\{a, b\} \cup \Sigma_{\mathcal{L}_1} \cup \Sigma_{\mathcal{L}_2}$, the correctness of the* Branching Rule *follows from the following fact:*

$$E_{\Sigma'}(a{\mathcal{L}_1}^*b)^* \cap E_{\Sigma'}(a{\mathcal{L}_2}^*b)^* = (a\,(\mathcal{L}_1|\mathcal{L}_2)^*\,b)^*$$

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.

$$\frac{(a\mathcal{L}_1b)^* \qquad (b\mathcal{L}_2c)^* \qquad (ac)^*}{(a\mathcal{L}_1b\mathcal{L}_2c)^*} \quad \text{[SEQUENCE]}$$

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

Proposition 3.6 (Correctness of Sequencing) *Redefining* Σ' *as* $\{a, b, c\} \cup \Sigma_{\mathcal{L}_1} \cup \Sigma_{\mathcal{L}_2}$*, the correctness of the* Sequencing Rule *follows from the following fact:*

$$E_{\Sigma'}(a\mathcal{L}_1 b)^* \cap E_{\Sigma'}(b\mathcal{L}_2 c)^* \cap E_{\Sigma'}(ac)^* = (a\mathcal{L}_1 b\mathcal{L}_2 c)^*$$

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 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 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.

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.

3.3.1 A Socket API, Ammons et al.

Strauss, a tool developed by Ammons *et al.* [4], mines arbitrarily complex specifications from dynamic traces. Consider the socket API in Figure 5. This figure has been reproduced from the original paper [4] 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.

3.3.2 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 6 are reproduced from their paper [27] and their online supplement [26], 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.

4. IMPLEMENTATION AND RESULTS

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

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 [14]. This algorithm leverages Binary Decision Diagrams [7] 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

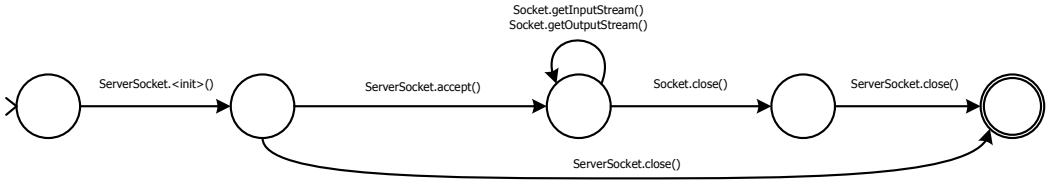

Figure 5: Socket API, Ammons *et al.* [4]

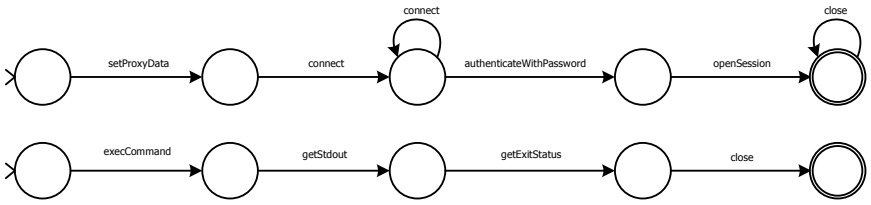

Figure 6: Ganymed APIs, Shoham *et al.* [26, 27]

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`.

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 paper describes temporal relationships between two or more types. Attempting to address this by considering more dataflow is non-trivial (see Scenario Extraction in [4]), 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.

Note, though, that although Javert can handle buggy or imprecise traces, it would certainly thrive with more accurate input. Techniques like SMArTIC [19] perform preprocessing and clustering on traces to isolate and remove false behavior, reducing the incidence of false positives. Dynamic slicing [2] over traces could also serve to this end by removing unrelated flows of data. Any technique for

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

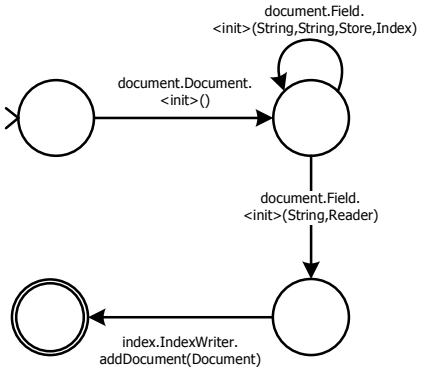

Figure 9: Specification of the use of Lucene's indexer.

improving the accuracy of Javert's input is compatible and likely to improve results, but we sought to design for the common case.

Availability Javert is available as an open source release from `http://wwwcsif.cs.ucdavis.edu/~gabel/research/javert`. In addition to the symbolic specification miner and Javert's pattern composer, the release also contains our trace collection framework.

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 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.

Our quantitative results are displayed in Figure 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, $|\Sigma| \geq 4$, 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 4.1 (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.

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 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 10 presents one particularly interesting specification from Javert's JGnash solution set. JGnash is a personal finance package; when saving bank account 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

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

Project	Description	Num. of Traces	Total Trace Events	Execution Time	
				Pattern Mining	Composition
Lucene Indexer	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 7: Trace data and analysis times.

	$\|\Sigma\| \geq 4$			$\|\Sigma\| \geq 7$			$\|\Sigma\| \geq 10$		
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 8: Quantitative results.

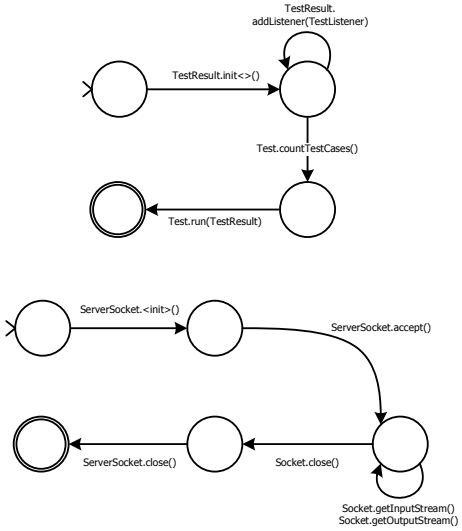

Figure 11: Two properties extracted from executions of Apache Ant.

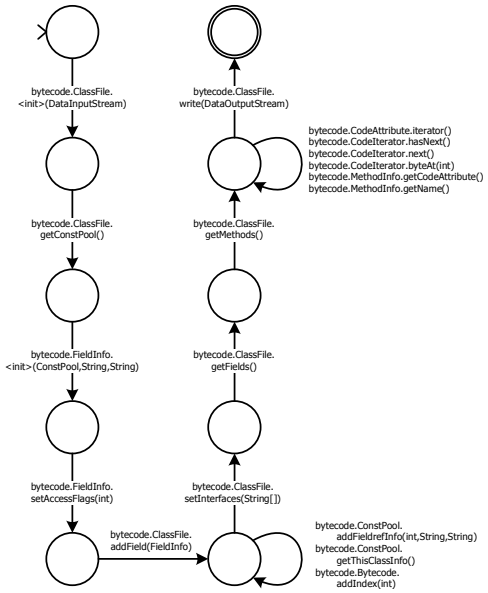

Figure 12: A larger temporal property of the JavaAssist bytecode framework, extracted from its use by Hibernate.

few false positives. Figure 11 displays two compact properties discovered from Ant, including the server-side TCP socket API. Figures 12 and 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 orders of magnitude greater than Javert's. For example, large portions of papers [29, 32] 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 tempo-

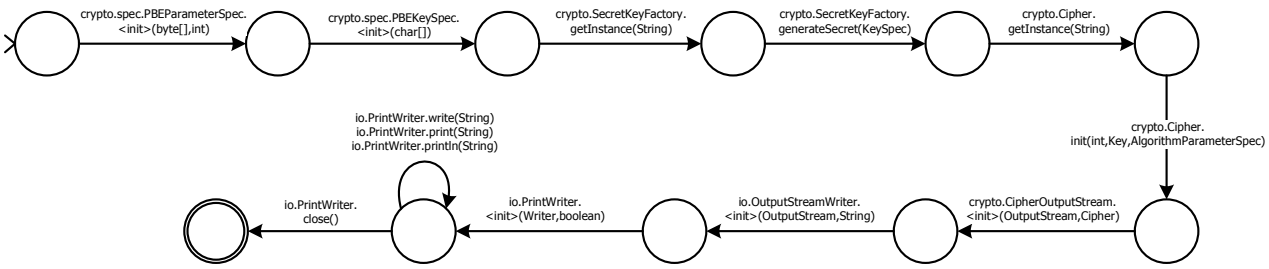

Figure 10: A specification for the use of a symmetric cipher, extracted from JGnash.

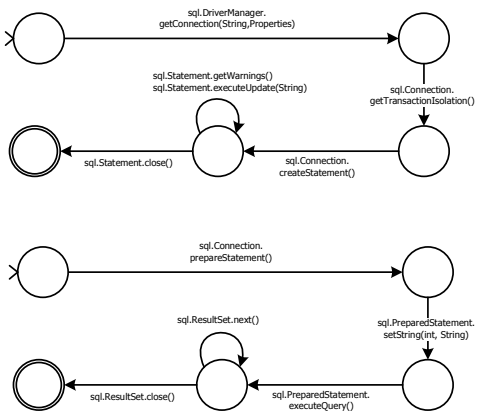

Figure 13: Two specifications extracted from Hibernate.

ral 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 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 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 [22] 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.

5. RELATED WORK

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

5.1 Temporal Specification Mining

Ammons *et al.* [4] 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 [5] by applying concept analysis to debug the learned specifications. Shoham *et al.* recently presented a static analysis with the same general goal [27]. 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 [3,16,30] 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 [11]. This approach suffered from imprecision, so the authors used statistical methods to rank the possible properties. Weimer and Necula [29] 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 [14] and greater.

Yang *et al.* [32] 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 [9], developed by Dallmeier *et al.*, 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 [24] 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 [25]. 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 [1]. 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 Wasylkowkski *et al.* [28], 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.

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 [18]. 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 [21]. 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 [17]. 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.

5.3 Dynamic Analysis

The Daikon project [8, 12] 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 [15].

Combining the ideas of invariant detection and temporal property mining, Lorenzoli *et al.* have developed a dynamic analysis algorithm for extracting software behavioral models [20]. 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.

6. CONCLUSIONS

In this paper, 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 paper. 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.

7. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful comments and helpful suggestions.

8. REFERENCES

[1] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining API patterns as partial orders from source code: from usage scenarios to specifications. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 25–34, New York, NY, USA, 2007. ACM.

[2] H. Agrawal and J. 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.

[3] R. Alur, P. Černý, P. Madhusudan, and W. 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.

[4] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 4–16, 2002.

[5] G. Ammons, D. Mandelin, R. Bodík, and J. 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, New York, NY, USA, 2003. ACM.

[6] T. Ball and S. 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.

[7] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986.

[8] C. Csallner and Y. Smaragdakis. Dynamically discovering likely interface invariants. In *Proceedings of ICSE*, pages 861–864, 2006.

[9] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller. Mining object behavior with adabu. In *WODA '06: Proceedings of the 2006 international workshop on Dynamic systems analysis*, pages 17–24, New York, NY, USA, 2006. ACM.

[10] M. Das, S. Lerner, and M. 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.

[11] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 57–72, 2001.

[12] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *Proceedings of ICSE*, pages 449–458, 2000.

[13] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of PLDI*, pages 234–245, 2002.

[14] M. Gabel and Z. Su. Symbolic mining of temporal specifications. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, New York, NY, USA, 2008. ACM.

[15] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of ICSE*, pages 291–301, 2002.

[16] T. A. Henzinger, R. Jhala, and R. 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, New York, NY, USA, 2005. ACM.

[17] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler. From uncertainty to belief: inferring the specification within. In *USENIX'06: Proceedings of the 7th conference on USENIX Symposium on Operating Systems Design and Implementation*, pages 12–12, 2006.

[18] Z. Li and Y. 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.

[19] D. Lo and S.-C. 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, New York, NY, USA, 2006. ACM.

[20] D. Lorenzoli, L. Mariani, and M. Pezze. Automatic generation of software behavioral models. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, New York, NY, USA, 2008. ACM.

[21] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. Muvi: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 103–116, New York, NY, USA, 2007. ACM.

[22] J. W. Nimmer and M. 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, New York, NY, USA, 2002. ACM.

[23] 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, New York, NY, USA, 1989. ACM.

[24] M. K. Ramanathan, A. Grama, and S. Jagannathan. Path-sensitive inference of function precedence protocols. In *Proceedings of ICSE*, pages 240–250, 2007.

[25] M. K. Ramanathan, A. Grama, and S. Jagannathan. Static specification inference using predicate mining. In *Proceedings of PLDI*, pages 123–134, 2007.

[26] S. Shoham, E. Yahav, S. Fink, and M. Pistoia. Gallery of mined specifications. http://tinyurl.com/23qct8 or http://docs.google.com/View?docid=ddhtqgv6_10hbczjd.

[27] S. Shoham, E. Yahav, S. Fink, and M. Pistoia. Static specification mining using automata-based abstractions. In *Proceedings of ISSTA*, pages 174–184, 2007.

[28] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 35–44, New York, NY, USA, 2007. ACM.

[29] W. Weimer and G. Necula. Mining temporal specifications for error detection. In *Proceedings of TACAS*, pages 461–476, 2005.

[30] J. Whaley, M. C. Martin, and M. 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.

[31] Y. Xie and A. 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.

[32] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: Mining temporal API rules from imperfect traces. In *Proceedings of ICSE*, pages 282–291, 2006.