Online Inference and Enforcement of Temporal Properties*

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

ABSTRACT

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.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Software/Program Verification

General Terms

Languages, Algorithms, Experimentation, Reliability

Keywords

Dynamic analysis, Temporal properties, Online algorithm

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

ICSE'10, May 2–8 2010, Cape Town, South Africa

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* [26] 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 [8]) either specifically target these specifications [16] or use them as their primary example. Similarly, dynamic tools, such as runtime monitoring frameworks [6], 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 paper, 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 [17] 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 paper 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 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 [29]) 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 [25] as well as temporal association rules of function calls and field accesses [22].

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).

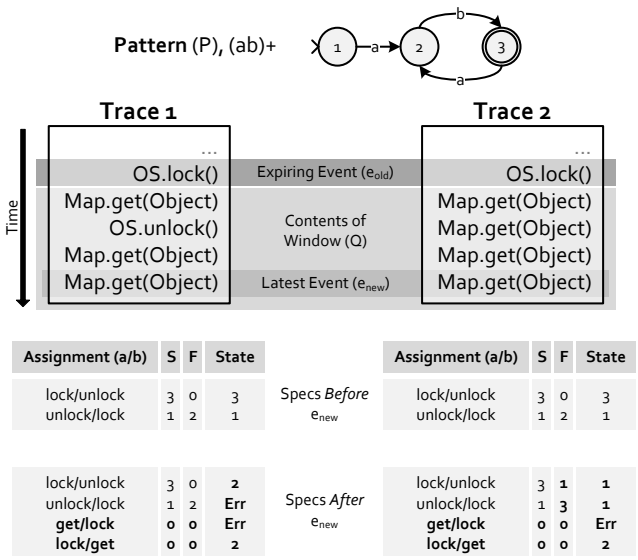

Figure 1: Execution of Algorithm 1 on two example traces.

This paper is organized as follows. The following section (Section 2) describes our general approach and algorithm, while Section 3 discusses the realization of our algorithm as a practical defect detection system. Section 4 contains an empirical evaluation of our work, and Sections 5 and 6 discuss related work and our plans for continuing this research, respectively.

2. APPROACH

This section describes our basic approach. We first describe our algorithm in its simplest form (Section 2.1). We then expand on the basic definition with a series of generalizations (Sections 2.2–2.4) that form the final algorithm implemented in our tool, OCD.

2.1 Basic Algorithm

Our algorithm, described in pseudocode as Algorithm 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.

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 2.2 and 2.3 sections discuss generalizations that consider additional information, *e.g.* receiver objects.) Two example traces appear in Figure 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 paper 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 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 1, one possible concrete assignment into the alternating pattern is $\{a \mapsto \mathsf{OS.lock}, b \mapsto \mathsf{OS.unlock}\}$, forming the potential specification $(\mathsf{OS.lock\ OS.unlock})+$.

Finite Window Our algorithm operates over a finite *window*: a bounded view of a trace's history. The window is a standard FIFO

Algorithm 1 Online inference and enforcement algorithm.

Constants: P : Two-letter pattern automaton over $\{\mathsf{a},\mathsf{b}\}$ with states $\{\text{INIT},\ldots\}$
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 τ
$specs$: Asgn $\mapsto$ Spec
Require: $e_{new} : \tau$

```
 1: Q ← ADD(Q, e_new)
 2: e_old ← REMOVE(Q)

 3: for all e_fut in Q do
 4:     if (e_old, e_fut) ∉ domain(specs) then
 5:         specs(e_old, e_fut) ← ((e_old, e_fut), 0, 0, INIT)
 6:         specs(e_fut, e_old) ← ((e_fut, e_old), 0, 0, INIT)
 7:     end if
 8: end for

 9: for all spec in specs(e_old, ∗) ∪ specs(∗, e_old) do
10:     if spec.asgn.a = e_old then
11:         spec.st ← NEXT(spec.st, a)
12:     else
13:         spec.st ← NEXT(spec.st, 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 ← spec.sat + 1
18:         else
19:             spec.fail ← spec.fail + 1
20:             if ISENFORCING(spec.sat, spec.fail) then
21:                 REPORTANOMALY( )
22:             end if
23:         end if
24:         spec.st ← INIT
25:     end if
26: end for
```

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*. 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 1 and the pseudocode of Algorithm 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 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 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 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.[1] 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:

$$\text{ISENFORCING}(sat, fail) \equiv \frac{sat}{sat + fail} > THRESHOLD$$

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 3. Finally, in the event that ISENFORCING returns true, we report the current instance as an anomaly.

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.

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

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

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 [6]).

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

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.

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

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."

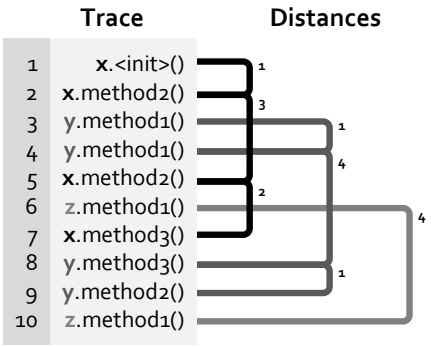

Figure 2: An example of our methodology for measuring temporal locality.

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 3.1). Next, we discuss the various pattern templates we use (Section 3.2). We then discuss the design and implementation of *learning strategies* (Section 3.3), which have thus far been presented in terms of the predicate ISENFORCING. Finally, we discuss our automatic *multivariate self-tuner* (Section 3.4), which allows OCD to function well on a wide variety of target programs without the danger of "overtraining" its various parameters.

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. 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 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.

Benchmark	Window Size 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 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.

Benchmark	Window Size 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 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.

We performed our study of temporal locality on the DaCapo workload [3], 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 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 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.

Tables 1 and 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.

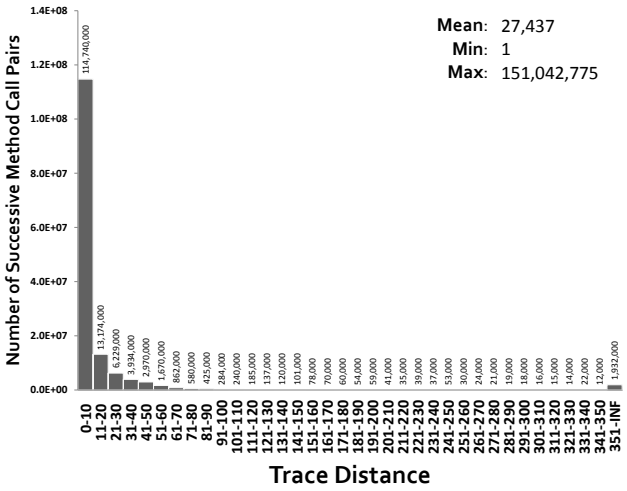

Figure 3: A histogram of the distances between successive method calls on the same object during the execution of Eclipse.

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 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.

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 [17]. 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 [17], 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:[2]

$$ab \qquad \text{SEQUENCING} \qquad (1)$$

$$ab^+ \qquad \text{LOOPBEGIN} \qquad (2)$$

$$a^+b \qquad \text{LOOPEND} \qquad (3)$$

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

$$ab? \qquad \text{PRECONDITION} \qquad (4)$$

$$a?b \qquad \text{POSTCONDITION} \qquad (5)$$

$$a^+b^* \qquad \text{GENERALIZED PRECOND.} \qquad (6)$$

$$a^*b^+ \qquad \text{GENERALIZED POSTCOND.} \qquad (7)$$

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 [22] 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.

$$(ab|ba) \qquad \text{ASSOCIATION RULE} \qquad (8)$$

$$(a^+b^+)|(b^+a^+) \qquad \text{GENERALIZED ASSOC. RULE} \qquad (9)$$

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.

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:

$$\text{ISENFORCING} : (sat : \text{int}, fail : \text{int}) \mapsto (\text{true}|\text{false})$$

[2]We showed previously [18] that it was generally impossible to precisely decompose three-letter patterns into a set of two-letter patterns. However, safe *approximations* are possible.

OCD implements a slight generalization of this function:

$$\text{ISENFORCING} : (sat : \text{int}, fail : \text{int}) \mapsto (\text{ENFORCING}|\text{LEARNING}|\text{DEAD})$$

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:

Join

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

Meet

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

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 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.

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.

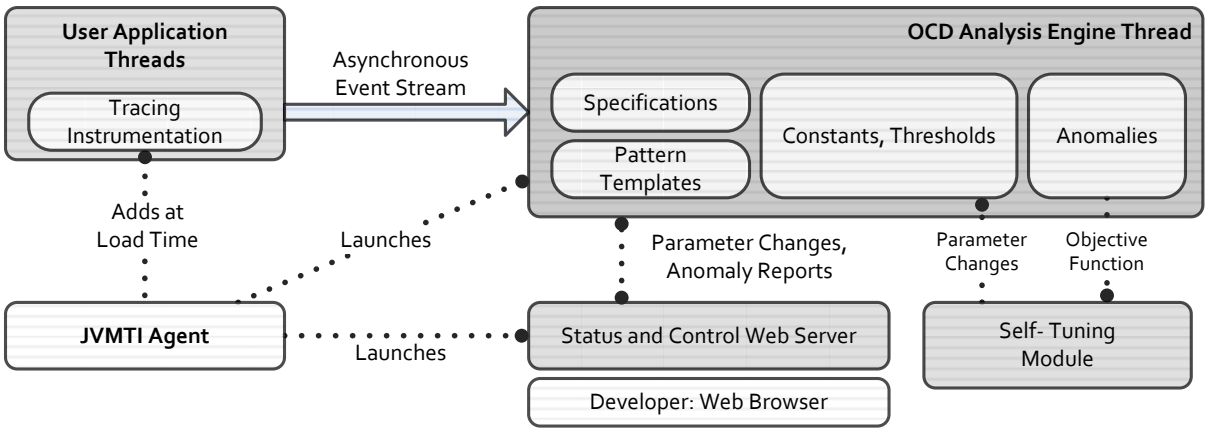

Figure 4: Implementation architecture.

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:

$$\text{OBJECTIVEFUNC} \equiv \textit{anomaly\_budget} - \textit{anomaly\_count}$$

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. EVALUATION

This section describes the implementation and evaluation of OCD. We start with a brief description of 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.

4.1 Implementation

OCD's high-level architecture is depicted in Figure 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.

4.2 The DaCapo Workload

We performed our first evaluation on the DaCapo workload [3], a benchmarking suite consisting of several widely-used Java applications.[3] 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 5.

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

Benchmark	Specifications Considered	Enforced	Anomalies	Overhead (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.

Benchmark	Specifications Considered	Enforced	Anomalies	Overhead (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 5: Results on the "known good" DaCapo suite.

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

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.

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 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.

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. 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 [11] 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.

4.3 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.

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 InputStreamReader 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 [9], 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 Generality: Associated Field Accesses

Existing tools that search for inconsistent *field* accesses, *e.g.* MUVI [23], 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[4] (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 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.

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.

Specification Inference Ammons *et al.* [2] 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 [7] 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.* [1] 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 [21] 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 [24]. 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 [27], 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 [6], have seen dramatic improvements in performance in recent years. Often with the help of static information [4, 12, 19], 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.* [10] 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

[4]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.

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.* [14]. Hangal and Lam's DIDUCE [20] hypothesizes and learns invariants over program values, much like Daikon [15], 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 [5], 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 [13]. 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 [28], 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.

6. CONCLUSIONS 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.

7. 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 *Proceedings of ESEC-FSE '07*, 2007.

[2] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *Proceedings of POPL '02*, 2002.

[3] 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.

[4] E. Bodden, P. Lam, and L. Hendren. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In *Proceedings of SIGSOFT '08/FSE-16*, 2008.

[5] R.-Y. Chang, A. Podgurski, and J. Yang. Finding what's not there: a new approach to revealing neglected conditions in software. In *Proceedings of ISSTA '07*, 2007.

[6] F. Chen and G. Roşu. Mop: an efficient and generic runtime verification framework. In *Proceedings of OOPSLA '07*, 2007.

[7] 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*, 2006.

[8] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proceedings of PLDI*, 2002.

[9] I. Dillig, T. Dillig, E. Yahav, and S. Chandra. The CLOSER: automating resource management in Java. In *ISMM '08: Proceedings of the 7th international symposium on Memory management*, 2008.

[10] M. B. Dwyer, M. Diep, and S. G. Elbaum. Reducing the cost of path property monitoring through sampling. In *ASE*, 2008.

[11] M. B. Dwyer, A. Kinneer, and S. Elbaum. Adaptive online program analysis. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, 2007.

[12] M. B. Dwyer and R. Purandare. Residual dynamic typestate analysis exploiting static analysis: results to reformulate and reduce the cost of dynamic analysis. In *Proceedings of ASE*, 2007.

[13] S. Elbaum, S. Kanduri, and A. Andrews. Trace anomalies as precursors of field failures: an empirical study. *Empirical Softw. Engg.*, 12(5), 2007.

[14] 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*, 2001.

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

[16] S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.*, 17(2), 2008.

[17] M. Gabel and Z. Su. Javert: Fully automatic mining of general temporal properties from dynamic traces. In *Proceedings of SIGSOFT '08/FSE-16*, 2008.

[18] M. Gabel and Z. Su. Symbolic mining of temporal specifications. In *Proceedings of ICSE '08*, 2008.

[19] M. Gopinathan and S. K. Rajamani. Enforcing object protocols by combining static and runtime analysis. In *Proceedings of OOPSLA '08*, 2008.

[20] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of ICSE*, 2002.

[21] C. Le Goues and W. 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.

[22] 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*, 2005.

[23] 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 *Proceedings of SOSP '07*, 2007.

[24] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based mining of multiple object usage patterns. In *Proceedings of ESEC/FSE '09*, 2009.

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

[26] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1), 1986.

[27] S. Thummalapenta and T. Xie. Mining exception-handling rules as sequence association rules. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, 2009.

[28] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *Proceedings of ESEC-FSE*, 2007.

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