Finding Programming Errors Earlier by Evaluating Runtime Monitors Ahead-of-Time

Eric Bodden[1], Patrick Lam[2] and Laurie Hendren[1]

[1] Sable Research Group, School of Computer Science, McGill University
[2] Department of Electrical and Computer Engineering, University of Waterloo

ABSTRACT

Runtime monitoring may require many program runs to find errors. Errors may therefore be discovered late in the development process or—often—only after deployment. We present analysis techniques to evaluate runtime monitors ahead-of-time. We show that our static evaluation techniques can (1) prove the absence of error conditions on all program runs, or (2) flag locations where errors may occur. Our work focuses on tracematches, an expressive runtime monitoring notation that allows programmers to reason about groups of correlated objects. We describe a novel flow-sensitive static analysis for analyzing program states; our abstraction captures both positive information—a set of objects could be in a particular monitor state—and negative information—the set is known not to be in a state. The analysis resolves references to the heap by combining the results of three points-to and alias analyses. We also propose an optional machine learning phase, which filters out likely false positives, and describe a visualization for the remaining points of failure. We applied a set of 13 tracematches to the DaCapo benchmark suite and SciMark2. Our static analysis was able to rule out all potential points of failure in 50% of the cases, and 75% of false positives on average. Our machine learning algorithm correctly classifies the remaining potential points of failure in all but 3 of 461 cases.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Validation*; F.3.2 [**Logics and Meaning of Programs**]: Semantics of Programming Languages—*Program analysis*

General Terms

Algorithms, Experimentation, Reliability, Verification

Keywords

Static analysis, static verification, runtime verification, machine learning, points-to analysis

1. INTRODUCTION

A software system's sequence of actions over an execution is a rich source of information about the system's behaviour on that execution. Certain sequences of runtime events indicate defects in the system; for instance, systems must not read from files after they are closed. Runtime monitoring can detect such sequences of events, enabling developers to handle the sequences with code that reports errors or enables the system to recover from faults.

However, runtime monitoring depends on the availability of test cases to detect errors. Errors may therefore be hard to find and can remain unnoticed until late in the development process or even until after a system is deployed.

Our research therefore aims to verify runtime monitoring properties ahead-of-time, through static analysis. Static verification can (1) prove the absence of error conditions on all program runs, or (2) flag all code locations where errors may possibly occur. We designed a sound static analysis (no missed violations) that would report as few false positives as possible. To further assist the programmer, we also expose a simple machine learning algorithm to distinguish between likely program errors and likely false positives. Our approach enables the programmer to (1) manually inspect all likely errors, and (2) specify recovery code to gracefully handle all unlikely errors (which could also be manually inspected, given enough time).

We focus on one particular approach to runtime monitoring: tracematches [1], a Java language extension. Tracematches are concise yet expressive; they enable developers to specify interesting error situations. Using tracematches, developers can specify traces of interest via regular expressions of symbols with free variables, along with some code to execute if the trace occurs on a program run. A symbol's free variables bind heap objects at runtime. A tracematch executes its associated code if its regular expression matches a suffix of the current execution trace with a consistent variable binding. For example, tracematches can monitor for files `f` that are read after being closed using the regular expression `close(f) read(f)`.

We show our complete analysis approach for tracematches in Figure 1 (explanation below). A key challenge for any static analysis of tracematches is the fact that tracematch events bind variables to heap objects at runtime. In the above example, the variable `f` would be bound at `close` and matched at `read`. Because heap objects are often shared by different methods in a program, and because any method could cause an event on a bound object, one might think that any nontrivial static analysis of tracematches would have to

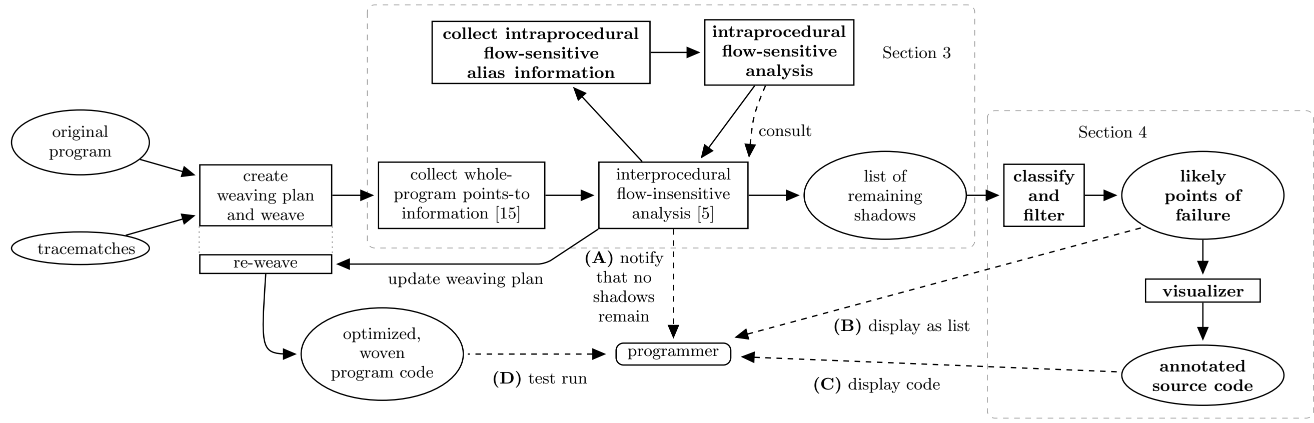

Figure 1: Our complete approach to static verification using tracematches; novel phases shown in boldface; solid arrows represent data flow, dashed arrows possible interaction

be both interprocedural and flow-sensitive. Indeed, in previous work [5] we showed that an interprocedural but flow-insensitive analysis can only eliminate those partial matches of a tracematch that cannot be completed for trivial reasons.

The most important finding of the methods presented in *this* paper is that we can successfully reason about tracematch states by combining inexpensive whole-program summary information with a suite of carefully-designed intraprocedural flow-sensitive analyses (top of Figure 1). Our abstraction associates two kinds of information with each state: positive information (an object is known to be in a particular state) and negative information (an object is not in a state). Our results show that the combination of positive and negative information enables us to eliminate most false positives.

As the Figure shows, our static analysis can have multiple outcomes. In the ideal case **(A)**, the analysis reports that the program can never trigger the tracematch at runtime, e.g. if we can prove that no file is read from after being closed. We use the term *shadow* [13] to denote a program point corresponding to a runtime event of interest. If no shadows remain, the program satisfies the monitored property and no further action is required.

If shadows do remain, we next identify potential points of failure. Such points correspond to the remaining shadows for a *final* events—ones may trigger the tracematch, e.g. any `read`-shadow in the file example. Using a machine learning approach, we then classify each potential point of failure, to determine whether or not it is a likely point of failure. The programmer can then inspect **(B)** the filtered list of likely points of failure along with useful context information. In addition, we provide a prototype of a visualization tool that the programmer can use to display **(C)** all potential points of failure in program code.

To enable the programmer to recover from runtime errors at points where errors are unlikely to occur (but are not impossible), we provide an option to **(D)** generate an optimized instrumented program that includes runtime monitoring code only at positions where our static analysis could not rule out potential matches statically. Because our approach is sound, this monitor catches *all* actual errors.

We have implemented our static analysis using the AspectBench Compiler [2]. We applied the analysis to 103 combinations of tracematches with SciMark2 [14] and the entire DaCapo benchmark suite [4], and found that 38 of these combinations showed potential property violations. Our static analysis ruled out all potential points of failure in 50% of the cases; overall, it ruled out 75% of the potential points of failure. For all but 5 benchmark/tracematch combinations, fewer than 10 points of failure remained.

We found actual property violations in 5 cases. In Jython and bloat, this violation pointed out an actual error. In PMD, we found dubious code which was fixed in a later revision. Our machine learning algorithm correctly classified the remaining potential points of failure in all but 3 of 461 cases. The contributions of this paper include:

- an analysis abstraction for tracking runtime monitor states, including the information that an object is or is not in a certain state, along with
- a unified set of rules for manipulating the analysis abstraction based on static alias information;
- a machine learning algorithm to identify likely points of failure among potential points of failure;
- a visualization tool to display potential points of failure in program code; and
- an implementation of our static analyses and their evaluation on realistically-sized benchmark programs.

We next proceed with two introductory examples before explaining our static analysis approach in Section 3 and our classification approach using machine learning in Section 4. The evaluation follows in Section 5, followed by a discussion of related work and by conclusions.

2. TRACEMATCHES AT RUNTIME

In this section, we describe tracematches [1], a mechanism for runtime monitoring; explain how a compiler creates code that implements tracematches at runtime; and present the runtime system's evaluation of tracematches. Although this section presents the situation at runtime, our two illustrative examples foreshadow our static analyses, which we describe in Section 3.

Figure 2 presents our running example, a program we would like to partially verify using tracematches. The program populates a collection, which is then passed to another

```
1  void main() {
2      Collection c1 = new LinkedList();
3      c1.add(''something'');                    //update(c1)
4      Collection c2 = c1;
5      c2.add(''somethingElse'');                //update(c2)
6      print(c2);
7  }
8
9  void print(Collection c3) {
10     Iterator i1 = c3.iterator();              //create(c3,i1)
11     while(i1.hasNext()) {                     //hasNext(i1)
12         Iterator i2 = i1;
13         System.out.println(i2.next());        //next(i2)
14     }
15 }
```

Figure 2: Example program with shadows

```
1  tracematch(Iterator i) {
2      sym hasNext before:
3          call(* java.util.Iterator+.hasNext()) && target(i);
4      sym next before:
5          call(* java.util.Iterator+.next()) && target(i);
6
7      next next { System.err.println("Trouble with "+i); } }
```

Figure 3: HasNext tracematch: do not call `next()` twice without an intervening call to `hasNext()`.

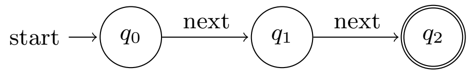

Figure 4: Automaton for HasNext from Figure 3.

method for printing. We explicitly added copy statements at lines 4 and 12 to emphasize the problem of aliasing.

Like many Java programs, our example makes extensive use of iterators, which come with implicit API usage contracts. Tracematches can verify such contracts.

2.1 HasNext example tracematch

Figure 3 presents the `HasNext` verification tracematch, matching suspicious traces where a program calls `i.next()` twice in a row without any intervening call to `i.hasNext()`. Tracematches include an alphabet of *symbols*, a *regular expression* over this alphabet and a *body*. Symbols associate abstract tracematch events with concrete program events. Developers define symbols using AspectJ pointcuts [17].

Symbols may bind variables; line 1 of the tracematch declares that symbols in the `HasNext` tracematch may bind an `Iterator i`. Lines 2–5 define symbols `hasNext` and `next`, which capture calls to the `hasNext()` and `next()` methods of `i`. These two symbols establish the alphabet for the tracematch's regular expression "`next next`" at line 7. Any occurrence of the `hasNext` symbol on a iterator `i` discards partial matches for `i`. Line 7 also holds the body of code to be executed every time the regular expression matches. In this work, we focus on verification tracematches, which typically encode API usage rules. Our tracematch bodies report errors, but could instead contain error-recovery code.

The tracematch runtime matches the regular expression against each *suffix* of the abstract (symbol-based) execution trace. For instance, symbols map the concrete call sequence

`hasNext() next() next() next()`

to an event sequence

`hasNext next next next`,

which the regular expression matches twice, executing the body at the second and third `next` events.

One feature of tracematches is that matches require consistent variable bindings. Our example therefore would only match if two calls to `next` occur on the same iterator. Hence, with iterators `i1` and `i2`, the call sequence we considered earlier could actually be

`i1.hasNext() i2.next() i1.next() i2.next()`,

giving an abstract event sequence of

`hasNext(i=i1) next(i=i2) next(i=i1) next(i=i2)`.

Conceptually, tracematches project the event sequence onto distinct sub-sequences separated by variable bindings. Our example sequence contains two projections: (1) "`hasNext next`" for `i=i1`, and (2) "`next next`" for `i=i2`. Projection (1) does not match, but projection (2) does, and the runtime would execute the tracematch body only once, at the last call to `next()`, with the binding `i=i2`.

Tracematch implementation.

In our approach the programmer expresses verification properties as tracematches and then feeds the tracematches and the program under test to the AspectBench compiler [2] (`abc`). `abc` first creates an automaton from each tracematch's regular expression. Figure 4 presents the automaton for `HasNext`. The compiler then identifies a set of instrumentation points, or *shadows* [13], as described by the tracematch's symbols. At shadows, the compiler adds code to update the tracematch state in response to program events. Figure 2 includes shadows as comments. In line with symbol declarations, shadows may bind tracematch variables.

Since different tracematch symbols may bind different subsets of tracematch variables and since heap objects may simultaneously be in many automaton states, the runtime must store mappings from variable bindings to states. It does so by attaching a constraint to each automaton state [1]. Constraints are logical formulae which can be evaluated to determine whether a given variable-to-object binding is in the given state.

The runtime system initially associates *true* (**tt**) with the initial automaton state and *false* (**ff**) with all other states, since all objects start at the initial automaton state. The constraint at the initial state is always **tt** because tracematches may start a match anytime. For the `HasNext` automaton, the initial configuration is (**tt**, **ff**, **ff**).

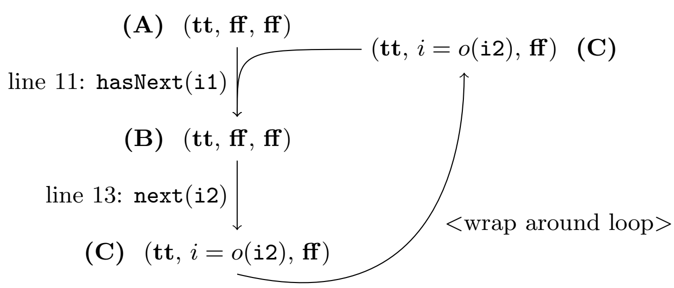

Figure 5: Effect of `print` on `HasNext` automaton.

As the program executes, the runtime updates constraints as events occur. Figure 5 shows the evaluation of the `HasNext` automaton at the start of the `print` method from Figure 2 with initial configuration $(\mathbf{tt}, \mathbf{ff}, \mathbf{ff})$ **(A)**. The `hasNext` shadow at line 11 has no effect on this configuration, since q_0 has no `hasNext` transition and all other states contain **ff** **(B)**. At line 13, the `next` shadow binds tracematch variable `i` to the object stored in `i2`, denoted $o(\texttt{i2})$. The `next` transition from q_0 to q_1 in the automaton causes the following update:

$$
\begin{aligned}
c'(q_1) \;&\equiv\; c(q_1) \vee (\; c(q_0) \wedge i = o(\texttt{i2}) \;) \\
&\equiv\; \mathbf{ff} \vee (\; \mathbf{tt} \wedge i = o(\texttt{i2}) \;) \\
&\equiv\; i = o(\texttt{i2}).
\end{aligned}
$$

Here $c(q_i)$ denotes the original constraint at q_i and $c'(q_i)$ the constraint after executing line 13. The update results in the configuration $(\mathbf{tt}, i = o(\texttt{i2}), \mathbf{ff})$ **(C)**. Another call to `next()` on the same iterator $o(\texttt{i2})$ would propagate $i = o(\texttt{i2})$ to the final state q_2, and the runtime would execute the tracematch body. However, the example program contains a loop, so control flow wraps around to line 11, again hitting the first event `hasNext(i=`o`(i1))`. Because q_1 has no `hasNext` self-loop, object $o(\texttt{i1})$ cannot possibly be in q_1 after this event, and we conjoin q_1's constraint with a negative binding $i \neq o(\texttt{i1})$. Since the incoming configuration is $(\mathbf{tt}, i = o(\texttt{i2}), \mathbf{ff})$, and because $o(\texttt{i2}) = o(\texttt{i1})$, we get:

$$
\begin{aligned}
c'(q_1) \;&\equiv\; c(q_1) \wedge i \neq o(\texttt{i1}) && (1) \\
&\equiv\; i = o(\texttt{i2}) \wedge i \neq o(\texttt{i1}) && (2) \\
&\equiv\; \mathbf{ff}, && (3)
\end{aligned}
$$

which again yields the configuration $(\mathbf{tt}, \mathbf{ff}, \mathbf{ff})$ **(B)**. Observe that this configuration, at line 13, has not changed from the previous iteration.

Note that aliasing information is critical for any static analysis that approximates the runtime configurations: in the above example, any analysis must know that $o(\texttt{i2}) = o(\texttt{i1})$ to conclude that the updates inside the loop have no effect (as in Equation (2)).

2.2 FailSafeIter example tracematch

Tracematches differ from previous approaches in that they enable developers to bind *multiple* variables. Moreover, not all symbols need to bind all variables; the only requirement is that for each *complete* match and each tracematch variable v there must be some matched symbol that binds v.

We demonstrate this feature with the `FailSafeIter` tracematch in Figure 6. This tracematch reports cases where the program modifies a `Collection c` while an `Iterator i` is active on `c`. The figure also shows the corresponding automaton.

```
pointcut collection_update(Collection c):
( call(* java.util.Collection+.add*(..)) || ... ||
  call(* java.util.Collection+.remove*(..)) ) && target(c);

tracematch(Collection c, Iterator i) {
    sym create after returning(i):
      call(* java.util.Collection+.iterator()) && target(c);
    sym next before:
      call(* java.util.Iterator+.next()) && target(i);
    sym update after: collection_update(c);

    create next* update+ next { ... } }
```

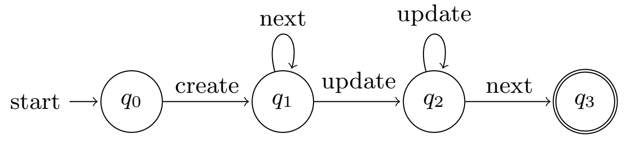

Figure 6: FailSafeIter tracematch and automaton: detect updates to a Collection which is being iterated over.

3. STATICALLY DETECTING FALSE POSITIVES

We now move to our static analysis. We first situate the static analysis in the context of the compilation process for tracematches and then explain our static abstraction.

3.1 Weaving process

We have implemented our analysis using the AspectBench Compiler (`abc`) [2] and the Soot [18] program analysis and transformation framework, which underlies `abc`. Figure 1 presents selected compiler stages and illustrates where our analysis fits in. First, the compiler reads the program under analysis and the tracematch definitions, and weaves the tracematches into the program. Next, we compute a call graph and points-to information for the whole program, using Sridharan and Bodik's context-sensitive, demand-driven, refinement-based points-to analysis [15]. Context information distinguishes multiple heap objects which are allocated at the same allocation site, as is often the case for iterators. We use the points-to information to apply a simple flow-insensitive checker [5]. For instance, consider the `HasNext` tracematch, together with a program that only calls `hasNext` on an iterator i (and never `next` on i). Clearly, the tracematch can never reach the final state for i. Our flow-insensitive checker identifies the shadows that do not contribute to a potential match because the objects at those shadows are lacking other—required—shadows to reach a final state.

At this point, we are left with a program where the remaining shadows all potentially contribute to triggering the tracematch on some object. However, there are typically still too many shadows to report to the user, since the flow-insensitive analysis is quite coarse. This paper proposes the use of flow-sensitive information to rule out further false positives (top of Figure 1).

We do so on a per-method, per-tracematch basis, first computing flow-sensitive alias information, to then detect and remove all "unnecessary" shadows based on this information. Such unnecessary shadows either do not affect the runtime monitor state or are subsumed by other shadows. Ruling out one shadow might render shadows in other methods unnecessary. Therefore, in principle, iterating analysis phases might help, as suggested by the loop in Figure 1. However, in practice, we have found it sufficient to just rerun the flow-insensitive checker [5].

Finally, we re-weave the program with an optimized version of the original runtime monitor. This monitor only processes shadows that have not been ruled out by the static analyses. We then emit a list of all shadows remaining in the program. In Section 4, we will explain how to extract potential points of failure from this list, and how we can successfully classify and visualize these program points, so that developers can more easily find program points that are likely to actually trigger the monitor.

3.2 Static analysis algorithm

Our flow-sensitive static analysis algorithm processes one method at a time. For each pair of method m and tracematch we apply two static analyses:

1. an intraprocedural flow-sensitive abstract interpretation of the runtime configuration. (Sections 3.2.1-3.2.4); and

2. an interprocedural flow-insensitive analysis which analyzes each final automaton configuration separately. (Section 3.2.5)

The first (intraprocedural) analysis determines what effect a method m could have on configurations that reach m. The analysis computes, for each of m's statements, all possible automaton configurations at that statement. If one of m's statements is associated with a configuration that has reached a final automaton state, we know that all shadows that generated this configuration need to be kept alive.

The second analysis models the continuation of control flow after the execution of m has finished, which ensures soundness: even if m itself did not make the tracematch automaton reach a final state, m could have generated (or discarded) a partial match which is then completed (or prevented from being completed) in the continuation of the execution.

3.2.1 Intraprocedural worklist algorithm

At the heart of the intraprocedural flow-sensitive analysis lies a standard worklist algorithm for computing configurations at statements, Algorithm 1.

Algorithm 1 Algorithm for computing configurations

let *initial* be the set of initial configurations
let *tm* be the tracematch to analyze
$wl := \emptyset$ // empty set
$computed := \emptyset$ // empty mapping
Let *head* be the entry statement of the current method
for initial configuration c in *initial* **do**
    $wl := wl \cup \{\ (head, c)\ \}$
end for
while wl non-empty **do**
    pop element $(stmt, c)$ from wl
    $cs' := \bigcup_{\substack{\text{shadow } s \\ \text{at } stmt}} \underline{transition}(c, s, tm)$
    for $c' \in cs'$ **do**
        if $c' \notin computed(s)$ **then**
            $computed(s) := computed(s) \cup \{\ c'\ \}$
            for $stmt' \in successors(stmt)$ **do**
                $wl := wl \cup \{\ (stmt', c')\ \}$
            end for
        end if
    end for
end while

We initialize the algorithm with a set of initial configurations. At runtime, there is only a single intial automaton configuration, which assigns **tt** to the initial, and **ff** to any non-initial state, since the porgram starts in the initial state only. However, at compile time, we do not know which partial matches may already have been produced at method entry by past program executions. Because our update rules act independently on each state, it is sound to start with a set of configurations where each configuration sets a different non-initial, non-final state to **tt**. For an automaton with four states, where the first state is initial and the last state is final, we obtain the following set of initial configurations:

$$\{(\mathbf{tt}, \mathbf{ff}, \mathbf{ff}, \mathbf{ff}), (\mathbf{tt}, \mathbf{tt}, \mathbf{ff}, \mathbf{ff}), (\mathbf{tt}, \mathbf{ff}, \mathbf{tt}, \mathbf{ff})\}.$$

The algorithm then intializes the worklist with a set of jobs, where each job associates a possible initial configuration with the entry statement of the control flow graph. The algorithm further propagates these jobs, computing new configurations by calling the function *transition* for every shadow at statement *stmt*. In our examples there is only at most one shadow per statement; however, in general, there can be multiple shadows, as multiple AspectJ pointcuts may match overlapping sets of statements. Whenever Algorithm 1 computes a configuration c' at a statement, and c' has not yet been computed at this statement before, the algorithm creates new jobs associating c' with each successor statement in the control flow graph.

Algorithm 2 Algorithm *transition*

let $p := (\mathbf{ff}, \ldots, \mathbf{ff})$ // empty configuration
let n be a copy of the current configuration c
let $l := label(s)$ be the label of the current shadow s
for edge (q_s, l, q_t) in automaton of *tm* **do**
    $p(q_t) := \underline{p(q_t) \vee (c(q_s) \wedge bind(s))}$ // positive update
end for
for non-initial, non-final state q in automaton of *tm* **do**
    if $\neg\exists$ loop (q, l, q) in automaton of *tm* **then**
        $n(q_t) := \underline{n(q_t) \wedge \neg bind(s)}$ // negative update
    end if
end for
return $(p \vee n)$ //state-wise disjunction

Algorithm 2 describes the function *transition*, which is the analogue of the runtime update rules for tracematches. We underlined the crucial calculations. The algorithm computes a successor configuration for c in two steps. First, it propagates disjuncts along automaton edges using a positive update rule. The new constraint $p(q_t)$ at the target state q_t is obtained by disjoining its old value with the constraint of the source state q_s, refined with the variable binding $bind(s)$ of the current shadow. In the second step we take care of "missing loops". If the tracematch automaton reads a shadow s with symbol l and on a state q there is no l-loop, then every binding that is compatible with the binding of s has to leave state q. This is implemented via the negative update rule. We return the state-wise disjunction of all bindings p that were propagated and all bindings n that remain at their current state q, due to a loop on q.

3.2.2 Analysis abstraction using instance keys

The crucial difference between Algorithm 2 and the runtime treatment arises when computing the underlined positive and negative update rules in Algorithm 2. At runtime, a conjunction $x = o_1 \wedge x \neq o_1$ is a contradiction and reduces to **ff**. We also want to deduce **ff** whenever possible at compile time, since **ff** certainly is not a false positive. However, at compile time, we have no access to runtime objects, potentially making it more difficult to find contradictions.

$i_1 \approx i_2$	May-alias
$i_1 = i_2$	Must-alias
$i_1 \neq i_2$	Must-not-alias

Figure 7: Aliasing relations between instance keys i_1 and i_2.

We therefore approximate runtime objects with *instance keys*. Instance keys are static representatives of heap objects, which we compute from a combination of (1) the flow-insensitive context-insensitive whole-program points-to analysis (which we re-use from previous stages), (2) a flow-sensitive intraprocedural must-not-alias analysis, and (3) a flow-sensitive intraprocedural must-alias analysis. Points-to and must-not-alias information allows us to determine that two instance keys i_1 and i_2 cannot possibly point to the same runtime object. We denote this relationship by $i_1 \neq i_2$. Furthermore, we say that two instance keys are equal, and write $i_1 = i_2$, if the keys are must-aliased; that is, the keys must represent the same runtime object. Finally, if two keys are neither must-aliased nor must-not-aliased, they are may-aliased, and we write $i_1 \approx i_2$. Figure 7 summarizes instance key notation for easy reference. In the following, we will distinguish a runtime object $o(\texttt{x})$ stored in variable `x` from the instance key $i(\texttt{x})$ that we use to model this object at compile time. A more detailed explanation of instance keys is out of the scope of this paper. We refer the reader to [6].

Precise information about the aliasing relationship between instance keys is crucial to our approach. Recall the `HasNext` tracematch from Section 2, and consider configuration $(\mathbf{tt}, \mathbf{ff}, \mathbf{ff})$ at the start of the `print` method from Figure 2. This is exactly the same situation as in Figure 5, but we now consider the compile-time abstraction. Because instance keys closely model runtime objects, states propagate exactly like at runtime, except that we bind instance keys ($i = i(\texttt{i2})$) instead of objects ($i = o(\texttt{i2})$). Consider the edge from **(C)** to **(B)**. At runtime we regain the configuration $(\mathbf{tt}, \mathbf{ff}, \mathbf{ff})$ because $o(\texttt{i1})$ and $o(\texttt{i2})$ are the same object. At compile time, we use our must-alias analysis to decide that $i(\texttt{i1}) = i(\texttt{i2})$. As a result, at compile time we are able to reconstruct the exact same evaluation as in Figure 5.

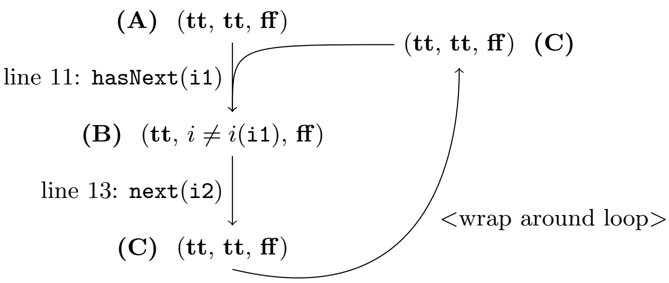

Figure 8: Effect of `print` on `HasNext` automaton.

Figure 8 shows the situation for the second initial configuration that we propagate, $(\mathbf{tt}, \mathbf{tt}, \mathbf{ff})$. This configuration "loops" just like the configuration in Figure 5: the must-alias analysis gives $i(\texttt{i1}) = i(\texttt{i2})$, so at line 13 we compute $c'(q_1) \equiv i \neq i(\texttt{i1}) \vee i = i(\texttt{i2}) \equiv \mathbf{tt}$. Because the `print` method can never reach the final state regardless of the initial configurations, we can conclude that `print` can never trigger the tracematch.

$\mathbf{i_1 \neq i_2}$	$x = i_1$	$x \neq i_1$
$x = i_2$	$\mathbf{ff}$	$x = i_2$
$x \neq i_2$	$x = i_1$	$x \neq i_1 \wedge x \neq i_2$

(a) Resulting disjuncts when i_1 and i_2 must-not-alias

$\mathbf{i_1 = i_2}$	$x = i_1$	$x \neq i_1$
$x = i_2$	$x = i_1 \equiv x = i_2$	$\mathbf{ff}$
$x \neq i_2$	$\mathbf{ff}$	$x \neq i_1 \equiv x \neq i_2$

(b) Resulting disjuncts when i_1 and i_2 must-alias

Table 1: Analysis-aware reduction rules.

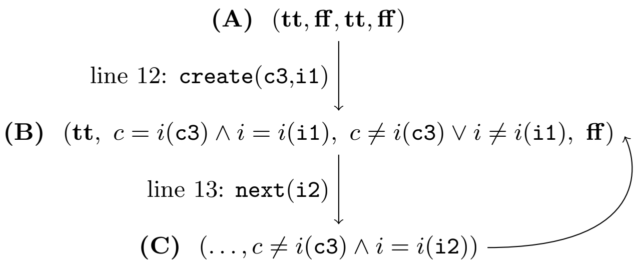

Figure 9: Effect of `print` on `FailSafeIter` automaton.

Table 1 summarizes the rules that we can use in the presence of aliasing information. With seven of these rules, aliasing information enables us to deduce that certain constraints are redundant or contradictory and therefore that the shadows that generated these constraints cannot contribute to reaching a final state. The rule shown in gray does not benefit from this information. When we conjoin two negative bindings for potentially different values, we have to store both negative bindings (as is also the case at runtime).

3.2.3 *Unique-shadow analyses and rules*

During the course of our experiments we found that two additional rules enabled us to avoid spurious false positives in the multiple-bound-variable case. Consider again the `print` method from Figure 2, but instead with the `FailSafeIter` tracematch from Figure 6. The `print` method contains a `create` shadow binding both `c3` and `i1` at line 10 and a `next` shadow binding `i2` at line 13. Assume that we start the static analysis with $(\mathbf{tt}, \mathbf{ff}, \mathbf{tt}, \mathbf{ff})$ **(A)** as shown in Figure 9. The `create` shadow would give the configuration **(B)** after line 12. Because of the `next`-edge from q_2 to q_3 (Figure 6), the `next` shadow at line 13 would then update the configuration at the final state q_3 to

$$(\ c \neq i(\texttt{c3}) \wedge i = i(\texttt{i2})\ ) \vee (\ i \neq i(\texttt{i1}) \wedge i = i(\texttt{i2})\ ).$$

Must-alias information eliminates the second disjunct:

$$\mathbf{(C)}_{|q_3}: \quad c \neq i(\texttt{c3}) \wedge i = i(\texttt{i2})$$

which means that the `next` shadow might trigger the final state q_3 on some collection (but not $i(\texttt{c3})$) and iterator $i(\texttt{i2})$. Now, we as programmers know that $i(\texttt{i3})$ can only be associated with the collection $i(\texttt{c3})$: an `Iterator` object will only be bound together with a single `Collection` object. However, the analysis seen so far has no way to infer that

$$i = i(\texttt{i2}) \Rightarrow c = i(\texttt{c3}). \quad (4)$$

Inferring this equation would allow us to reduce $\mathbf{(C)}_{|q_3}$ to $\mathbf{ff}$, proving that `print` never violates the `FailSafeIter` prop-

Algorithm 3 Algorithm *contradictsUniqueBinding*

1: let $x = i(\mathtt{v})$ be the incoming binding, $bind(s)$.
2: $bindLab := \emptyset$
3: **for** edge (q_s, l, q_t) in the automaton of tm **do**
4: **if** x may be unbound at q_s and l binds x **then**
5: $bindLab := bindLab \cup \{\ l\ \}$
6: **end if**
7: **end for**
8: $bindShadows := \{s \mid s \text{ shadow}, label(s) \in bindLab\}$
9: **for** positive binding $y = i(\mathtt{w})$ in current disjunct **do**
10: **if** $\forall s \in bindShadows\ .\ s$ may bind $(x = i(\mathtt{v}))$ $\implies s$ cannot bind $(y = i(\mathtt{w}))$ **then**
11: **return** true
12: **end if**
13: **end for**
14: **for** negative binding $y \neq i(\mathtt{w})$ in current disjunct **do**
15: **if** $\forall s \in bindShadows\ .\ s$ may bind $(x = i(\mathtt{v}))$ $\implies s$ must bind $(y = i(\mathtt{w}))$ **then**
16: **return** true
17: **end if**
18: **end for**
19: **return** false

erty. A key insight is that equation (4) holds if we can establish that all shadows in the program that bind i to $i(\mathtt{i2})$ must also bind c to $i(\mathtt{c3})$ (if they bind c at all). If this holds, then $\mathbf{(C)}_{|q_3} = \mathbf{ff}$, which guarantees that the `print` method never violates the tracematch.

Algorithm 3 presents the generic uniqueness check. We apply this check whenever computing conjunctions in the positive and negative update rules of Algorithm 2. Assume that the binding of the current shadow is $bind(s)$ and $\{x = i(\mathtt{v})\} \subseteq bind(s)$. The check first determines the set of all shadows that may possibly bind x. Next (lines 9-13), if there already exists a positive binding $y = i(\mathtt{w})$ in the current disjunct, and for all shadows binding x, we have $x = i(\mathtt{v}) \implies y \neq i(\mathtt{w})$, then this contradicts the binding $x = i(\mathtt{v}) \wedge y = i(\mathtt{w})$, which we are trying to generate. Therefore we return *true*, which will instruct Algorithm 2 to reduce the conjunction to **ff**. Lines 14-19 perform the inverse check for an existing negative binding.

This concludes our discussion of constraint update rules. Next we describe how we can actually eliminate shadows based on the analysis results.

3.2.4 Tracking shadows with shadow histories

When the intraprocedural worklist algorithm terminates, it leaves us with a set of possible configurations at each statement. If any of these configurations has a constraint at a final state that is different from **ff**, this means that this automaton configuration may have triggered the tracematch at runtime. We therefore need to keep all shadows alive that may have lead to this configuration being generated. But how can we determine these shadows? We chose to record shadow information on-the-fly. Whenever Algorithm 2 performs a positive or negative update for a shadow s, generating a new binding b (of the form $x = i$ or $x \neq i$) which cannot be avoided by any of the reduction rules, we attach to b the information that b was generated by s. We call this shadow information b's "shadow history". If our analysis later eliminates b, e.g. due to one of the reduction rules, we drop the corresponding entry in the history entry as well. This is sound because our reduction rules have implied that s was unnecessary. After the worklist algorithm finishes, the necessary shadows are the shadows in the history of all (positive and negative) bindings at all final states of all configurations at every statement in the method. The remaining shadows are unnecessary, at least at an intraprocedural level, and can therefore in principle be removed. However, to maintain soundness we must also consider interprocedural control flow and data flow.

3.2.5 Conservative assumptions made for soundness

The analysis algorithm which we presented in the last sections is truly intraprocedural, i.e. only regards one single method at a time. To maintain sound results for a method m, we have to take interprocedural control flow and data flow into account, from three different directions.

1. the execution preceding the execution of m,
2. methods called by m,
3. the execution following the execution of m.

We have already taken care of (1): because we initialize our abstraction with the set of all possible initial configurations, with a different non-initial, non-final state set to **tt** each time, we soundly over-approximate every execution preceding the execution of m.

We solve (2) in the following way. First, m might call only "harmless" methods. For example, in case of the `HasNext` tracematch, if m only uses an iterator $o(\mathtt{i1})$, m calls m', which only uses iterator $o(\mathtt{i2})$, and $i(\mathtt{i1}) \neq i(\mathtt{i2})$, then we know that m' cannot jeopardize the soundness of the analysis results for m. We can query our flow-insensitive analysis from earlier work [5] to enumerate all shadows that have bindings that may be compatible with the binding of any shadow in m. We use this information as follows. Whenever Algorithm 1 processes a configuration c at a statement *stmt*, and *stmt* may (transitively) call a method with such a compatible shadow, we "taint" c. Algorithm 1 then further propagates taintedness. Any successor configuration of a tainted configuration c will be tainted as well. Then, after Algorithm 1 has finished, we protect all shadows at any statement that holds a tainted configuration from being removed. This approximation is very conservative, but it makes our results sound because we never remove a shadow based on a possibly unsound configuation.

Algorithm 4 handles all potential traces following m. The intraprocedural worklist algorithm leaves us with a set of possible configurations C at each statement. We collect all configurations at each exit statement of m's control flow graph. Then for each configuration $c \in C$ and each automaton state q we determine the labels *haveLbls* of all shadows in the program that are compatible with any shadow in $history(c(q))$. If in the automaton there exists a path from q to a final state q_F and all labels of this path are in *haveLbls*, i.e. exist somewhere in the program, with compatible bindings, we have to retain all shadows in $history(c(q))$, as $c(q)$ could potentially reach q_F. Note that this algorithm subsumes the simple intraprocedural case: when $q = q_F$, then such a path exists trivially, namely the empty path.

Although the analysis for each method will always terminate, usually within a second, it may take too long for some very large methods with complicated aliasing relationships. To reduce waiting time, we give programmers the op-

Algorithm 4 Algorithm *soundContinuation*

$endConfigs = \{config(stmt) \mid stmt$ is exit node of $m\}$
for $c \in endConfigs$ **do**
  for state q of the automaton of tm **do**
    $haveLbls = \{\ l \ \mid \exists$ shadow s . $\exists\ s_h \in history(c(q))$. s compatible with $s_h \ \wedge\ l = label(s)\}$
    if $\exists$ path p from q to final state q_F $\wedge$ $\forall (q_s, l, q_t) \in p$. $l \in haveLbls$ **then**
      retain all shadows in $history(c(q))$
    end if
  end for
end for

tion to abort the iteration after N worklist jobs have been processed, retaining all the method's shadows. For our experiments we chose N=3000. This value made the analysis abort on only 12 out of 1053 methods in total. Our analysis would not have found an exact result for these methods in any case, due to their complicated aliasing relationships.

The output of our analysis is a list of potential points of failure, i.e. shadows that can immediately trigger a tracematch body. For each such point we also report the point's "context shadows", i.e. shadows that build up the partial match which is then completed by the point of failures. For `HasNext`, each remaining `next` shadow would be a potential point of failure, while its context shadows could be both `next` and `hasNext` shadows on aliased iterators.

4. CLASSIFICATION & VISUALIZATION

We have explored two ways of making our results more accessible. First, we used machine learning algorithms to create a decision tree that would distinguish between false positives and true positives, based on auxiliary information that we collected during analysis. Second, we developed a visualization tool to graphically display potential matches, thereby enabling developers to understand sequences of events potentially leading to failures.

4.1 Decision Trees

Although our static analyses removes many false positives, factors like dynamic class loading and interprocedural data flow can still lead to imprecise analysis results; on our benchmark set, we still report 461 points of potential interest. To help the developer focus on the most important points first, we modified our analysis to report the approximations it made in each particular case—that is, why it reported each positive—and used this information as a feature vector for a machine learning algorithm. We then inspected all 461 potential points of failure by hand and marked actual matches, giving us training data. We also ran the respective benchmarks with monitoring instrumentation to validate the results gained by manual inspection. Interestingly, most matches we had identified manually were not exercised by the benchmark harness. We then used the Weka machine learning toolkit [19] to create decision trees which would distinguish false positives from true positives. We will present a detailed discussion of our methodology and the results of our machine learning approach in Section 5.

4.2 Visualization

Our system helps developers better understand interactions between their programs and tracematches by providing a graphical view of tracematches. In particular, our compiler can emit tracematch automata in GraphViz [10] format and create imagemaps associated with these automata. Our imagemaps enable developers to inspect the automata and select edges of interest—both potential points of failure and transitions leading up to the PPFs—and view the lines of code responsible for executing those transitions. We emit one copy of the automaton per potential failure group and rank the groups by the number of approximations they encompass (as defined in Section 5. Visualizations for our benchmark programs are available on our results webpage:

`http://www.aspectbench.org/tmahead/tms`.

5. EXPERIMENTS

We evaluated the effectiveness of our analysis and classification algorithm using the 99 combinations of nine tracematches with the ten benchmarks of the current version 2006-10-MR2 of the DaCapo benchmark suite [4] and with SciMark2 [14]. To SciMark2 we applied four additional tracematches to validate the correct use of stopwatches, which the benchmark uses to collect execution times. Altogether this leads to a total of 103 tested tracematch/benchmark combinations. Table 2 gives brief descriptions of the tracematches we used.

For many of the 103 benchmark/tracematch combinations, the flow-insensitive analysis from previous work [5] detected that the tracematch could never apply. However, 38 benchmark/tracematch combinations had remaining shadows. We applied our flow-sensitive analysis and our classification to these cases. We uploaded all benchmarks, tracematches and raw data to: `http://www.aspectbench.org/tmahead`

Potential and reported points of failure.

Columns two and three of Table 3 on page 9 present the number of potential points of failure remaining after the flow-insensitive analysis (fi) [5] and the new flow-sensitive analysis presented in this paper (fs). Note that we are able to rule out *all* potential points of failure in 19 of the 38 cases. In average, the flow-sensitive stage removed 58% of all potential points of failure, or 75% of false positives. Column four shows the number of actual points of failure that we identified through manual inspection of the results. For benchmarks where many potential points of failure remain, we assist the programmer through our machine learning based filtering and ranking.

Filtering and ranking using decision trees.

We extended our static analysis to attach up to seven features to each potential point of failure. A feature is present on a point of failure if the static analysis encounters the feature on the point of failure itself or on a "context shadow" (see end of Section 3). Although our feature vectors contain seven features, the decision tree Weka generated only uses four of these features to classify points of failure. `ABORTED` means that the intraprocedural analysis was aborted after 3000 worklist iterations (see end of Section 3). `CALL` means that the static analysis retained a shadow due to tainting at unsafe method calls (Section 3). `DELEGATE` identifies shadows at delegating calls, for instance at a call `inner.hasNext()` within `wrapper.hasNext()`. Such calls are usually uninteresting because clients use `inner` correctly whenever they use `wrapper` correctly. `NO_CONTEXT` means that one of the shad-

ASyncIterC	only iterate a synchronized collection c when owning a lock on c
ASyncIterM	only iterate a synchronized map m when owning a lock on m
FailSafeEnum	do not update a vector while iterating over it
FailSafeIter	do not update a collection while iterating over it
HasNextElem	always call hasNextElem before calling nextElement on an Enumeration
HasNext	always call hasNext before calling next on an Iterator
LeakingSync	only access a synchronized collection using its synchronized wrapper
Reader	do not use a Reader after its InputStream was closed
Writer	do not use a Writer after its OutputStream was closed
ResetRead	on a stopwatch, do not call reset followed by read without calling start in between
StartResume	on a stopwatch, do not call start followed by resume without calling stop in between
StartStart	on a stopwatch, do not call start twice without calling stop in between
StopStop	on a stopwatch, do not call stop twice without calling start or resume in between

Table 2: The generic and domain specific tracematch patterns we used; code at: `http://www.aspectbench.org/tmahead/tms`

benchmark-tracematch	PPF fi	PPF fs	APF	filtered
antlr-HasNextElem	11			
antlr-Reader	5			
bloat-FailSafeIter	282	245		
bloat-HasNext	308	82	1	1
bloat-Writer	26			
chart-FailSafeIter	41	23		
chart-HasNext	41			
eclipse-ASyncIterC	3			
eclipse-ASyncIterM	5			
eclipse-FailSafeEnum	5	2		
eclipse-FailSafeIter	3	1		
eclipse-HasNextElem	17	7	5	6 (+1)
eclipse-HasNext	4	1	1	1
eclipse-LeakingSync	1	1		
eclipse-Reader	2	1		
fop-FailSafeEnum	2	1		
fop-FailSafeIter	1			
fop-HasNextElem	4			
fop-HasNext	2			
jython-FailSafeEnum	1	1		
jython-FailSafeIter	12	8		
jython-HasNextElem	21	12		
jython-HasNext	12	2		
jython-Reader	2	1	1	0 (-1)
lucene-FailSafeEnum	3			
lucene-FailSafeIter	17			
lucene-HasNextElem	7			
lucene-HasNext	17			
pmd-FailSafeEnum	2			
pmd-FailSafeIter	104	63		
pmd-HasNextElem	3			
pmd-HasNext	119	6	4	3 (-1)
pmd-Reader	5			
xalan-HasNextElem	1			
scimark-ResetRead	2	2		
scimark-StartResume	1			
scimark-StartStart	2	2		
scimark-StopStop	2			
sum	1096	461	12	11

Table 3: Potential points of failure (PPF) after flow-insensitive analysis (fi, [5]) and flow-sensitive analysis (fs, this paper); actual points of failure (APF); PPF remaining after filtering (filtered), false positives/negatives in brackets

ows holds points-to sets without context information—that is, the demand-driven analysis failed to find context information in its time budget. Weka reports the following decision tree for our data set:

```
CALL = 0
|   ABORTED = 0
|   |   DELEGATE = 0
|   |   |   NO_CONTEXT = 0: TRUE_POSITIVE (11.0/1.0)
|   |   |   NO_CONTEXT = 1: FALSE_POSITIVE (4.0/1.0)
|   |   DELEGATE = 1: FALSE_POSITIVE (10.0)
|   ABORTED = 1: FALSE_POSITIVE (30.0)
CALL = 1: FALSE_POSITIVE (406.0/1.0)
```

The tree tells us that it is best to classify a potential point of failure as `FALSE_POSITIVE`, if any of `CALL`, `ABORTED`, `DELEGATE` or `NO_CONTEXT` are attached.

Weka evaluates its classifiers with 10-fold stratified cross-validation [11] to ensure that the classifiers do not overfit the data. Under cross-validation, 458 of the 461 instances are correctly classified, while three of the instances are misclassified (one false positive and two missed true positives). In these cases, our feature vectors do not contain enough information to deduce the correct answer. The last column of Table 3 shows the number of filtered point of failures. Note that we manage to filter out *all* false positives for benchmarks that have no actual points of failure.

The most striking feature of the decision tree is the fact that `CALL` almost always indicates a false positive. Hence our tracematch properties are almost always violated intraprocedurally, with only one exception in our test data. We realize that this is the case because most actual violations in our benchmarks occur on the HasNext and HasNextElem tracematches, which bind iterators and enumerations, both of which hardly ever leak out of the method in which they are created. This suggests that it might be useful to train the machine learning algorithm for each tracematch separately. We decided to answer this question in future work, as this would require yet more benchmarks.

benchmark-tracematch	PPF un-filtered	APF ranks
bloat-HasNext	82	1
eclipse-HasNextElem	7	1,2,3,4,6
eclipse-HasNext	1	1
jython-Reader	1	1
pmd-HasNext	6	1,2,3,4

Table 4: PPF un-filtered; ranks of APF in list of PPFs

When we present the un-filtered list to the user, we rank it such that potential points of failure that have fewer of the four negative features attached than others appear further up the list. The last column of Table 4 shows the ranks that we assign to actual points of failure. As the results show, actual points of failure are ranked close to the top. Very compelling is bloat-HasNext: without filtering, its actual point of failure ranks on place 1 of 82, with filtering enabled, it is the only reported point of failure—a perfect match.

Generally, we propose the following workflow: the developer runs our static analysis on a program; if there are a small number of potential points of failure, then the developer can verify each of these points manually and get a sound analysis result. However, if there are many points, then the developer can inspect just the top or filtered results and let runtime monitoring handle the remaining cases.

Suspicious code and defects.

CLEAN THIS TEXT

In pmd-HasNext a method passes an iterator to another method as an argument. The callee method then extracts the first element without any further check. While this is not a direct bug, the missing precondition on the callee might cause problems for long-term software maintenance. We therefore marked this case as a true positive. Interestingly, the method was fixed in a later version of pmd, when Java5 for-loops were introduced to the code, which avoid the explicit use of iterators. The actual point of failure in jython-Reader indicates an actual defect. The code sometimes closes `Reader` objects and then reads from them. The developers "cured" this defect by returning `null` from the method that reads from the Reader, in case of an `IOException`. bloat-HasNext extracts two elements from a collection without any checks.

Analysis times.

We ran our analyses on IBM's J9 VM (build 2.3, J2RE 1.5.0 IBM J9 2.3 Linux amd64-64), with 3GB of maximal heap space on a machine with a AMD Athlon 64 X2 Dual Core Processor 3800+ running Ubuntu 7.10 with kernel version 2.6.22-14. We found out that on average only 4% of the total compilation and analysis time was spent in our novel flow-sensitive analysis. On average, 50% of the time was spent on computing points-to sets. The remaining 46% were spent on normal compiler passes. An average total compilation, including all analyses, took 6 minutes. By far the worst case was bloat-FailSafeIter, where we had to apply the flow-sensitive analysis to 240 methods, significantly more than the average of 18 methods. This resulted in a total runtime of eight minutes and 17 seconds for the flow-sensitive stage, or 40% of this benchmark's total compilation time of 20:51.

6. RELATED WORK

We compare our work to software for anomaly detection and on the alternate specification languages encapsulated in typestate-based approaches and PQL.

Java Anomaly Detector (JADET).

Wasylkowski, Zeller and Lindig expose the Java Anomaly Detector (JADET) to detect object usage anomalies in a given program. JADET first generates a list of common usage patterns from the program, considering control flow, one object at a time. The tool then produces a list of uncommon usages contradicting the pattern, ranked by a confidence value. The authors show that seven uncommon usages represent actual defects. JADET's analysis is unsound.

JADET's mining phase makes the tool fully automated, admittedly a great advantage over our approach, as no specification is necessary. Tracematches can however validate more complicated usage patterns, in particular patterns involving multiple objects, such as `FailSafeIter`. Furthermore, regular expressions are more expressive than JADET's pattern language, which only allows relationships of the form "event e may precede event f". It would be interesting to see an extension of JADET to tracematch-like patterns.

Typestate.

Typestate properties [16] have been enjoying renewed interest. Typestate describes the state of heap objects but, as JADET, one at a time. Because of this restriction, typestate is less general than tracematches. All of the following approaches are sound, i.e. do not miss true positives.

DeLine and Fähndrich [7] check typestate specifications statically, in the presence of aliasing. The authors implemented their approach in the Fugue tool for specifying and checking typestates in .NET-based programs.

Fink et al. present a static analysis for runtime checking of typestate properties [9]. Their approach, like ours, uses a staged analysis which starts with a flow-insensitive pointer-based analysis, followed by flow-sensitive checkers. Their analyses all rely on the fact that typestate specifies properties of a single object at a time. Like us, Fink et al. aim to verify properties fully statically. However, our approach enables the use of specialized instrumentation and recovery code, while their approach emits a compile-time warning. Also, tracematches let developers specify the properties to be verified, while Fink et al. do not say how developers might specify their properties.

Bierhoff and Aldrich [3] recently presented an intraprocedural approach which enables the checking of typestate properties in the presence of aliasing. Their system propagates access permissions with references, which permits reasoning about the scope of the program that has access to any given reference. The authors use reference counters to reclaim permissions and enhance precision. Their abstraction is based on linear logic, and it can relate the states of one object (e.g. an iterator) with the state of another object (e.g. a collection) using access permissions, but only if that object is stored in a field. In our approach, objects do not have to be related in the heap. A key difference between their approach and ours is that they require annotations describing access permissions at method boundaries, while we have found that worst-case assumptions coupled with side-effect information are surprisingly powerful.

Dwyer and Purandare use existing typestate analyses to specialize runtime monitors [8], as we do. Their work identifies *safe regions* in the code using an out-of-the-box typestate analysis. Safe regions can be methods, single statements or compound statements (e.g. loops). A region is safe if its transition function (1) does not drive the typestate automaton into a final state and (2) is deterministic. While our approach focuses on identifying instrumentation that has no "net effect", Dwyer and Purandare summarize transitions from regions that may have a net effect, if this effect can be statically determined. This speeds up runtime evaluation of typestate properties, similar to our optimized monitors. To statically remove false positives however, a region must have no net effect.

Program Query Language.

The Program Query Language [12] resembles tracematches in that it enables developers to specify properties of Java programs, where each property may bind free variables to runtime heap objects. PQL supports a richer specification language than tracematches: it uses stack automata rather than finite state machines. PQL proposes a flow-insensitive approach (like [5]); no flow-insensitive analysis can remove the shadows that interest us here.

To summarize, our approach is the first to present a static flow-sensitive analysis to handle temporal properties involving more than one object at a time. Moreover it is the first flow-sensitive static analysis approach that provides programmers with a specialized version of a runtime monitor. To our best knowledge, we are the first to provide a visualization tool that links abstract events on the automaton level to corresponding concrete program locations and locations related through aliasing.

7. CONCLUSIONS

Analyzing tracematches ahead-of-time clearly is a non-trivial problem. Because tracematches can refer to multiple objects and at various places in the program, one might expect that an interprocedural flow-sensitive analysis would be required. Therefore we were surprised to find that combining inexpensive whole-program summary information with a suite of carefully-designed intraprocedural flow-sensitive analyses was successful for the majority of our benchmarks.

Our approach allows programmers to gain confidence in their code, in cases where we report that the property stated by the tracematch can never be violated. This is the case for 50% of our benchmarks. When we do report potential points of failure, we use machine learning to sort or filter the report so that unlikely points of failure move to the bottom of the list or filtered out. Our approach revealed several defects in large software systems. In all these cases, the actual points of failure were ranked at the top of our report.

If the programmer even wants to play safe at unlikely points of failure, she can instruct our verification tool to instrument the program under test with an optimized runtime monitor. This monitor only updates its internal state at code positions which our analysis failed to rule out statically. Yet, we guarantee that the monitor is sound, i.e. it will report every actual violation at runtime.

We believe that this combined approach will greatly support programmers in finding erroneous uses of application interfaces in their programs.

8. REFERENCES

[1] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding Trace Matching with Free Variables to AspectJ. In *OOPSLA*, pages 345–364. ACM Press, 2005.

[2] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. *abc*: An extensible AspectJ compiler. In *AOSD*, pages 87–98. ACM Press, 2005.

[3] K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In *OOPSLA*, pages 301–320. ACM Press, 2007.

[4] 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. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, pages 169–190, 2006.

[5] E. Bodden, L. J. Hendren, and O. Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *ECOOP*, 2007.

[6] E. Bodden, P. Lam, and L. Hendren. Instance keys: A technique for sharpening whole-program pointer analyses with intraprocedural information. Technical Report SABLE-TR-2007-8, Sable Research Group, McGill University, October 2007.

[7] R. DeLine and M. Fähndrich. Typestates for objects. In *ECOOP*, volume 3086 of *LNCS*, pages 465–490. Springer, 2004.

[8] M. B. Dwyer and R. Purandare. Residual dynamic typestate analysis: Exploiting static analysis results to reformulate and reduce the cost of dynamic analysis. In *ASE*, pages 124–133. ACM Press, 2007.

[9] S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. In *ISSTA*, 2006.

[10] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Software — Practice and Experience*, 30(11):1203–1233, 2000.

[11] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In M. Kaufmann, editor, *Fourteenth International Conference on Artificial Intelligence (IJCAI), San Mateo, CA*, pages 1137–1143, 1995.

[12] M. Martin, B. Livshits, and M. S. Lam. Finding application errors using PQL: a program query language. In *OOPSLA*, pages 365–383, 2005.

[13] H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In *CC*, volume 2622 of *LNCS*, pages 46–60, 2003.

[14] R. Pozo and B. Miller. Scimark 2.0, June 2000. `http://math.nist.gov/scimark`.

[15] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for Java. In *PLDI*, pages 387–400, 2006.

[16] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12(1):157–171, 1986.

[17] The AspectJ Team. The AspectJ Programming Guide.

[18] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[19] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques (Second Edition)*. Morgan Kaufmann, 2005. June.