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 allows programmers to validate, for instance, the proper use of application interfaces. Given a property specification, a runtime monitor tracks appropriate runtime events to detect violations and possibly execute recovery code. Although powerful, runtime monitoring inspects only one program run at a time and so may require many program runs to find errors. Therefore, in this paper, we present ahead-of-time techniques that can (1) prove the absence of property violations on all program runs, or (2) flag locations where violations are likely to occur. Our work focuses on tracematches, an expressive runtime monitoring notation for reasoning about groups of correlated objects. We describe a novel flow-sensitive static analysis for analyzing monitor 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 heap references by combining the results of three points-to and alias analyses. We also propose a machine learning phase to filter out likely false positives. We applied a set of 13 tracematches to the DaCapo benchmark suite and SciMark2. Our static analysis rules 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 three of 461 cases. The approach revealed defects and suspicious code in three benchmark programs.

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

Experimentation, Reliability, Verification

Keywords

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

1. INTRODUCTION

A program's sequence of events over an execution is a rich source of information about the program's behaviour on that execution. Some sequences of runtime events indicate defects in the program: for instance, programs must not advance iterators with no more elements. Runtime monitoring can detect such sequences of events, enabling developers to handle the sequences with reporting or recovery code.

However, runtime monitoring can only detect errors as they occur, and furthermore inspects only one program execution at a time and so may require many executions to find errors. Errors may therefore be hard to find and can remain unnoticed until late in the development process or even until after a program is deployed.

Our research aims to verify runtime monitoring properties ahead-of-time, through static analysis. Static verification can (1) prove the absence of error conditions on all executions, or (2) flag all code locations where errors may possibly occur. We designed a "complete" static analysis—no missed violations—that would report as few false positives as possible. However, to design a complete static analysis, we had to make conservative assumptions, which potentially lead to false positives. To mitigate the impact of false positives, we developed a new (optional) machine learning approach that filters out likely false positives, which enables developers to concentrate on program points that are likely points of failure. Our combined approach enables the programmer to (1) manually inspect all potential points of failure (starting with the likely points of failure), and (2) specify recovery code to gracefully handle all remaining potential points of failure (which she could also inspect manually).

We focus on one particular approach to runtime monitoring: tracematches [2], 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 such a trace occurs on a program execution. A symbol's free variables bind heap objects at runtime. A tracematch triggers when its regular expression matches a suffix of the current execution trace with a consistent variable binding. For example, tracematches can monitor for inappropriate use of iterators; the regular expression "`next(i) next(i)`" over the alphabet {`hasNext(i)`, `next(i)`} matches if a program calls the `next()` method twice on an iterator `i` without any intervening call to `hasNext()`.

Figure 1 shows our complete analysis approach for tracematches (an explanation follows). The fact that tracematch

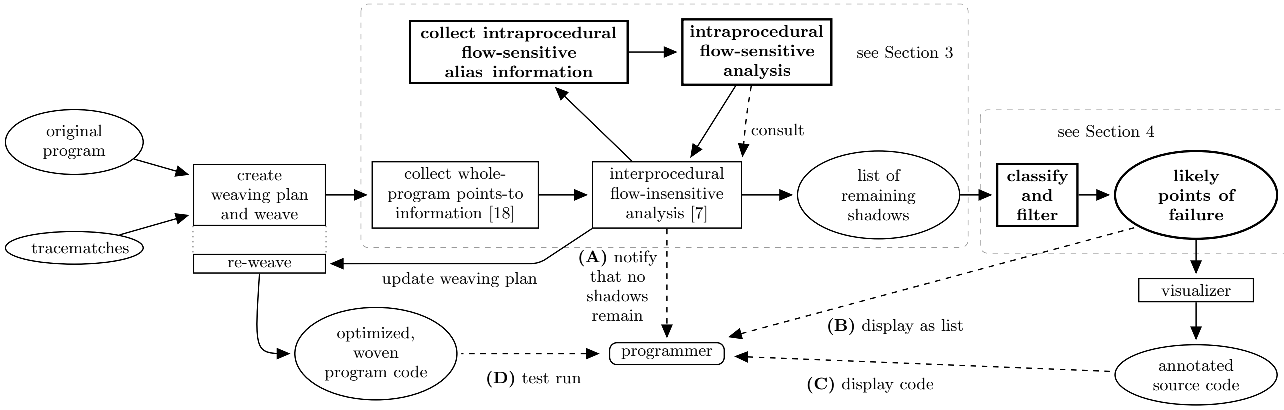

Figure 1: Our complete approach to static verification using tracematches; novel phases shown in boldface; solid arrows represent data flow, dashed arrows possible interaction

events bind variables to heap objects at runtime is *the* key challenge for any static analysis of tracematches. In the above example, the variable `i` would be bound at the first `next(i)` event and matched at the second `next(i)` event. 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 be both interprocedural and flow-sensitive. In previous work [7] we tried such an approach and failed: we found that a precise analysis of tracematches requires must-alias information for strong updates. Such information is very expensive to compute at a whole-program level. Our earlier work further showed that an interprocedural but flow-insensitive analysis could only eliminate the partial matches of tracematches that could not be completed for trivial reasons, e.g. iterators which receive calls to `hasNext()` but never to `next()`.

We therefore took a step back and determined where and how tracematches matched in our benchmarks. To our surprise, many tracematches described mostly-local patterns, often confined to a single method. This led us to the principal finding of *this* paper: A suite of carefully-designed *intra*procedural flow-sensitive analyses (top of Figure 1) can successfully reason about tracematch states, if intelligently combined with precise local alias information and inexpensive whole-program summary information.

Note that tracematches are much harder to analyze than typestate [19]. In typestate, the type system associates a state with each individual object of a given type. Static typestate analyses can therefore track the possible states of each of a program's allocation sites individually [11]. Many tracematches, however, have multiple free variables. In such situations, multiple objects share a joint state. A naive attempt must therefore track the states of all combinations of allocation sites—an exponential blowup. To avoid this blowup, we instead encode knowledge about entire *equivalence classes* of objects using constraints. Our abstraction associates two kinds of information with each state q: positive information $x = o$ (object o is in state q, bound to tracematch variable x) and negative information $x \neq o$ (any object except o may be in q and bound to x). In our benchmarks, the combination of positive and negative information enables us to eliminate most false positives.

As the Figure shows, our static analysis can be useful in various cases. 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 all iterators are checked for having a next element before they are advanced. We use the term *shadow* [15] to denote a program point corresponding to a runtime event of interest. If no shadows remain, the program satisfies the monitored property.

If shadows do remain, we identify potential points of failure. Such points may directly cause the tracematch body to execute, e.g. any `next`-shadow in the iterator 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 the future, we plan to provide a visualization tool that the programmer can use to browse **(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 still possible), we provide an option to **(D)** generate an optimized instrumented program that includes runtime monitoring code only at points where our static analysis could not rule out potential matches statically. Because our approach is complete, this monitor catches *all* actual errors.

We have implemented our static analysis as an extension to the AspectBench Compiler [3]. We applied the analysis to 103 combinations of tracematches with SciMark2 [17] and the entire DaCapo benchmark suite [5], and found that 38 of these combinations showed potential property violations. Our static analysis ruled out all potential points of failure in 50% of these cases; overall, it ruled out 75% of the potential points of failure. For all but 5 benchmark/tracematch combinations, fewer than 10 such points remained.

We found actual property violations in 5 cases. In Jython and bloat, this violation pointed out actual errors. 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;

```
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

- a machine learning algorithm to identify likely points of failure among potential points of failure; 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 conclusions.

2. TRACEMATCHES AT RUNTIME

In this section, we describe tracematches [2], a mechanism for runtime monitoring; explain how a compiler creates code that implements tracematches at runtime; and describe how the runtime system tracks tracematch states. Although this section presents the situation at runtime, our two illustrative examples foreshadow our static analyses from 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 the method `print` for printing. We explicitly added copy statements at lines 4 and 12 to emphasize the problem of aliasing.

Like many Java programs, our example uses iterators and collections, which come with implicit API usage contracts (see below). Tracematches can verify such contracts.

2.1 HasNext example tracematch

Figure 3 presents the `HasNext` verification tracematch. This tracematch identifies 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. The examples in this paper use `call` pointcuts. In principle, programmers could use any AspectJ pointcut in their tracematch symbols. However, some pointcuts, like `cflow`, may make less sense in a tracematch symbol definition than in an advice definition, since tracematches can partly subsume the `cflow` construct. The `let` pointcut is most useful within tracematch symbol definitions: `let(v,exp)` binds an expression `exp` to tracematch variable `v` when the symbol matches, which allows the programmer to bind context information to a variable that is not accessible via `this`, `target` or `args`. The `let` pointcut is only available in the AspectBench Compiler.

```
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); } }
```

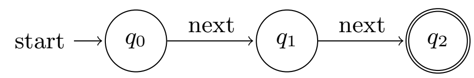

Figure 3: HasNext tracematch and automaton: do not call `next()` twice without an intervening call to `hasNext()`.

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.

In the following we distinguish the concrete program trace, which consists of all AspectJ joinpoints (including method calls, field assignments and the execution of exception handlers), from the abstract event sequence, as seen by tracematches. The abstract sequence consists only of symbol names and, as we will see later, variable bindings. This sequence therefore abstracts from AspectJ's concrete joinpoint model. The tracematch runtime matches the regular expression against each *suffix* of this 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) is not matched, but projection (2) is, 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 using tracematches, and then feeds the tracematches and the program under test to the AspectBench Compiler [3] (`abc`). `abc` first creates an automaton from each tracematch's regular expression. Figure 3 presents the automaton for `HasNext` below the tracematch definition. `abc` then identifies instrumentation points, or *shadows* [15], as described by the tracematch's symbols. At each shadow, the compiler adds code to update the tracematch state in response to program events. Figure 2 includes shadows as comments. In line with their symbol declarations, shadows may bind tracematch variables.

Since different tracematch symbols may bind different subsets of tracematch variables, heap objects may simultaneously be in many automaton states, and the runtime must store mappings from variable bindings to states. It does so by attaching a constraint to each automaton state [2]. Constraints are logical formulae which can be evaluated to determine whether a given variable-to-object binding holds in the given state.

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

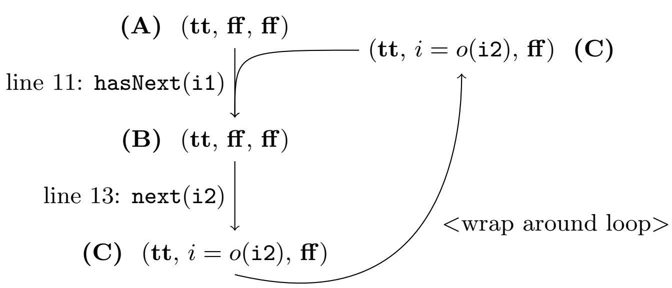

Figure 4: Effect of `print` on `HasNext` automaton.

As the program executes, the runtime updates constraints as events occur. Figure 4 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 $\mathbf{ff}$ **(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 constraint updates inside the loop have no effect (as in Equation (2)).

```
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 { ... } }
```

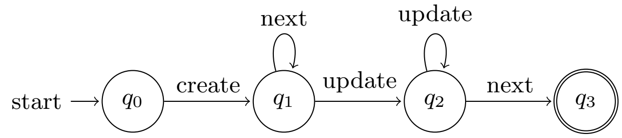

Figure 5: FailSafeIter tracematch and automaton: detect updates to a Collection which is being iterated over.

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 5. 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. Note that traditional typestate [19] approaches cannot bind multiple variables and therefore cannot directly describe or verify such properties.

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`) [3]. 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 [18]. We query the points-to analysis once for every program variable that assigns a value to a tracematch variable (at a shadow). The points-to analysis returns a points-to set for the variable (at this shadow), possibly annotated with context information. Context information distinguishes multiple heap objects which are allocated at the same allocation site, as is the case for iterators. In some rare cases, e.g. when an object is referenced by a static field, context information cannot be constructed. In other cases, e.g. when a program involves complicated forms of dynamic class loading, the context may take a long time to compute. We used the points-to analysis's default settings, traversing at most 75,000 nodes per query, divided amongst 10 iterations (see [18] for an explanation of the points-to analysis). Should the analysis exceed its quota, it simply returns a points-to set without context information, which is potentially less precise than a points-to set with context information.

We use the points-to information to apply a simple flow-insensitive checker [7]. For instance, consider the `HasNext` tracematch, together with a program that only calls `hasNext` on some iterator `i` (in particular, it never calls `next` on `i`). Clearly, `HasNext` can never reach the final state for `i`. The flow-insensitive checker identifies and removes "orphan" shadows that do not contribute to a potential match because the objects at those shadows lack critical shadows—those required to reach a final state.

At this point, if any shadows remain, we are left with a program where the remaining shadows all potentially contribute to triggering the tracematch on at least one 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 rule out shadows on a per-method, per-tracematch basis, first computing flow-sensitive alias information, then detecting and removing all "unnecessary" shadows based on the flow-sensitive information. Unnecessary shadows only affect objects that never reach a final state, or are subsumed by other shadows. Ruling out one shadow might reveal that shadows in other methods are unnecessary. Therefore, in principle, iterating the analysis might help, as the loop in Figure 1 suggests. However, in practice, we have found it sufficient to just re-run the flow-insensitive checker [7].

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. At the same time, we 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 these program points, so that developers can more easily find program points that are likely to actually cause the tracematch body to execute.

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 tm we apply two static analyses:

1. an intraprocedural flow-sensitive abstract interpretation of tm's runtime configuration (3.2.1 to 3.2.4); and
2. an interprocedural flow-insensitive analysis which analyzes each final automaton configuration separately for effects caused by other methods (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.

All other shadows are candidates for removal. However, to ensure completeness, we still need to take the continuation of the control flow after executing m into account: even if m itself did not drive the tracematch automaton into 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. The second analysis models this continuation of control flow.

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 current method's entry statement
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
  $cs' := \bigcup_{\text{shadow } s \text{ at } stmt} \underline{transition}(c, s, tm)$
  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

At runtime, a method's initial configuration is the configuration that gets computed by the previously executing part of the program run. At compile time, however, we do not know this configuration because our analysis is intraprocedural—we do not know which partial matches may reach any method's entry point. However, because our equations update each state independently, it is sufficient to start with a set of configurations where each configuration sets a different non-initial, non-final state to $\mathbf{tt}$; this accounts for all possible configurations that may reach a method's entry. For a four-state automaton, where the first state is initial and the last state is final, we use these 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 initializes 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 previously been computed at this statement, the algorithm creates new jobs associating c' with each successor statement in the control flow graph.

Algorithm 2 Algorithm *transition*

1: let $p := (\mathbf{ff}, \ldots, \mathbf{ff})$ // empty configuration
2: let n be a copy of the current configuration c
3: let $l := \mathit{label}(s)$ be the label of the current shadow s
4: **for** edge (q_s, l, q_t) in automaton of tm **do**
5: $\quad p(q_t) := \underline{p(q_t) \vee (c(q_s) \wedge \mathit{bind}(s))}$ // positive update
6: **end for**
7: **for** non-initial, non-final state q in automaton of tm **do**
8: $\quad$ **if** $\neg\exists$ loop (q, l, q) in automaton of tm **then**
9: $\quad\quad n(q_t) := \underline{n(q_t) \wedge \neg \mathit{bind}(s)}$ // negative update
10: $\quad$ **end if**
11: **end for**
12: **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 have underlined the crucial calculations. The algorithm computes a successor configuration for c in two steps. First, we propagate disjuncts along automaton edges using the positive update rule (line 5). We obtain the new constraint $p(q_t)$ at the target state q_t by disjoining its old value with the constraint of the source state q_s, refined with the variable binding $\mathit{bind}(s)$ of the current shadow. Second, we take care of "missing loops". If the tracematch automaton reads a shadow s of 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; we implement removals via the negative update rule (line 9). 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 self-loops on q.

3.2.2 Abstraction using object representatives

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 $\mathbf{ff}$. At compile time, to prevent spurious matches which could reach the final automaton state (false positives), we would also like to deduce $\mathbf{ff}$ whenever possible. However, at compile time, we have no access to runtime objects, which make it more difficult to find contradictions.

We therefore approximate runtime objects with *object representatives* [8]. Object representatives are static representatives of heap objects, which we compute from a combination of (1) the flow-insensitive context-sensitive 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

$r_1 \approx r_2$	May-alias
$r_1 = r_2$	Must-alias
$r_1 \neq r_2$	Must-not-alias

Figure 6: Aliasing relations between object representatives r_1 and r_2.

to determine that two object representatives r_1 and r_2 cannot possibly represent the same runtime object. We denote this relationship by $r_1 \neq r_2$. Furthermore, we say that two object representatives are equal, and write $r_1 = r_2$, if the representatives are must-aliased; that is, the representatives must represent the same runtime object. Finally, if two object representatives are neither must-aliased nor must-not-aliased, they are may-aliased, and we write $r_1 \approx r_2$. Figure 6 summarizes object representative notation for easy reference. In the following, we will distinguish a runtime object $o(\texttt{x})$ referenced by variable `x` from the object representative $r(\texttt{x})$ that we use to model this object at compile time.

Precise information about the aliasing relationship between object representatives 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 4, but we now consider the compile-time abstraction. Because object representatives closely model runtime objects, states propagate exactly like at runtime, except that we bind object representatives ($i = r(\texttt{i2})$) 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 $r(\texttt{i1}) = r(\texttt{i2})$. Object representatives therefore enable us to compute, at compile time, the same configurations as at runtime (Figure 4).

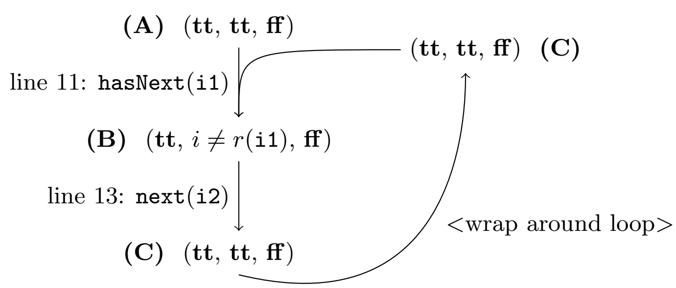

Figure 7: Effect of `print` on `HasNext` automaton.

Figure 7 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 4: the must-alias analysis gives $r(\texttt{i1}) = r(\texttt{i2})$, so at line 13 we compute $c'(q_1) \equiv i \neq r(\texttt{i1}) \vee i = r(\texttt{i2}) \equiv \mathtt{tt}$. 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.

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

$\mathbf{r_1 \neq r_2}$	$x = r_1$	$x \neq r_1$
$x = r_2$	$\mathbf{ff}$	$x = r_2$
$x \neq r_2$	$x = r_1$	$x \neq r_1 \wedge x \neq r_2$

(a) Resulting disjuncts when r_1 and r_2 must-not-alias

$\mathbf{r_1 = r_2}$	$x = r_1$	$x \neq r_1$
$x = r_2$	$x = r_1 \equiv x = r_2$	$\mathbf{ff}$
$x \neq r_2$	$\mathbf{ff}$	$x \neq r_1 \equiv x \neq r_2$

(b) Resulting disjuncts when r_1 and r_2 must-alias

Table 1: Analysis-aware reduction rules.

$$\mathbf{(A)} \quad (\mathbf{tt}, \mathbf{ff}, \mathbf{tt}, \mathbf{ff})$$

line 12: `create(c3,i1)`

$$\mathbf{(B)} \quad (\mathbf{tt},\ c = r(\mathtt{c3}) \wedge i = r(\mathtt{i1}),\ c \neq r(\mathtt{c3}) \vee i \neq r(\mathtt{i1}),\ \mathbf{ff})$$

line 13: `next(i2)`

$$\mathbf{(C)} \quad (\ldots, c \neq r(\mathtt{c3}) \wedge i = r(\mathtt{i2}))$$

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

bindings for potentially different values, we must store both negative bindings (as is 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 presence of multiple bound variables. Consider again the `print` method from Figure 2, but instead with the `FailSafeIter` tracematch from Figure 5. 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 8. The `create` shadow would give the configuration **(B)** after line 12. Because of the `next` edge from q_2 to q_3 (Figure 5), the `next` shadow at line 13 would then update the configuration $\mathbf{ff}$ at the final state q_3 to

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

Must-alias information discards the second disjunct, yielding

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

which means that the `next` shadow might trigger the final state q_3 on some collection (but not $r(\mathtt{c3})$) and iterator $r(\mathtt{i2})$. Now, we as programmers know that $r(\mathtt{i2})$ can only be associated with the collection $r(\mathtt{c3})$: each `Iterator` object binds to exactly one `Collection` object. However, the analysis seen so far has no way to infer that

$$i = r(\mathtt{i2}) \Rightarrow c = r(\mathtt{c3}). \tag{4}$$

Inferring this equation would allow us to reduce $\mathbf{(C)}_{|q_3}$ to $\mathbf{ff}$, proving that `print` never violates the `FailSafeIter` property. A key insight is that equation (4) holds if we can establish that all shadows in the program that bind i to $r(\mathtt{i2})$ must also bind c to $r(\mathtt{c3})$ (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 approach, the "uniqueness check". We apply this check whenever computing conjunctions in the underlined update rules of Algorithm 2. Assume that the binding of the current shadow is $bind(s)$ and $\{x = r(\mathtt{v})\} \subseteq bind(s)$. The check first determines the set of all shadows in the entire program that may possibly bind x. Next (lines 9-13), if there already exists a positive binding $y = r(\mathtt{w})$ in the current disjunct, and for all shadows binding x, we have $x = r(\mathtt{v}) \implies y \neq r(\mathtt{w})$, then this contradicts the binding $x = r(\mathtt{v}) \wedge y = r(\mathtt{w})$, which we are trying to generate. Therefore we return *true*, which will instruct Algorithm 2 to reduce the conjunction to $\mathbf{ff}$. Lines 14-19 perform the inverse check for an existing negative binding. Note that this algorithm is interprocedural. Nevertheless, the algorithm usually computes its result in a few milliseconds, due to its low complexity and the fact that all of its inputs have previously been computed.

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

Algorithm 3 Algorithm *contradictsUniqueBinding*

1: let $x = r(\mathtt{v})$ 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 $y = r(\mathtt{w})$ in current disjunct **do**
10:    **if** $\forall s \in bindShadows\ .\ s$ may bind $(x = r(\mathtt{v}))$ $\implies s$ cannot bind $(y = r(\mathtt{w}))$ **then**
11:       **return** true
12:    **end if**
13: **end for**
14: **for** negative binding $y \neq r(\mathtt{w})$ in current disjunct **do**
15:    **if** $\forall s \in bindShadows\ .\ s$ may bind $(x = r(\mathtt{v}))$ $\implies s$ must bind $(y = r(\mathtt{w}))$ **then**
16:       **return** true
17:    **end if**
18: **end for**
19: **return** false

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 $\mathbf{ff}$, this means that this automaton configuration may have triggered the tracematch body at runtime. We therefore need to keep all shadows alive that may have lead to this configuration being generated. But how can we determine which shadows we must keep alive? 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 as well. This is possible 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 completeness we must also consider interprocedural control flow and data flow.

3.2.5 *Conservative assumptions made for completeness*

The analysis algorithm which we presented above is intraprocedural, i.e. it only considers one method at a time. To compute "complete" results for a method m, we must also account for all program events which may occur:

1. before the execution of m,
2. within methods called by m,
3. after the execution of m.

We have already handled (1): since 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 over-approximate all possible events before m.

We solve (2) as follows. We first determine whether m calls only "harmless" methods—methods which cannot affect m's automaton state. Consider `HasNext`: if (a) m only uses an iterator o(`i1`); (b) m calls m', which only uses iterator o(`i2`); and (c) r(`i1`) $\neq r$(`i2`), then we know that m' cannot jeopardize the completeness of the analysis results for m. To determine harmlessness, we ask the flow-insensitive analysis from [7] to enumerate all shadows that have bindings that are compatible with the binding of any shadow in m. Next, whenever Algorithm 1 processes a configuration c at a statement *stmt*, and *stmt* may (transitively) call a method with a compatible shadow, we "taint" c. Algorithm 1 then further propagates taintedness: successor configurations of tainted configurations are also tainted. Then, after Algorithm 1 finishes, we protect all shadows at any statement that holds a tainted configuration from being removed. While this approximation is conservative, it is complete: we never remove shadows based on possibly incomplete configurations.

Algorithm 4 handles all potential traces following m. Intuitively, it checks that m cannot leave the tracematch automaton in a configuration which may reach a final state in some other method. Recall that the intraprocedural worklist algorithm computes a set of possible configurations after each statement. We form *endConf*, the set of all configurations at each exit statement of m. For each configuration $c \in$*endConf* and each automaton state q, we determine the labels *haveLbls* of all shadows in the rest of the program that are compatible with any shadow in $history(c(q))$. (We do not need to regard shadows in m itself because these have already been handled by our selection of initial configurations.) If there exists an automaton 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 must 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, and usually terminates within a second, some large methods with complicated aliasing relationships may take too long to analyze. We therefore allow programmers to

Algorithm 4 Algorithm *completeContinuation*

$endConf = \{c \in computed(stmt) \mid stmt$ exit node of $m\}$
for $c \in endConf$ **do**
  for state q of the automaton of tm **do**
    $haveLbls = \{\ l\ \mid \exists$ shadow $s\ .\ method(s) \neq m\ .$
      $\exists\ s_h \in history(c(q))\ .$
      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

abort the analysis after it processes a specified number, N, of worklist jobs; the compiler then retains all of that method's shadows. For our experiments we chose N=3000. This N aborts the analysis on only 12 of 1053 methods. 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 contribute to the partial match which is subsequently completed by the point of failure. For `HasNext`, each remaining `next` shadow would be a potential point of failure, while `next` and `hasNext` shadows on aliased iterators are context shadows.

For our concrete example in Figure 2, our analysis removes all shadows for the `HasNext` tracematch, because the intraprocedural analysis never reaches the final state (Figures 4 and 7), and there are no shadows in other methods (the continuation) that could drive partial matches to a final state. For `FailSafeIter`, the analysis first removes the shadow at line 5 in `main` because it is subsumed by the shadow at line 3. Then the analysis removes all shadows in `print`. Our uniqueness check prevents the intraprocedural analysis from hitting the final state (Section 3.2.3). The continuation only contains `update`-shadows, which cannot drive any of the configurations at `print`'s exit statements to a final state. When we then re-iterate the flow-insensitive checker [7], it removes the orphan shadow at line 3 as well.

4. AUTOMATIC CLASSIFICATION

Although our static analysis 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 program points by hand and marked actual matches, giving us training data. Some actual matches only occurred due to delegation, for instance at an (unchecked) call `inner.next()` within `wrapper.next()`. Such calls are uninteresting for error detection because clients use `inner` correctly whenever they use `wrapper` correctly. We therefore marked such matches as false positives. 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 [21] 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.

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 [5] and with SciMark2 [17]. SciMark2 uses stopwatches to collect execution times and we applied four additional tracematches to validate the correct use of these stopwatches. Altogether this leads to a total of 103 tested tracematch/benchmark combinations. Table 2 briefly describes our tracematches.

For many of the 103 benchmark/tracematch combinations, the flow-insensitive analysis from previous work [7] detected that the tracematch never triggers. However, 38 combinations have remaining potential points of failure. We applied our flow-sensitive analysis and classification to these cases.

Potential and actual points of failure.

Columns two and three of Table 3 present the number of potential points of failure remaining after the flow-insensitive analysis (fi) [7] 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. On 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, as identified through manual inspection. For benchmarks with many remaining potential points of failure, 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 at 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 3.2.5). `DELEGATE` identifies shadows at delegating calls (Section 4). `NO_CONTEXT` means that one of the shadows holds points-to sets without context information—that is, the demand-driven analysis failed to find context information in its time budget. For our data set, Weka computes the decision tree shown in Figure 9.

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 to ensure that the classifiers do not over-fit the data. Cross-validation is a statistical technique that allows the estimation of error without distinct training and testing sets. Cross-validation estimates the error by using a subset of a data set as training data and then applying the trained model to the remainder of the data set. Ten-fold stratified cross-validation is a cross-validation technique that has been proven statistically stable [13].

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, [7]) and flow-sensitive analysis (fs, this paper); actual points of failure (APF); PPF remaining after filtering (filtered), false positives/negatives in brackets

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 points of failure. Note that we manage to filter out *all* 449 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; our test data only contains one interprocedural violation. The reason is that, in our benchmarks, most actual violations occur on the HasNext and HasNextElem tracematches, which bind iterators and enumerations, which hardly ever leak out of their defining method. We believe that it might be useful to train the machine learning algorithm for each tracematch separately, but leave this question to future work, as it would require yet more benchmarks.

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

```
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)
```

Figure 9: Decision tree computed by Weka

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 work flow: the developer runs our static analysis on a program; if there remain a small number of potential points of failure, then the developer can verify each of these points manually and get a complete analysis result. However, if there remain 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.

Using our filtered results we identified the following defects and pieces of suspicious code in our benchmarks by manually inspecting the program code. In pmd-HasNext a method passes an iterator i to another method. The callee method then extracts i's first element without further checks. While this is not an actual bug, the undocumented precondition (that i has a next element) on the callee might cause problems for long-term software maintenance. Interestingly, PMD's developers fixed the method in a later version of PMD by using Java5's for-each loops, which avoid the explicit use of iterators. The actual point of failure in jython-Reader indicates an actual defect. The code may close `Reader` objects and then read from them. The developers "cured" this defect by returning `null` from the method that reads from the Reader, in case of an `IOException`. bloat extracts two elements from a collection without any checks, leading to an actual match in the bloat-HasNext benchmark.

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. Algorithm 1 processed 1.3 jobs per statement and loop on average, with a variance of 0.63.

The version of the AspectBench Compiler that we used is available as revision 4790 in the compiler's Subversion repository and integrated into `abc` release 1.3.0. We uploaded all benchmarks, tracematches and raw data to:

`http://www.aspectbench.org/benchmarks`

6. RELATED WORK

We compare our work to software for anomaly detection, to typestate-based approaches and PQL, to another recent static analysis for tracematches, as well as a shape analysis and the `restrict` pointer annotation.

Java Anomaly Detector (JADET).

Wasylkowski, Zeller and Lindig propose the Java Anomaly Detector (JADET) for detecting object usage anomalies in programs [20]. JADET first generates common usage patterns from the program, considering control flow, one object at a time. The tool lists uncommon usages contradicting the pattern, ranked by confidence. The authors show that seven uncommon usages in their benchmarks are actual defects. While useful, JADET's analysis is not complete.

JADET's mining phase makes the tool fully automated. Tracematches require specifications, but can validate more complicated usage patterns, namely patterns involving multiple objects like `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". Extending JADET to tracematch-like patterns would be an interesting project.

Typestate.

Typestate properties [19] have been enjoying renewed interest. Typestate describes the state of heap objects but, as in JADET, one at a time. Because of this restriction, typestate is less general than tracematches. All of the following approaches are complete, i.e. never miss true positives.

DeLine and Fähndrich [9] 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 [11]. 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 [4] 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 [10]. 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 deterministic transition function does not drive the typestate automaton into a final state. They then summarize the effect of a region and update the typestate with the region's effects all at once. Our static analysis does not attempt to determine regions; we instead decide if each method is safe, independently, and combine this information with an approximation of the effect of the rest of the program. Our static analysis enables the compiler to estimate the points at which a program may violate the safety property described in a tracematch.

Extended typestate analysis for tracematches.

Naeem and Lhoták recently proposed an entirely different approach to evaluating tracematches ahead-of-time [16]. Our design decision was to re-use our fast flow-insensitive analysis from earlier work [7] and add must-alias information and local flow-sensitivity for relatively little compile-time cost. Naeem and Lhoták instead designed a new context-sensitive flow-sensitive whole-program analysis from scratch. Their approach is potentially more precise because they can track must-alias information across procedure boundaries, and because their call-graph is flow-sensitive. However, since they do not use global points-to analysis, they also flag some false positives that our approach avoids. It would be an interesting piece of future work to see how one can combine the best aspects of both approaches to gain maximal precision at minimal cost.

Program Query Language.

The Program Query Language [14] 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 [7]); no flow-insensitive analysis can remove the shadows that interest us here.

Shape analysis and pointer analysis.

Hackett and Rugina propose a region-based shape analysis that uses tracked locations [12]. A tracked location is an abstract configuration that characterizes the state of a single heap location. The authors' approach is similar to ours in that it consists of two layers: (1) a region abstraction that encapsulates whole-program points-to information, and (2) a shape abstraction that encapsulates local knowledge about a single location's shape in an individual configuration. The analysis propagates configurations through the whole program, gaining precision through additional context information, and querying the global heap abstraction during this process. This interprocedural approach is possible in their analysis because it determines the shape of each single heap location individually. The authors' abstract configurations therefore each encode knowledge about a location, which does not depend on the configurations of other locations. Tracematches with multiple variables, however, track states for multiple objects simultaneously; it is therefore not feasible to track the state of each object independently, which led to our use of constraints. Another difference is that Hackett and Rugina's region abstraction contains may-alias information only. The authors store must-alias information in the shape abstraction. We instead encapsulate both may-alias and must-alias information inside object representatives. This decouples pointer analysis from the abstraction, allowing each to evolve separately.

Aiken et al. propose the program annotation `restrict` [1]. Programmers can augment a statement s with an annotation `restrict p = v {`s`}` to denote that s only accesses the object referenced by pointer `p` directly through the name `p`, not through aliases. Such annotations allow program analyses to perform strong updates on the restricted pointer `p` even without global knowledge. As the authors show, the necessary program annotations can often be inferred. In our work we used object representatives to model aliasing relationships. We can perform strong updates because object representatives provide must-alias queries and because we use a side-effects analysis (Section 3.2.5) to decide whether pointers are restricted in the above sense.

7. CONCLUSIONS & FUTURE WORK

Analyzing tracematches ahead-of-time is clearly 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. 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 ranking and filtering approach helped greatly in distinguishing actual points of failure from remaining false positives. As an alternative to inspecting all remaining 461 remaining potential points of failure, our machine learning algorithm identified almost exactly the program points of interest. It was equally straightforward to instruct our verification tool to instrument the program under test with an optimized runtime monitor to guard unlikely points of failure. We believe that this combined approach will greatly support programmers in finding erroneous uses of application interfaces in their programs. We further believe that machine learning approaches like ours can be generally useful for weeding out false positives caused by overly conservative approximations.

This work poses some interesting research questions that we wish to address in the near future. We are currently conducting a case study that aims to identify which kinds of specification patterns naturally exist in application interfaces of some well-known open-source projects. We plan to express these patterns using tracematches, which should give us insight into the usefulness of tracematches as a property specification language, and should furthermore significantly enlarge our database of tracematches.

Furthermore, we are extracting the essence of the analysis algorithms presented here, to determine how well these algorithms can be applied to other specification languages. Our flow-insensitive checkers from earlier work [7] apply directly to other languages [6]. To generalize the results of this paper, we would need to adapt the abstract automaton configurations to the concrete runtime model used to evaluate the specification language in question. Other specification languages might need alternatives to constraints—either more or less sophisticated abstractions.

Finally, we will investigate how to present our analysis results to the user. We anticipate that connecting our analysis to an integrated development environment will enable developers to quickly visualize potential points of failure.

Acknowledgements.

We owe thanks to Manu Sridharan for help with his demand-driven points-to analysis. Stephen Fink provided valuable information about instance keys and SSA form. Brian Demsky and Nomair Naeem provided useful comments on an earlier version of this paper. We also thank the entire `abc` group for their continuing support, in particular Pavel Avgustinov and Julian Tibble for providing and maintaining their tracematch implementation. We would like to thank the anonymous referees for their detailed and very helpful comments.

8. REFERENCES

[1] A. Aiken, J. S. Foster, J. Kodumal, and T. Terauchi. Checking and inferring local non-aliasing. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 129–140, New York, NY, USA, 2003. ACM Press.

[2] 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.

[3] 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.

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

[5] 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. ACM Press, 2006.

[6] E. Bodden, F. Chen, and G. Roşu. Dependent advice: A general approach to optimizing history-based aspects (Extended version). Technical Report abc-2008-2, `http://www.aspectbench.org/`, March 2008.

[7] E. Bodden, L. J. Hendren, and O. Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *ECOOP*, volume 4609 of *LNCS*, pages 525–549. Springer, 2007.

[8] E. Bodden, P. Lam, and L. Hendren. Object representatives: a uniform abstraction for pointer information. In *1st International Academic Research Conference of the British Computer Society (BCS 2008)*, Sept. 2008. To appear.

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

[10] 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.

[11] S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. In *ISSTA*, pages 133–144. ACM Press, 2006.

[12] B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In J. Palsberg and M. Abadi, editors, *POPL*, pages 310–323. ACM Press, 2005.

[13] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *IJCAI, San Mateo, CA*, pages 1137–1143, 1995.

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

[15] 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.

[16] N. A. Naeem and O. Lhoták. Extending typestate analysis to multiple interacting objects. In *OOPSLA*. ACM Press, Oct. 2008. To appear.

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

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

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

[20] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *ESEC-FSE*, pages 35–44, New York, NY, USA, 2007. ACM Press.

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