The abc Group

Static Analysis Techniques for Evaluating Runtime Monitoring Properties Ahead-of-Time

abc Technical Report No. abc-2007-6

Eric Bodden, Patrick Lam, Laurie Hendren

Sable Research Group
School of Computer Science
McGill University

November 23, 2007

Contents

1 Introduction **3**

2 Runtime evaluation of tracematches **5**

2.1 HasNext example tracematch 5

2.2 FailSafeIter example tracematch 8

3 Static analysis setting and abstraction **9**

3.1 Weaving process 9

3.2 Analysis abstraction 9

4 Using static analyses to increase precision **11**

4.1 Refinement rules based on alias information 12

4.2 Distributive law 13

4.3 Unique-shadow analysis 13

5 Static checkers and transformations **15**

5.1 Unnecessary-shadows analysis 15

5.2 Run-once loop optimization 16

6 Experiments **17**

6.1 Unnecessary shadows 17

6.2 Suspicious code, defects and sources of imprecision 19

6.3 Run-once shadows 19

6.4 Remaining runtime overheads 20

7 Related Work **20**

7.1 Previous whole-program analyses for tracematches 20

7.2 Typestate 22

7.3 Program Query Language 22

7.4 Eliminating runtime errors 22

8 Conclusions **23**

A Suspicious pieces of code found in DaCapo benchmark programs **25**

List of Figures

1 Example program, annotated with instrumentation points . . . 5
2 HasNext tracematch . . . 6
3 Automaton for the HasNext tracematch from Figure 2. . . . 7
4 Effect of `print` on `HasNext` automaton. . . . 7
5 FailSafeIter tracematch . . . 8
6 Automaton for FailSafeIter from Figure 5. . . . 8
7 Weaving process . . . 10
8 Aliasing relations between instance keys . . . 12
9 Effect of `print` on `HasNext` automaton. . . . 13
10 Derivations leading to our strong update rules . . . 14
11 Effect of `print` on `FailSafeIter` automaton. . . . 14
12 Code in Eclipse not checking `hasNext()` but rather tests `size()==0` on the collection . . . 25
13 Code in Eclipse passing a reader to other methods . . . 25
14 Code in PMD not checking `hasNext()` but rather tests `isEmpty()` on the collection . . . 26
15 Delegating `Iterator` in Jython . . . 26
16 Suspicious code in PMD where a method extracts an iterator's first element without a check 27
17 Code in Jython reading from a potentially closed reader. . . . 28

List of Tables

I Update rules when instance keys may-alias . . . 10
II More precise analysis-aware reduction rules. . . . 12
III The generic and domain specific tracematch patterns which required flow-sensitive analysis . 17
IV Number of shadows affected by the static analyses . . . 18
V Impact of the static analyses on the run-time performance of our benchmarks . . . 21

Abstract

Runtime monitoring enables developers to specify code that executes whenever certain sequences of events occur during program execution. In particular, runtime monitors can check for illegal API uses, such as attempts to read from already-closed files. This paper presents techniques for evaluating runtime monitoring properties ahead-of-time. Statically evaluating runtime monitors is advantageous for two reasons: 1) it enables the optimization and removal of unnecessary monitoring code; furthermore, 2) it can increase developers' confidence that their programs conform to their stated correctness properties. In the best case, static analysis can successfully guarantee that a program never violates those properties at all.

Our work focuses on tracematches, a monitoring notation based on regular expressions with free variables over program events that bind these variables to heap objects. Statically deciding properties of tracematches is difficult: tracematches bind heap objects throughout the course of their evaluation. One might expect that an interprocedural flow-sensitive analysis would be required. However, our approach successfully analyzes tracematches by combining inexpensive whole-program summary information with a suite of carefully-designed intraprocedural flow-sensitive analyses. Our analyses use a novel abstraction which captures both positive information, indicating that an object could be associated with a particular monitor state, and negative information, indicating that the object is known not to be in a state. This abstraction enables us to eliminate unnecessary monitoring instrumentation.

We implemented our analyses and applied them to 43 program/tracematch combinations. In 18 cases, our analysis removes all instrumentation from the programs, statically verifying the tracematch property. In 12 more cases, our analysis leaves no more than 10 instrumentation points, and we easily verified (by hand) that these programs satisfied (or falsified) the stated property. In 36 of 43 cases, our techniques reduce runtime overhead to below 10%.

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 and often gives insight into the system's behaviour in general. 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.

Our research aims to evaluate runtime monitoring properties at compile time. When successful, our static analysis approach has a number of advantages over runtime monitoring. For instance, it can enable a compiler to omit unnecessary monitoring code or to specialize monitors that are unnecessarily general. However, our primary focus is on software verification, and we have found our proposed techniques especially valuable for verification. In particular, static analysis can automatically certify that programs will never violate stated correctness properties on any execution. When such a guarantee is not possible, static analysis can at least identify all program points which may contribute to a property violation, enabling the developer to verify the property by hand.

Tracematches [1] are a Java language extension for runtime monitoring. Using tracematches, programmers can specify traces 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 a suffix of the symbols in the current execution trace contains 1) the right symbols with 2) a consistent variable binding (*i.e.* symbols' bound objects match up) in 3) an order which the regular expression matches. Tracematches can, for instance, monitor for files that are read after being closed using the regular expression `close read`.

At the implementation level, the compiler and runtime system implement tracematches using runtime monitors based on finite-state machines. Compiler-generated instrumentation code updates the monitor's internal state each time an event in the execution matches a declared symbol from the tracematch. When the monitor finds a match in the program's execution trace, it triggers the code associated with the tracematch.

A key challenge for any static analysis of tracematches is the fact that tracematch events bind variables to heap objects at runtime. For instance, a tracematch might bind a File object o and state that o is never read after being closed. In previous work, we have proposed filtering tracematches [6] according to a simple

condition: simply discard tracematches which cannot possibly match because they lack a complete set of symbols on potentially-aliased heap objects. Their results showed that their approach is effective in a limited number of cases.

The goal of our research is to develop an analysis that is powerful enough to use in a static verification tool for tracematches. 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 interprocedural and flow-sensitive. We have found, however, that combining inexpensive whole-program summary information with a suite of carefully-designed intraprocedural flow-sensitive analyses permits us to reason about tracematch states successfully. Our abstraction associates two kinds of information with each tracematch state: positive information (an object is in a particular state) and negative information (an object is known not to be in a state).

Our abstraction stores this information using logical constraints, mirroring the tracematch runtime [1]. However, at compile time, our abstraction can only track static approximations to heap objects, not the objects themselves. We mitigate this problem by disambiguating references to the heap statically. Precise points-to and alias analyses help make our abstraction more precise. Specifically, we present reduction rules over our logical constraints. These rules are based on must-alias information, must-not-alias information, and a custom-built uniqueness analysis.

Our static analysis enables compilers to optimize runtime monitor implementations. The analysis gives the compiler enough information to 1) identify monitoring points that never cause a tracematch to trigger; and 2) optimize monitoring points in loops that can only trigger on their first execution. If a runtime monitor guards against a pathological sequence of events and the compiler statically verifies that the runtime monitor never triggers (by removing all instrumentation points), then the compiler has proven that the program never executes the pathological sequence of events.

We have implemented our static analysis and two transformations based on this analysis in the context of the AspectBench Compiler [2]. To test the efficacy of our analysis, we applied it to 43 benchmark/tracematch combinations, each of which contained potential instrumentation points that previous analyses were unable to prove unnecessary. Our benchmarks were mostly drawn from the DaCapo benchmark suite [4], plus two additional benchmarks and their domain-specific safety properties. We found that in 18 of the 43 cases, the compiler could show that the program never triggered the tracematch (it removed all of the instrumentation points). Furthermore, in an additional 12 cases, the compiler was able to greatly reduce the number of instrumentation points, leaving no more than 10 points in these cases. With so few points remaining, we easily verified all outstanding points manually. As we expected, some of the points could not be verified because our analysis was not precise enough. However, some instrumentation points highlighted suspicious code which could violate the properties stated by the tracematch patterns under certain conditions.

The contributions of this paper include:

- an analysis abstraction for tracking runtime monitor states, which combines interprocedural may-alias information with intraprocedural flow-sensitive must-alias and must-not alias information;
- a unified, clean set of rules for manipulating the analysis abstraction based on static analysis information;
- transformations and optimizations based on the static analysis information; and
- an implementation of the proposed static analyses and their evaluation on a suite of realistically-sized benchmark programs.

The remainder of the paper is structured as follows. Section 2 presents a pair of examples and explains how our static analysis represents the state of our runtime monitors in these cases. Section 3 describes the static analysis abstraction and Section 4 explains rules for manipulating the abstraction. Section 5 presents program transformations based on our analysis. Section 6 summarizes our experimental setup and gives results. Finally, Section 7 discusses related work and Section 8 concludes.

2 Runtime evaluation of tracematches

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, we include two illustrative examples to foreshadow our static analyses, later described in Sections 3 through 5.

Figure 1 presents our running example, a program we would like to partially verify using tracematches. The program populates a collection, which is then passed to methods for copying and printing. The code contains a set of auxiliary copy statements, e.g. at lines 4 and 14. We explicitly added these copy statements to emphasize the problem of aliasing. Any static verification of Java programs has to deal with aliasing, mostly occurring in intermediate code and whenever objects are stored in fields or passed to methods.

```
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      Iterator i1 = c2.iterator ();         //create(c2,i1)
7      Collection copy = iteratorToList(i1 );
8      print(copy);
9  }
10
11 void print(Collection c3) {
12     Iterator i2 = c3.iterator ();         //create(c3,i2)
13     while(i2.hasNext()) {                 //hasNext(i2)
14         Iterator i3 = i2;
15         System.out.println(i3 .next ());  //next(i3)
16     }
17 }
18
19 List iteratorToList (Iterator i4) {
20     List c5 = new LinkedList();
21     while(i4.hasNext()) {                 //hasNext(i4)
22         Iterator i5 = i4;
23         Object o = i5.next();             //next(i5)
24         c5.add(o);                        //update(c5)
25     }
26     return c5;
27 }
```

Figure 1: Example program, annotated with instrumentation points

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

2.1 HasNext example tracematch

Figure 2 presents the `HasNext` verification tracematch, which matches (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* of code. 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` object named `i`. Lines 2–5 define symbols `hasNext` and `next`, which capture method calls to the `hasNext()` and `next()` methods of `i`. Symbols establish an alphabet for the tracematch's regular

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

Figure 2: HasNext tracematch: do not call `next()` twice without an intervening call to `hasNext()`.

expression. Line 7 declares the regular expression "`next next`" and the body of code to be executed every time the regular expression matches. Any occurrence of the `hasNext` symbol on a iterator `i` resets partial matches for `i`.

The tracematch runtime attempts to match each suffix of the abstract (symbol-based) execution trace with the regular expression. 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 once at the second `next` event and once at the third `next` event.

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.

One distinguishing 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 could 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. The AspectBench compiler [2] (`abc`) implements tracematches by compiling Java source or bytecode, plus any desired tracematches, into Java programs augmented with runtime monitors. `abc` first creates an automaton from the tracematch's regular expression. Figure 3 presents the automaton for `HasNext`. The compiler then identifies a set of instrumentation points, or *shadows* [14], corresponding to the points in the code where symbols potentially execute. At runtime, shadows update tracematch state in response to program events. Figure 1 includes shadows as comments. 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 (possibly partial 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 an object is in a given state with a given binding.

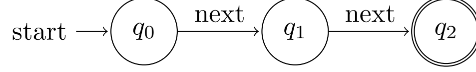

Figure 3: Automaton for the HasNext tracematch from Figure 2.

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 is always $\mathbf{tt}$ because tracematches may start a match anytime. For the `HasNext` automaton, the initial configuration is $(\mathbf{tt}, \mathbf{ff}, \mathbf{ff})$.

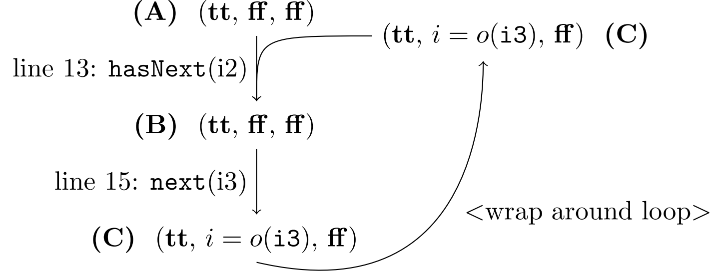

Figure 4: Effect of `print` on `HasNext` automaton.

As the program executes, the runtime updates constraints as events occur. In Figure 4 we evaluate the `HasNext` automaton at the start of the `print` method from Figure 1 with initial configuration $(\mathbf{tt}, \mathbf{ff}, \mathbf{ff})$ **(A)**. The `hasNext` shadow at line 13 has no effect on this configuration, since q_0 has no `hasNext` transition and all other states contain $\mathbf{ff}$ **(B)**. At line 15, the `next` shadow binds tracematch variable `i` to the object stored in `i3`, denoted $o(\mathtt{i3})$. The `next` transition from q_0 to q_1 in the automaton causes the following update:

$$\begin{aligned} c'(q_1) &\equiv c(q_1) \vee (\ c(q_0) \wedge i = o(\mathtt{i3})\ ) \\ &\equiv \mathbf{ff} \vee (\ \mathbf{tt} \wedge i = o(\mathtt{i3})\ ) \\ &\equiv i = o(\mathtt{i3}). \end{aligned}$$

Here $c(q_i)$ denotes the incoming constraint at q_i and $c'(q_i)$ the constraint after executing line 15. The update results in the configuration $(\mathbf{tt}, i = o(\mathtt{i3}), \mathbf{ff})$ **(C)**. Another call to `next()` on the same iterator $o(\mathtt{i3})$ would propagate $i = o(\mathtt{i3})$ to the final state q_2, and the runtime would execute the tracematch body.

However, the example program contains a loop, and control flow wraps around to line 13 again, with event `hasNext(i=`o`(i2))`. Because q_1 has no `hasNext` self-loop, object $o(\mathtt{i2})$ cannot possibly be in q_1 after this event, and we conjoin q_1's constraint with a negative binding $i \neq o(\mathtt{i2})$. Since the incoming configuration is $(\mathbf{tt}, i = o(\mathtt{i3}), \mathbf{ff})$, and because $o(\mathtt{i2}) = o(\mathtt{i3})$, we get:

$$\begin{aligned} c'(q_1) &\equiv c(q_1) \wedge i \neq o(\mathtt{i2}) \\ &\equiv i = o(\mathtt{i3}) \wedge i \neq o(\mathtt{i2}) \\ &\equiv \mathbf{ff}, \end{aligned}$$

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

Static Optimization Potential. We designed our static optimization to exploit cases where transitions have no net effect, as seen above. Our static analysis identifies shadows that 1) bind objects which never reach a final tracematch state, or 2) have no runtime effect on any input configuration. Note that we can profitably reason about the `print` method in isolation and find that it never triggers a final state, regardless of the incoming tracematch configuration. Sections 3 and 4 present our static analysis for determining such

```
1  pointcut collection_update(Collection c):
2      (   call(* java.util.Collection+.add*(..))  ||   ...   ||
3          call(* java.util.Collection+.remove*(..)) )  && target(c);
4
5  tracematch(Collection c, Iterator i) {
6      sym create after returning(i):
7              call(* java.util.Collection+.iterator())  && target(c);
8      sym next before:
9              call(* java.util.Iterator+.next()) && target(i);
10     sym update after: collection_update(c);
11
12     create next* update+ next { ... } }
```

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

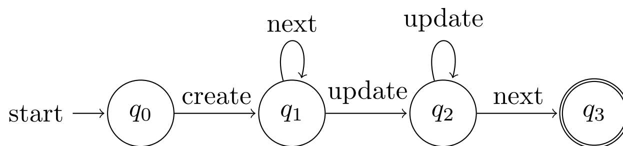

Figure 6: Automaton for FailSafeIter from Figure 5.

information. Aliasing is critical for our analysis: in the above example, we must know that o(`i2`) = o(`i3`) to conclude that the updates inside the loop have no effect.

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 each complete match must bind each tracematch variable somewhere. Typestate analysis [3, 10, 11, 16] instead tracks each object in isolation or has to encode one object's state in the state of another [3].

Our second example demonstrates the ability of tracematches to bind multiple variables. Figure 5 presents the `FailSafeIter` tracematch, which reports cases where the program modifies a `Collection c` while an `Iterator i` is active on `c`. Figure 6 shows the corresponding automaton.

Complications due to aliasing. Aliasing is a fundamental problem in reasoning about objects at compile time, even in the typestate approach, which only tracks a single variable. To reason about an object o(`x`), we must know about all state changes on o(`x`), even changes through aliases, or make conservative assumptions.

The situation is even more complicated in the presence of multiple variables. Consider, for instance, the method `iteratorToList` from Figure 1. This method takes an iterator `i4`, puts a copy of `i4` into `i5` (line 22) and advances (line 23). But we must ensure that the `update` shadow at line 24 does not invalidate the iterator `i4` before it is advanced again. How do we know that o(`i5`) is not an iterator of o(`c5`)? In this case, we used our flow-insensitive consistent-shadows analysis from previous work [6] to deduce that the shadows on o(`i5`) and o(`c5`) are independent, using interprocedural points-to information.

Need for run-once loop optimization. The `iteratorToList` example demonstrates a fundamental limitation of intraprocedural analyses: no intraprocedural analysis can determine the state of iterator o(`i5`) upon method entry. We cannot tell whether its associated collection has already been updated, and therefore do not attempt to remove the `next` shadow. Instead, we specialize it: we assume safely that the `next` shadow could trigger the tracematch's final state. We then observe that the `next` shadow can have no further effect

in this method. It can only trigger the tracematch in the first loop iteration. Section 5 presents a static analysis and transformation that detects shadows that only need to run once. The transformation optimizes such shadows by guarding them with a special Boolean flag, which ensures that the shadow only generates an abstract event in the first loop execution.

While our loop optimization cannot certify that the `next` shadow does not trigger the final state, it does speed up runtime evaluation of the tracematch. Furthermore, information from the loop optimization is useful for test case generation: test cases only need to exercise the functionality of the loop once to get complete coverage with respect to the tracematch property.

3 Static analysis setting and abstraction

We now move to the compile-time setting of static analysis and optimization. We first situate our static analyses and transformations in the context of the compilation process for tracematches and then explain our static abstraction. Our analysis extends the flow-insensitive whole-program approach from previous work [6] with precise flow-sensitive intraprocedural analyses using flow-sensitive must-alias and must-not-alias information.

3.1 Weaving process

We have implemented our analysis in the context of the AspectBench Compiler. Figure 7 presents selected compiler stages and illustrates where our analyses and transformations fit in. First, the compiler reads the program under test and the tracematch definitions, and weaves the tracematches into the program. Next, we apply two stages from [6], the quick check and the flow-insensitive consistent-shadows analysis. The quick check removes tracematches that cannot trigger because the program as a whole does not contain all necessary events, while the consistent-shadows analysis uses flow-insensitive pointer analysis to identify shadows that cannot contribute to a potential match because the objects at those shadows are lacking other shadows to reach a final state.

After the stages from [6] finish, we are left with a program where the remaining instrumentation points (shadows) all potentially contribute to triggering the tracematch. We must use flow-sensitive information to further analyze these points.

We proceed on a per-method basis. We first remove all “unnecessary” shadows, as inferred by our analysis. Such shadows either do not affect the runtime monitor state, or they are subsumed by other shadows. If any shadows remain, we next find shadows within reducible loops which only need to be executed in the first loop iteration. Because removing shadows in one method can make the analysis of other methods more precise, we iterate analysis phases until we reach a fixed point (interestingly, this does not help in practice; see. Section 6). Finally, we re-weave the program using the updated shadow information and emit optimized code.

3.2 Analysis abstraction

Both unnecessary shadow elimination and our loop optimization rely on a common analysis abstraction. Our abstraction statically models dynamic tracematch automaton configurations. Recall that, at runtime, each automaton state carries a constraint (recording which heap objects are in which states), and that the runtime monitoring code transforms these constraints at events. When heap objects reach the final state, the runtime calls the tracematch body with the objects fulfilling the constraint at that state.

The main challenge in moving from run-time configurations to compile-time configurations is to handle heap objects precisely. We thus introduce the notion of *instance keys* [7,11], to serve as static representatives for heap objects. We describe instance keys in more detail in Section 4.1. Aside from the substitution of instance keys for heap objects, our static abstraction directly follows the run-time configuration (operational semantics given in [1]).

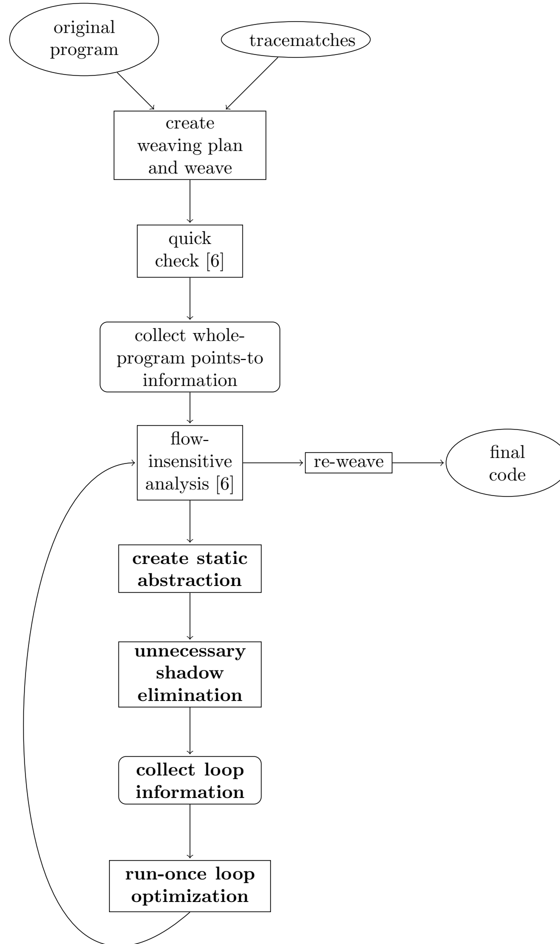

Figure 7: Weaving process; new phases in bold

The abstraction consists of the tracematch automaton and a constraint for each automaton state. At run-time, the constraints store information in the form $x = o(\texttt{v})$ or $x \neq o(\texttt{v})$, where x is a free tracematch variable and o a heap object. At compile-time, we instead store constraints as $x = i(\texttt{v})$ or $x \neq i(\texttt{v})$. We adopt the convention that $o(\texttt{v})$ refers a runtime object stored in local variable `v`, while $i(\texttt{v})$ refers to an instance key associated with `v`.

The tracematch runtime system updates automaton states by conjoining constraints with $x = o(\texttt{v})$ or $x \neq o(\texttt{v})$ whenever an event may move $o(\texttt{v})$ into, or must move $o(\texttt{v})$ out of, a given automaton state. Our compile-time analysis symbolically executes the program's events and propagates instance keys $i(\texttt{v})$ in response to the events. Although our full static analysis exploits alias information to increase precision, in Table I we first present update rules for our abstraction that conservatively assume that no such information is available. (Section 4 contains the full rules.) The table refers to two instance keys i_1 and i_2, which, conservatively, may alias each other (that is, they may refer to the same heap object), denoted by $i_1 \approx i_2$. The rules in Table I simply state that, in the absence of any further information, it is always safe to conjoin a new constraint (*e.g.* $x = i_2$) with an existing constraint (*e.g.* $x = i_1$).

$i_1 \approx i_2$	$x = i_1$	$x \neq i_1$
$x = i_2$	$x = i_1 \wedge x = i_2$	$x \neq i_1 \wedge x = i_2$
$x \neq i_2$	$x = i_1 \wedge x \neq i_2$	$x \neq i_1 \wedge x \neq i_2$

Table I: Update rules when instance keys i_1 and i_2 may-alias.

The soundness of these update rules stems from the way we constructed them—directly modelling the runtime machinery—and from properties of instance keys [7]. In particular, if a tracematch automaton would hold a constraint $x = o(\texttt{v})$ at state s at runtime, then our analysis will hold the constraint $x = i(\texttt{v})$ at s at compile time, and $i(\texttt{v})$ is an instance key modelling the object $o(\texttt{v})$. This reasoning also applies to bindings at final tracematch states, so our static analysis will not miss potential tracematch hits.

Static analysis setup. The analyses in Section 5 use the above update rules in the context of a fixed point iteration over a method's or loop's statements. We next describe the initial configuration and merge operators for our static analyses.

Our analyses track sets of abstract configurations per program point. First, due to the suffix property for tracematches—any tracematch can start matching at any time—start states for all configurations always contain the constraint true (**tt**). Because our update rules act independently on each state, we can start with a set of configurations where each configuration sets a different non-initial, non-final state to **tt**. This conservatively models the possibility that any object can be in any non-final state upon method entry.

For an automaton with four states, where the first state is initial and the last state is final, we would obtain the following set of initial configurations:

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

At control-flow merges, we combine incoming configurations with set union ($\cup$).

Possible side-effects from calling other methods. Our analysis handles method calls as follows. At a method call we inspect the methods transitively reachable from the call. (Our compiler makes a call graph available to clients.) We then search for transitively reachable shadows that bind objects potentially interfering with shadows in the current method, using the results of our flow-insensitive consistent-shadows analysis from previous work [6]. If no such shadows exist, then the method call is harmless and does not affect the results of our current analysis. Otherwise, we *taint* all outgoing configurations at the method call. The optimizations in Section 5 will not remove shadows based on tainted configurations.

Room for improvement. As we have presented it so far, our static abstraction is not yet precise. Consider again the example from Figure 1 and the `HasNext` automaton depicted in Figure 3. In our example evaluation in Figure 4 we showed that no shadow in this method can drive an object into the final state. However, as we pointed out, this only holds because at runtime $o(\texttt{i2}) = o(\texttt{i3})$. If at compile time we just know that $i(\texttt{i2}) \approx i(\texttt{i3})$, we can only perform weak updates on those bindings, by the rules of Table I. With such weak updates, $i(\texttt{i2})$ would reach the final state during the fixed point iteration. The next section presents our suite of rules for leveraging analysis results to enable more precise estimates of abstract automaton states.

4 Using static analyses to increase precision

In this section we present nine reduction rules for weeding out false positives from our static abstraction. Our rules identify parts of constraints that are either redundant or contradictory. Our abstraction stores constraints in Disjunctive Normal Form, so the atomic unit of a constraint is a *disjunct*. Whenever the analysis creates a new constraint, we run this constraint through our reduction rules. These rules drop redundant disjuncts and replace disjuncts that lead to a contradictions with **ff**.

Our reduction rules fall into the following four categories:

1. three rules based on must-not-alias information, supported by:
    - a whole-program, context-sensitive, flow-insensitive points-to analysis [15]; and
    - an intraprocedural flow-sensitive must-not-alias analysis [7];

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

Figure 8: Aliasing relations between instance keys i_1 and i_2.

2. four rules based on must-alias information, computed by an intraprocedural, flow-sensitive must-alias analysis [7];
3. one generic rule, based on the distributive law for Boolean algebra; and
4. one rule based on a special-purpose unique-shadows analysis, which can enable us to strengthen aliasing assumptions.

We continue by describing each reduction rule in turn.

4.1 Refinement rules based on alias information

Because tracematches bind heap objects, pointer analysis is important for understanding the compile-time behaviour of tracematches. We now describe how intraprocedural flow-sensitive must-alias and must-not alias analyses, in combination with an interprocedural may-alias analysis, sharpen the update rules for our static analysis.

Our static analysis uses *instance keys* to statically represent heap objects. Instance keys support must-alias (via equality) and must-not-alias queries. The must-alias query ($=$) only returns useful information on queries about two instance keys from the same method, as it depends on the intraprocedural must-alias analysis. The must-not-alias ($\neq$) query combines results from the intraprocedural flow-sensitive must-not-alias analysis and the interprocedural flow-insensitive points-to analysis. Note that the two must-not-alias analyses have different strengths, so we combine them to achieve better results overall. The default relation is $\approx$. Figure 8 summarizes the different possible relations between instance keys.

To illustrate our use of instance keys, consider again the `HasNext` tracematch, with configuration $(\mathbf{tt}, \mathbf{ff}, \mathbf{ff})$ at the start of the `print` method from Figure 1. This is exactly the same situation as in Figure 4, but this time we consider the compile-time abstraction. Because instance keys closely model runtime objects, states propagate exactly like at runtime, only with the binding $i = i(\texttt{i3})$ instead of $i = o(\texttt{i3})$. The point of interest here is the edge from **(C)** to **(B)**. At runtime we regain the configuration $(\mathbf{tt}, \mathbf{ff}, \mathbf{ff})$ because $o(\texttt{i2})$ and $o(\texttt{i3})$ are in fact the same objects. At compile time, we make use of our must-alias analysis to decide that $i(\texttt{i2}) = i(\texttt{i3})$. As a result, at compile time we are able to reconstruct the exact same evaluation as in Figure 4. Figure 9 shows the situation for the second initial configuration that we propagate, $(\mathbf{tt}, \mathbf{tt}, \mathbf{ff})$. This configuration "loops" in the very same way because our must-alias analysis tells us that $i(\texttt{i2}) = i(\texttt{i3})$ and hence at line 15 we compute $c'(q_1) \equiv i \neq i(\texttt{i2}) \vee i = i(\texttt{i3}) \equiv \mathbf{tt}$. Because the configurations here do not reach the final state either, and because $i(\texttt{i2})$ is not used elsewhere in the program, the analysis we present in Section 5 will be able to statically recognize that through the shadows within the `print` method no final state can be reached.

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

(a) Table I 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) Table I when i_1 and i_2 must-alias

Table II: More precise analysis-aware reduction rules.

Table II summarizes the rules that we can use in the presence of aliasing information. Aliasing information enables us to deduce that certain constraints will always be redundant or contradictory. We give the derivations for these rules in Figure 10.

We have presented seven reduction rules in Table II. Note that the flow-sensitive analysis from [6] was only able to use the must-not-alias rules from Table IIa, since it did not have any must-alias information. Our results show that this paper's flow-sensitive, intraprocedural must-alias analysis (and the strong updates enabled by must-alias analysis) greatly enhances precision in many cases.

4.2 Distributive law

Now reconsider the `print` method from Figure 1, and assume that we instead start with the configuration $(\mathtt{tt}, \mathtt{tt}, \mathbf{ff})$ before the `hasNext` shadow at line 13. (We have changed the constraint at q_1 to be $\mathtt{tt}$ rather than $\mathbf{ff}$.) The `hasNext` shadow on the expression `i2.hasNext()` generates a constraint $i \neq i(\mathtt{i2})$ on state q_1, yielding configuration $(\mathtt{tt}, i \neq i(\mathtt{i2}), \mathbf{ff})$. The `next` shadow attached to `i3.next()` on line 15 then generates a new disjunct $i = i(\mathtt{i3})$ on q_1. Our alias analyses will give $i(\mathtt{i2}) = i(\mathtt{i3})$ (must-aliasing), so our abstraction will store $i \neq i(\mathtt{i2}) \vee i = i(\mathtt{i3})$ as $i \neq i(\mathtt{i2}) \vee i = i(\mathtt{i2})$. The distributive law from Boolean algebra then implies that $i \neq i(\mathtt{i2}) \vee i = i(\mathtt{i2}) \equiv \mathtt{tt}$. This gives the configuration $(\mathtt{tt}, \mathtt{tt}, \mathbf{ff})$ after line 15, which implies that the shadows at lines 13 and 15, in combination, are again redundant.

In general, the distributive law over the Boolean algebra states that for Boolean predicates A, B, C,

$$(A \wedge B) \vee (A \wedge C) = A \wedge (B \vee C).$$

The distributive law implies, in particular, that

$$(A \wedge X) \vee (A \wedge \neg X) = A \wedge (X \vee \neg X) = A$$

for any Boolean variables A, X. We pattern-match on constraints containing this sub-expression and simplify the constraints accordingly in our abstraction.

4.3 Unique-shadow analysis

Consider again the `print` method from Figure 1, this time with the `FailSafeIter` tracematch from Figure 5. This method contains a `create` shadow binding `c3` and `i2` at line 12 and a `next` shadow binding `i3` at line 15. Assume that we start the static analysis with $(\mathtt{tt}, \mathbf{ff}, \mathtt{tt}, \mathbf{ff})$ **(A)** as shown in Figure 11. The `create` shadow would give the configuration **(B)** after line 12. Because of the `next`-edge from q_2 to q_3, the `next` shadow at line 15 would then update the configuration at the final state q_3 to

$$(\ c \neq i(\mathtt{c3}) \wedge i = i(\mathtt{i3})\ ) \vee (\ i \neq i(\mathtt{i2}) \wedge i = i(\mathtt{i3})\ ).$$

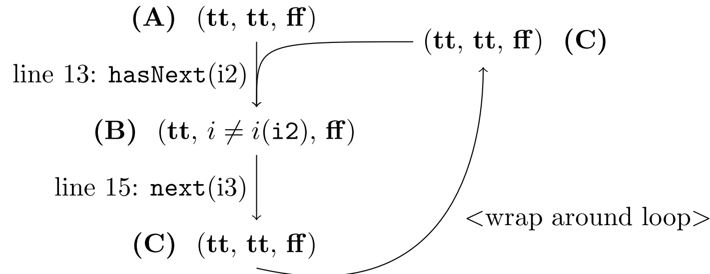

Figure 9: Effect of `print` on `HasNext` automaton.

$$\frac{x = i_1 \quad x = i_2 \quad i_1 \neq i_2}{\mathbf{ff}} \qquad \frac{x = i_1 \quad x \neq i_2 \quad i_1 \neq i_2}{x = i_1}$$

$$\frac{x = i_1 \quad x = i_2 \quad i_1 = i_2}{x = i_1 \equiv x = i_2} \qquad \frac{x \neq i_1 \quad x \neq i_2 \quad i_1 = i_2}{x \neq i_1 \equiv x \neq i_2}$$

$$\frac{x = i_1 \quad x \neq i_2 \quad i_1 = i_2}{\mathbf{ff}}$$

Figure 10: Derivations leading to our strong update rules

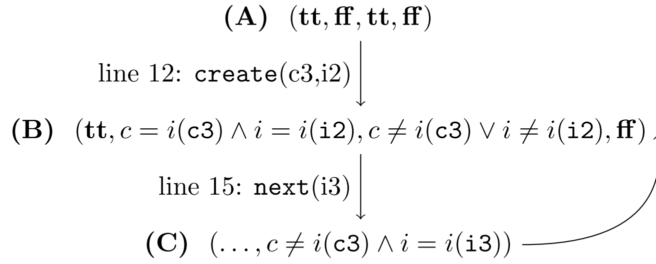

Figure 11: Effect of `print` on `FailSafeIter` automaton.

Must-alias information eliminates the second disjunct, leaving

$$c \neq i(\texttt{c3}) \wedge i = i(\texttt{i3}) \quad \textbf{(C)},$$

which means that the `next` shadow might trigger the final state q_3 on some collection (different from $i(\texttt{c3})$) and iterator $i(\texttt{i3})$.

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 one single `Collection` object. However, the current analysis has no way to infer this, i.e. to infer

$$i = i(\texttt{i3}) \Rightarrow c = i(\texttt{c3}). \tag{1}$$

Inferring this equation would allow us to reduce $c(q_3)$ in **(C)** to **ff**, proving that `print` never violates the `FailSafeIter` property.

A key insight is that equation (1) holds if we can establish that the whole program has a unique `create` shadow binding `i3` and its aliases. Once we establish the uniqueness of the `create` shadow, then we can simplify the disjunct at q_3 to **ff**, guaranteeing that the `print` method never violates the tracematch.

Unique-shadow Refinement Rule. Abstractly, our unique shadows analysis refines the conjunction of $x \neq i_1$ with $y = i_2$. When the static analysis attempts to carry out such a conjunction, it first tests whether $x = i_1 \Leftrightarrow y = i_2$ using unique-shadows analysis. If so (that is, if $\langle x, y \rangle$ are always bound together to $\langle i_1, i_2 \rangle$), then the static analysis can conclude $x \neq i_1 \wedge y = i_2 \Rightarrow \mathbf{ff}$.

We use the following algorithm to establish the unique shadows property $x = i_1 \Leftrightarrow y = i_2$:

- Verify that at least one shadow binds both x and y. If not, fail.
- Enumerate all shadows $\mathcal{S}$ that bind x and y, such that $i_x \approx i_1$ and $i_y \approx i_2$.

- Verify must-aliasing: for each shadow $s \in \mathcal{S}$ binding i_x and i_y, check that $i_x = i_1$ and $i_y = i_2$. If all shadows pass, the unique shadows property holds. Conclude that

$$x = i_1 \Leftrightarrow y = i_2.$$

Our must-alias analysis currently requires all shadows in $\mathcal{S}$ to reside in the method being analyzed to return useful information.

5 Static checkers and transformations

This section presents two static analyses and optimizations that rely on our static abstraction of tracematch states. The first identifies shadows in the program that can never contribute to reaching a final state in any execution. We remove all such shadows. The second analysis optimizes shadows in loops, even if they may trigger their tracematches. The insight is that many shadows in loops only have any effect on the first loop iteration. Our analysis identifies such situations and disables the shadow on all subsequent loop iterations.

5.1 Unnecessary-shadows analysis

Our unnecessary-shadows analysis detects shadows that cannot contribute to reaching a tracematch automaton's final state. It handles three cases:

1. Shadow s causes no state changes (1-step invariance).
2. Shadow s changes the automaton state, but its change is always undone by a future state transition (n-step invariance).
3. Shadow s changes the automaton state, but no execution path containing s hits the final automaton state (cannot-trigger-final).

Our static abstraction makes it easy to detect all three cases. The static analysis propagates the abstraction through each shadow-bearing method's control flow graph until it reaches a fixed point. To model information about possible future actions, we extend the control flow graph with a synthetic empty loop, which gets copies of all shadows that possibly refer to the same values as the shadows in the current method. We then change each original exit statement of the graph to point to the synthetic loop instead. The synthetic loop models the continuation of the computation after method exit.

Static verification of tracematches. We determine whether or not any object potentially reaches a final state by inspecting the static abstraction after the fixed-point iteration. If all nodes (including the synthetic node) in a method's control-flow graph contain **ff** at all final states, then we have statically verified that the method cannot trigger the tracematch, and we remove all shadows in the method.

Static optimization of tracematches. We also handle the case where some shadows contribute to a final state, but others do not. Shadows that do not contribute to a final state should be discarded, easing the manual verification task and reducing runtime overhead.

To distinguish between necessary and unnecessary shadows, we augment our abstraction: whenever we generate a variable binding $x = i$ or $x \neq i$, we also store the shadow that generates the binding. (If a symbol binds no variables, we store origin information directly on the constraint.) When inspecting the final states, we retain only the shadows that contribute to reaching the final states.

For example, given a constraint $x = i_1 \wedge y \neq i_2$, we look up the shadows that generated $x = i_1$ and $y \neq i_2$. If the constraint holds no additional shadow information, only the shadows generating $x = i_1$ and $y \neq i_2$ could have had an impact on whether or not the tracematch triggers. We would remove all other shadows.

The reduction rules from Section 4 apply as follows: each reduction rule reduces the number of bindings in a constraint. The more reduction rules we can find and apply, the fewer bindings will end up at final states and the fewer shadows need to be kept alive.

We handle calls to other shadow-bearing methods as follows. Our optimization will not remove any shadows that contributed bindings to *any* state, including non-final states, at any tainted (Section 3) configuration.

5.2 Run-once loop optimization

Of course, some shadows are in fact necessary (or not provably unnecessary). When such shadows reside inside hot loops, runtime monitoring can become quite expensive, as we reported in earlier work [6]. Furthermore, shadows inside loops complicate testing: it is much harder to get good coverage for loops than for straight-line code, since the tests must generally exercise the loop's corner cases.

Fortunately, we observed that monitor configurations often remained stable after the first iteration. We therefore devised an analysis that exploits our static abstraction to identify shadows that only need to run once per loop execution. The transformation based on this analysis speeds up runtime monitoring and simplifies testing.

We first describe a simple extension to our static abstraction which enables it to support the run-once optimization. The issue is that the abstraction does not capture the fact that the tracematch body executes, an observable side-effect. (The abstraction records all other changes to the runtime monitor's state). We therefore add a *hit counter*, which increases whenever a new disjunct hits the final state. Two abstract automaton configurations are equal when they have the same constraints at each state and the same hit counter.

In our running example from Figure 1, the `next` shadow at line 23 only needs to run once with the `FailSafeIter` tracematch. Because our analysis cannot determine whether any collection associated with iterator $i(\texttt{i5})$ was updated prior to entering `iteratorToList`, it must assume that the `next` shadow could potentially trigger the tracematch. We end up with a configuration

$$[\ (\mathbf{tt}, \mathbf{ff}, i \neq i(\texttt{i5}), i = i(\texttt{i5})),\ 1] \tag{2}$$

after analyzing the `next` shadow, where 1 is the hit counter. Note that, because the state q_2 of the automaton (Figure 6) has no self-loop labelled "`next`", our analysis generates the constraint $i \neq i(\texttt{i5})$ at q_2 following `next`: object $i(\texttt{i5})$ must move to the final state and cannot possibly remain in q_2. The configuration in (2) therefore stays stable in subsequent loop iterations. Our analysis can therefore identify the `next` shadow as "run-once".

In general, our analysis of a method m proceeds as follows.

- Find all reducible loops in m that contain shadows. (We currently restrict ourselves to loops with a single exit, where the configuration at loop exits is uniquely determined.)
- For each loop ℓ, inner loops first:
  - Perform a fixed-point iteration over the statements of ℓ.
  - If the analysis result after the first iteration equals the analysis result at the fixed point, and if no configuration is tainted, the shadows in ℓ only need to run once.

We find loops using a standard dominator analysis. We guard each run-once shadow that lies on a path from the loop entry to the loop exit with a Boolean flag. This flag is initialized to `true` before loop entry and set to `false` once the shadow runs. The shadow only runs if the flag is `true`, i.e. only on the first iteration.

(Note that in both the unnecessary-shadows analysis and the run-once analysis, we handle redefinitions of variables within loops soundly by using "strong instance keys" [7].)

pattern name	description
FailSafeEnum	do not update a vector while iterating over it
FailSafeIter	do not update a collection while iterating over it
HashMap	do not change an object's hash code while it is in a hash map
HashSet	do not change an object's hash code while it is in a hash set
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
PathPaint	ensure a path is stroked after it is drawn
SetWidth	must set table width before querying height
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 III: The generic and domain specific tracematch patterns which required flow-sensitive analysis

6 Experiments

We evaluated the effectiveness of our analyses and optimizations by applying them to several combinations of tracematches with benchmarks from the current version 2006-10-MR2 of the DaCapo benchmark suite [4]. Our choice of benchmarks enables a comparison with our flow-insensitive analysis from previous work [6]. In the previous paper, we presented nine tracematches applied to the 10 benchmarks of the DaCapo suite, resulting in 90 different configurations.

For this work we have added two more benchmark programs with domain-specific tracematches (vs. generic Java API tracematches used with DaCapo). SciMark2 is a benchmark for scientific computing in Java, provided by the National Institute of Standards and Technology (NIST). This benchmark uses stopwatches to collect execution times. We applied a set of four tracematches which check illegal usage patterns; for instance, stopwatches may not be stopped without first being started, and vice-versa. We included additional timers in the benchmark program to make the problem more interesting.

Pentaho Reporting provides reporting, analysis, dashboard, data mining and workflow capabilities to Java applications. Pentaho Reporting produces PDF output using the iText library for PDF manipulation. We extracted iText's usage properties from a book by one of its principal developers [12] (and found that people asked about their buggy property-violating programs on the iText list!). We encoded seven iText properties with tracematches, but Pentaho Reporting only potentially violates two of these properties.

In our experiments, we applied all of our tracematches to all DaCapo benchmarks, plus Pentaho Reporting and SciMark2. For many of these benchmark/tracematch combinations, the quick check and flow-insensitive analysis from previous work detected that the tracematch could never apply. However, 43 benchmark/tracematch combinations had remaining shadows. We applied our flow-sensitive analyses to these cases. Table III gives brief descriptions of the tracematches used in these 43 combinations. All benchmarks, tracematches and raw data are available on our website:

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

6.1 Unnecessary shadows

Table IV presents static shadow counts before and after our unnecessary-shadows and run-once analyses. The *reachable* column contains the total number of reachable shadows in the program, the *fi* column presents the number of shadows after the flow-insensitive optimizations from previous work [6], the *us* column presents

benchmark-tracematch	reachable	fi	us	ro
antlr-HasNextElem	23	23	0	0
antlr-Reader	46	15	0	0
bloat-FailSafeIter	1034	1005	845	0
bloat-HashMap	203	197	197	0
bloat-HashSet	202	171	165	0
bloat-HasNext	640	640	158	0
bloat-Writer	88	9	0	0
chart-FailSafeIter	110	110	62	0
chart-HasNext	63	63	0	0
eclipse-FailSafeEnum	55	41	39	0
eclipse-FailSafeIter	92	6	0	0
eclipse-HashMap	19	14	14	0
eclipse-HasNextElem	38	33	24	1
eclipse-HasNext	10	10	2	0
eclipse-LeakingSync	424	5	4	0
eclipse-Reader	72	5	5	5
fop-FailSafeEnum	11	6	6	0
fop-HashMap	35	35	35	0
fop-HasNextElem	9	9	0	0
fop-HasNext	5	5	0	0
jython-FailSafeEnum	58	10	9	0
jython-FailSafeIter	50	8	8	0
jython-HasNextElem	35	35	31	15
jython-HasNext	5	5	3	0
jython-Reader	32	5	5	0
lucene-FailSafeEnum	43	7	0	0
lucene-FailSafeIter	82	49	0	0
lucene-HashSet	6	4	3	0
lucene-HasNextElem	16	16	0	0
lucene-HasNext	33	29	0	0
pmd-FailSafeEnum	13	11	0	0
pmd-FailSafeIter	130	81	48	0
pmd-HashSet	25	10	10	0
pmd-HasNextElem	7	7	0	0
pmd-HasNext	88	86	9	0
pmd-Reader	33	10	0	0
xalan-HasNextElem	3	3	0	0
pentaho-PathPaint	138	92	92	0
pentaho-SetWidth	15	15	15	0
scimark-ResetRead	9	9	0	0
scimark-StartResume	8	6	0	0
scimark-StartStart	7	7	3	0
scimark-StopStop	8	8	0	0

Table IV: Number of shadows in reachable code (reachable), remaining after flow-insensitive stage (fi) [6] and unnecessary-shadow elimination (us); shadows changed to run-once (ro)

the number of shadows after the unnecessary shadow elimination optimization in this work, and the *ro* column counts the number of run-once shadows. In 18 of 43 cases, we statically prove that the benchmark programs are sound with respect to the given tracematch specification. In 12 further cases, no more than 10 shadows remain in the program. With so few remaining shadows a programmer can easily inspect each shadow by hand. We investigated those shadows in detail and found that some of the benchmarks actually do violate the patterns, indicating suspicious code and program defects.

6.2 Suspicious code, defects and sources of imprecision

We cannot remove shadows for the pmd and eclipse benchmarks, as they do not call the `hasNext()` method before calling `next()`; they instead use the underlying collection's `isEmpty()` method to guarantee that it is safe to call `next()`. Figure 12 in the appendix shows the suspicious code for eclipse, Figure 14 shown the code in the case of pmd.

The pmd-HasNext case exposes another oddity: one of its methods takes an iterator as a parameter, and simply extracts the first element with an unchecked call to `next`. This suspicious code was fixed in a later version of pmd: when the authors updated pmd for Java 5, their use of enhanced for-loops hid the iterators, thereby forcing a rewrite of the suspicious code. Figure 16a in the appendix shows the suspicious code before the refactoring, Figure 16b shows the repaired code.

The combination eclipse-Reader really requires an interprocedural analysis since a reader is used by multiple methods. See Figure 13 in the appendix for details.

Jython uses many delegating collections and iterators. In such cases, a `next()` method delegates to another iterator by calling `otherIter.next()`. Due to a lack of context, our intraprocedural analysis cannot effectively handle such delegating `next()` calls in jython-HasNext and jython-FailSafeIter. Figure 15 in the appendix shows the code for such a delegating iterator.

The shadows in jython-Reader indicate an actual defect. The code does sometimes close `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`. Figure 17 shows the source code related to this defect.

scimark-StartStart has three shadows remaining. This benchmark uses stopwatches safely, but our analysis fails to recognize this fact for the following reason. `start()` is called on a fresh stopwatch object. Because the object is fresh, `start()` cannot possibly have been called on this object before. Right now our analysis has however no notion of "freshness". Having discovered this shortcoming, we found that our abstraction was flexible enough to support a lightweight auxiliary analysis and a new reduction rule which successfully remove the remaining shadows in this benchmark. Nevertheless we chose to present the experimental results as they were before the discovery of this new rule.

There are a number of other causes for remaining shadows. Dynamic class loading forces our underlying points-to analysis to make overly conservative assumptions, leading to imprecision in our static abstraction. The `HashMap` and `HashSet` patterns work by matching the tracematch and executing tracematch bodies at many program locations. However, a part of their contract is encoded in the tracematch body, which only reports a violation when the old and new hash codes of the bound object differ. Finally, the bloat benchmark is a special case, as previously reported [6, 8]. It uses a few very long-lived collections, maps and iterators all over the program. This leads to a complex aliasing situation, with many possible side-effects, which our analysis has to conservatively handle. Nevertheless, our novel abstraction and analyses were quite successful with bloat-HasNext: whereas we failed to remove a single shadow [6] in previous work, we can now remove around 75% of the shadows from bloat-HasNext.

6.3 Run-once shadows

Our run-once optimization was only effective in three cases. This is largely due to the fact that in many other cases most shadows were already removed. Some of the remaining shadows are difficult to optimize because they belong to loops that bind values which are reassigned in different loop iterations. We believe

that the run-once optimization greatly reduces runtime overhead if it optimizes shadows belonging to hot loops.

6.4 Remaining runtime overheads

Table V summarizes runtime overheads for benchmarks with remaining shadows. (The overhead is naturally zero for the other benchmarks.) The *notm* column presents running times for the program without tracematch, the *fi* column presents relative slowdowns after the flow-insensitive stage, the *us* column presents slowdowns after the unnecessary-shadows optimization, and the *us+ro* column presents slowdowns after both the unnecessary-shadows and run-once optimizations.

Because some benchmarks require a Java 1.4 Virtual Machine, we collected runtime numbers by executing the benchmarks on Sun's HotSpot 32-Bit Client VM (build 1.4.2_12-b03), with 2GB of maximal heap space on a machine with a AMD Athlon 64 X2 Dual Core Processor 3800+ running Ubuntu 6.06 with kernel version 2.6.22-14. For the DaCapo benchmarks we used the standard workload size and enabled the `-converge` option, which repeatedly runs each benchmark until the normalized standard deviation of the runs is less than 3%. For pentaho and scimark we report an average of five runs, which also varied by less than 3%. In Table V we show values within the error margins in gray. Note that lucene occurs twice in DaCapo with different workloads (luindex and lusearch); we therefore list two distinct runtime numbers with those names in the table. The benchmark chart renders objects to the screen. In order to minimize distortion by that fact, we opened a VNC server with a virtual screen that is not actually displayed. Screen output of chart was then redirected to that server. The two benchmarks luindex/lusearch use the same binaries (the program lucene) which are compiled/optimized only once. Just their run configuration differs. The benchmark setup described here is the same as we used in our own previous work [6] and therefore enabled a direct comparison.

While removing more shadows generally leads to a faster-running program, the correlation is imperfect. In pmd-FailSafeIter, for instance, 48 shadows remain, yet the running time with instrumentation increases by only less than 5%, as compared to the un-instrumented running time. In luindex-HashSet, on the other hand, we removed all but 3 shadows. Unfortunately, these 3 shadows are hot and contribute a runtime overhead of about 14%. The run-once optimization only improved runtime for eclipse-Reader. Furthermore, experimental noise (for instance, through varying code layouts caused by our transformations) seems to cause some optimized programs to run slightly more slowly (e.g. pentaho-PathPaint).

Generally, we can see that after applying our optimizations, only 7 out of the original 43 cases show a runtime overhead of more than 10%. Furthermore, only 4 out of 43 cases show a more than twofold increase. This number looks even smaller when we take into account that we originally started with a total of 102 benchmark/tracematch combinations. While we believe that techniques for handling the remaining cases would be very interesting in theory, the practical impact seems to be be restricted to a rather small fraction of possible program/tracematch combinations.

7 Related Work

We compare our current work on tracematches to our own previous research on this topic and on the alternate specification languages encapsulated in typestate-based approaches and PQL.

7.1 Previous whole-program analyses for tracematches

In previous work we have presented a staged static analysis to improve the performance of runtime monitoring with tracematches [6]. The analysis we presented there consists of three stages, a very fast syntactic "Quick Check", a flow-insensitive pointer analysis and a flow-sensitive whole-program optimization. The first two stages are very effective at weeding out shadows that cannot lead to matches for obvious reasons, primarily those on objects that never have enough symbols to reach final states. Because these stages are relatively cheap, it makes sense to apply them before doing further tracematch evaluation, and we do so in our work.

benchmark-tracematch	notm	fi	us	us+ro
bloat-FailSafeIter	4066	>4h	>4h	
bloat-HashMap	4066	4.85	4.83	
bloat-HashSet	4066	2.87	2.83	
bloat-HasNext	4066	38.71	39.93	
chart-FailSafeIter	14621	1.07	1.07	
eclipse-FailSafeEnum	44157	1.06	1.01	
eclipse-HashMap	44157	0.99	0.99	
eclipse-HasNextElem	44157	1.04	1.02	1.02
eclipse-HasNext	44157	1.05	1.01	
eclipse-LeakingSync	44157	1.00	1.01	
eclipse-Reader	44157	1.05	1.04	1.00
fop-FailSafeEnum	2572	1.04	1.03	
fop-HashMap	2572	1.03	1.03	
jython-FailSafeEnum	11012	1.00	1.02	
jython-FailSafeIter	11012	1.00	1.01	
jython-HasNext	11012	1.05	1.03	
jython-HasNextElem	11012	1.00	1.00	0.99
jython-Reader	11012	1.04	1.00	
luindex-HashSet	17223	1.16	1.14	
lusearch-HashSet	13916	1.04	1.04	
pmd-FailSafeIter	12859	1.12	1.04	
pmd-HashSet	12859	1.05	1.03	
pmd-HasNext	12859	1.70	1.03	
pentaho-PathPaint	6116	1.67	1.71	
pentaho-SetWidth	6116	1.02	1.01	
scimark-StartStart	5190	1.27	1.15	

Table V: Raw runtimes without instrumentation in milliseconds (notm) and relative slowdowns after applying the flow-insensitive stage (fi) [6], after unnecessary-shadow elimination (us) and run-once optimization (us+ro; for cases where run-once applied)

The third stage was completely ineffective, and did not remove any shadows at all. As formulated in that paper, the third stage could not use must-alias information nor flow-sensitive must-not-alias information, both of which are crucial to the analysis presented here.

7.2 Typestate

Typestate properties [16] have been enjoying renewed interest. Typestate describes the state of heap objects one at a time, similar to a tracematch where each symbol has exactly one variable. Because of this restriction, typestate is less general than tracematches. Tracematches allow reasoning about multiple objects at once.

DeLine and Fähndrich [9] presented a method for statically checking typestate specifications 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. The most important analyses in [11], rely on the fact that typestate specifies properties of a single object at a time; our analyses allow symbols to bind multiple objects simultaneously. 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 [10], as we do. Their work uses *safe regions* in the code, as identified by typestate analyses. 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. We believe that our approach better suited to manual inspection, as it preserves crucial information about where updates occur in the source code. Summary transitions break this direct correlation.

7.3 Program Query Language

The Program Query Language [13] 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 [6]); no flow-insensitive analysis can remove the shadows that interest us here.

7.4 Eliminating runtime errors

The ASTREE static analyzer [5] has verified millions of lines of automatically generated C code for the absence of runtime errors. ASTREE ensures that programs never trigger the runtime errors defined in the C language specification, e.g. out-of-bounds array accesses and arithmetic overflow. It combines a number of

different static analyses to statically verify program properties; ASTREE uses abstract interpretation over a number of specialized domains.

While, like ASTREE, we use static analyses to detect cases where error conditions might occur, our goals differ substantially from those of ASTREE. ASTREE detects a set of runtime errors fixed in the C language specification; our specification language is flexible. We enable developers to choose the properties that are important to them (as specified as regular expressions over symbols). ASTREE verifies that integers and floating-point numbers fall within acceptable ranges, while we verify relationships between events on heap objects and their states.

8 Conclusions

We have presented a novel analysis for evaluating runtime monitoring properties ahead-of-time. Our approach is based on tracematches, which enable developers to write runtime monitors using regular expressions over program events with free variables. Event executions bind variables to heap objects. Heap objects are difficult to reason about statically, and especially so when tracematches bind heap objects as needed.

We focus on verification-oriented tracematches and evaluate them statically using a novel abstract domain. Our domain tracks object membership and, critically, non-membership in monitor states. This symmetric approach enables us to reason about single methods in isolation. We use lightweight interprocedural summary information to increase precision and ensure soundness. We also use precise, easily collected, flow-sensitive intraprocedural alias information to disambiguate heap references.

Our results show that this combined approach is surprisingly effective, despite focusing on one method at a time. In 18 out of 43 program/tracematch combinations, our analysis was able to prove that the monitor can never trigger at runtime. 12 of the other cases have at most 10 instrumentation points remaining. We found it easy to manually inspect those cases and through inspection we found defects and suspicious code in multiple programs.

Acknowledgements. We owe thanks to Manu Sridharan for help with using 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.

Downloads All benchmarks, tracematches and raw data are available on our website:

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

The download package also contains the version of `abc` we experimented with. All our analyses were already incorporated into our main branch of `abc` and will be contained in the next official release.

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 *Annual ACM SIGPLAN Conferences on Object-Oriented Programming, Systems, Languages, and Applications (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 *International Conference on Aspect-Oriented Software Development (AOSD)*, pages 87–98. ACM Press, 2005.

[3] K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In *Annual ACM SIGPLAN Conferences on Object-Oriented Programming, Systems, Languages, and Applications (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] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A Static Analyzer for Large Safety-Critical Software. In *ACM PLDI*, San Diego, California, June 2003. ACM.

[6] E. Bodden, L. J. Hendren, and O. Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *European Conference on Object-Oriented Programming (ICSE)*, 2007.

[7] 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.

[8] M. D. Bond, N. Nethercote, S. W. Kent, S. Z. Guyer, and K. S. McKinley. Tracking bad apples: reporting the origin of null and undefined value errors. In *Annual ACM SIGPLAN Conferences on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 405–422, 2007.

[9] R. DeLine and M. Fähndrich. Typestates for objects. In *ECOOP*, volume 3086 of *Lecture Notes in Computer Science (LNCS)*, pages 465–490. Springer, 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 *International Conference on Automated Software Engineering (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 *International Symposium on Software Testing and Analysis (ISSTA)*, 2006.

[12] B. Lowagie. *iText in Action*. Manning, 2007.

[13] M. Martin, B. Livshits, and M. S. Lam. Finding application errors using PQL: a program query language. In *Annual ACM SIGPLAN Conferences on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 365–383, 2005.

[14] H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In *International Conference on Compiler Construction (CC)*, volume 2622 of *Lecture Notes in Computer Science (LNCS)*, pages 46–60, 2003.

[15] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for Java. In *Conference on Programming Language Design and Implementation (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.

A Suspicious pieces of code found in DaCapo benchmark programs

```
String [] selectNativeCode(org.osgi.framework.Bundle bundle) {
    ...
    if (bundleNativeCodes.size() == 0)
        return noMatches(optional);
    Iterator iter = bundleNativeCodes.iterator();
    BundleNativeCode highestRanking = (BundleNativeCode) iter.next();
    ...
}
```

Figure 12: Suspicious code in class org.eclipse.osgi.framework.internal.core.Framework, CVS revision 1.110
The code does not check `hasNext()` but rather tests `size()==0` on the collection.

```
private void setOutput(File newOutFile, Writer newWriter, boolean append) {
    ...
    Reader fileIn = null;
    try {
        openFile();
        fileIn = new InputStreamReader(secureAction.getFileInputStream(oldOutFile),"UTF-8");
        copyReader(fileIn, this.writer);
    } catch (IOException e) {
    ...
}
```

Figure 13: Code in class org.eclipse.core.runtime.adaptor.EclipseLog, Eclipse version 3.1
The code creates a reader and then passes it to another method. After doing so, `setOutput` has to make sure not to close the reader, nor its input stream.

```
protected NameDeclaration findVariableHere(NameOccurrence occurrence) {
    if (occurrence.isThisOrSuper() || occurrence.getImage().equals(className)) {
        if (variableNames.isEmpty() && methodNames.isEmpty()) {
            return null;
        }
        if (!variableNames.isEmpty()) {
            return variableNames.keySet().iterator().next();
        }
        return methodNames.keySet().iterator().next();
    }
    ...
}
```

Figure 14: Suspicious code in class net.sourceforge.pmd.symboltable.ClassScope, SVN revision 5111 The code does not check `hasNext()` but rather tests `isEmpty()` on the collection.

```
public Iterator iterator () {
    return new Iterator() {
        Iterator i = list . iterator ();
        public void remove() {
            throw new UnsupportedOperationException();
        }
        public boolean hasNext() {
            return i.hasNext();
        }
        public Object next() {
            return i.next();
        }
    };
}
```

Figure 15: Delegating Iterator in class org.python.core.PyTuple, SVN revision 3673

```
private List markUsages(IDataFlowNode inode) {
    ...
    for (Iterator k = ((List) entry.getValue()).iterator(); k.hasNext();) {
        addAccess(k, inode);
    }
    ...
}

...

private void addAccess(Iterator k, IDataFlowNode inode) {
    NameOccurrence occurrence = (NameOccurrence) k.next();
    ...
}
```

(a) SVN revision 4797

```
private List markUsages(IDataFlowNode inode) {
    ...
    for (NameOccurrence occurrence: entry.getValue()) {
        addAccess(occurrence, inode);
    }
    ...
}

...

private void addAccess(NameOccurrence occurrence, IDataFlowNode inode) {
    ...
}
```

(b) SVN revision 4993; code was fixed when switching to enhanced Java 5 for-loops

Figure 16: Suspicious code in class net.sourceforge.pmd.dfa.variableaccess.VariableAccessVisitor
The method `addAccess` extracts the iterator's first element without a check. This program is only sound because `addAccess` is only called by `markUsages`.

```
static String getLine(BufferedReader reader, int line) {
    if (reader == null)
        return "";
    try {
        String text=null;
        for(int i=0; i < line; i++) {
            text = reader.readLine();
        }
        return text;
    } catch (IOException ioe) {
        return null;
    }
}
```

(a) Suspicious code in class org.python.core.parser; reads from potentially closed reader

```
private final void FillBuff() throws java.io.IOException
{
    ...
    try {
        if ((i = inputStream.read(buffer, maxNextCharInd, available − maxNextCharInd)) == −1) {
            inputStream.close();
            throw new java.io.IOException();
        }
        else
            maxNextCharInd += i;
        return;
    }
    ...
}
```

(b) Suspicious code in class org.python.parser.ReaderCharStream; closes reader when it hits the end of file (the variable inputStream is actually of type Reader)

Figure 17: Suspicious code in jython, SVN revision 3673
Reads from a potentially closed reader.