Typestate-like Analysis of Multiple Interacting Objects

Nomair A. Naeem Ondřej Lhoták

D. R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
{nanaeem,olhotak}@uwaterloo.ca

Abstract

This paper presents a static analysis of typestate-like temporal specifications of groups of interacting objects, which are expressed using tracematches. Whereas typesate expresses a temporal specification of one object, a tracematch state may change due to operations on any of a set of related objects bound by the tracematch. The paper proposes a lattice-based operational semantics equivalent to the original tracematch semantics but better suited to static analysis. The paper defines a static analysis that computes precise local points-to sets and tracks the flow of individual objects, thereby enabling strong updates of the tracematch state. The analysis has been proved sound with respect to the semantics. A context-sensitive version of the analysis has been implemented as instances of the IFDS and IDE algorithms. The analysis was evaluated on tracematches used in earlier work and found to be very precise. Remaining imprecisions could be eliminated with more precise modeling of references from the heap and of exceptional control flow.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification

General Terms Verification

Keywords typestate, static analysis, tracematches

1. Introduction

An object is not isolated; it interacts with other objects. For an object, a temporal specification can be expressed using typestate [35]. At any time, the object is in some state, and the state changes when an operation is performed on the object. Many programming errors can be detected by checking whether undesirable states are reachable. A multitude of typestate checking tools, both dynamic and static, have been developed [1, 5, 6, 10, 12, 13, 17–22, 24, 29]. Temporal specifications can be applied to express constraints on the interactions between software components. In this case, the specified protocol may involve multiple interacting objects from different components. Some newer specification mechanisms can express temporal properties of multiple objects [1, 10, 20, 29]. These formalisms are mainly intended for dynamic checking. In this paper, we extend techniques from static typestate verification to formulate and implement a static analysis of such multi-object temporal specifications.

The static analysis has two classes of applications. First, it can be used for sound static program verification. The analysis is intended to be precise: in the ideal case, all possible violations are ruled out statically, and the program is therefore guaranteed to observe the specified protocol. However, it is not always possible to rule out all violations statically. In this case, the program can be instrumented with dynamic checks that report violations at run time. The second application of the static analysis is to reduce the overhead of these dynamic checks. If the analysis proves that some instrumentation points cannot possibly lead to a violation, no instrumentation is required at those points. Thus, the runtime overhead at those program points is reduced.

We have chosen tracematches [1] as the formalism for specifying the temporal properties to be checked. A tracematch specifies which operations are relevant to the specification, how the operations identify the objects involved, the sequence of operations leading to an undesirable state, and what should be done when a violation is detected at run time. For our analysis, tracematches have two advantages over similar formalisms. First, they are widely applicable because their semantics is intuitive and highly expressive compared to other regular-expression-based formalisms. A key issue in defining such formalisms is how to tease apart the interactions between operations on different objects; in some other systems, operations on different objects are not cleanly separated. Conceptually, a tracematch executes a separate copy of a finite automaton for every possible combination of runtime objects. While other systems require each automaton to bind all objects on the first state transition, tracematches

OOPSLA'08, October 19–23, 2008, Nashville, Tennessee, USA.

do not have this restriction. Second, the semantics of tracematches has been formally specified, which allows us to formally prove that the static analysis soundly abstracts the semantics. The original tracematch paper motivates the design of a declarative semantics from the programmer's point of view, then proves it equivalent to an operational semantics better suited for implementation [1]. The operations, and how they bind objects, are specified using AspectJ pointcuts, which are in widespread use and have a formal specification [3].

While the operational tracematch semantics is convenient for a dynamic implementation, it is difficult to abstract statically because it is defined in terms of manipulating and simplifying boolean formulas, a relatively complicated concrete domain. Thus, we have defined a new, equivalent semantics based on sets and lattices, which are more convenient to reason about and to abstract. We have proven the two semantics bisimilar. The static analysis uses a provably sound abstraction of the lattice-based semantics.

The formal definitions and correctness proofs are important because reasoning about interacting objects is subtle. Allan et al. wrote this about their dynamic implementation:

> In our experience it is very hard to get the implementation correct, and indeed, we got it wrong several times before we formally showed the equivalence of the declarative and operational semantics. [1]

Similar pitfalls apply when defining a static analysis.

A key difference between our analysis and previous work on typestate verification is that in a tracematch, typestate is associated not with a single object, but with a group of objects. Existing work on typestate verification (e.g. [17, 18]) generally uses some abstraction of objects and adds the current state to each abstract object. This approach cannot be applied when there is no single object to which the state can be attached. Thus, our analysis uses two separate abstractions: the first models individual objects and the second models tracematch state of related groups of objects. The first analysis uses a storeless heap abstraction [14, 27] similar to earlier work [11, 17, 18, 23, 34]. The focus of the paper is on the second analysis, which is novel. Indeed, we present a specific object analysis only for the sake of concreteness; the object analysis could be replaced with more precise or cheaper variants if necessary for a particular application.

The example in Figure 1 illustrates the kind of property that the analysis verifies. The method `flatten` takes a list of lists `in`, and adds all of their elements to the list `out`. The automaton below the code checks that a list is not updated during iteration, and that every call to `next` on an iterator is preceded by a call to `hasNext`. A violation of the property causes the automaton to enter one of the final states. The tracematch associated with this automaton (shown in Figure 2) has two parameters, the list (c) and the iterator (i). The next and hasNext operations bind the iterator i, update binds the list c, and makeiter binds both.

```
1  void flatten(List in, List out) {
2    Iterator it = in.iterator();
3    while(it.hasNext()) {
4      List l = (List) it.next();
5      Iterator it2 = l.iterator();
6      while(it2.hasNext()) {
7        Object o = it2.next();
8        out.add(o);
9      }
10   }
11 }
```

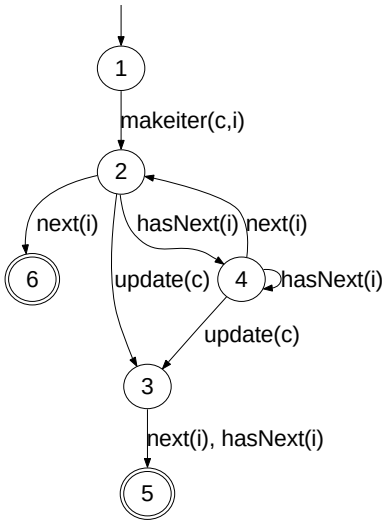

Figure 1. Tracematch example: iterator safety

According to the declarative tracematch semantics, a copy of the automaton is made for every possible runtime pair of list and iterator. Each operation causes a transition in those automata consistent with the bindings. For example, the update(c) operation on runtime list object o_c causes an update transition in all automaton copies having o_c as their list c.

Consider what information a static analysis needs to prove the absence of a violation. First, it needs precise may-alias information to determine that the list `out` updated in line 8 is not aliased with the list `in` or any of the lists it contains, over which the loops iterate. Interprocedural information is necessary because aliases may be made elsewhere; for example, the caller of the method could pass in the same list as both `in` and `out`. In fact, since the method could be called several times on different lists, context sensitivity is useful. In addition, the analysis must ensure that each call to `hasNext` occurs on the same iterator as the subsequent call to `next`. Although some have suggested using must-alias analysis, proving this fact requires more than just knowing that a pair of variables must be aliased. A must-alias analysis can prove that whenever execution reaches a given program point, two variables point to the same object. A must-alias analysis does not say anything about the values of variables

```
1  tracematch(Collection c, Iterator i) {
2     sym makeiter after returning(i): call(* Collection+.iterator()) && target(c);
3     sym next before: call(* Iterator+.next()) && target(i);
4     sym hasNext before: call(* Iterator+.hasNext()) && target(i);
5     sym update after : (call(* Collection+.add*(..)) ||
6                         call(* Collection+.clear()) ||
7                         call(* Collection+.remove*(..)) ) && target(c);
8
9     makeiter (hasNext+ next)* ( next | hasNext* update+ (next | hasNext) )
10   {
11       throw new RuntimeException(``Violated safety property.'');
12   }
13 }
```

Figure 2. Tracematch source code

at *different* times during execution. It would be difficult to extend the notion of must-aliasing to an unambiguous definition of the relationship between variables at different times. For example, it is *not* true that `it2` in line 6 always points to the same object as `it2` in line 7. When control flows from line 6 to line 7, `it2` continues to point to the same iterator, but when control flows from line 7 around the outer loop and back to line 6, the object to which `it2` points changes. Thus, a statement about the relationship between `it2` at line 6 and `it2` at line 7 would be ambiguous unless it somehow considered specific control flow paths between the two points. Instead, in order to reason about the objects pointed to by variables at different points in time, our analysis must track the flow of individual objects along specific control flow paths. To summarize, the analysis requires:

1. precise may-alias information,
2. precise context-sensitive interprocedural information, and
3. flow-sensitive tracking of individual objects along control flow paths.

The analysis presented in this paper satisfies all three requirements.

The main contributions of this paper are:

1. We define a lattice-based operational semantics of tracematches which is better suited to static analysis than the original semantics of Allan et al. [1]. We have proven that the two semantics are bisimilar. (Section 2)
2. We define a precise static abstraction of the lattice-based operational semantics. We have proven that the overall abstraction is sound with respect to the operational semantics. (Section 3)
3. We express the static analysis as instances of the IFDS [32] and IDE [33] frameworks which efficiently support context-sensitive interprocedural analysis. (Section 4)
4. We report experimental results from our implementation of the static analysis. We implemented the analysis in Scala, using the tracematch implementation in the abc compiler [1,2] to provide the intermediate representation to be analyzed. (Section 5)

Due to space constraints, complete formal details and proofs are presented in a separate technical report [30].

2. Tracematch Semantics

Allan et al. [1] define a tracematch as follows:

DEFINITION 1. *A* tracematch *is a triple* $\langle F, A, P \rangle$*, where*

F *is a finite set of tracematch parameters,*
A *is a finite alphabet of symbols (operations), and*
P *is a regular language over* A*.*

Figure 2 shows the source code that a programmer would write to define the example tracematch discussed in Section 1. This tracematch has two parameters, a `Collection c` and an `Iterator i`. Lines 2-7 define the four tracematch symbols. Each symbol is accompanied by an AspectJ pointcut that specifies where in the base code the symbol occurs. A pointcut may also bind objects from the base code to tracematch parameters. For example, the `makeIter` pointcut binds the target of the call (the collection) to `c` and the returned iterator to `i`. Line 9 defines the regular language of the tracematch and lines 10-12 provide the code to be executed when the tracematch matches at run time.

When writing a tracematch, the programmer specifies P using a regular expression. Internally within the abc compiler, P is represented as a non-deterministic finite automaton accepting the same language. To refer to this NFA, we use the customary notation $\langle Q, A, q_0, Q_f, \delta \rangle$, where Q is a finite set of states, A is the finite alphabet of tracematch symbols, $q_0 \in Q$ is the start state, $Q_f \subseteq Q$ is a set of final states, and $\delta \subseteq Q \times A \times Q$ is a transition relation.

A tracematch is applied to a program in an existing language such as Java or AspectJ. The program executes ac-

cording to the semantics of the base language, but the dynamic tracematch implementation maintains additional state to keep track of the configuration of the tracematch. Allan et al. defined a declarative semantics of how tracematches ought to work, as well as an operational semantics that they proved equivalent [1].

Next, we review the declarative semantics. We then define a new operational semantics based on sets and lattices which is more amenable to static analysis. In the technical report, we have proven the lattice-based semantics equivalent to the semantics of Allan et al. Thus, all three semantics are equivalent.

2.1 Declarative Semantics of Tracematches

The essential part of a tracematch is a regular expression over operations of interest (symbols). The dynamic tracematch implementation checks, for each suffix of the program trace, whether the suffix is a word in the language specified by the regular expression. Each such word is a *match* and causes the tracematch body to be executed. When a tracematch defines a safety property, each violation of the specified property is a match of the tracematch.

Much of the expressive power of tracematches comes from their parameters, to which symbols can bind specific objects. The tracematch body executes for each suffix of the trace that matches the specified regular expression with a consistent set of object bindings. The declarative semantics makes this precise: a separate *version* of the tracematch automaton is instantiated for each possible set of objects that could be bound to the tracematch parameters. These automaton versions run independently of each other. An automaton version makes a transition on each event in the trace if the parameters bound by the event are bound to the objects associated with that automaton version. The tracematch body is executed whenever an automaton version reaches an accepting state; at that point, the automaton version is discarded.

We illustrate with an example. Figure 3 shows a possible trace of the events declared in the tracematch from Figure 2. Each `hasNext` and `next` event binds an iterator object, `update` binds a list object and `makeiter` binds both a list and an iterator. We assume the program creates two list objects `x` and `y` and two iterator objects `a` and `b`. Thus, there are four possible ways in which these objects could be bound to the parameters, which correspond to the four automaton versions shown as columns in Figure 3. Each column includes only those events from the trace that are consistent with the object bindings of each version. The example trace results in matches of two automaton versions: the version with `c=x` and `i=a`, and the version with `c=y` and `i=b`. The first of these signals that the collection was modified while it was being iterated. The second signals two consective `next` events without an intervening `hasNext` event on the same iterator.

Trace	`c=x` `i=a`	`c=x` `i=b`	`c=y` `i=a`	`c=y` `i=b`
makeiter(x,a)	makeIter			
hasNext(a)	hasNext		hasNext	
makeiter(y,b)				makeIter
next(a)	next		next	
hasNext(b)		hasNext		hasNext
update(x)	update	update		
next(b)		next		next
next(a)	next		next	
next(b)		next		next
	match	no	no	match

Figure 3. Declarative semantics of tracematches. Column 1 shows the program trace. Columns 2 to 5 show automaton versions for different runtime objects bound to tracematch parameters.

2.2 A Lattice-Based Operational Semantics

The abc compiler includes a transformation that implements tracematch semantics at run time. This is done by inserting additional code, which we call *transition statements*, at each point in the base program where a tracematch symbol could match. In the dynamic implementation, the effect of each transition statement is to update the tracematch state to reflect the corresponding state transition and parameter bindings. The operational semantics is defined on the code that results after transition statements have been inserted.

Before performing the static analysis, we simplify the code to an intermediate representation (IR) containing only instructions relevant to tracematch semantics. The intraprocedural instructions in the IR are:

$$s ::= \textbf{tr}\,\langle a, b\rangle \mid \textbf{body}$$
$$\mid v_1 \leftarrow v_2 \mid v \leftarrow \textbf{h} \mid \textbf{h} \leftarrow v \mid v \leftarrow \textbf{null} \mid v \leftarrow \textbf{new}$$

In addition, the IR contains method call and return instructions. In the IR, v can be any variable from the set **Var** of local variables of the current method. The symbol **h** represents any heap location, such as a field of an object or an array element.

The two instructions directly relevant to tracematches are **tr** (*transition statement*) and **body** (*body statement*). Each transition statement contains a pair[1] a, b where $a \in A$ is one of the symbols of the tracematch and $b : F \hookrightarrow \textbf{Var}$ is a partial map specifying the object to be bound to each tracematch parameter. The map b binds a subset of the parameters; any of the parameters may be left unbound. When

[1] Allan et al. [1] allow each transition statement to contain *multiple* transitions, each a pair $\langle a, b\rangle$. This is necessary because their implementation allows a single instruction to be matched by multiple tracematch symbols. We fully handle this general case in the technical report. The generality does not add expressivity, nor does it make the analysis any more interesting, only more complicated. We therefore restrict our discussion in this paper to the common case of a single pair.

tr $\langle a, b\rangle$ is executed, each automaton version whose object bindings are consistent with the objects currently pointed to by the variables specified by b performs a transition on the symbol a.

A body statement is generated immediately after every transition statement **tr** $\langle a, b\rangle$ in which a is a symbol on which the tracematch automaton contains a transition into an accepting state. The effect of **body** is to find each automaton version in an accepting state, execute the tracematch body for it, and discard it.

The remaining IR instructions are self-explanatory: they copy object references between variables and the heap, and create new objects.

In the declarative semantics, the number of automaton versions that must be maintained is unbounded because the number of objects that could be created by the program is unbounded. This unboundedness hinders both a practical dynamic implementation and a static analysis. Therefore, Allan et al. defined an equivalent operational semantics. For the same reason, we define a different operational semantics that is well suited for static analysis. All three semantics have been proven equivalent.

The core construction of our semantics is a *binding lattice*. Figure 4 illustrates a sample binding lattice for a program with three objects o_1, o_2, o_3; in general, the binding lattice is defined analogously for the unbounded number of objects that the program may allocate. Thus, the binding lattice is infinite. In Section 3.2, we will define a finite abstraction of the binding lattice for use in the static analysis. The binding lattice comprises the element $\bot$, positive bindings (which are a single object), and negative bindings (which contain zero or more objects). The interpretation of each element of the binding lattice is a set of objects: $\bot$ represents the empty set, a positive binding represents a single object, and a negative binding represents the set of all objects other than those in the binding. We write $\top$ as a synonym for the empty set of negative bindings (which represents all objects). The lattice order corresponds to the subset order on sets of objects: for any pair of bindings $d_1 \sqsubseteq d_2$, every object in the set represented by d_1 is also in the set represented by d_2. As a reminder that a set of objects indicates negative bindings, we will always write such a set with a bar above it: $\overline{O}$. The bar is only a reminder; it has no semantic meaning.

We extend the binding lattice pointwise to the space of functions that map each tracematch parameter to an element of the binding lattice. We say that a mapping $m \in F \rightarrow$ **Bind** is *consistent* with a given automaton version if the object it associates with each parameter f is in the set represented by $m(f)$. Thus, each mapping m can be interpreted as a set of automaton versions. For example, consider the mapping $\texttt{c} \mapsto \texttt{x}, \texttt{i} \mapsto \overline{\{\texttt{b}\}}$. Of the automaton versions shown in Figure 3, only the one corresponding to `c=x` and `i=a` is consistent with this mapping. Again, the lattice order on

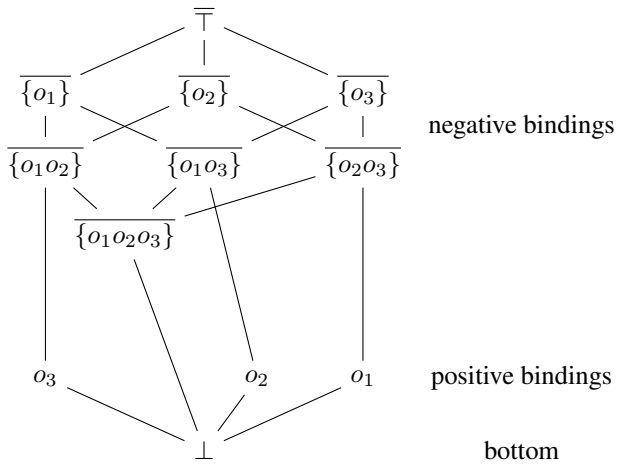

Figure 4. Concrete Binding Lattice **Bind**

$F \rightarrow$ **Bind** corresponds to the subset order on automaton versions.

The runtime state of a tracematch is then defined as a set σ of pairs $\langle q, m\rangle$, where q is a tracematch state, and $m \in F \rightarrow$ **Bind**. Each pair $\langle q, m\rangle$ indicates that all automaton versions consistent with m are in the state q.

When execution begins, the initial tracematch state is the single pair $\langle q_0, \lambda f.\top\rangle$. The binding map $\lambda f.\top$ is consistent with every version of the automaton, and q_0 indicates that all these versions are in the initial state.

Whenever a transition statement executes, some automaton versions change state and others keep their old state. A mapping m in the runtime state must be refined to distinguish the versions whose state changes from those whose state remains the same. In both cases, this refinement is done using the meet operator of the lattice.

For example, consider a tracematch with a single parameter f and the automaton in Figure 5, and suppose that the transition $\langle a, f \mapsto o_1\rangle$ occurs. The automaton version for o_1 should move to state qa and all others should remain in state q. From the initial map $\lambda f.\top$, we perform meets with $\lambda f.o_1$ and $\lambda f.\overline{\{o_1\}}$ to obtain the desired pairs $\langle qa, \lambda f.o_1\rangle$ and $\left\langle q, \lambda f.\overline{\{o_1\}}\right\rangle$. Suppose the transition $\langle b, f \mapsto o_2\rangle$ occurs next. We again perform the meets of the existing states with both $\lambda f.o_2$ and $\lambda f.\overline{\{o_2\}}$ to obtain $\langle qab, \lambda f.\bot\rangle, \langle qa, \lambda f.o_1\rangle, \langle qb, \lambda f.o_2\rangle, \left\langle q, \lambda f.\overline{\{o_1o_2\}}\right\rangle$. Since the binding in the first pair is $\bot$, it is not consistent with any automaton version and can be discarded. The next two pairs correspond to the two automaton versions for o_1 and o_2 in states qa and qb, respectively, and the final pair corresponds to all other automaton versions still in the initial state.

In the general case of a tracematch with multiple parameters, there is an additional difference between negative and positive bindings. In the declarative semantics, only the au-

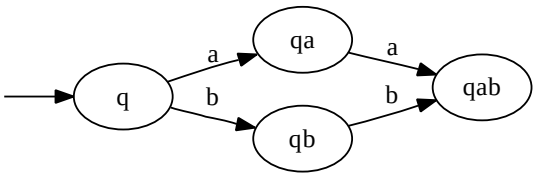

Figure 5. Example automaton

tomaton versions consistent in *all* the parameters bound by the transition statement change state; if an automaton version is inconsistent in *any* parameter, its state remains the same. Thus, for the automaton versions that change state, the new map is computed by replacing each $m(f)$ with the meet $m(f) \sqcap o$, where o is the object bound to f by the transition statement. However, for the automaton versions that do not change state, multiple maps must be computed, one for each parameter bound by the transition statement. The map computed for each parameter f reflects the condition that the object bound to f in the automaton version differs from the object bound to f by the transition statement. Thus, the new map for parameter f is constructed by replacing only $m(f)$ with $m(f) \sqcap \overline{\{o\}}$, where o is the object bound to f by the transition statement.

A formal definition of the transition function e that is applied to each pair $\langle q, m\rangle$ in the tracematch state is given in Figure 6. In the technical report, the semantics is fully formalized and proven equivalent to the operational semantics of Allan et al.

$$e_0^+(b,\rho) \triangleq \lambda f. \begin{cases} \rho(b(f)) & \text{if } f \in \text{dom}(b) \\ \top & \text{otherwise} \end{cases}$$

$$e_0^-(b,\rho,f) \triangleq \lambda f'. \begin{cases} \overline{\{\rho(b(f))\}} & \text{if } f = f' \\ \top & \text{otherwise} \end{cases}$$

$$e^+[a,b,\rho](q,m) \triangleq \left\{\left\langle q', m \sqcap e_0^+(b,\rho)\right\rangle : \delta(q,a,q')\right\}$$

$$e^-[b,\rho](q,m) \triangleq \left\{\left\langle q, m \sqcap e_0^-(b,\rho,f)\right\rangle : f \in \text{dom}(b)\right\}$$

$$e[a,b,\rho](q,m) \triangleq e^+[a,b,\rho](q,m) \cup e^-[b,\rho](q,m)$$

Figure 6. Transition function, in the Lattice-based operational semantics, for **tr** $\langle a, b\rangle$ in local variable environment ρ, which is applied to each pair $\langle q, m\rangle$ in the tracematch state.

3. Static Abstraction

The abstraction is presented in two parts. The first abstraction computes object aliasing relationships. This information is needed to determine which objects are pointed to by the variables in each transition statement. The second abstraction models the tracematch state. Using this abstraction, the analysis can prove that at certain body statements, the tracematch cannot be in an accepting state.

3.1 Object Abstraction

The object abstraction represents each concrete object by the set of local variables pointing to it. This is the same abstraction as the nodes in Sagiv et al.'s shape analysis [34]. However, our abstraction tracks only the nodes, not the pointer edges between objects.

The set of variables in the abstraction of each object is exact; it is neither a may-point-to nor a must-point-to approximation. Since it may not be known statically whether a given pointer points to the object, the analysis maintains a set $\rho^\sharp$ of abstract objects. This set is an overapproximation of all possible objects. That is, if it is possible for some concrete object to be pointed to by the set of variables $o^\sharp$, then the set $o^\sharp$ must be an element of $\rho^\sharp$. Converesely, the presence of $o^\sharp$ in $\rho^\sharp$ indicates that there may exist zero or more concrete objects which are pointed to by the variables in $o^\sharp$ and no others. For example, consider a concrete environment in which variables x and y point to distinct objects and z may be either null or point to the same object as x. The abstraction of this environment would be the set $\{\{x\}, \{x, z\}, \{y\}\}$.

The abstraction subsumes both may-alias and must-alias relationships. If variables x and y point to distinct objects, $\rho^\sharp$ will not contain any set containing both x and y. If variables x and y point to the same object, every set in $\rho^\sharp$ will contain either both x and y, or neither of them.

At run time, a variable cannot point to more than one object at a time. Thus, every abstract object except the empty set $\emptyset$ represents at most one concrete object at any given point of execution. This enables precise flow-sensitive analysis including strong updates.

Specifically, if s is any statement in the IR except a heap load, and if $o^\sharp$ is the set of variables pointing to a given concrete object o, then it is possible to compute the exact set of variables which will point to o after the execution of s. The transfer function $[\![s]\!]_{o^\sharp}$ that performs this computation is shown in Figure 7. This property enables the analysis to flow-sensitively track individual objects along control flow paths; this was one of the three requirements motivated in the introduction. We formalize the property in the accompanying technical report [30, Proposition 2].

To precisely handle the uncertainty in heap loads we use the materialization or focus operation [11,17,18,23,34]. The abstract object $o^\sharp$ is split into two, one representing the single concrete object that was loaded, and the other representing all other objects previously represented by $o^\sharp$. Focus is important to regain the precision lost when an object is no longer referenced from any local variables, in which case the analysis lumps it together with all other such objects. In order for a tracematch operation to be performed on such an object, the object must first be loaded into a variable. At the load, the focus operation separates the loaded object from the other objects. If multiple tracematch operations are then performed on the object, the analysis knows that they are

$$
\begin{array}{rcl}
[\![s]\!]_{o^\sharp}(o^\sharp) & \triangleq & \begin{cases} o^\sharp \cup \{v_1\} & \text{if } s = v_1 \leftarrow v_2 \wedge v_2 \in o^\sharp \\ o^\sharp \setminus \{v_1\} & \text{if } s = v_1 \leftarrow v_2 \wedge v_2 \notin o^\sharp \\ o^\sharp \setminus \{v\} & \text{if } s \in \{v \leftarrow \textbf{null}, v \leftarrow \textbf{new}\} \\ o^\sharp & \text{if } s \in \{e \leftarrow v, \textbf{tr}(T), \textbf{body}\} \\ \text{undefined} & \text{if } s = v \leftarrow e \end{cases} \\
\mathit{focus}[h^\sharp](v, o^\sharp) & \triangleq & \begin{cases} \{o^\sharp \setminus \{v\}\} & \text{if } o^\sharp \notin h^\sharp \\ \{o^\sharp \setminus \{v\}, o^\sharp \cup \{v\}\} & \text{if } o^\sharp \in h^\sharp \end{cases} \\
[\![s]\!]_{O^\sharp}[h^\sharp](O^\sharp) & \triangleq & \begin{cases} \{[\![s]\!]_{o^\sharp}(o^\sharp) : o^\sharp \in O^\sharp\} & \text{if } s \neq v \leftarrow e \\ \bigcup_{o^\sharp \in O^\sharp} \mathit{focus}[h^\sharp](v, o^\sharp) & \text{if } s = v \leftarrow e \end{cases} \\
[\![s]\!]_{\rho^\sharp}(\rho^\sharp, h^\sharp) & \triangleq & \begin{cases} [\![s]\!]_{O^\sharp}[h^\sharp](\rho^\sharp) \cup \{\{v\}\} & \text{if } s = v \leftarrow \textbf{new} \\ [\![s]\!]_{O^\sharp}[h^\sharp](\rho^\sharp) & \text{otherwise} \end{cases} \\
[\![s]\!]_{h^\sharp}(\rho^\sharp, h^\sharp) & \triangleq & [\![s]\!]_{O^\sharp}[h^\sharp] \left(\begin{cases} h^\sharp \cup \{o^\sharp \in \rho^\sharp : v \in o^\sharp\} & \text{if } s = e \leftarrow v \\ h^\sharp & \text{otherwise} \end{cases} \right)
\end{array}
$$

Figure 7. Transfer function for the object abstraction

performed on the same concrete object as long as the local variable continues to point to it.

In addition to the set $\rho^\sharp$ of possible abstract objects, the analysis tracks a subset $h^\sharp \subseteq \rho^\sharp$ of abstract objects which may have escaped to the heap. The focus operation is performed only on these escaped abstract objects. Since focus splits one abstract object into two, it can theoretically lead to exponential growth in the abstraction. The escape information was necessary and sufficient to control this growth in the benchmarks that we evaluated.

The technical report formally defines a correctness relation that ensures that for any concrete object o occurring at run time, its abstract counterpart $o^\sharp$ is included in $\rho^\sharp$, as well as in $h^\sharp$ if o is referenced from the heap. We have proven that the transfer function for $\rho^\sharp$ and $h^\sharp$ preserves the correctness relation [30, Theorem 2].

We illustrate the effect of the transfer functions using the example statement sequence shown in Figure 8. Statement 1 creates a new concrete object and assigns it to variable x. Correspondingly, the transfer function $[\![s]\!]_{\rho^\sharp}$ creates the abstract object $\{x\}$. Statement 2 assigns the value of x to some pointer in the heap. The transfer function $[\![s]\!]_{h^\sharp}$ adds the abstract object $\{x\}$ to $h^\sharp$ since the concrete object represented by this abstract object has been assigned to a heap location. The value of x is then assigned to a local variable w in statement 3. The transfer function $[\![s]\!]_{o^\sharp}$ adds the variable w to the abstract object $\{x\}$ since after statement 3 executes, w and x point to the same concrete object. Statement 4 creates a new concrete object and assigns it to y. Like in statement 1, a new abstract object, $\{y\}$, is added to $\rho^\sharp$. Statement 5 is a load from the heap. The transfer function $[\![s]\!]_{O^\sharp}$ applies the focus operation to both $\{y\}$ and $\{x, w\}$. Since the abstract object $\{y\}$ is not in $h^\sharp$, $\mathit{focus}[h^\sharp](z, \{y\})$ is simply $\{\{y\}\}$. However, since $\{x, w\} \in h^\sharp$, this abstract object is split into two: $\{x, w, z\}$ and $\{x, w\}$. After statement 5, $\rho^\sharp$ contains three abstract objects: $\{x, w\}, \{y\}$, and $\{x, w, z\}$. Statement 6 assigns y to x. The transfer function $[\![s]\!]_{o^\sharp}$ is applied to each of the three abstract objects, yielding $\{w\}, \{x, y\}$, and $\{w, z\}$. Statement 7 assigns **null** to z. This changes $\{w, z\}$ to simply $\{w\}$, yielding the abstract environment $\{w\}, \{x, y\}$.

	$\rho^\sharp$	$h^\sharp$
1: $x \leftarrow$ **new**	$\{x\}$	
2: **h** $\leftarrow x$	$\{x\}$	$\{x\}$
3: $w \leftarrow x$	$\{x, w\}$	$\{x, w\}$
4: $y \leftarrow$ **new**	$\{x, w\}, \{y\}$	$\{x, w\}$
5: $z \leftarrow$ **h**	$\{x, w\}, \{y\}, \{x, w, z\}$	$\{x, w\}, \{x, w, z\}$
6: $x \leftarrow y$	$\{w\}, \{x, y\}, \{w, z\}$	$\{w\}, \{w, z\}$
7: $z \leftarrow$ **null**	$\{w\}, \{x, y\}$	$\{w\}$

Figure 8. Example statement sequence to illustrate transfer functions

3.2 Tracematch Abstraction

Typestate associates a state with each runtime object. Existing typestate analyses (e.g. [17, 18]) model each runtime object using an abstraction similar to the one defined in the previous section. The typestate analysis models the state of a runtime object by maintaining a set of possible states for each abstract object. A runtime object o can only be in state q if the abstract object $o^\sharp$ representing o has q in its set of possible states. When the analysis encounters an instruction that changes the state of an object, it updates the possible states of the appropriate abstract objects.

In our setting, a state is not associated with any single object, but with multiple objects. Thus, we cannot just add the state to any given object abstraction. Therefore, our analysis uses a second abstraction to represent the tracematch state. Each such abstract tracematch state contains within it the abstractions of the objects bound by the tracematch.

We begin by presenting a simple but inefficient abstraction of the tracematch state, then discuss the refined version that we have implemented in our analysis. Thanks to the lattice-based design of our tracematch semantics, a basic tracematch state abstraction would be straightforward to define. Recall that a concrete tracematch state is a set of pairs $\langle q, m\rangle$, where m maps each tracematch parameter to an element of the **Bind** lattice. An abstraction of this state could be defined by replacing all concrete objects in the **Bind** lattice with their abstract counterparts as defined in the previous section. The resulting abstract lattice **Bind**$^\sharp$ has the same structure as **Bind**, but each positive binding is an *abstract* object, and each negative binding is a set of *abstract* objects. The overall abstraction is a set of pairs $\langle q, m^\sharp\rangle$, where $m^\sharp$ maps each tracematch parameter to an element of **Bind**$^\sharp$. After working out some details, we defined a transfer function on this domain, proved that it correctly abstracts the semantics, and implemented it. However, on tracematches with multiple parameters, the implementation did not scale to large benchmarks. The key reason for this is that the focus operation was applied to every abstract object bound by a tracematch state. Since each focus splits the state into two, the growth was exponential in the number of abstract objects appearing in the tracematch state.

In fact, there is little benefit to performing the focus operation once the object has been bound in a tracematch state. The benefit of the focus operation is that it singles out one object, so that if a sequence of transition statements occurs, we know that they occur on the same concrete object. Thus, focus is needed for precise aliasing information at the transition statement before an object is bound. However, after the object is bound, focusing it simply causes both resulting objects to appear in two separate tracematch states, and does not improve precision of the tracematch abstraction.

Therefore, in the tracematch state, we replaced the object abstraction (the precise set of variables pointing to the object) with an under- and over-approximation: a pair of a must set $o^!$ and may set $o^?$ represents every concrete object pointed to by all variables in $o^!$ and only by variables in $o^?$. In the special case when the must and may sets are equal, we recover the precise set of variables pointing to the object. The resulting abstract lattice **Bind**$^\sharp$ is illustrated for two variables x, y in Figure 9. We use the notation $x?$ to say that the variable x is in the may set but not the must set, and x to say that it is in both sets. Suppose that a tracematch state has bound an object pointed to by x and a heap load to y occurs. Instead of focusing the bound object to x and xy, we instead use the join of these two, namely $xy?$, to represent both possibilities. Thus, we avoid focusing objects already bound in the tracematch state.

Efficiency can be further improved for negative bindings. It turns out that the transfer function is independent of the may sets of negatively-bound objects; thus, we need only maintain the must sets. This is because a negative binding

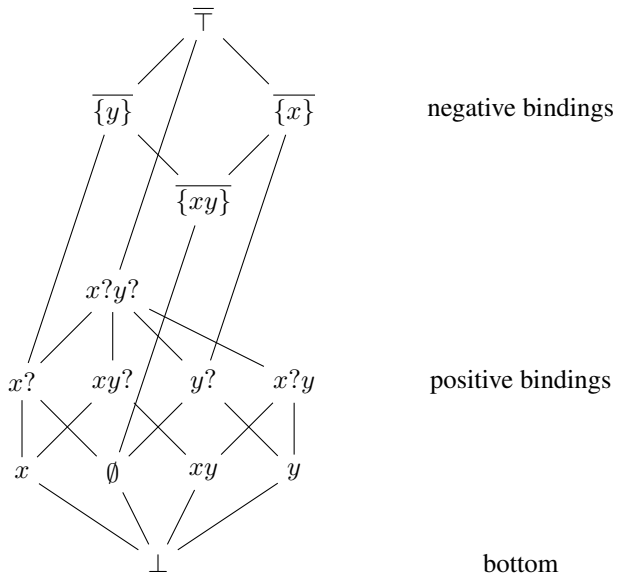

Figure 9. Abstract Binding Lattice **Bind**$^\sharp$

indicates that some object o' is not the object o bound by a given automaton version; knowing that a given variable v may not point to o' gives no information about the identity of o, since v could still point to some other object o'' that is also not o. In addition, although a concrete negative binding is a *set* of objects, all the must sets representing these objects can be replaced with their intersection without affecting precision of the analysis. Thus, the **Bind**$^\sharp$ lattice illustrated in Figure 9 represents a negative binding as simply a set of variables that definitely point to every concrete object that may have been negatively bound.

In the technical report, we formally define **Bind**$^\sharp$, show that it is a finite lattice, and that the abstraction preserves the partial order from **Bind** in **Bind**$^\sharp$ [30, Propositions 3 and 4].

The transfer function for the tracematch state abstraction for all statements except transition statements is shown in Figure 10. We again draw a bar over each negative binding like we did for the concrete tracematch lattice. The helper function $[\![s]\!]_{d^\sharp}$ is similar to $[\![s]\!]_{o^\sharp}$ from the object abstraction, but it updates both the must and may set of each abstract binding. On a heap load instruction, it introduces uncertainty into the binding instead of focusing it. The transfer function is extended pointwise to maps of bindings by $[\![s]\!]_{m^\sharp}$ and to sets of abstract state pairs by $[\![s]\!]_{\sigma^\sharp}$. Like for $[\![s]\!]_{o^\sharp}$, we have proven that the adapted function $[\![s]\!]_{d^\sharp}$ also tracks each concrete object flow-sensitively along control flow paths [30, Proposition 5].

The transfer function for transition statements is more complicated. In the operational semantics, all variables mentioned in each transition statement are looked up in the concrete environment. How should this lookup be performed in the abstract domain? A sound but imprecise and therefore

$$[\![s]\!]_{d^\sharp}(\bot) \triangleq \bot \text{ for all statements } s$$

$$[\![s]\!]_{d^\sharp}\left(\left\langle o^!, o^? \right\rangle\right) \triangleq \begin{cases} \left\langle o^! \cup \{v_1\}, o^? \cup \{v_1\} \right\rangle & \text{if } s = v_1 \leftarrow v_2 \wedge v_2 \in o^! \\ \left\langle o^! \setminus \{v_1\}, o^? \cup \{v_1\} \right\rangle & \text{if } s = v_1 \leftarrow v_2 \wedge v_2 \notin o^! \wedge v_2 \in o^? \\ \left\langle o^! \setminus \{v_1\}, o^? \setminus \{v_1\} \right\rangle & \text{if } s = v_1 \leftarrow v_2 \wedge v_2 \notin o^! \wedge v_2 \notin o^? \\ \left\langle o^! \setminus \{v\}, o^? \setminus \{v\} \right\rangle & \text{if } s \in \{v \leftarrow \textbf{null}, v \leftarrow \textbf{new}\} \\ \left\langle o^! \setminus \{v\}, o^? \cup \{v\} \right\rangle & \text{if } s = v \leftarrow e \\ \left\langle o^!, o^? \right\rangle & \text{if } s \in \{e \leftarrow v, \textbf{body}\} \end{cases}$$

$$[\![s]\!]_{d^\sharp}\left(\overline{V^\sharp}\right) \triangleq \begin{cases} \overline{V^\sharp \cup \{v_1\}} & \text{if } s = v_1 \leftarrow v_2 \wedge v_2 \in \overline{V^\sharp} \\ \overline{V^\sharp \setminus \{v_1\}} & \text{if } s = v_1 \leftarrow v_2 \wedge v_2 \notin \overline{V^\sharp} \\ \overline{V^\sharp \setminus \{v\}} & \text{if } s \in \{v \leftarrow \textbf{null}, v \leftarrow \textbf{new}, v \leftarrow e\} \\ \overline{V^\sharp} & \text{if } s \in \{e \leftarrow v, \textbf{body}\} \end{cases}$$

$$[\![s]\!]_{m^\sharp}(q, m^\sharp) \triangleq \left\{\left\langle q, \lambda f.[\![s]\!]_{d^\sharp}(m^\sharp(f))\right\rangle\right\}$$

$$[\![s]\!]_{\sigma^\sharp}(\sigma^\sharp) \triangleq \bigcup_{\langle q, m^\sharp\rangle \in \sigma^\sharp \cup \{\langle q_0, \lambda f.\top\rangle\}} [\![s]\!]_{m^\sharp}(q, m^\sharp)$$

Figure 10. Transfer functions for the tracematch state abstraction for $s \neq \textbf{tr}\,\langle a, b\rangle$.

costly approach is to consider that each variable v could point to any abstract object containing v, and to handle all possible combinations of variable values independently. We use a more precise approach that considers *compatibility* [34], the notion that some abstract objects cannot possibly correspond to concrete objects in the same execution. For example, the abstract environment may contain both $\{x\}$ and $\{x, y\}$ if the object pointed to by x is also pointed to by y in some but not all executions. However, at any given instant at run time, y cannot both point and not point to the object pointed to by x; thus, the two abstract objects are incompatible. The analysis therefore considers *reduced* environments, which are subsets of the abstract environment $\rho^\sharp$ satisfying the following constraints:

- The objects must all be *compatible* with each other, and with all objects in the tracematch state being updated.
- The objects must be *relevant*: each object must be pointed to by some variable in the transition statement.
- The subset must contain some object pointed to by each variable in the transition statement.

These constraints guarantee that each variable points to a unique abstract object, so every variable can be looked up in the reduced abstract environment. In addition, the constraints reduce the otherwise possibly exponential number of subsets of the abstract environment to a small number, usually only one. To be sound, the analysis considers all reduced environments satisfying the constraints.

Consider, for example, a transition statement binding x and y to two tracematch parameters. Suppose that the abstract environment contains abstract objects $\{x\}, \{y\}, \{x, y\}$ and $\{z\}$. The subsets $\{\{x\}, \{y\}\}$ and $\{\{x, y\}\}$ satisfy the constraints of the reduced environment. The subsets $\{\{x\}, \{x, y\}\}$ and $\{\{y\}, \{x, y\}\}$ are not compatible. The subset $\{\{x\}, \{y\}, \{z\}\}$ is compatible but not relevant since the transition statement does not bind z. The subset $\{\{x\}\}$ is not in the reduced environment because it does not contain any object pointed to by y.

It would be expensive to construct the reduced environment by considering all subsets of the abstract environment and selecting those that satisfy the constraints. Instead, we use Algorithm 1, which, by construction, only generates environments satisfying the constraints. The algorithm works as follows: at each step, it chooses some abstract object $o^\sharp$ to remove from the abstract environment $\rho^\sharp$, and calls itself recursively to construct all reduced environments not containing $o^\sharp$ and all reduced environments containing $o^\sharp$. The set of all reduced environments not containing $o^\sharp$ is simply the set of all reduced environments of the smaller abstract environment $\rho^\sharp \setminus \{o^\sharp\}$. A reduced environment can contain $o^\sharp$ only if $o^\sharp$ is relevant and compatible with other abstract objects in the environment. To check that $o^\sharp$ is relevant, the algorithm checks that $o^\sharp \cap \mathit{relevantVars}$ is non-empty. To ensure that $o^\sharp$ is compatible with other abstract objects in the environment, the algorithm uses a parameter called *forbiddenVars* to keep track of variables which already appear in some abstract object. When it calls itself recursively to construct the reduced environments to which $o^\sharp$ will be added, it adds all the variables in $o^\sharp$ to *forbiddenVars*. Thus, the abstract objects in the environments returned by the recursive call cannot contain any of the variables in $o^\sharp$, so they are compatible with $o^\sharp$. To each of the reduced environments returned by the recursive call, the algorithm adds $o^\sharp$, and the environments are then returned. In the base case, when $\rho^\sharp$ is empty, the algorithm returns either the empty environment if every relevant vari-

Algorithm 1: reducedEnvs($\rho^\sharp$, *relevantVars*, *forbiddenVars*)

Input: $\rho^\sharp$: abstract environment
Input: *relevantVars*: set of variables
Input: *forbiddenVars*: set of variables
Output: set of reduced environments (i.e. sets of abstract objects)

```
1  if ρ♯ ≠ ∅ then
2  |  choose any o♯ from ρ♯
3  |  r1 = reducedEnvs(ρ♯ \ {o♯}, relevantVars, forbiddenVars)
4  |  if (o♯ ∩ relevantVars ≠ ∅) ∧ (o♯ ∩ forbiddenVars = ∅) then
5  |  |  r2 = reducedEnvs( ρ♯ \ {o♯}, relevantVars \ o♯, forbiddenVars ∪ o♯)
6  |  |  r3 = {ρ'♯ ∪ {o♯} : ρ'♯ ∈ r2}
7  |  |  return r1 ∪ r3
8  |  else
9  |  |  return r1
10 |  end
11 else
12 |  if relevantVars = ∅ then return { {} }
13 |  else return {}
14 end
```

able has already been included in some abstract object, or no environments if some relevant variable remains.

Since Sagiv et al.'s notion of compatibility [34] is defined only for the precise object abstraction, we generalized it for the must-may abstraction. The generalized *compatible* predicate is formally defined in Figure 11. In order for two abstract objects to be compatible, they must either be abstractions of distinct concrete objects, or of the same concrete object. In the former case, the two must sets need to be disjoint. In the latter case, the must set of each abstract object needs to be a subset of the may set of the other. Before computing the reduced environments using Algorithm 1, we use the generalized compatibility predicate to remove from the abstract environment any abstract objects that are incompatible with an abstract object already bound in the tracematch state.

$$\begin{aligned} \mathit{same}(\langle o_1^!, o_1^? \rangle, \langle o_2^!, o_2^? \rangle) &\triangleq o_1^! \subseteq o_2^? \wedge o_2^! \subseteq o_1^? \\ \mathit{diff}(\langle o_1^!, o_1^? \rangle, \langle o_2^!, o_2^? \rangle) &\triangleq o_1^! \cap o_2^! = \emptyset \\ \mathit{compatible}(o_1^{!?}, o_2^{!?}) &\triangleq \mathit{same}(o_1^{!?}, o_2^{!?}) \vee \mathit{diff}(o_1^{!?}, o_2^{!?}) \end{aligned}$$

Figure 11. Generalized compatibility predicate.

The transfer function for transition statements is defined in Figure 12. At a high level, it mirrors the semantics of $\mathbf{tr}\,\langle a, b\rangle$ presented in Section 2. Having defined abstract variable lookup, the abstract tracematch transition functions $e_0^{+\sharp}, e_0^{-\sharp}, e^{+\sharp}, e^{-\sharp}, e^\sharp$ are exactly like their concrete counterparts, but with abstract lookup $\mathit{lookup}(O^\sharp, v)$ substituted for concrete lookup in ρ. The overall transfer function $[\![\mathbf{tr}\,\langle a, b\rangle]\!]_{m^\sharp}$ joins the results of $e^\sharp$ for all reduced abstract environments $O^\sharp \subseteq \rho^\sharp$. Finally, $[\![s]\!]_{\sigma^\sharp}$ extends $[\![s]\!]_{m^\sharp}$ to sets of abstract tracematch state pairs; it is the same as in Figure 10. At control flow merge points, the join operator used on sets of tracematch state pairs is set union.

We illustrate the effect of the tracematch state transfer function using the example statements shown in Figure 13. In this example, the must and may sets in every positive binding are the same, so we only show the set of variables once in each positive binding. Consider the tracematch from Figure 2 whose automaton was shown in Figure 1. The right side of Figure 13 shows some of the tracematch state computations performed when analyzing the statements in the left side of the figure. Once statement 3 has executed, $\rho^\sharp$ contains the abstract objects $\{l_1\}, \{l_2\}$ and $\{it_1\}$. The tracematch state $\sigma^\sharp$ contains the initial tracematch state pair $\langle 1, \{c \mapsto \top, i \mapsto \top\}\rangle$. The execution of statement 4 creates a state pair in state 2 with a positive binding for the two tracematch parameters. Two state pairs with negative bindings for the parameters are also created. Hence, after statement 4, the abstract tracematch state $\sigma^\sharp$ contains the 4 state pairs shown in the figure. Of the computations required to model statement 5, the figure shows only the computations needed to handle the state pair $\langle 2, \{c \mapsto \{l_1\}, i \mapsto \{it_1\}\}\rangle$. The positive binding of l_2 yields one state pair in state 3 which has a binding of $\bot$ for the tracematch parameter c. This represents an inconsistent binding and is discarded by the analysis. If $\rho^\sharp$ had contained an abstract object $\{l_1, l_2\}$, indicating that l_1 and l_2 may have been aliased, the transfer function would

$$
\begin{aligned}
\mathit{lookup}(O^\sharp, v) &\triangleq o^\sharp \in O^\sharp : v \in o^\sharp \\
e_0^{+\sharp}(b, O^\sharp) &\triangleq \lambda f. \begin{cases} \left\langle o^\sharp, o^\sharp \right\rangle \text{ where } o^\sharp = \mathit{lookup}(O^\sharp, b(f)) & \text{if } f \in \mathrm{dom}(b) \\ \top & \text{otherwise} \end{cases} \\
e^{+\sharp}[a, b, O^\sharp](q, m^\sharp) &\triangleq \left\{ \left\langle q', m^\sharp \sqcap e_0^{+\sharp}(b, O^\sharp) \right\rangle : \delta(q, a, q') \right\} \\
e_0^{-\sharp}(b, O^\sharp, f) &\triangleq \lambda f'. \begin{cases} \overline{\mathit{lookup}(O^\sharp, b(f))} & \text{if } f = f' \\ \top & \text{otherwise} \end{cases} \\
e^{-\sharp}[b, O^\sharp](q, m^\sharp) &\triangleq \left\{ \left\langle q, m^\sharp \sqcap e_0^{-\sharp}(b, O^\sharp, f) \right\rangle : f \in \mathrm{dom}(b) \right\} \\
e^\sharp[a, b, O^\sharp](q, m^\sharp) &\triangleq e^{+\sharp}[a, b, O^\sharp](q, m^\sharp) \cup e^{-\sharp}[b, O^\sharp](q, m^\sharp) \\
[\![\mathbf{tr}\ \langle a, b \rangle]\!][\rho^\sharp](q, m^\sharp) &\triangleq \bigcup_{O^\sharp \in \textit{reduced environments of } \rho^\sharp} e^\sharp[a, b, O^\sharp](q, m^\sharp)
\end{aligned}
$$

Figure 12. Transfer function for a transition statement $\mathbf{tr}\ \langle a, b \rangle$, which is applied to each pair $\left\langle q, m^\sharp \right\rangle$ in the tracematch state abstraction.

1 2 3	$l_1 \leftarrow \mathbf{new}$ $l_2 \leftarrow \mathbf{new}$ $it_1 \leftarrow \mathbf{new}$	
		$\sigma^\sharp = \{\langle 1, \{c \mapsto \top, i \mapsto \top\}\rangle\}$
4	$\mathbf{tr}\langle makeIter, \{c \mapsto l_1, i \mapsto it_1\}\rangle$	$O^\sharp = \{\{l_1\}, \{it_1\}\}$ $e_0^{+\sharp} = \{\ c \mapsto \{l_1\}, i \mapsto \{it_1\}\ \}$ $e^{+\sharp} = \langle 2, \{c \mapsto \top, i \mapsto \top\} \sqcap \{c \mapsto \{l_1\}, i \mapsto \{it_1\}\}\rangle$ $= \langle 2, \{c \mapsto \{l_1\}, i \mapsto \{it_1\}\}\rangle$ $e^{-\sharp} = \left\{ \left\langle 1, \{c \mapsto \overline{\{l_1\}}, i \mapsto \top\} \right\rangle, \left\langle 1, \{c \mapsto \top, i \mapsto \overline{\{it_1\}}\} \right\rangle \right\}$
		$\sigma^\sharp = \left\{ \langle 1, \{c \mapsto \top, i \mapsto \top\}\rangle, \left\langle 1, \{c \mapsto \overline{\{l_1\}}, i \mapsto \top\} \right\rangle, \left\langle 1, \{c \mapsto \top, i \mapsto \overline{\{it_1\}}\} \right\rangle, \langle 2, \{c \mapsto \{l_1\}, i \mapsto \{it_1\}\}\rangle \right\}$
5	$\mathbf{tr}\langle update, c \mapsto l_2\rangle$	$O^\sharp = \{\{l_2\}\}$ For tracematch state pair $\langle 2, \{c \mapsto \{l_1\}, i \mapsto \{it_1\}\}\rangle$ only: $e_0^{+\sharp} = \{c \mapsto \{l_2\}\}$ $e^{+\sharp} = \langle 3, \{c \mapsto \{l_1\}, i \mapsto \{it_1\}\} \sqcap \{c \mapsto \{l_2\}\}\rangle$ $= \langle 3, \{c \mapsto \bot, i \mapsto \{it_1\}\}\rangle$ $e^{-\sharp} = \left\langle 2, \{c \mapsto \{l_1\}, i \mapsto \{it_1\}\} \sqcap \{c \mapsto \overline{\{l_2\}}\} \right\rangle$ $= \langle 2, \{c \mapsto \{l_1\}, i \mapsto \{it_1\}\}\rangle$
		$\sigma^\sharp = \left\{ \langle 1, \{c \mapsto \top, i \mapsto \top\}\rangle, \left\langle 1, \{c \mapsto \overline{\{l_1, l_2\}}, i \mapsto \top\} \right\rangle, \left\langle 1, \{c \mapsto \overline{\{l_1\}}, i \mapsto \overline{\{it_1\}}\} \right\rangle, \left\langle 1, \{c \mapsto \overline{\{l_1\}}, i \mapsto \top\} \right\rangle, \langle 2, \{c \mapsto \{l_1\}, i \mapsto \{it_1\}\}\rangle, \langle 3, \{c \mapsto \bot, i \mapsto \{it_1\}\}\rangle \right\}$

Figure 13. Example illustrating tracematch state transfer function

have generated the state pair $\langle 3, \{c \mapsto \{l_1, l_2\}, i \mapsto \{it_1\}\}\rangle$, indicating a transition to state 3 with consistent bindings. The negative binding creates a state pair in state 2. The additional state pairs appearing in the final tracematch state $\sigma^\sharp$ arise from the other state pairs that were present before statement 5. The computation for these state pairs is not shown in the figure.

We have proven [30, Theorem 3] that the transfer function $[\![s]\!]_{\sigma^\sharp}$ preserves correctness. The correctness relation relating concrete and abstract binding lattice elements is defined as follows. An abstract state $\sigma^\sharp$ soundly approximates a concrete state σ if for every pair $\langle q, m\rangle$ in σ, there is a corresponding pair $\langle q, m^\sharp\rangle$ in $\sigma^\sharp$ that soundly approximates it. A pair $\langle q, m^\sharp\rangle$ soundly approximates $\langle q, m\rangle$ if for every tracematch parameter f, $m^\sharp(f)$ is higher in the binding lattice than the abstraction of $m(f)$ obtained by replacing each concrete object with the set of variables that point to it. Recall that a body statement completes a match only if the concrete state contains a pair $\langle q, m\rangle$ such that q is a final state and $m(f)$ is not $\perp$ for any f. The correctness relation ensures that if this happens, the abstract state $\sigma^\sharp$ must also contain a pair $\langle q, m^\sharp\rangle$ satisfying the same conditions. In the absence of such a pair in the abstract state, the analysis concludes that the body statement cannot complete a match.

4. Context-Sensitive Interprocedural Analysis

We implemented the analysis as an instance of the IFDS algorithm of Reps et al. [32] with some small modifications. The IFDS algorithm implements a fully context-sensitive interprocedural dataflow analysis provided that:

- the analysis domain is the powerset of a finite set **Dom**,
- the merge operator is union, and
- the transfer function is distributive.

IFDS is an efficient dynamic programming algorithm that uses $O(E|\mathbf{Dom}|^3)$ time in the worst case, where E is the number of control-flow edges in the program. The key reason for its efficiency is that it evaluates transfer functions on each individual element of **Dom** at a time, rather than on a subset of **Dom** at a time (recall that each element of the analysis domain is a subset of **Dom**). As a result, any distributive function $f : \mathcal{P}(\mathbf{Dom}) \to \mathcal{P}(\mathbf{Dom})$ can be efficiently represented as a graph with at most $(|\mathbf{Dom}| + 1)^2$ edges [32, Section 3]. The IFDS algorithm starts with a graph representation of the transfer function for each instruction in the program, and works by composing them into transfer functions for ever longer sequences of instructions. Specfically, it uses a typical worklist algorithm to complete two tables of transfer functions: the PathEdge table gives the transfer function from the start node of each procedure to every other node in the same procedure, and the SummaryEdge table gives the transfer function that summarizes the effect of each call site in the program.

To formulate the tracematch analysis as an IFDS problem, we must define the set **Dom** and the transfer functions on individual elements of **Dom**. This cannot be done for the overall flow function that computes both the object and tracematch abstractions because it is not distributive. This is because the tracematch state depends on abstractions of multiple objects, which could come from different control flow paths. Individually, however, each of the transfer functions for the object abstraction and for the tracematch state abstraction is distributive. Thus, we can first perform the object analysis as one instance of IFDS, then use the result to perform the tracematch state analysis as a second instance of IFDS. Moreover, the decomposition into transfer functions on individual elements of a finite set **Dom** comes naturally from the definition of the overall transfer functions. For the object abstraction, **Dom** is two copies of the set of all possible abstract objects, one copy to represent each of $\rho^\sharp$ and $h^\sharp$. Thus, the decomposed transfer function specifies the effect of an instruction on a single abstract object at a time. For the tracematch state abstraction, **Dom** is the set of all possible pairs $\langle q, m^\sharp\rangle$. Thus, the decomposed transfer function specifies the effect of an instruction on one pair at a time. The complete decomposed transfer functions and a proof of their equivalence to the overall transfer functions appears in the technical report [30].

In addition to the transfer functions, an instantiation of the IFDS algorithm must specify how to map elements of **Dom** between the caller and callee at a call site. Our mapping into the callee simply replaces actuals with formals, and removes all caller-side variables from both the object and tracematch state abstractions. In order to map objects from the callee back to the caller, we need to know which caller-side variables pointed to the object prior to the call. This information is readily available in the IFDS algorithm. Although [32] did not anticipate making this information available to the transfer function, it is straightforward to modify it to do so. The same modification is also used in the typestate analysis of Fink et al. [17, 18], and is likely to be useful in general in other IFDS analyses.

4.1 Collecting Useful Update Shadows

The analysis presented thus far can prove that the tracematch will never be in an accepting state at a given body statement. If this can be proved for all body statements in the program, the property expressed by the tracematch has been fully verified statically, and all dynamic instrumentation can be removed. However, the analysis may not be successful in ruling out *all* body statements. In this case, it is useful to compile a list of all transition statements that may contribute to a match at each body statement. In static verification, this list helps the user identify the source of the bug, or to decide that the error report is a false positive. For example, if a collection is updated during iteration, the body statement is the failing `next` call on the iterator; more useful to the programmer would be the location of the collection update. We are

currently developing an Eclipse plugin to present this information to the programmer. In optimizing the dynamic tracematch implementation, all transition statements not leading to a potentially matching body statement can be removed, thereby reducing the runtime overhead of matching.

The analysis can be extended to keep track of relevant transition statements by using the IDE [33] algorithm instead of IFDS. The IDE algorithm is an extension of IFDS to analysis domains of the form $\mathbf{Dom} \rightarrow L$, where **Dom** satisfies the same conditions as for IFDS and L is a lattice of finite height. Indeed, IFDS is a special case of IDE with L chosen as the two-point lattice $\bot \sqsubseteq \top$. The IFDS version of the tracematch analysis presented thus far determines only whether a given pair $\langle q, m^\sharp \rangle$ is $(\top)$ or is not $(\bot)$ present at each program point. To keep track of transition statements leading to a match, we keep the same set $\mathbf{Dom} = Q \times (F \rightarrow \mathbf{Bind}^\sharp)$, and define L to contain $\bot$ along with all subsets of the set of all transition statements. For each pair $\langle q, m^\sharp \rangle$ present at a program point, the IDE version of the analysis maintains the set of transition statements that may have contributed to its presence.

The IDE transfer functions are extensions of the IFDS transfer functions that we have already presented. The transfer functions are fomally defined in the technical report; we briefly summarize them here. The transfer function for every statement other than a transition statement keeps the set of relevant transition statements for each tracematch state pair unchanged. The transfer function for a transition statement adds the current transition statement to the set of relevant transition statements for each tracematch state pair. There is one exception: when the transition statement transforms a tracematch state pair $\langle q, m^\sharp \rangle$ to itself, the transition statement is not added to the set of relevant transition statements for that pair. A transition statement that does not change the *concrete* tracematch state at run time is not considered relevant because removing it would not change the program behaviour. Such a statement occurs when the tracematch regular expression contains a subexpression of the form a^*, which causes a self-loop in the finite automaton. We have proved that a transition statement that does not change the *abstract* tracematch state cannot change the *concrete* tracematch state [30, Proposition 7]. It is therefore sound to omit a transition statement that does not change the abstract tracematch state from the relevant transition statements.

It may happen that a transition statement in a loop changes the tracematch state in the first iteration but not in any subsequent iteration. An optimized dynamic implementation should execute the first, relevant transition, but should avoid executing the redundant transitions in subsequent iterations of the loop. This can be achieved by peeling one iteration of every loop containing a transition statement prior to performing the IDE analysis. The analysis will mark the transition as relevant in the peeled iteration and unnecessary in the remaining loop.

5. Empirical Evaluation

We empirically evaluated the precision of our analysis and compared it to Bodden et al.'s existing tracematch analysis [8], which uses may-point-to information to rule out possibly matching transition statements. The evaluation was performed on the tracematches from [8] plus one new one (FailSafeEnumHashtable), summarized below:

ASyncIteration: A synchronized collection should not be iterated over without owning its lock.

FailSafeEnum: A vector should not be updated while enumerating it.

FailSafeEnumHashtable: A hashtable should not be updated while enumerating its keys or values.

FailSafeIter: A collection should not be updated while iterating over it.

HasNext: The `hasNext` method should be called prior to every call to `next` on an iterator.

HasNextElem: The `hasNextElem` method should be called prior to every call to `nextElement` on an enumeration.

LeakingSync: A synchronized collection should only be accessed through its synchronized wrapper.

Reader: A `Reader` should not be used after its `InputStream` has been closed.

Writer: A `Writer` should not be used after its `OutputStream` has been closed.

We applied the above tracematches to the benchmarks antlr, bloat, hsqldb, luindex, jython, and pmd from the DaCapo benchmark suite, version 2006-10-MR2 [7]. Most of the benchmarks use reflection to load key classes. We instrumented the benchmarks using ProBe [28] and *J [15] to record actual uses of reflection at run time, and provided the resulting reflection summary to the static analysis. The jython benchmark generates code at run time which it then executes; for this benchmark, we made the unsound assumption that the generated code has no effect on aliasing or tracematch state.

Each of the 6 benchmarks was analyzed with each of the 9 tracematches, a total of 54 cases (tracematch/benchmark pairs). The 54 cases evaluated contained a total of 5409 final transition statements. We define a transition statement $\langle a, b \rangle$ as *final* if the tracematch automaton contains a transition to an accepting state on a. Thus, a match can be completed only at a final transition statement and implies a violation of the specified property. We count only final transition statements in the reachable part of the call graph. Of these, our analysis proved that 4815 (89 %) will never complete a match. Thus, a programmer wishing to check the tracematch properties need only examine 11 % of the uses of the features checked by the tracematches.

Bodden's analysis comprises three stages. The first stage (QC) considers only the set of tracematch symbols present in the program; if every word satisfying the tracematch pattern contains a given symbol and that symbol does not appear anywhere in the program, the tracematch cannot match and hence the safety property enforced, cannot be violated. The second stage (FI) considers the may-point-to sets of the variables in each transition statement. If a sequence of transitions is to lead to a violation, they must have consistent bindings, which is possible only if their points-to sets overlap. Bodden observed this stage to reduce the number of matching transition statements in seven of nine cases (tracematch/benchmark pairs); in one case, it completely eliminated all possibility of a match. The third stage (FS) considers the order in which symbols occur during execution, but does not coordinate this order with the flow of individual objects; Bodden observed no precision improvement over FI. Since our analysis subsumes QC and the precision of FI and FS is equivalent in practice, the evaluation in this section compares our analysis with FI.

Of the 54 cases, 36 actually used the features described by the tracematch, in the sense that QC did not rule out a match. These cases contained 1509 final transition statements, and our analysis proved that 915 will never complete a match and hence do not violate the tracematch property. Each of the 36 cases is represented by a circle in Table 1. Beside each circle is a fraction giving the number of transition statements at which a match could not be ruled out and the total number of final transition statements. In 15 of the 36 cases, our analysis ruled out all matches; i.e. it successfully verified that the benchmark is free of any violations of the property specified by the tracematch. These cases are represented by the 15 fully white circles. In comparison, the FI analysis ruled out all matches in only 1 of the 36 cases where QC was unsuccessful.

However, the two analyses are complementary in that they are successful on *different* transition statements. Our analysis fares better when the temporal order in which events occur is relevant in ruling out the match. When the feature monitored by the tracematch is used in many distinct ways in different parts of the program, like iterators, FI is sometimes better at distinguishing the different uses based on the allocation sites of the objects involved. More specific examples are discussed in the rest of this section. The two analyses can be run together, and the combination is more precise than each analysis on its own.

5.1 Discussion of Results

In this section we take a closer look at some of the results from Table 1. Of the 21 remaining cases in which all violations could not be removed, 4 involve the HasNext and HasNextElem tracematches. In one case (HasNext/pmd), all possible matches are actual violations of the tracematch pattern. The code uses `isEmpty` to ensure that a collection is not empty, then calls `next` on an iterator without calling `hasNext` first. Similar violations occur in the other three cases (in jython and in HasNext/bloat). In addition, these cases contain false positives due to iterators stored only in fields and not local variables. In the HasNext and HasNextElem tracematches, flow-sensitive tracking of individual objects is crucial to ensure that the `hasNext` call occurs on the same object as the calls to `next`. Thus, while our analysis ruled out matches at 441 of the 476 final transition statements, FI could not rule out a match at any of them.[2]

In 11 cases involving the FailSafe* tracematches, the analysis found both violations and likely false positives due to aliasing. Some collections, such as `java.util.Hashtable`, keep a singleton enumeration and iterator which are reused every time the collection is empty. This violates the tracematch because an iterator is being used even though a collection with which it was previously associated has since been updated. This accounts for many but not all of the detected matches; the associated transition statements are shown in gray in Table 1.

At many of the other transition statements, a match cannot be ruled out because a loop iterating over a collection contains calls leading to very deep call chains comprising many methods, some of which update collections. The analysis is not able to prove that all these collections are distinct from the collection being iterated. In some of these loops, may point-to information would help: FI ruled out matches at 19 transition statements in 3 cases that our analysis did not. On the other hand, our analysis ruled out matches at 54 transition statements in 2 cases that FI did not. Since so many methods are transitively called from the loop, it is difficult to examine them all by hand to determine whether any of the updated collections may in fact alias the iterated collection. We are working on a convenient user interface to visualize the potential update locations and the call chains connecting them to the original loop.

The cases involving the Reader and Writer tracematches can be classified into three categories. The first category includes readers/writers of files, which are closed after the last access. In these instances, our analysis proved all accesses occur before the close, thereby ruling out a violation. Since FI ignores the order of the events, it could not rule out a violation. The second category includes readers/writers of the standard input/output streams. These are never closed, and the FI analysis proves this fact, thus ruling out a match. These streams are often referenced only by their static field in the System class, and not by any local variables. Therefore, our analysis cannot distinguish them from other readers/writers on which close is called, and cannot rule out a match. The third category includes readers/writers for which neither analysis can rule out a violation. We noticed the following pattern in several benchmarks. A loop repeatedly

[2] Some transition statements were ruled out in [8] because they were determined to be in code that could not be reached at run time. Our evaluation considers only reachable code.

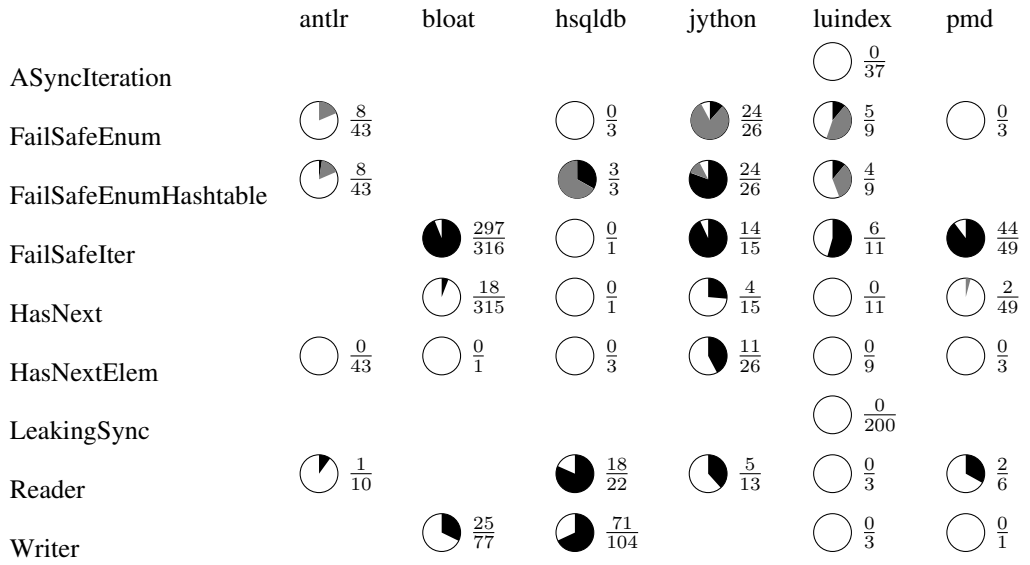

Table 1. Fraction of final transition statements that may complete a match. The white part of each circle represents those that cannot complete a match. The black part represents those at which a match cannot be ruled out, due either to analysis imprecision or an actual violation. The gray part represents those at which a violation is known to exist.

calls a helper method that uses the reader/writer. Both the loop and the helper method contain a try block. An exception during the input/output operation is caught in the helper, which closes the stream and re-throws the exception. The try block protecting the loop catches the exception, thereby terminating the loop and preventing any further use of the reader/writer. Because our analysis does not distinguish normal and exceptional returns, it conservatively assumes that the loop could continue iterating and therefore use the reader/writer after the stream was closed. Overall, our analysis proves three Reader/Writer cases correct compared to two for FI, but FI rules out slightly more final transition statements than our analysis.

In summary, although our analysis is often more precise than FI, the two are complementary in that each is more effective than the other on certain code patterns. In many practical cases, our analysis is precise enough to rule out a match. However, there remain cases where the abstraction looses all local variable references to an object. Thus, our analysis would benefit from some information about pointers from within the heap. We will investigate augmenting the abstraction with such information in future work.

6. Related Work

When tracematches were introduced, space and time overhead of their dynamic implementation was a concern [1]. In general, the overhead varied widely depending on the tracematch and the number of dynamic updates to the tracematch state that must be performed; in many cases, the overhead was prohibitive.

One approach to reduce the overhead has been to improve the dynamic tracematch implementation [4]. In this approach, the tracematch automaton (but not the base code to which it is applied) is analyzed statically to generate more efficient matching code. Specific attention has been paid to freeing bindings as soon as possible to reduce memory requirements and to detect statically when a tracematch may lead to unbounded space overhead. Freeing bindings early has the additional benefit of reducing the time required to find the binding requiring update when a transition statement is encountered. This time can be reduced further by maintaining suitable indexes on the binding set. On some realistic tracematches, these techniques yield speed improvements of multiple orders of magnitude. Thus, these techniques are necessary for a practical dynamic implementation of tracematches. A similar indexing technique is also applied in JavaMOP [10].

A second approach, of which our work is an example, is to use static analysis to reduce the number of transition statements that must be instrumented. Another example is the work of Bodden et al. [8], which we discussed in Section 5. In follow-on work to be presented at SIGSOFT/FSE 2008, Bodden et al. [9] have augmented the analysis with a suite of intraprocedural flow-sensitive analyses. The analyses combine local alias information with inexpensive whole program summary information. In their benchmark suite, tracematches described mostly local patterns, thus a careful combination of these analyses could detect many violations and with few false positives. Guyer and Lin's [21, 22] client-driven pointer analysis is also related. Their analysis is based on a subset-based may-point-to analysis followed

by flow-sensitive propagation of states on the abstract object represented by each allocation site. When a property cannot be proven, the analysis iteratively refines the context-sensitivity of the points-to analysis in order to improve precision and hopefully verify the property. Dwyer and Purandare [16] also use static analysis to reduce the cost of dynamic typestate verification by proving that certain transitions need not be instrumented because they cannot lead to a violation.

The static analysis most closely related to our analysis is Fink et al.'s typestate analysis [17, 18]. Their analysis also uses an object abstraction in which an abstract object represents at most one concrete object, and it uses the focus operation to achieve this. Their object abstraction is more precise but more costly than ours because it tracks access paths through fields, rather than only references from local variables. In addition, the object abstraction contains the allocation site of each object, which provides the same information as a subset-based may-point-to analysis. It would be possible to replace the object abstraction in our tracematch analysis with that of Fink et al. to improve precision. Unlike tracematches, typestate applies only to a single object. Therefore, rather than requiring a separate tracematch abstraction, Fink et al. simply augment the abstraction of each object with its typestate.

Another object abstraction similar to ours is used by Cherem and Rugina [11] to statically insert free instructions to deallocate some objects earlier than the garbage collector can get to them. This application makes use of the property that the abstract object corresponding to a given concrete object can be traced through the control flow graph. The object abstraction is also more precise than ours, but less so than Fink's; it maintains reference counts from individual fields rather than full access paths. This object abstraction could also be substituted in the tracematch analysis.

Multiobject temporal constraints have been studied by Jaspan and Aldrich [25, 26] in the context of plugins for object-oriented frameworks. The motivation for their work is that since large frameworks introduce complex constraints that are difficult to understand and document, it is difficult for programmers to develop plugins which conform to the constraints laid down by the framework. They present a lightweight specification system which allows the framework developers to specify runtime interactions between objects and the framework constraints that depend on these interactions. A static analysis is presented that uses these specifications and analyzes the plugin code for any violations of the constraints laid down by the framework developer. The analysis has been shown to work on real world examples from the ASP.NET and Eclipse framework.

Ramalingam et al. [31] present a verification technique for checking that a client program follows conventions required by an API. The effects and requirements of the API methods are specified using a declarative language. The system constructs a predicate abstraction of the API internals from the specification. The predicate abstraction is used to prove that a client program satisfies the requirements. The system was used to check correct usage of iterators in client programs of up to 2396 LOC.

The Metal system [24] is an unsound state-based bug finder for C. The core system does not consider aliasing; instead an automaton is maintained for each variable, regardless of the object to which it may be pointing. It uses heuristics such as *synonyms* (an unsound variation of must-alias analysis) to partially recover from this unsoundness. Metal was successful in finding many locking bugs in the Linux kernel.

An alternative to analyzing arbitrary aliasing is to use a specialized type system to restrict aliasing. An advantage of this approach is modularity: a violation of the type system is local, as are violations of the typestate property when the aliasing restrictions are obeyed. A disadvantage is that it is difficult to apply to existing, unannotated code, although sometimes annotations can be inferred automatically. The Vault system [12] uses *keys*, unique pointers to objects. The type system prevents duplication of keys, and each typestate change is correlated with a set of keys held at the point of the change. The same authors propose a system for specifying typestates of object-oriented programs, focusing especially on object-oriented features such as subtyping [13]. To handle aliasing, they allow objects to be either unaliased and updateable, or possibly aliased and non-updateable. CQual [19] is another system similar to but simpler than Vault. Bierhoff and Aldrich [5, 6] present a type system in which both aliasing and typestate information are specified using types. A key innovation of their system are *access permissions*, which specify whether a pointer is unique or whether it is aliased but with fine-grained restrictions on which aliases may read or write to the object. Access permissions can be split for multiple aliases and later recombined, making them more flexible than earlier aliasing control mechanisms.

7. Conclusions and Future Work

We have presented a static analysis of temporal specifications of multiple interacting objects expressed using tracematches. The analysis has been proven sound with respect to the tracematch semantics. A fully context-sensitive version of the analysis has been implemented as two instances of IFDS [32] and IDE [33] algorithms. The analysis was evaluated on the tracematches of Bodden et al. [8] and found to be very precise. The analysis ruled out the possibility of a violation at 89% of the final transition statements in the benchmarks that we evaluated. Thus, a programmer wishing to check the properties specified by the tracematches would have to examine only the remaining 11% of final transition statements. Of the 36 tracematch/benchmark pairs in which the benchmark used features checked by the tracematch, the analysis fully verified 15 to contain no possible violations.

Remaining imprecisions are mainly due to two factors. First, the analysis loses precision when all local variable references to an object are lost. This can be remedied either by making use of may-point-to information, or by adding more precise information about heap references to the object abstraction. Even type information may help in some cases. Second, the analysis fails to verify some tracematches due to imprecise handling of interprocedural exceptional control flow. The precision of exceptional control flow can be improved with suitable modifications to the IFDS and IDE algorithms. We plan to investigate these improvements in the future.

To make the analysis useful to programmers, and to ease our work of interpreting the results, we are developing an Eclipse plugin for presenting the analysis results and for navigating the call graph and control flow graph of the program. A screenshot from the current prototype of the plugin analyzing the example from Figures 1 and 2 is shown in Figure 14. The top view shows the code being analyzed in the Eclipse source editor and the bottom view shows the results of the tracematch analysis. In this example, the analysis has found one possible final transition statement, in line 23, that could violate the property. For this final transition statement, the view also displays the update shadows that could have led to the match, collected through the IDE analysis discussed in Section 4.1. A programmer investigating the violation in line 23 would examine the update transitions to find that the update in line 24 modified the list being iterated, leading to a violation the next time line 23 is executed.

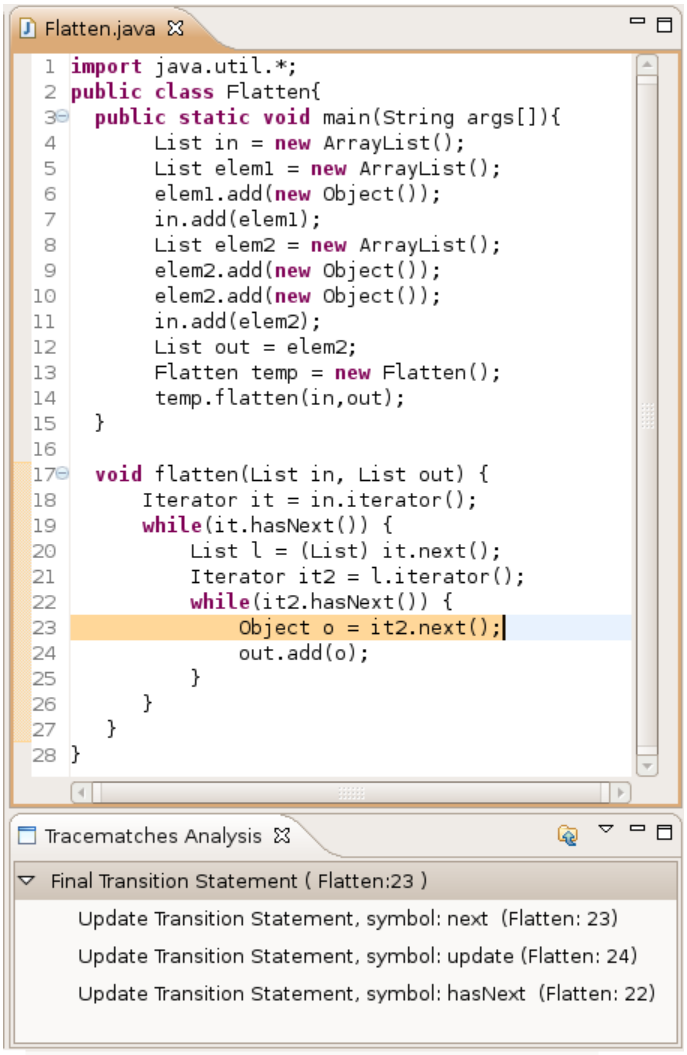

Figure 14. Screenshot of Eclipse Plugin Prototype

Acknowledgments

We are grateful to Matt Negulescu and Robert Burke for developing the Eclipse plugin prototype. We have had fruitful discussions about static verification of multi-object temporal properties with Jonathan Aldrich, Eric Bodden, Laurie Hendren, Ciera Jaspan, and Patrick Lam. We thank the abc team for maintaining the dynamic tracematch implementation in the abc compiler. Finally, we thank the anonymous reviewers for their useful remarks and suggestions for the paper. This research was supported by the Natural Sciences and Engineering Research Council of Canada.

References

[1] C. Allan, P. Avgustinov, A. 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. *OOPSLA '05: Proceedings of the 20th ACM SIGPLAN Conference on Object Oriented Programming Systems and Applications*, pages 345–364, 2005.

[2] P. Avgustinov, A. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. *abc* : An extensible AspectJ compiler. *Transactions on Aspect-Oriented Software Development I*, pages 293–334. 2006.

[3] P. Avgustinov, E. Hajiyev, N. Ongkingco, O. de Moor, D. Sereni, J. Tibble, and M. Verbaere. Semantics of static pointcuts in AspectJ. *POPL '07: Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 11–23, 2007.

[4] P. Avgustinov, J. Tibble, and O. de Moor. Making trace monitors feasible. *OOPSLA '07: Proceedings of the 22nd ACM SIGPLAN Conference on Object Oriented Programming Systems and Applications*, pages 589–608, 2007.

[5] K. Bierhoff and J. Aldrich. Lightweight object specification with typestates. *ESEC/FSE 2005: Proceedings of the Joint 10th European Software Engineering Conference (ESEC) and the 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-13)*, pages 217–226, 2005.

[6] K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. *OOPSLA '07: Proceedings of the 22nd ACM SIGPLAN Conference on Object Oriented Programming Systems and Applications*, pages 301–320, 2007.

[7] S. Blackburn, R. Garner, C. Hoffman, A. Khan, K. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, E. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. *OOPSLA '06: Proceedings of the 21st ACM SIGPLAN Conference on Object Oriented Programming Systems and Applications*, 2006.

[8] E. Bodden, L. Hendren, and O. Lhoták. A staged static program analysis to improve the performance of runtime monitoring. *ECOOP '07: Proceedings of the 21st European Conference on Object-Oriented Programming*, pages 525–549, 2007.

[9] E. Bodden, P. Lam, and L. Hendren. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. *FSE 2008: ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, Nov. 2008.

[10] F. Chen and G. Roşu. Mop: an efficient and generic runtime verification framework. *OOPSLA '07: Proceedings of the 22nd ACM SIGPLAN Conference on Object Oriented Programming Systems and Applications*, pages 569–588, 2007.

[11] S. Cherem and R. Rugina. Compile-time deallocation of individual objects. *ISMM '06: Proceedings of the 2006 International Symposium on Memory Management*, pages 138–149, 2006.

[12] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. *PLDI '01: Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 59–69, 2001.

[13] R. DeLine and M. Fähndrich. Typestates for objects. *ECOOP '04: Proceedings of the 18th European Conference on Object-Oriented Programming*, pages 465–490, 2004.

[14] A. Deutsch. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. *ICCL '92: Proceedings of the 4th IEEE International Conference on Computer Languages*, pages 2–13, 1992.

[15] B. Dufour. Objective quantification of program behaviour using dynamic metrics. Master's thesis, McGill University, 2004.

[16] M. B. Dwyer and R. Purandare. Residual dynamic typestate analysis: exploiting static analysis results to reformulate and reduce the cost of dynamic analysis. *ASE '07: Proceedings of the twenty-second IEEE/ACM International Conference on Automated Software Engineering*, pages 124–133, New York, NY, USA, 2007. ACM.

[17] S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. *ISSTA'06: Proceedings of the International Symposium on Software Testing and Analysis*, pages 133–144, 2006.

[18] S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.*, 17(2):1–34, 2008.

[19] J. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. *PLDI '02: Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 2002.

[20] S. Goldsmith, R. O'Callahan, and A. Aiken. Relational queries over program traces. *OOPSLA '05: Proceedings of the 20th ACM SIGPLAN Conference on Object Oriented Programming Systems and Applications*, pages 385–402, 2005.

[21] S. Guyer and C. Lin. Client-driven pointer analysis. *SAS '03: Proceedings of the 10th Annual International Static Analysis Symposium*, pages 214–236, 2003.

[22] S. Guyer and C. Lin. Error checking with client-driven pointer analysis. *Sci. Comput. Program.*, 58(1-2):83–114, 2005.

[23] B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 310–323, 2005.

[24] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. *PLDI '02: Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 69–82, 2002.

[25] C. Jaspan and J. Aldrich. Checking semantic usage of frameworks. *Library Centric Software Design Symposium*, 2007.

[26] C. Jaspan and J. Aldrich. Checking temporal relations between multiple objects. Technical Report CMU-ISR-08-119, Carnegie Mellon University, Dec. 2008.

[27] H. Jonkers. Abstract storage structures. de Bakker and van Vliet, editors, *Algorithmic Languages*, pages 321–343, 1981.

[28] O. Lhoták. Comparing call graphs. *PASTE '07: Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 37–42, 2007.

[29] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. *OOPSLA '05: Proceedings of the 20th ACM SIGPLAN Conference on Object Oriented Programming Systems and Applications*, pages 365–383, 2005.

[30] N. Naeem and O. Lhoták. Extending typestate analysis to multiple interacting objects. Technical Report CS-2008-04, University of Waterloo, 2008. `http://www.cs.uwaterloo.ca/research/tr/2008/CS-2008-04.pdf`.

[31] G. Ramalingam, A. Warshavsky, J. Field, D. Goyal, and M. Sagiv. Deriving specialized program analyses for certifying component-client conformance. *PLDI '02: Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 83–94, 2002.

[32] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61, 1995.

[33] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167(1–2):131–170, 1996.

[34] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM TOPLAS*, 20(1):1–50, Jan. 1998.

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