Efficient runtime monitoring through static analysis

Double-blinded submission

Abstract

Runtime monitors observe a sequence of events occurring during the execution of a program. They have been frequently used for the purpose of runtime verification. A compiler translates an initial requirements specification into an equivalent monitor implementation and instruments the program under test so that it emits the necessary events to this monitor at runtime. An inherent problem with this approach is that this can significantly slow down the program if many such instrumentation points have to be inserted.

In this work we concentrate on tracematches, a specification formalism based on regular expressions which induces finite-state monitors. We optimize each tracematch by statically analyzing the program to decide what parts of it are unsafe, i.e. can possibly violate the specification once it runs. For parts that prove safe, the optimization removes the related instrumentation points so that they are not evaluated at runtime. As proof of concept, we expose an implementation on top of the AspectBench Compiler

Our results show that, in our benchmark set, for some tracematches and programs this optimization can remove all of the instrumentation points, on average it removed more than two thirds. The instrumented programs are usually much more responsive when optimized. Also the analysis itself is efficient, requiring only seconds to run on sizable benchmarks in addition to a general points-to analysis that is required for any such kind of analysis.

As this work shows, runtime monitors can be effectively and efficiently specialized. We believe that this is one important step towards getting runtime monitors ready for widespread use in large-scale applications, leading to safer applications in general.

Categories and Subject Descriptors TODO [*TODO*]: TODO—TODO

General Terms TODO

Keywords Points-to analysis, runtime verification, model checking, compiler optimization, aspect-oriented programming

1. Introduction

A *runtime monitor* observes a sequence of events occurring during the execution of a piece of software. They have been frequently used for the purpose of Runtime Verification [**?**]. In the beginning of the verification process stands a formal, often declarative, specification based on regular expressions, temporal logics, or context free grammars. An automatic process then generates a monitor equivalent to this specification and alters the program under test so that it emits the necessary events to this monitor at runtime.

In 2005, Avgustinov et al. [1] propsed an implementation of such an approach under the name of *tracematches*. Figure 1 gives an example adapted from their paper.

Explain what this tracematch consists of and what it does. Explain that there is overhead involved because so many events have to be captured in the program. Also mention that it binds free variables.

```
public aspect FailSafeIter {

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

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

    create_iter call_next * update_source + call_next
   {
        throw new ConcurrentModificationException ();
   }
    }
}
```

Figure 1. Safe iteration tracematch

Although such trace monitors are certainly easy to use due to their declarative specification, the compiler has to do a good job to compiler the related runtime monitor efficiently. This problem is common to all runtime verification approaches. In [?], Avgustinov et al. compare different runtime monitoring systems and compare their performance. The authors further explain certain optimizations applied to the generated monitor, resulting in faster monitors compared to the other approaches. Those optimizations specialize the monitor with respect to the given specification.

The purpose of this work is now to specialize the monitor also with respect to the program being monitored. We propose an set of algorithms that analyze the program under test in order to restrict the set of potential points of failure for a runtime monitor to a feasible minimum.

Figure 2 gives our running example. We use it to show what our analysis is trying to achive. The example simulates a drawing package which has a canvas and shapes and some user interaction with it. First we have a class Canvas that implements some canvas that a collection of shapes. It exposes methods to add a shape, to read a set of shapes from an input stream and to animate all shapes on the canvas.

The class UI simulates some user interaction with the program. First a canvas is created. The user adds a rectangle to it. Then

PLDI '07, XXX
Copyright © 2007 ACM X-XXXXX-XXX-XXXXX...$5.00

the user adds shapes from a file. He clicks the animation button triggering the animation. Then he loads some other shapes, while the animation is running.

To this example we now apply the aforementioned tracematch from Figure 1 as well as the one from Figure 3, which checks that nothing is read from a closed stream.

When taking those two monitor specifications into consideration, we can identify the following potential points of failures: We have a call to Iterator .next() in line 30. This could potentially fail if the collection Shapes was modified since the last call to next. This might in fact be the case in our example, because the user adds shapes from a file while the animation is running. Consequently, the associated instrumentation points at the lines 11, 29 and 30 have to remain. The tracematch will actually be woven at those positions and an error message will be issued at runtime at the appropriate time. The shadow at line 5 can be removed, because it is only called from line 42, which takes certainly place before the shapes are iterated over.

For the input stream property, thinks look better: The only two reads that occur do so before the respective stream is closed. Furthermore, the stream is not affected by the animation thread. So here, the analysis should infer, that the tracematch from figure 3 can be removed entirely.

Internal remark: We can later on use lines 43 and 49 to describe that if we had reused the variable is1 in line 49, then the points-to info would have had represented the two streams as one abstract object (since it is flow-insensitive), which would have led to the optimization failing for this example.

1.1 Constributions

This paper provides the following original contributions:

- A set of algorithms to efficiently compute unnecessary instrumentation points for tracematches. Those algorithms are generic enought to be applied to other finite-state runtime verification systems.
- A set of sizeable (?) benchmarks in order to evaluate theapplicability of our approach and its precision.
- More?

The remainder of this paper is organized as follows: Section 2 gives an overview of our optimization. It enumerates all analysis steps we apply and how thos esteps interact. Section 3 gives the concrete dynamic semantics of tracematches. This is necessary in order to be able to reason about the correctness of our abstract semantics. Those are presented in Section 5. Here we explain the exact nature of our abstraction and how it is obtained. Section 6 then describes in detail how a given tracematch is evaluated over this abstraction in order to find the set of *active shadows*. This concludes our discussion of the analysis. We give benchmarks results in Section 7, evaluating on the performance of the different steps of our analysis. We close with a discussion of related work (Section 8) as well as our results and future work (Section 9).

2. Overview

Here we just outline the analysis. First name our requirements: Want to handle free variables, threads and recursion. Explain that we have two major steps. First we create a sound abstraction of the program which captures the abstract transition behaviour with respect to all given tracematches in form of a finite state machine. Then secondly, we evaluate the tracematch automaton over this state machine using a worklist algorithm.

Further outline step 1 (with a graphic?): We first we create a per-method state machine based on an exceptional unit graph.

```
 1 class Canvas {
 2
 3    Collection shapes = new ArrayList();
 4
 5    void add(Shape s) { shapes.add(s); }
 6
 7    void insertFromFile (InputStream is) {
 8        while(true) {
 9            Shape shape = readShape(is);
10            if (shape==null) return;
11            shapes.add(shape);
12        }
13    }
14
15    private Shape readShape(InputStream is) {
16        Shape shape = null;
17        do{
18            int i = is.read();
19            /* construct shape from content omitted */
20        }
21        return shape;
22    }
23
24    void animate() { new Animator().start(); }
25
26    class Animator extends Thread {
27
28        public void run() {
29            for ( Iterator iter = shapes.iterator(); iter.hasNext();) {
30                Shape s = (Shape) iter.next();
31                s.translateBy(generateOffset());
32            }
33        }
34
35        int generateOffset() { /* omitted */ }
36    }
37 }
38
39 class UI {
40     static void userInteraction() {
41        Canvas c = new Canvas();
42        c.add(new Rectangle());
43        InputStream is1 = new FileInputStream("myShapes");
44        c.insertFromFile(is1);
45        is1.close();
46
47        c.animate();
48
49        InputStream is2 = new FileInputStream("myOtherShapes");
50        c.insertFromFile(is2);
51    }
52 }
```

Figure 2. Running example of a UI drawing package

```
public aspect StreamClose {

  tracematch(InputStream is ) {

      sym close  after  :  call (* InputStream . close ())  && target(is );

       sym read  before :  call (* InputStream . read  (..))   && target(is );

       close  read  {
          throw new RuntimeException(
             "Tried to read from closed stream " + is  );
       }

   }

}
```

Figure 3. Tracematch checking for reads from closed streams

Then we combine those automata to thread summaries and in the end to one large state machine for the whole program.

3. Concrete semantics

The concrete semantics are basically defined in three steps (see three subsections). First we state our general assumptions and prerequisites. Then we define what the underlying NFA for a tracematch looks like: Essentially a normal NFA but with additional skip-edges. Note that here, as I proposed weeks ago, skip edges have labels! We assume (that will be mentioned eralier in the paper), that a state s has a skip-loop **skip**-l, if there is not already a loop (s, l, s) in the transition relation Δ. In the last step, we define the actual tracematch automaton, i.e. the machinery that is executed at runtime - on top of that given NFA.

3.1 Program representation

In the following, we assume the following sets as given. They are induced by the specification of the given tracematches.

We start off by stating some basic assumptions, such as that we have a given set of tracematch symbols with associated variable names plus a potentially infinite set of objects.

Sym	finite set of fully qualified tracematch-symbol names
Var_s	finite set of variable names for each $s \in Sym$
Var	defined by $Var := \bigcup_{s \in Sym} Var_s$
O	possibly infinite set of objects

Then we define the underlying NFA for a tracematch automaton. It is almost a usual NFA, with the only real difference, that its set of input labels can be splitted into labels l and **skip**-l. Also, the transition relation can be partitioned in the same way into skip transitions and "normal" transitions.

3.2 Underlying NFA

Definition 1 (Nondeterministic finite automaton, NFA) *We define a nondeterministic finite automaton $\mathcal{A}$ in the usual way as a quintuple $\mathcal{A} \in (Q^{\mathcal{A}}, \Sigma^{\mathcal{A}}, Q_0^{\mathcal{A}}, \Delta^{\mathcal{A}}, Q_F^{\mathcal{A}})$ with*

- $Q^{\mathcal{A}}$ *finite set of states*
- $\Sigma^{\mathcal{A}}$ *finite set of input symbols*
- $Q_0^{\mathcal{A}} \subseteq Q^{\mathcal{A}}$ *set of initial states*
- $\Delta^{\mathcal{A}} \subseteq Q^{\mathcal{A}} \times \Sigma^{\mathcal{A}} \times Q^{\mathcal{A}}$ *transition relation*
- $Q_F^{\mathcal{A}} \subseteq Q^{\mathcal{A}}$ *set of final states*

As usual, an input word $w \in \Sigma^*$ is accepted by $\mathcal{A}$ if $\mathcal{A}$ is in a final state (from $Q_F^{\mathcal{A}}$) after reading w letter by letter through the transition relation $\Delta^{\mathcal{A}}$, starting in any initial state $q_0 \in Q_0^{\mathcal{A}}$.

For the NFAs in this paper, we further assume that Σ is partitioned in two sets, $\Sigma = \Sigma_+ \cup \Sigma_-$ with the following property:

$$a \in \Sigma_+ \Leftrightarrow \textbf{skip-}a \in \Sigma_-$$

This also induces a partitioning of $\Delta = \Delta_+ \cup \Delta_-$ as:

$$\Delta_+ := \{(s, l, t) \in \Delta \mid l \in \Sigma_+\}$$

$$\Delta_- := \{(s, \textbf{skip-}l, t) \in \Delta \mid \textbf{skip-}l \in \Sigma_-\}$$

3.3 Definition of the tracematch automaton

Definition 2 (Powerset) *For each set s, we define its powerset as $\mathcal{P}(s)$.*

Definition 3 (Disjunct) *A disjunct d is a pair of positive and negative binding functions (d_+, d_-) with $d_+ : Var \rightarrow O$ and $d_- : Var \rightarrow \mathcal{P}(O)$.*

Note that the negative binding maps a variable to a *set* of objects. We denote the set of all disjuncts by $\mathcal{D}$.

Definition 4 (Constraint) *Any element of $\mathcal{P}(\mathcal{D})$ is a constraint.*

We denote the set of all constraints by $\mathcal{C}$. The constraint $\emptyset$ is also called $\mathbf{false}$ and the constraint $\{(\emptyset, \emptyset)\}$ is also called $\mathbf{true}$ (in line with the well-known definition of disjunctive normal form).

We model a configuration (i.e. complete state of a tracematch automaton) as a mapping from states to constraints.

Definition 5 (Configuration) *A configuration $\gamma(Q)$ with respect to a finite state set Q is a mapping from the states of Q to constraints. Hence, the set of all configurations over Q is defined as*

$$\Gamma(Q) := \{\gamma \mid \gamma : Q \rightarrow C\}$$

In the following we will often write Γ instead of $\Gamma(Q)$ if Q is clear from the context.

Definition 6 (Initial configuration) *For any set of configurations $\Gamma(Q)$, over a state set Q with subset of initial states $Q_0 \subseteq Q$, we define the initial configuration $\gamma_0 \in \Gamma(Q)$ as follows:*

$$\forall q \in Q \, . \, \gamma_0(q) := \begin{cases} \mathbf{true} & \textit{if } q \in Q_0 \\ \mathbf{false} & \textit{if } q \notin Q_0 \end{cases}$$

Note that here $\mathbf{true}$ and $\mathbf{false}$ are the corresponding constraints as defined above.

Definition 7 (Domain) *For any function φ, we define $dom(\varphi)$ as the domain of φ.*

The join operation just combines two configurations disjointly.

Definition 8 (Join of configurations) *We define a join operation $\oplus : \Gamma \times \Gamma \to \Gamma$ which combines two configurations disjointly. For $\gamma_1, \gamma_2 \in \Gamma$ with $dom(\gamma_1) = dom(\gamma_2)$ we define:*

$$\forall c \in dom(\gamma_1) \,.\, (\gamma_1 \oplus \gamma_2)(c) := \gamma_1(c) \cup \gamma_2(c)$$

The satisfying disjuncts are all disjuncts which are contained in a constraint at a final state… (but not **false**)

Definition 9 (Satisfying disjunct) *We identify the satisfying disjuncts of a configuration as the non-***false** *disjuncts at a final state.*

$$sat(\gamma) := \{d \in \mathcal{D} \mid \exists q \in Q_F \,.\, d \in \gamma(q)\} \backslash \{\textbf{false}\}$$

A configuration is accepting if there exists at least one satisfying disjunct in that configuration.

Definition 10 (Accepting configuration) *We define a configuration $\gamma \in \Gamma$ as accepting if it has any satisfying binding:*

$$\Gamma_F = \{\gamma \in \Gamma \mid sat(\gamma) \neq \emptyset\}$$

We still have to define what the input to the tracematch automaton is. In our notation here, a tracematch automaton reads "events". An event is basically the information we get from each matching joinpoint: A set of pairs (s, β) where s is a symbol name and β a variable binding…

Definition 11 (Binding and Event) *A binding β is a partial function of type $\beta : Var \rightharpoonup O$. We denote the set of all bindings by*

$$\mathcal{B} := \{\beta \mid \beta : Var \rightharpoonup O\}$$

An event e is an then element of $\mathcal{P}(Sym \times \mathcal{B})$, i.e. a set of symbol-labels, each associated with a variable binding. We denote the set of all events by $\mathcal{E}$.

A tracematch is now defined as follows. Its states are configurations. It reads events. It starts in the initial configuration. The transition function will be described later. Its accepting configurations are exactly those defined by Γ_F.

Definition 12 (Tracematch automaton) *A tracematch automaton $\mathcal{T}(\mathcal{A})$ is a deterministic automaton augmenting a given NFA $\mathcal{A}$. It is defined as a quintuple $\mathcal{T}(\mathcal{A}) := (\Gamma, \mathcal{E}, \gamma_0, \delta, \Gamma_F)$ where $\Gamma := \Gamma(Q^{\mathcal{A}})$ and δ as will be described in Definition 14.*

In particular this definition implies that, since Γ is a potentially infinite set of configurations, this automaton has a potentially infinite set of states. Yet, acceptance for $\mathcal{T}$ is defined as usual: An event-trace $t \in \mathcal{E}^*$ is accepted by $\mathcal{T}$ if after reading t, $\mathcal{T}$ is in an accepting configuration $\gamma \in \Gamma_F$, i.e. has a satisfying binding. When this is the case, the tracematch body is executed for any such binding $b \in sat(\gamma)$.

In the following we will often write $\mathcal{T}$ instead of $\mathcal{T}(\mathcal{A})$ if $\mathcal{A}$ is clear from the context.

Before we define the transition function δ, we first introduce some more notation…

We need to be able to update a binding. Hence, we define a functional update, replacing a mapping $s \mapsto t$ by $s \mapsto r$.

Definition 13 (Function update) *Further, for all $s \in dom(\varphi)$ we define an update operator $[s \mapsto t/s \mapsto r]$ (in postfix notation) as follows:*

$$\varphi[s \mapsto t/s \mapsto r](a) := \begin{cases} r & \textit{if } a = s \\ \varphi(a) & \textit{otherwise} \end{cases}$$

If before the update operation $s \notin dom(\varphi)$, we also simply write $\varphi[s \mapsto r]$ instead of $\varphi[s \mapsto t/s \mapsto r]$.

Definition 14 (Transition function δ)

The transition function δ is of type:

$$\delta : \Gamma \times \mathcal{E} \to \Gamma$$

It is defined as:

We compute the successor configuration by iterating over all transitions of $\mathcal{A}$. We do so separately for **skip**- and "normal" transitions, then in the end join the results. In the following,*underlined* parts of the definitions are new functions which are the defined right below.

$$\begin{aligned} & \delta(\gamma, e) && (1)\\ := \; & \delta(\gamma, e, \Delta^{\mathcal{A}}) && (2)\\ := \; & \underline{\delta(\gamma_0, e, \Delta^{\mathcal{A}}_{+})} \oplus \underline{\delta(\gamma, e, \Delta^{\mathcal{A}}_{-})} && (3)\\ & && (4) \end{aligned}$$

with $\delta(\gamma, e, \Delta)$ defined as:

For all transitions in Δ…

$$\begin{aligned} & \delta(\gamma, e, \Delta) && (5)\\ = \; & \delta(\gamma, e, \{(s_1, l_1, t_1), (s_2, l_2, t_2), \ldots, (s_n, l_n, t_n)\}) && (6)\\ := \; & \delta(\underline{\delta(\gamma, e, (s_1, l_1, t_1))}, e, \{(s_2, l_2, t_2), \ldots, (s_n, l_n, t_n)\}) && (7)\\ & && (8) \end{aligned}$$

with $\delta(\gamma, e, (s, l, t))$ defined as:

We introduce an accumulating parameter γ_a, which is initialized to γ_0, i.e. $(true, false, \ldots)$.

$$\delta(\gamma, e, (s,l,t)) := \delta(\gamma, \gamma_0, e, (s,l,t))$$

with $\delta(\gamma, \gamma_a, e, (s,l,t))$ defined as:

For pairs (s,β) in the event e…

$$\begin{aligned} & \delta(\gamma, \gamma_a, e, (s,l,t)) && (9)\\ = \; & \delta(\gamma, \gamma_a, \{(l_1,\beta_1),(l_2,\beta_2),\ldots,(l_m,\beta_m)\}, (s,l,t)) && (10)\\ := \; & \text{let } \gamma_1 := \underline{\delta(\gamma, \gamma_a, (l_1,\beta_1), (s,l,t))} && (11)\\ & \text{in } \delta(\gamma_1, \gamma_1, \{(l_2,\beta_2),\ldots,(l_m,\beta_m)\}, (s,l,t)) && (12) \end{aligned}$$

with $\delta(\gamma, \gamma_a, (l_e,\beta), (s,l,t))$ defined as:

If we process a "normal" edge with the right label, do update positively, if we see a skip edge with that label, update negatively, else do nothing.

$$\begin{aligned} & \delta(\gamma, (l_e,\beta), (s,l,t)) && (13)\\ := \; & \begin{cases} \underline{pos(\gamma, \gamma_a, \beta, s, t)} & \text{if } l = l_e \\ \underline{neg(\gamma, \beta, t)} & \text{if } l = \textbf{skip}\text{-}l_e \\ \gamma & \textit{otherwise} \end{cases} && (14) \end{aligned}$$

where we define $pos(\gamma, \gamma_a, \beta, s, t)$ as:

Do the positive update by replacing the constraint at the target state t by the result of $pos(\gamma(s), \beta)$, which is described below. Note that here we operate on the accumulating parameter γ_a (initialized to γ_0), which corresponds to the fact that the `temp` disjuncts in the actual implementation are all initialized to **false**.

$$pos(\gamma, \gamma_a, \beta, s, t) := \gamma_a[t \mapsto c/t \mapsto (c \cup \underline{pos(\gamma(s), \beta))}] \quad (15)$$

with $pos(c,\beta)$ defined as:

For all disjuncts…

$$pos(c,\beta) := \bigcup_{c \in d} pos(d,\beta)$$

For all variables…

$$\begin{aligned} & pos(c,\beta) && (16)\\ = \; & pos(c, \{v_i \mapsto o_i, \ldots\}) && (17)\\ & && (18) \end{aligned}$$

Return **false** if in the current positive binding is not the same as the one we see or if we have a negatiove binding to that same object. Else, recurse for the remaining variables, storing the positive binding.

$$:= \begin{cases} \textbf{false} \text{ , if } v_i \in dom(d_+) \wedge o_i \neq d_+(v_i) \\ \quad \vee \; v_i \notin dom(d_+) \wedge o_i \in d_-(v_i) \\ pos((d_+[v_i \mapsto o_i], d_-), \{v_{i+1} \mapsto o_{i+1}, \ldots\}) \text{ , else} \end{cases} \quad (19)$$

(20)

Now the negative binding, similarly. Here we pass in the current configuration γ.

and $neg(\gamma, \beta, t)$ defined as:

$$neg(\gamma, \beta, t) := \gamma[t \mapsto c/t \mapsto \underline{neg(c,\beta)}] \quad (21)$$

with $neg(c,\beta)$ defined as:

For all disjuncts…

$$neg(c,\beta) := \bigcup_{d \in c} \underline{neg(d,\beta)} \quad (22)$$

with $neg(d,\beta)$ defined as:

For all variables…

$$neg(d,\beta) := \bigcup_{v \in dom(\beta)} \{\underline{neg(d,\beta,v)}\} \quad (23)$$

Return **false** if the incoming (negative) binding clashes with the positive one we have stored. If we do have a positive binding to a different object, just return d unchanged. We do not need to store the negative binding. If the variable is unbound positively, add the negative binding and return.

with $neg(d,\beta,v)$ defined as:

$$\begin{aligned} & neg(d,\beta,v) \\ := \; & \begin{cases} \textbf{false} & \text{if } d_+(v) = \beta(v) \\ d & \text{if } d_+(v) \neq \beta(v) \\ (d_+, d_-[v \mapsto \{\beta(v)\}]) & \text{if } v \notin dom(d_+) \cap dom(d_-) \\ (d_+, d_-[v \mapsto (n \cup \beta(v))]) & \text{if } v \notin dom(d_+) \wedge d_-(v) = n \end{cases} \end{aligned}$$

4. Correctness proof for concrete semantics

4.1 Labeled versus generic skip edges

Instead of

$$\textbf{skip}(e) = \wedge a : a \in A : \neg a(e)$$

we now use symbols **skip**-a for all $a \in A$ with:

$$\textbf{skip}\text{-}a(e) = \neg a(e)$$

At each place where there used to be a skip-loop before, we now place the appropriate skip edges for each symbol. (We can leave out an edge $(s, \textbf{skip}\text{-}a, s)$ on a state s if there is already an edge (s,a,s), but let's forget about this optimization for now.) This is a correct transformation, as long as those edges are updated conjointly, which we are going to make sure now…

We have the following definition:

$$\begin{aligned} lab(s,te) \quad = \quad & (\vee s' : s' \rightarrow^{\textbf{skip}} s : lab(s',t) \wedge \textbf{skip}(e)) \\ & \vee (\vee a, s' : a \in A \wedge S' \rightarrow^{a} s : lab(s',t) \wedge a(e)) \end{aligned}$$

which we can rewrite to:

$$\begin{array}{rcl} lab(s,te) & = & (\vee s' : \wedge a : s' \rightarrow^{\textbf{skip-}a} s : lab(s',t) \wedge \textbf{skip-}a(e)) \\ & & \vee\ (\vee a, s' : a \in A \wedge S' \rightarrow^{a} s : lab(s',t) \wedge a(e)) \end{array}$$

Or in the optimized version instead of...

$$\begin{array}{l} lab(s,te) = \\ (\textbf{if } s \rightarrow^{\mathbf{skip}} s \textbf{ then } lab(s,t) \wedge \mathbf{skip}(e) \textbf{ else } \mathit{false}) \\ \vee\ (\vee a, s' : a \in A \wedge s' \rightarrow^{a} s : lab(s',t) \wedge a(e)) \end{array} \quad (24)$$

... we can write:

$$\begin{array}{l} lab(s,te) = \\ (\wedge a \in A : \textbf{if } s \rightarrow^{\textbf{skip-}a} s \textbf{ then } lab(s,t) \wedge \textbf{skip-}a(e) \textbf{ else } \mathit{false}) \\ \vee\ (\vee a, s' : a \in A \wedge s' \rightarrow^{a} s : lab(s',t) \wedge a(e)) \end{array} \quad (25)$$

What differences still remain to the actual implementation?

1. There can be multiple events at one an the same joinpoint. In this case, we have to "or" over those.
2. The way bindings are stored.

The above equation suggests that it is possible to derive an implementation as described by equation (3): We first process all skip loops, then separately all other edges and combine the results disjointly.

4.2 What to prove

A correct implementation based on equation (3) would then have to fulfill the following requirements:

1. When processing skip-loops, this has to be done in a manner such that the result of processing each loop is conjoined with the previous results. Also, the updates have to be performed on (a copy of) the current configuration, which holds the $lab(s,t)$ for all states s.
2. When processing a normal edge, this has to be done in a disjoint manner. Also, the previous configuration $lab(s,t)$ plays no role here. Hence, one must operate on temporary labels initialized to $\mathbf{false}$ (as $\mathbf{false}$ is the neutral element for the operation $\vee$).
3. For the initial state s_0, we are always going to have $lab(s_0,t) = \mathbf{true}$ for any t, due to the implicit A^* loop on the state. Hence, it does not matter, what we initialize the temporary variable for s_0 to. It is hence safe, to assume a temporary *configuration* $(\mathbf{true}, \mathbf{false}, \ldots)$.

4.3 Proof of correctness

4.3.1 Negative updates

Skip-loops are processed using $neg(\gamma, \beta, t)$. Since skip-loops are loops, we can ignore the source state s, as we know that $s = t$.

As noted above, negative updates have to be performed on a copy of the current configuration, which we do by equation (21). Equation (22) is correct because we are acting on constraints in DNF. Hence, we have to disjoin the result of the operation for all disjuncts.

Equation (23) produces one set of disjuncts representing the negative update of the original disjunct. This implements the fact that updates have to be in a conjoint manner.

About the innermost overloaded definition of neg, $neg(d, \beta, v)$:

The case $d_+(v) = \beta(v)$, corresponds to the case where $lab(s,t) \wedge \textbf{skip-}a(e) = \mathbf{false}$.

In the case $d_+(v) \neq \beta(v)$, we would actually have to return a disjunct representing $(v = d_+(v)) \wedge (v \neq \beta(v))$. However, since v is already bound in this disjunct and by the definition of pos (pos only binds if there is nothing bound yet), there is no need to store the negative binding. We can just return the original disjunct.

In the third case, where there is neither a positive nor a negative binding for v, we generate a negative binding for v holding $\{\beta(v)\}$. This will prevent pos from binding v to this value in the future.

In the last case where there is already some negative binding for v but v is positively unbound, we just add the value to the negative bindings for the same reason.

4.4 Positive updates

Positive updates have to be performed on a temporary configuration γ_a which is initialized to γ_a. This is because the result of the positive update is combined *disjointly* (!) with the one of the negative update to form the global result. So if we did perform the positive update ont he current configuration, we would at each state have an implicit $\vee lab(s,t)$, which would be incorrect.

Hence, we include an accumulating parameter γ_a, which is initialized to $\gamma_0 = (\mathbf{true}, \mathbf{false}, \ldots)$. By the definition of equation (11), this parameter γ_a is replaced in each consecutive positive update with the result of the last such update, hence accumulating all updates in γ_a.

By equation (15), return a constraint based on the accumulating parameter, replacing the constraint of

5. Static abstraction

Explain that our analysis has to assure three things in order to be correct. First we have to make sure that for any dynamic execution path which leads from one shadow to another, there also exists a path in our abstraction. Secondly, we have to make sure that we take thread interleavings into account. Last but not least we have to update the bingings in our abstraction correctly. This last step will be explained in detail in Section 6. Here we first concentrate on the creation of the program abstraction.

Outline the different steps involved: 1.) create per-method state machines 2.) explain the concept of a thread context and define the contents of a thread summary 3.) give algorithm to combine state machines interprocedurally

6. Flow analysis

Here we consistently refer to the concrete semantics explained in Section 3. Explain that our abstraction is essentially very similar to the actual implementation. The only difference is that instead of mappings $var \mapsto object$ we now have mappings $var \mapsto pointsToSet$. Explain how those mappings have to be updated in order to be correct. Explain the concept of the "shadow history", which helps us to identify the trace of all shadows that can possibly lead to a match.

Give algorithm for the optimized fixed-point iteration. Explain merge operation in detail. (has to preserve the shadow history)

7. Benchmarks

Here we evaluate around 10 medium-sized benchmarks, probably taken of the dacapo and ashes suites, in combination with at least 5 tracematches. Questions to answer: 1.) How many shadows can be removed for each combination? 2.) How long do different parts of the analysis take for each combination? 3.) Is it worth minimizing the abstraction or the tracematch automaton? 4.) How muc do those benchmarks run faster with the reduced instrumentation? 5.) Does the analysis scale to multiple tracematches in on and the same program? 6.) What role does context-sensitivity play? 7.) ... more?

Discussion: Where do we fail? Why? What could be improved?

8. Related work

8.1 (Generalized) Typestates

- Do not handle threads (unsound).
- The non-generalized version used in [2] can only handle properties that talk about one single object.
- [2] does handle must-alias information nicely, however

8.2 PQL

no idea what they are doing so far. have to look into this!

8.3 Bandera (slicing)

no idea what they are doing so far. have to look into this!

8.4 Compiler optimizations using temporal logic

by this I mean [3] - kind of the inverse approach: Express dataflow analyses as finite-state properties which are then evaluated over a program. Could we do similar stuff in our approach? (express other flow analyses in this framework)

8.5 More?

9. Discussion and future work

- Were able to remove many instrumentation points for certain properties.
- Worse precision for other properties. Need must-alias/flow analyis (future work).
- Bulk of time spent in points-to and callgraph computation.
- Can be applied to virtually any finite-state runtime verification system.
- more?

References

[1] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding Trace Matching with Free Variables to AspectJ. In *Object-Oriented Programming, Systems, Languages and Applications*, pages 345–364. ACM Press, 2005.

[2] Stephen Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. In *ISSTA'06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 133–144, New York, NY, USA, 2006. ACM Press.

[3] David Lacey, Neil D. Jones, Eric Van Wyk, and Carl Christian Frederiksen. Proving correctness of compiler optimizations by temporal logic. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 283–294, New York, NY, USA, 2002. ACM Press.