Partially evaluating finite-state runtime monitors ahead of time[1]

ERIC BODDEN
Technische Universität Darmstadt
PATRICK LAM
University of Waterloo
and
LAURIE HENDREN
McGill University

Finite-state properties account for an important class of program properties, typically related to the ordering of operations invoked on objects. Many library implementations therefore include manually-written finite-state monitors to detect violations of finite-state properties at runtime. Researchers have recently proposed the explicit specification of finite-state properties and automatic generation of monitors from the specification. However, runtime monitoring only shows the presence of violations, and typically cannot prove their absence. Moreover, inserting a runtime monitor into a program under test can slow down the program by several orders of magnitude.

In this work, we therefore present a set of four static whole-program analyses that partially evaluate runtime monitors at compile time, with increasing cost and precision. As we show, ahead-of-time evaluation can often evaluate the monitor fully statically. This may prove that the program cannot violate the property on any execution or may prove that violations do exist. In the remaining cases, the partial evaluation converts the runtime monitor into a residual monitor. This monitor only receives events from program locations that the analyses failed to prove irrelevant. As we show, this makes the residual monitor much more efficient than a full monitor, while still capturing all property violations at runtime.

We implemented the analyses in Clara, a novel framework for the partial evaluation of AspectJ-based runtime monitors, and validated our approach by applying Clara to finite-state properties over several large-scale Java programs. Clara proved that most of the programs never violate our example properties. Some programs required monitoring, but in those cases Clara could often reduce the monitoring overhead to below 10%. We observed that several programs did violate the stated properties.

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

General Terms: Algorithms, Experimentation, Performance, Verification

Additional Key Words and Phrases: typestate analysis, static analysis, runtime monitoring

1. INTRODUCTION AND CONTRIBUTIONS

Finite-state properties constrain the set of acceptable operations on a single object or a group of inter-related objects, depending on the object's or group's history. Typestate systems [Strom and Yemini 1986], one particular instantiation of the idea of finite-state properties, enable the specification and (potentially static) ver-

[1]This submission consolidates and extends previous publications on the same subject. Section 11.3 explains the relationship of this work with our previous work.

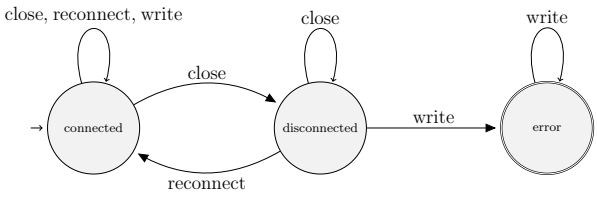


Fig. 1: "ConnectionClosed" finite-state property: no write after close.

ification of finite-state properties, in the service of program understanding and verification. One can define type systems [Bierhoff and Aldrich 2007; DeLine and Fähndrich 2004] that prevent programmers from writing code that may cause typestate errors. Unfortunately, current typestate systems require elaborate program annotations that tell the type checker either 1) which statements may modify an object's typestate or 2) which variables may or must point to the same objects (i.e., may or must alias). Such annotations are hard to maintain, perhaps explaining in part why such type systems have not been adopted in practice.

A pragmatic approach is to instead monitor programs for violations of finite-state properties at runtime. Several researchers have proposed notations and tools to support monitoring for finite-state properties expressed using tracematches or other formalisms [Allan et al. 2005; Bodden 2005; Chen and Roşu 2007; Maoz and Harel 2006; Krüger et al. 2006]. One notation, tracematches, combines regular expressions with AspectJ [aspectj 2003] pointcuts to provide a high-level specification language for runtime monitors. JavaMOP [Allan et al. 2005; Chen and Roşu 2007] is an open framework for notations, which can generate monitors from high-level specifications written in different concrete notations such as linear temporal logic, regular expressions or context-free grammars. Runtime monitoring is appealing because monitor specifications can be very expressive: they can reason about concrete program events and concrete runtime objects, and thus completely avoid false warnings. However, runtime monitoring basically amounts to testing, where the runtime monitor merely provides a principled way to insert high-level assertions into the program under test. Testing, however, has several drawbacks. A suite of sufficiently varying test runs may be able to identify errors or strengthen a programmer's confidence in her program by not identifying errors, but it does not constitute a correctness proof if the test suite is not complete. Secondly, runtime monitoring requires program instrumentation, and, as we show in this paper, this instrumentation may slow down the program under test by several orders of magnitude, making exhaustive testing even less of an option in many cases.

In this work we therefore propose a hybrid approach that starts with a runtime monitor but then uses static analysis results to convert this monitor into a residual runtime monitor. This residual monitor captures actual property violations as they occur, but updates its internal state only at relevant statements, as determined through static analysis. Unlike static type systems, our approach requires no program annotations; it is fully automatic.

Static analyses for optimizing monitors have special requirements; consider our running example, in which programmers must not write to a connection handle that is currently in its "closed" state. Figure 1 shows a non-deterministic finite-state machine for this property. It monitors a connection's "close", "reconnect"

and "write" events and signals an error at its accepting state. A correct runtime monitor must observe events like "close" and "write" that can cause a property violation, but also events like "reconnect" that may prevent the violation from occurring. Missing the former causes false negatives while missing the latter causes false positives, i.e., false warnings. Both are unacceptable, as runtime monitors must handle property violations exactly when they occur. A correct static analysis for partially evaluating runtime monitors must therefore determine program locations that can trigger either kind of relevant event. As we will show later, this is different from static analyses that only attempt to prove the absence of property violations but have no runtime monitoring component [Fink et al. 2006].

In this work we present a set of four static-analysis algorithms that evaluate finite-state runtime monitors ahead of time, with increasing precision. All algorithms analyze so-called *shadows* [Masuhara et al. 2003]. This term is popular in the aspect-oriented programming community and refers to program locations that can trigger runtime events of interest. The first analysis, the Quick Check, uses simple syntactic checks only. In our example, the Quick Check may be able to infer that a program opens and closes connections but never writes to a connection. Such a program cannot violate the property—if there are no writes, then no write can ever follow a close operation. The second analysis stage, the Orphan-shadows Analysis, applies a similar check on a per-object basis. If a program opens and closes some connection c, but never writes to c, then the analysis can rule out violations on c (but not on other connections, based on this information). This stage uses points-to analysis to handle aliasing, i.e., to decide whether or not two variables may point to the same runtime connection object. The third stage, the Nop-shadows Analysis, takes the program's control-flow into account. Using a backward analysis, it first computes, for every transition statement s (e.g. close, reconnect or write), sets of states that are, at s, equivalent with respect to all possible continuations of the control flow following s. The analysis then uses a forward pass to find transition statements that only switch between equivalent states. Switching between equivalent states is unnecessary, and the analysis removes such transitions.

As we prove, all three analysis stages are sound, i.e., when an analysis asserts that a program location requires no monitoring then this is indeed the case: removing transitions from this location can never alter the program locations at which the runtime monitor will (or will not) reach its error state. However, all three analyses are also incomplete: they may fail to identify program locations that actually require no monitoring. We therefore investigated and developed a fourth analysis, the Certain-match Analysis, which reports no false positives but may miss actual violations. This analysis is thus similar in flavour to unsound static checkers as implemented, for instance, in FindBugs [Hovemeyer and Pugh 2004] or PMD [Copeland 2005]. The Certain-match Analysis applies the same forward pass as the Nop-shadows Analysis, but instead identifies program locations at which the program certainly triggers a property violation. Such certain matches help programmers find true positives in a larger set of potential false positives.

We have implemented our analyses in Clara (CompiLe-time Approximation of Runtime Analyses), our novel framework for partially evaluating runtime monitors

ahead of time [Bodden et al. 2010; Bodden 2010]. We developed CLARA to facilitate the integration of research results from the static analysis, runtime verification and aspect-oriented-programming communities. CLARA features a formally specified abstraction, Dependency State Machines, which function as an abstract interface, decoupling runtime monitors from their static optimizations. The analyses that we present in this paper therefore apply to any runtime monitor implemented as an AspectJ aspect which uses Dependency State Machines. Our analyses are therefore compatible with a wide range of state-of-the-art runtime verification tools [Allan et al. 2005; Bodden 2005; Chen and Roşu 2007; Maoz and Harel 2006; Krüger et al. 2006], if they are extended to produce Dependency State Machines.

To evaluate our approach, we applied the analysis to the DaCapo benchmark suite [Blackburn et al. 2006]. In our experiments, in 68% of all cases CLARA's analyses can prove that the program is completely free of program locations that could drive the monitor into an error state. In these cases, CLARA gives the strong static guarantee that the program can never violate the stated property, eliminating the need for runtime monitoring of that program. In many other cases, the residual runtime monitor will require much less instrumentation than the original monitor, therefore yielding a greatly reduced runtime overhead. For monitors generated from a tracematch [Allan et al. 2005] specification, in 65% of all cases that showed overhead originally, no overhead remains after applying the analyses. The Certain-match Analysis, on the other hand, does not appear to be very effective: in our benchmark set, it could only identify a single match as *certain*, even though our runtime monitors signal several matches at runtime. This suggests that static analyses can be far more effective in determining that programs cannot violate finite-state properties than in proving that a program will violate such properties. Because of the design of our analyses, and of the CLARA framework, our analyses are equally effective on any runtime monitor for a given property, no matter which approach or tool was used to generate this monitor.

To summarize, this paper presents the following contributions:

—A set of three static whole-program analyses that can partially evaluate finite-state monitors ahead of time, with increasing precision.

—Soundness proofs for those three analyses.

—A novel Certain-match Analysis that can determine inevitable property violations on an intra-procedural level.

—A system for presenting analysis results to the user to support manual code inspection in the Eclipse integrated development environment.

—An open-source implementation of these analyses in the context of the CLARA framework and a large set of experiments that validates the effectiveness of our approach based on large-scale, real-world benchmarks drawn from the DaCapo benchmark suite.

2. RUNNING EXAMPLE

We continue by presenting a specification for the ConnectionClosed finite-state property and explaining how our static analyses can analyze programs for conformance with this property. In particular, we will demonstrate how our analyses

behave on code that always violates the property, code that sometimes violates the property and code that never violates the property (for two different reasons).

Recall that the ConnectionClosed property specifies that programs may not write to closed connections. Figure 2 presents a textual specification of the ConnectionClosed property using *Dependency State Machines* , CLARA's intermediate representation for runtime monitors. Specifications in CLARA consist of an AspectJ aspect (implementing a runtime monitor for the property), augmented with additional annotations that aid static analysis. The example aspect consists of three pieces of advice (lines 4–13) which intercept close, reconnect and write events. When closing a connection, the "close" advice places the closed connection object into the set `closed`. When the connection is re-connected, the "reconn" advice removes it from the set again. Finally, the "write" advice issues an error message upon any write to a connection in the `closed` set. Note that the aspect uses its own data structure—the set in line 2—to keep track of closed connections. CLARA seeks to be independent of such internal implementation details. The aspect therefore carries an additional, CLARA-specific annotation in lines 15–25: the monitor's Dependency State Machine. This state machine encodes the internal transition structure of the pieces of advice that implement the monitoring logic. Note that this is a textual representation of the state machine from Figure 1. We will formally define the semantics of Dependency State Machines in Section 4. The first author's dissertation [Bodden 2009] presents the formal syntax for these annotations.

The design of Dependency State Machines in CLARA allows them to function as an abstract interface, bridging the efforts of the static analysis community to efforts of the runtime verification community. Many state-of-the-art runtime verification tools generate monitors in the form of AspectJ aspects, because such aspects offer a convenient and declarative way to define the program points which require instrumentation. Figure 3, for example, shows a high-level specification for ConnectionClosed in the form of a tracematch [Allan et al. 2005]. Tracematches allow programmers to match on the execution history via a regular expression over AspectJ pointcuts. Line 2 states that the tracematch reasons about one connection `c` at a time. Lines 3–5 define the set of events that the monitor wishes to process. The events form symbols of an alphabet, and line 7 uses this alphabet to define the regular expression "disconn+ write". The body (lines 7–9) will therefore execute after one or more disconnects, followed by a write, as long as there is no intervening reconnect. The tracematch implementation generates an AspectJ aspect similar to the one we showed in Figure 2 from this specification. Other runtime verification tools also generate similar aspects from their specification languages. By augmenting an aspect with a Dependency State Machine annotation, the tool can easily make its generated aspects optimizable with CLARA, and therefore with all our analyses. In our experiments, we will focus on optimizing runtime monitors from tracematches, but in previous work we have shown [Bodden et al. 2009] that our analyses are equally effective on monitors generated from other types of high-level specifications.

Next, we discuss our analysis of ConnectionClosed on the code in Figure 4. Figure 4a always triggers the final state of the monitor, since it contains a connection close followed by a write on the same connection. Our Certain-match Analysis

```
1  aspect ConnectionClosed {
2      Set closed = new WeakIdentityHashSet();
3
4      dependent after close(Connection c) returning:
5          call(* Connection.disconnect()) && target(c) { closed.add(c); }
6
7      dependent after reconn(Connection c) returning:
8          call(* Connection.reconnect()) && target(c) { closed.remove(c); }
9
10     dependent after write(Connection c) returning:
11         call(* Connection.write(..)) && target(c) {
12         if(closed.contains(c))
13             error("May not write to "+c+": it is closed!"); }
14
15     dependency {
16         close, write, reconn;
17         initial connected:    close -> connected,
18                               write -> connected,
19                               reconn -> connected,
20                               close -> disconnected;
21                 disconnected: reconn -> connected,
22                               close -> disconnected,
23                               write -> error;
24         final   error:        write -> error;
25     }
26 }
```

Fig. 2: "ConnectionClosed" aspect with Dependency State Machine.

```
1  aspect ConnectionClosed {
2      tracematch(Connection c) {
3          sym disconn after returning: call(* Connection.disconnect()) && target(c);
4          sym reconn after returning: call(* Connection.reconnect()) && target(c);
5          sym write  after returning: call(* Connection.write(..)) && target(c);
6
7          disconn+ write {
8              error("May not write to "+c+", as it is closed!");
9          }
10     }
11 }
```

Fig. 3: "ConnectionClosed" tracematch.

will determine that it always triggers the final state. It does so by performing a flow-sensitive propagation of possible states for the connection `c`; after line 2, the connection is in the initial "connected" state. Following the `close` and `write` transitions, our analysis can deduce that the connection is sure to reach the final "error" state.

Our remaining analyses are staged: CLARA runs a series of analyses in turn, from least computationally expensive to most expensive. The idea is to do as little work

as possible to try to guarantee that programs do not violate properties. The first stage, Quick Check, therefore only collects the labels of transitions in the program, and eliminates the transitions which never affect whether or not the program triggers a final state. The second stage, Orphan-shadows Analysis, sharpens this information with pointer information. Finally, the third stage, Nop-shadows Analysis, is flow-sensitive: it uses information about the ordering of potential transitions in the program to rule out transitions which can never trigger the final state. CLARA then groups the remaining transitions into *potential failure groups*, using points-to information: transitions that can potentially affect the same object appear in the same group. As we explain in Section 9, this eases manual code inspection.

Figure 4b presents one example of a program which never triggers the final transition. In this case, the program contains both `write` and `close` transitions, so the Quick Check cannot remove these transitions. However, our pointer analysis finds that the connection objects `c1` and `c2` are distinct, so that no single object executes both the `write` and `close` transition. The Orphan-shadows Analysis therefore instructs CLARA to remove both transitions in Figure 4b.

Figure 4c also never triggers the final state, even though it contains all of the necessary transitions on appropriate objects. The analysis must track object abstractions through their potential states. In particular, our Nop-shadows Analysis establishes that the connection starts in its initial state after instantiation at line 2. Next, it follows the transitions in lines 3 and lines 4 to reason that these lines never trigger the monitor. Finally, since connection `c` does not escape the `main` method, the analysis can conclude that no other transition in the program affects `c`, so that none of the transitions on `c` in the `main` method affect a possible match. Note that it is much harder to prove that transitions are unnecessary than that they are necessary (as we did for Figure 4a).

Finally, Figure 4d illustrates a program for which no static analysis can determine whether or not the final state triggers, in this case because the transitions taken depend on program input. Each of our analyses would state that each transition could occur and has a potential effect on the state machine.

Parameterized traces. Every program run generates a parameterized trace over the pieces of advice applicable to that run. We reason about these traces by using abstract *parameterized* runtime traces, which are sequences of sets of abstract events. Sets of abstract events enable us to account for the fact that every concrete program event (e.g. method calls), can potentially be matched by a number of overlapping pieces of advice. Section 4 formally defines the semantics of dependency state machines over abstract parameterized runtime traces.

For instance, consider the program from Figure 4b. This program generates the parameterized trace "{close($c \mapsto o(\texttt{c1})$)}{write($c \mapsto o(\texttt{c2})$)}": the "close" method call is only matched by the "close" piece of advice, and this piece of advice binds the aspect's variable c to $o(\texttt{c1})$, the object referenced by `c1`. Similarly, the "write" method is only matched by the "write" piece of advice and binds c to $o(\texttt{c2})$. Parameterized traces give rise to "ground" traces by projection onto consistent variable bindings. In the above example, we can project onto $c \mapsto o(\texttt{c1})$ and $c \mapsto o(\texttt{c2})$. Projection onto $c \mapsto o(\texttt{c1})$ yields the trace "close", while $c \mapsto o(\texttt{c2})$ yields the trace "write". Neither projected trace belongs to the language that the Dependency

```
1 public static void main(String args[]) {
2     Connection  c = new Connection(args[0]);
3     c.close ();          // close(c)
4     c.write(args [0]);  // write(c): violation−write after close on c
5 }
```

(a) Example program which always matches

```
1 public static void main(String args[]) {
2     Connection  c1 = new Connection(args[1]),
3                 c2 = new Connection(args[2]);
4     c1.close ();          // close(c1)
5     c2.write(args [0]);  // write(c2): write, but on c2, hence independent of 4
6 }
```

(b) Example program which never matches due to incompatibility of transitions

```
1 public static void main(String args[]) {
2     Connection  c = new Connection(args[0]);
3     c.write(args [0]);  // write(c)
4     c.close ();          // close(c): no violation, since it always follows 3
5 }
```

(c) Example program which never matches due to ordering of transitions

```
1 public static void main(String args[]) {
2     Connection  c = new Connection(args[0]),
3     if (args.length > 1)
4         c.close ();      // close(c)
5     c.write(args [0]);  // write(c): violation−write after close possible
6 }
```

(d) Example program which sometimes matches

Fig. 4: Example programs

State Machine in Figure 2 accepts.

The program from Figure 4a yields the trace "{close($c \mapsto o(\mathsf{c})$)} {write($c \mapsto o(\mathsf{c})$)}". Here, projection onto $c \mapsto o(\mathsf{c})$ yields the ground trace "close write", which is in the language of the Dependency State Machine, indicating that this program may (and indeed will) violate the property that this Dependency State Machine describes.

3. CLARA FRAMEWORK

CLARA (CompiLe-time Approximation of Runtime Analyses) is a novel research framework for partially evaluating runtime monitors ahead of time. We developed CLARA to support easy implementations of the analyses presented in this paper, and to facilitate the integration of research results from the static-analysis, runtime-verification and aspect-oriented-programming community in general. CLARA's major design goal is to decouple code generation for efficient runtime monitors from the static analyses that convert these monitors into optimized, residual monitors which are triggered at fewer program locations. In this work, we provide a brief summary of CLARA's design; previous work [Bodden et al. 2010] and the first au-

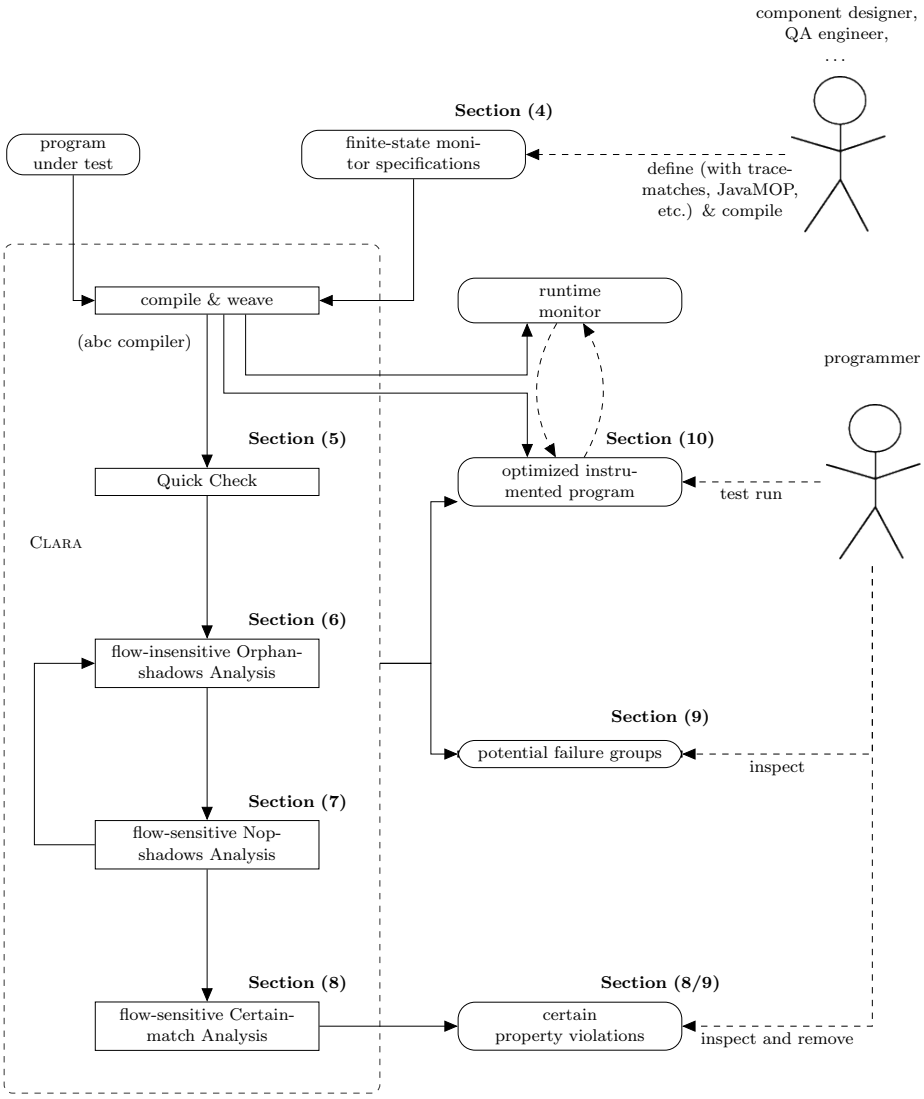


Fig. 5: Overview of Clara

thor's dissertation [Bodden 2009] give a more detailed account. Clara is available as open-source software at `http://bodden.de/clara/`.

Figure 5 gives an overview of Clara. At the beginning of the work flow (top right) stands a component designer who wrote an application interface (API) for which clients must fulfil certain finite-state properties to use this interface correctly. In our running example, this would be the programmer who initially provides the "Connection" API. As part of the API, the designer specifies a runtime monitor which captures property violations at runtime, for instance as a tracematch. Further, the programmers uses a runtime-verification tool to automatically translate the high-level specification into an AspectJ aspect, annotated with a Dependency

State Machine. We extended the tracematch implementation so that it automatically annotates the aspects that it generates. The authors of JavaMOP [Chen and Roşu 2007] are currently in the process of extending their tool accordingly, and it is likely that many other runtime verification tools will support Dependency State Machines in the near future.

The programmer invokes CLARA with the aspect-based monitor definition and a program as inputs. CLARA compiles the code of the runtime monitor and "weaves" the runtime monitor into the program under test, i.e., instruments the program with dispatch code that notifies the monitor about state transitions that the program performs. (CLARA extends the AspectBench Compiler [Avgustinov et al. 2005] for this purpose.) CLARA then invokes its static analysis engine. Researchers can add static analyses to CLARA and apply them in any order. These analyses collect information about the finite-state property to precisely approximate the set of relevant instrumentation points. When an analysis declares that an instrumentation point is irrelevant to a property, i.e., the program can neither violate the property nor prevent a property violation at this point, then CLARA automatically disables the instrumentation for this property at this point. The result is an optimized instrumented program that updates the runtime monitor only at program points at which instrumentation remains enabled.

Our Certain-match Analysis also reports certain matches to the programmer. Such matches denote program locations that certainly lead to a property violation if executed. In the end, CLARA outputs a list of *potential failure groups*. Each such group is a set of shadows containing a single potential point of failure, i.e., a shadow at which the program may violate the stated property at runtime, and a set of context shadows, i.e., shadows that trigger events on the same objects as the potential point of failure and therefore could contribute to reaching the property violation at this point. We found this presentation of our analysis results particularly useful for manual inspection.

4. DEFINITIONS

We now provide a formal description of finite-state runtime monitors. These definitions allow us to reason formally about the correctness of our static analyses.

4.1 Runtime Monitors

We begin by stating standard definitions from automata theory.

Definition 1 *Finite-state machine.* A non-deterministic *finite-state machine* $\mathcal{M}$ is a tuple $(Q, \Sigma, q_0, \Delta, Q_F)$, where Q is a set of states, Σ is a set of input symbols, q_0 the initial state, $\Delta \subseteq Q \times \Sigma \times Q$ the transition relation and $Q_F \subseteq Q$ the set of accepting (or final) states.

In our context, we will also call final states "error states". We consider that a finite-state property has been violated when the state machine associated with the property reaches a final state. (In that sense, our properties are negative properties which describe forbidden behaviour.)

Definition 2 *Words, runs and acceptance.* A *word* $w = (a_1, \ldots, a_n)$ is an element of Σ^*; word w has length $|w| = n$. We define a *run* ρ of $\mathcal{M}$ on w to

be a sequence $(q_0, q_{i_1}, \ldots, q_{i_n})$ which respects the transition relation Δ; that is, $\forall k \in [0, n).\ \exists a_k.\ (q_{i_k}, a_{k+1}, q_{i_{k+1}}) \in \Delta$, with $i_0 := 0$. A run ρ is *accepting* if it finishes in an accepting state, i.e. $q_{i_n} \in Q_F$. We say that $\mathcal{M}$ accepts w, and write $w \in \mathcal{L}(\mathcal{M})$, if there exists an accepting run of $\mathcal{M}$ on w.

We further require the notion of a prefix.

Definition 3 *Set of prefixes.* Let $w \in \Sigma^*$ be a Σ-word. We define the set $\mathit{pref}(w)$ as:

$$\mathit{pref}(w) := \{p \in \Sigma^* \mid \exists s \in \Sigma^* \text{ such that } w = ps\}.$$

Definition 4 *Matching prefixes of a word.* Let $w \in \Sigma^*$ be a Σ-word and $\mathcal{L} \subseteq \Sigma$ a Σ-language. Then we define the matching prefixes of w (with respect to $\mathcal{L}$) to be the set of prefixes of w also belonging to $\mathcal{L}$:

$$\mathit{matches}_{\mathcal{L}}(w) := \mathit{pref}(w) \cap \mathcal{L}.$$

We write $\mathit{matches}(w)$ instead of $\mathit{matches}_{\mathcal{L}}(w)$ if $\mathcal{L}$ is clear from the context.

In Clara, we use finite-state machines to model and implement runtime monitors. Clara first generates a finite-state machine from the provided monitor definition. It then instruments the program under test such that the program will issue a trace (effectively a word in Σ^*) when executed. The finite-state machine then reads this trace as input. The instrumented program executes the monitor's associated error handler whenever the machine reaches an accepting state, i.e. whenever the prefix of the trace read so far is an element of $\mathcal{L}$.

Generalizing to multiple monitor instances. It would be a severe restriction to allow for only one monitor instance at runtime. Consider the ConnectionClosed example from Section 1. Programs typically use many simultaneously-active connection objects; furthermore, each connection object could be in a different state, depending on its history. To avoid this restriction, modern runtime monitoring systems allow the user to define *parametric* runtime monitors [Chen and Roşu 2007]. A parametric runtime monitor effectively comprises a set of monitors, one monitor for every variable binding. To support such tools, Clara's semantics are defined over parametric monitors, which we now define.

Definition 5 *Variable Binding.* Let $\mathcal{O}$ be the set of all runtime heap objects and $\mathcal{V}$ a set of variables appearing in monitor specifications. Then we define a variable binding β as a partial function $\beta : \mathcal{V} \rightharpoonup \mathcal{O}$. We call the set of all possible variable bindings $\mathcal{B}$.

Due to variable bindings, runtime monitoring does not operate on a single trace from Σ^*, but rather on a parametrized trace, consisting of a trace of parametrized events. Parametrized events associate bindings with events.

Definition 6 *Parametrized event.* A parametrized event $\hat{e}$ is a set of pairs $(a, \beta) \in \Sigma \times \mathcal{B}$. We call the set of all parametrized events $\hat{\Sigma}$. A parametrized trace is a word from $\hat{\Sigma}^*$.

We use sets of pairs because one program event can yield multiple monitoring events. This occurs when multiple monitors listen for the same program events.

Every monitored program will generate a parametrized event trace when it executes. The instrumentation that CLARA inserts into the program notifies the runtime monitor at every event of interest about the monitor transition symbol $a \in \Sigma$ as well as the variable binding $\beta \in \mathcal{B}$ identifying all monitor instances that need to process the event.

For instance, executing the program from Figure 4b with the ConnectionClosed monitoring aspect from Figure 2 yields the following parametrized trace:

$$\{(close, c \mapsto o(\texttt{c1}))\} \ \cdot \ \{(write, c \mapsto o(\texttt{c2}))\}$$

Here, $o(\texttt{v})$ represents the object referred to by program variable `v`.

However, monitor instances are ordinary finite-state machines, which process symbols from the base alphabet Σ, rather than parametrized events from $\hat{\Sigma}$. We therefore project the unique parametrized program trace that the program generates onto a set of ground traces—words over Σ. Every ground trace is associated with a binding β and contains all events whose bindings are compatible with β.

Definition 7 *Compatible bindings.* Let $\beta_1, \beta_2 \in \mathcal{B}$ be two variable bindings. These bindings are compatible when they agree on the objects that they jointly bind:

$$\mathit{compatible}(\beta_1, \beta_2) := \forall v \in (\mathit{dom}(\beta_1) \cap \mathit{dom}(\beta_2)).\ \beta_1(v) = \beta_2(v).$$

Definition 8 *Projected event.* For every parametrized event $\hat{e}$ and binding β we define a projection of $\hat{e}$ with respect to β:

$$\hat{e}{\downarrow}\beta := \{a \in \Sigma \mid \exists (a, \beta_a) \in \hat{e} \text{ such that } \mathit{compatible}(\beta_a, \beta)\}.$$

Definition 9 *Parametrized and projected event trace.* Any finite program run induces a parametrized event trace $\hat{t} = \widehat{e_1} \ldots \widehat{e_n} \in \hat{\Sigma}^*$. For any variable binding β we define a projected trace $\hat{t}{\downarrow}\beta \subseteq \Sigma^*$ by only keeping events compatible with β. Formally, $\hat{t}{\downarrow}\beta$ is the smallest subset of Σ^* for which, if $e_{f(i)} := \widehat{e_i}{\downarrow}\beta$ for all $i \in \{1, \ldots, n\}$, with $f : [1, n] \to [1, m]$ order-preserving and $m \leq n$, then

$$e_1 \ldots e_m \in \hat{t}{\downarrow}\beta.$$

In the following we will call traces like t, which are elements of Σ^*, ground traces, as opposed to parametrized traces, which are elements of $\hat{\Sigma}^*$.

In our semantics, a runtime monitor will execute its error handler whenever the prefix read so far of one of the ground traces of any variable binding is in the language described by the state machine. We exclude the empty trace (with no events) because this trace cannot possibly cause the handler to execute: empty traces contain no events, while we require handlers to see at least one event before executing. This yields the following definition.

Definition 10 *Set of non-empty ground traces of a run.* Let the trace $\hat{t} \in \hat{\Sigma}^*$ be the parametrized event trace of a program run. Then the set $\mathit{groundTraces}(\hat{t})$ of non-empty ground traces of $\hat{t}$ is:

$$\mathit{groundTraces}(\hat{t}) := \left(\bigcup_{\beta \in \mathcal{B}} \hat{t}{\downarrow}\beta \right) \cap \Sigma^+.$$

We intersect with Σ^+ to exclude the empty trace.

Definition 11 *Matching semantics of a finite-state runtime monitor.* Let $\mathcal{M} := (Q, \Sigma, q_0, \Delta, Q_F)$ be a finite-state machine. Let $\hat{t} \in \hat{\Sigma}^*$ be a parametrized event trace generated by a program under test. We say that $\hat{t}$ violates the property described by $\mathcal{M}$ at position i when:

$$\exists t \in \mathit{groundTraces}(\hat{t}).\ \exists p \in \mathit{matches}_{\mathcal{L}(\mathcal{M})}(t).\ |p| = i.$$

By our definition of runtime monitoring, the monitor will execute its error handler at every position i at which $\hat{t}$ violates the monitored property.

4.2 Statically optimizing parametrized monitors

The above definition of the matching semantics for finite-state runtime monitors states exactly when a runtime monitor needs to trigger on an input trace $\hat{t}$. Any sound static optimization of such runtime monitors must obey this semantics, i.e., must guarantee that the monitors trigger (or don't trigger) exactly at the same times with or without the optimization.

We next define a runtime predicate called *mustMonitor* that, for every symbol $a \in \Sigma$, every parametrized trace $\hat{t}$ and every position $i \in \mathbb{N}$ in this trace, returns **true** when a-transitions must be monitored at position i of $\hat{t}$ according to the above semantics and **false** when the transition need not be monitored, i.e., when processing of the a-transition may safely be omitted.

$$\mathit{mustMonitor}: \quad \Sigma \times (\hat{\Sigma})^* \times \mathbb{N} \quad \rightarrow \quad \mathbb{B}$$

$\mathit{mustMonitor}(a, \hat{t}, i) := \exists t \in \mathit{groundTraces}(\hat{t})$ such that $\mathit{necessaryTransition}(a, t, i)$.

The *mustMonitor* predicate depends on the predicate $\mathit{necessaryTransition}(a, t, i)$. This predicate is a free parameter to our semantics, enabling the use of any suitable definition of *necessaryTransition*. Our semantics demand that *necessaryTransition* must meet the following soundness condition.

Condition 1 *Soundness condition for necessaryTransition predicate.* Any sound implementation of *necessaryTransition* must satisfy:

$$\begin{aligned}&\forall a \in \Sigma.\ \forall t = t_1 \ldots t_i \ldots t_n \in \Sigma^+.\ \forall i \in \mathbb{N}.\\ &\qquad a = t_i \wedge \mathit{matches}_{\mathcal{L}}(t_1 \ldots t_n) \neq \mathit{matches}_{\mathcal{L}}(t_1 \ldots t_{i-1} t_{i+1} \ldots t_n)\\ &\qquad\qquad \Longrightarrow \mathit{necessaryTransition}(a, t, i).\end{aligned}$$

In other words, a transition a must be monitored at position i whenever not monitoring a at i would change the set of matches for a runtime monitor. Note that it is only possible to approximate *necessaryTransition*; the optimal function is uncomputable because it would require knowledge about future events: the most accurate possible *necessaryTransition* requires that, while observing the i-th event, one would need to know whether the remainder of the trace will or will not lead to further matches.

Sections 5 through 7 define three different approximations to *necessaryTransition* that are computable at compile-time. We will be able to prove these approximations sound by showing that they imply the soundness condition.

5. SYNTACTIC QUICK CHECK

The Quick Check is, as the name suggests, a very simple analysis that can execute within milliseconds. This quick analysis rules out transitions that obviously have no effect because they are unreachable or have no effect in the program under analysis. The Quick Check only uses syntactic information which is readily available to the compiler after it inserts instrumentation for all runtime monitors.

Algorithm 1 presents pseudo-code which computes required data for the Quick Check algorithm. The algorithm first iterates through the transitions of the monitor's finite-state machine. For every a-transition, if the compiler determines that the program cannot generate any a-events, then the Quick Check removes the transition from the state machine. Next, the Quick Check removes all unproductive states—states that have become unreachable from the initial state or from which no final state can be reached. The algorithm then computes the output set *symbolsThatNeedMonitoring* in two steps. In the first step, the Quick Check adds all symbols that the finite-state machine still contains transitions for, except symbols that are only associated with trivial loops. (Trivial loop symbols do not need to be monitored because they can never change the automaton's state.) Then, in a last step, the Quick Check adds the symbols that may cause an input word to be rejected, i.e., symbols which prevent spurious matches. Such symbols originate at productive states that have no outgoing a-transition.

To connect the Quick Check to CLARA's optimization engine, we simply define a new predicate *necessaryTransitionQC* as an instantiation of the predicate *necessaryTransition*:

$$necessaryTransitionQC(a, t, i) := a \in symbolsThatNeedMonitoring.$$

As an example, consider again the ConnectionClosed automaton from Figure 1 in combination with a program that closes and perhaps reconnects connection objects but never writes to them (for instance, the program from Figure 4a without line 4). In this case, the Quick Check would first remove all `write`-transitions from the finite-state machine. Next, the algorithm would find that all states have become unproductive, yielding an empty state machine. Therefore, the Quick Check would correctly determine that no symbol requires monitoring.

The Quick Check sometimes eliminates only some of the transitions in a finite state machine. Consider the automaton in Figure 6. In this example, if the program can produce all events except b events, then the Quick Check will reduce the

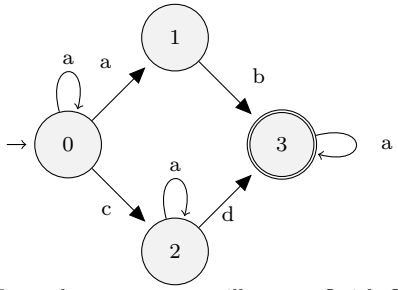


Fig. 6: Example automaton to illustrate Quick Check.

Algorithm 1 Compute necessary data for Quick Check.

Input: set L of labels of all transition statements in the program
Output: set *symbolsThatNeedMonitoring* of symbols that require monitoring

```
1: for (q_s, a, q_t) ∈ Δ do
2:   if a ∉ L then // no a-transition in the entire program
3:     Δ := Δ\{(q_s, a, q_t)}
4:   end if
5: end for
6: remove all unproductive states from Q and transitions to these states from Δ
7: symbolsThatNeedMonitoring := ∅
8: // find all symbols that transition from one state to another
9: for (q_s, a, q_t) ∈ Δ do // note that Δ may have changed
10:  if q_s ≠ q_t then // disregard symbols that only loop
11:    symbolsThatNeedMonitoring := symbolsThatNeedMonitoring ∪ {a}
12:  end if
13: end for
14: // find all symbols that may cause a word to be rejected
15: for q_s ∈ Q, a ∈ Σ do
16:  if ∄q_t ∈ Q such that (q_s, a, q_t) ∈ Δ then
17:    symbolsThatNeedMonitoring := symbolsThatNeedMonitoring ∪ {a}
18:  end if
19: end for
20: return symbolsThatNeedMonitoring
```

state machine to states $0, 2$, and 3. Along the remaining acyclic path through the automaton, the symbols c and d change the machine's state, and hence require monitoring. The symbol a, however, does not require monitoring because on the remaining productive states $0, 2$, and 3 a-transitions always loop. Symbol b would actually be a member of the set *symbolsThatNeedMonitoring* returned by Quick Check, but that is a moot point, since b events cannot occur in our program.

Similarly, consider the case where the program can produce all events except d-events. In that case, the Quick Check would determine that symbols a, b and c require monitoring: a and b because they transition from one productive state to another, and c because states 1 and 3 have no outgoing c-transition and therefore would possibly reject words when reading a c.

To clarify the last point, assume a program that generates a trace "$a\ c\ b$". The monitor should not trigger on this trace because "$a\ c\ b$" is not in the language of this finite-state machine. But if we failed to monitor c then the state machine would effectively only observe the partial trace "$a\ b$". This trace would drive the finite-state machine into its final state, which would be incorrect.

The Quick Check generally works well in cases where properties do not apply to the program under test. For instance, in the ConnectionClosed example, the Quick Check would succeed only if the program either never closes connections or never writes to them. One may wonder why monitors would track properties which a program can obviously never violate. However, we envision a scenario in which monitors and programs are written by different people. In our example, the

monitor for the ConnectionClosed property would be written by the developers of the Connection interface, and distributed along with that interface. The library developers cannot know in advance which parts of the distributed properties actual programs will use.

Soundness of the Quick Check

To show that the Quick Check meets the soundness condition from Section 4 we need to show the following:

$$\forall a \in \Sigma \quad \forall t = t_1 \dots t_i \dots t_n \in \Sigma^+ \quad \forall i \in \mathbb{N} : \\ a = t_i \wedge \mathit{matches}_{\mathcal{L}}(t_1 \dots t_n) \neq \mathit{matches}_{\mathcal{L}}(t_1 \dots t_{i-1} t_{i+1} \dots t_n) \\ \implies \mathit{necessaryTransitionQC}(a, t, i)$$

PROOF. Assume $\mathit{matches}_{\mathcal{L}}(t_1 \dots t_n) \neq \mathit{matches}_{\mathcal{L}}(t_1 \dots t_{i-1} t_{i+1} \dots t_n)$. Because the *matches* sets differ, we know that after having read the prefix $t_1 \dots t_{i-1}$, the automaton must either move from one productive state to another or it must move to no state at all because no current state has a t_i-transition. By the construction of *symbolsThatNeedMonitoring*, we know that $a \in \mathit{symbolsThatNeedMonitoring}$ and therefore $\mathit{necessaryTransitionQC}(a, t, i)$ holds. □

6. FLOW-INSENSITIVE ORPHAN-SHADOWS ANALYSIS

The flow-insensitive Orphan-shadows Analysis sharpens the results of the Quick Check by taking pointer information into account; basically, it removes transitions which are ineffective because they are bound to objects that never match. The Orphan-shadows Analysis models runtime objects using the static abstraction of points-to sets. For any program variable p, the points-to set $\mathit{pointsTo}(p)$ is the set of all allocation sites, as represented by `new` statements, that can reach p through a chain of assignments.

Recall the example from Figure 4b, which never matched because the `close` and `write` events occurred on different objects. In that program, the points-to set for variable `c1` referenced at line 4, would contain the `new` statement at line 2, while the points-to set for variable `c2` referenced at line 5 would contain the `new` statement at line 3. The points-to sets $\mathit{pointsTo}(\texttt{c1})$ and $\mathit{pointsTo}(\texttt{c2})$ would be disjoint, since it is impossible for `c1` and `c2` to point to the same object.

Recall that, at runtime, variable bindings β connect monitor variables to runtime heap objects. Points-to sets can serve in the place of these heap objects, and we denote static variable bindings which use points-to sets by $\tilde{\beta}$. Furthermore, the static variable bindings summarize runtime bindings. That is, let v be a monitor specification variable and o be a runtime object. Then $\beta(v) = o$ implies that o was created at one of the `new` statements n such that $n \in \tilde{\beta}(v)$.

Bindings are critical to matching. A runtime monitor only reaches its error state after having processed appropriate transitions with a "consistent variable binding". That is, each pair of transitions comes with bindings β_i and β_j; we require that, for each pair of transitions leading to a match, $\mathit{compatible}(\beta_i, \beta_j)$.

We statically approximate variable bindings using points-to sets. To do so, we define a static approximation *stCompatible* of the compatibility predicate *compatible*. Instead of requiring equality as in the runtime case, we instead declare two bindings

to be statically compatible when their points-to sets *overlap* on their joint domains. That is, variables in common may possibly be assigned the same objects at runtime:

$$\mathit{stCompatible}(\tilde{\beta}_1, \tilde{\beta}_2) := \forall v \in (\mathit{dom}(\tilde{\beta}_1) \cap \mathit{dom}(\tilde{\beta}_2)) \;.\; \tilde{\beta}_1(v) \cap \tilde{\beta}_2(v) \neq \emptyset$$

Shadows. Programs may contain *shadows*, which are static program points causing finite-state-machine transitions. Shadows occur as specified by the pointcuts in the declaration of the dependency state machine. Each shadow binds some number of variables. At runtime, shadows cause events, and their variables become bound to heap objects. As we alluded to above, we use points-to sets to approximate the heap objects occurring in variable bindings. We denote the set of all shadows by $\mathcal{S}$.

We say that two shadows s_1 and s_2 are compatible, and write $\mathit{stCompatible}(s_1, s_2)$, if their bindings are statically compatible. As with usual applications of points-to analyses, our static analysis exploits the negation of *stCompatible*: it soundly disregards transitions or events that can, in combination, only lead to inconsistent variable bindings. Such events clearly cannot drive the runtime monitor into an error state.

Algorithm 2 outlines the algorithm for the Orphan-shadows Analysis. For every shadow s, we first compute the set *compSyms* of labels of all shadows compatible with s. (Note that every shadow s is, by definition, also compatible with itself.) Next, we invoke the Quick Check algorithm, Algorithm 1, to compute the set of necessary symbols, under the assumption that *compSyms* is the set of labels of all transitions in the entire program. In other words, the Orphan-shadows Analysis projects out the set of transitions compatible with s. If s is necessary, then it can cause a transition that may contribute to reaching the final state (as guaranteed by the Quick Check) on compatible bindings (as guaranteed by the projection), so its label will belong to the computed set of necessary symbols. We therefore add such shadows s to the set *necessaryShadows*.

To connect the Orphan-shadows Analysis to Clara's optimization engine, just like for the Quick Check, we define a new predicate *necessaryTransitionOSA*, as a second instantiation of the predicate *necessaryTransition*:

$$\mathit{necessaryTransitionOSA}(a, t, i) := \mathit{shadow}(t_i) \in \mathit{necessaryShadows}$$

Algorithm 2 Compute necessary data for Orphan-shadows Analysis

Output: set *necessaryShadows* of shadows that require monitoring

```
1: necessaryShadows := ∅
2: for s ∈ S do
3:    compSyms := { l | ∃s' ∈ S . stCompatible(s, s') ∧ l = label(s)}
4:    necessarySyms := QuickCheck(compSyms)
5:    if label(s) ∈ necessarySyms then
6:       necessaryShadows := necessaryShadows ∪ {s}
7:    end if
8: end for
9: return necessaryShadows
```

6.1 Soundness of the Orphan-shadows Analysis

To show that the Orphan-shadows Analysis meets the soundness condition from Section 4 we now need to show:

$$\forall a \in \Sigma \quad \forall t = t_1 \dots t_i \dots t_n \in \Sigma^+ \quad \forall i \in \mathbb{N} : \\ a = t_i \wedge \mathit{matches}_{\mathcal{L}}(t_1 \dots t_n) \neq \mathit{matches}_{\mathcal{L}}(t_1 \dots t_{i-1} t_{i+1} \dots t_n) \\ \Longrightarrow \mathit{necessaryTransitionOSA}(a, t, i)$$

Proof. Assume $\mathit{matches}_{\mathcal{L}}(t_1 \dots t_n) \neq \mathit{matches}_{\mathcal{L}}(t_1 \dots t_{i-1} t_{i+1} \dots t_n)$. Just as for the Quick Check, we know that after having read the prefix $t_1 \dots t_{i-1}$ the automaton must either move from one productive state to another or it must move to no state at all because no current state has a t_i-transition. In either case, the disequality implies that the transition at position i must have a variable binding that is compatible with all bindings of all transitions at positions $1, \dots, n$. Therefore, by construction, it must hold that $\mathit{shadow}(t_i) \in \mathit{necessaryShadows}$ and therefore $\mathit{necessaryTransitionOSA}(a, t, i)$ must hold too. □

6.2 The benefits of a demand-driven pointer analysis

To compute points-to sets we use the demand-driven, refinement-based, context-sensitive, flow-insensitive pointer-analysis by Sridharan and Bodík [Sridharan and Bodík 2006]. A context-sensitive analysis helps us distinguish multiple objects that are allocated in different program contexts but using the same allocation sites, e.g., multiple iterators that are all instantiated by calling the same `iterator()` method in the Java runtime library. Sridharan and Bodík's analysis starts with context-insensitive information computed by Spark [Lhoták and Hendren 2003] and then refines the context-insensitive results with additional context information on demand. This is relatively fast because the refinement only needs to be computed for variables that we are interested in, i.e., for program variables that the monitor actually refers to. The pointer analysis is also demand-driven: it computes context information up to a certain level, defined by a user-provided quota. If the refined information is precise enough to distinguish the computed points-to sets from others, then we are done. Otherwise, we can opt to have the points-to set refined further with a higher quota.

To use the demand-driven approach, we augmented points-to sets with wrappers. Upon a emptiness-of-intersection query for points-to sets $\mathit{pointsTo}(\texttt{c1})$ and $\mathit{pointsTo}(\texttt{c2})$, the wrappers will compute a first approximation of $\mathit{pointsTo}(\texttt{c1})$ and $\mathit{pointsTo}(\texttt{c2})$. If this approximation is sufficient to determine that $\mathit{pointsTo}(\texttt{c1}) \cap \mathit{pointsTo}(\texttt{c2}) = \emptyset$, then the wrappers return **false** right away. Otherwise, the wrappers refine the approximations of both points-to sets and re-iterate until finding two approximations with empty intersection (yielding **false**), or until exhausting a pre-defined quota, yielding **true**.

7. FLOW-SENSITIVE NOP-SHADOWS ANALYSIS

We next motivate the Nop-shadows Analysis by discussing the need for flow-sensitiveity on some code which exercises our running example, the ConnectionClosed tracematch. Our first example is straight-line code involving a single connection object; however, in Section 7.4, we discuss how our analysis handles loops, multiple methods, and events on arbitrary combinations of aliased objects.

Figure 7 presents our example code. It is annotated with the analysis information that our analysis will compute—an explanation of this information follows. The code creates a connection and executes some operations on that connection, all of which cause transitions on the ConnectionClosed aspect.

While this example is clearly contrived, it demonstrates the possibilities for optimization by taking control flow into account. (Note that the flow-insensitive Orphan-shadows Analysis does not apply since both disconn and reconn events occur on the same connection.) The only events that must be monitored to trigger the monitor for this example at the right time are 1) the write at line 7 and 2) one of the two disconnect events at lines 5 and 6. In particular, the disconnect and reconnect operations at lines 3 and 4 do not need to be monitored: they are on the prefix of a match, but the match can be completed even without monitoring this prefix. Conversely, the operations at lines 8 to 10 do not lead to a pattern violation and hence do not need to be monitored either. Of the the two disconnects at lines 5 and 6, it is sound to omit monitoring one of them, but not both.

Our next static analysis eliminates the monitoring of the events identified above as unnecessary—*nop shadows*. As a result, instrumentation will only remain at lines 5 and 7, or 6 and 7—the minimal set of instrumentation points guaranteeing optimized instrumented program will report an error if and only if the un-optimized program would have reported an error.

In summary, the Nop-shadows Analysis identifies nop shadows one at a time, using two analysis phases. The first phase computes, for every statement in a method, sets of states at that statement which may possibly lead (in the rest of the program) to a final state—"hot states"—and states which will never give rose to a final state—"cold states". The first phase uses a backward dataflow analysis to compute these sets of states. Then, the second phase uses a forward dataflow analysis to compute the possible monitor states at each statement s.

Combining the analysis information allows us to determine whether s is a nop shadow. Assume that s can transition from q to q'. Then s needs to be monitored, i.e., is not a nop shadow, if there is some continuation for which q and q' are not

```
1  public static void main(String args[]) {
2      Connection c1 = new Connection(args[0]);
                                  {0}   { {}  ···  {0,1,2} }
3      c1.disconnect();           {1}   { {}  ···  {0,1,2} }
4      c1.reconnect();            {0}   { {}  ···  {0,1,2} }
5      c1.disconnect();           {1}   { {}  ···  {0,1,2} }
6      c1.disconnect();           {1}   { {}  ···  {1} }
7      c1.write(args [1]);        {2}   { {}  ···  {2} }
8      c1.disconnect();           {1}   { {} }
9      c1.reconnect();            {0}   { {1} }
10     c1.write(args [1]);        {0}   { {2} }
11 }
```

Fig. 7: Example program, annotated with combined analysis information.

in the same equivalence class.

Two situations require shadows to remain enabled: (1) a transition at s may move the automaton from a hot to a cold state, possibly leading to false positives; or (2) a transition from a cold state to a hot state, leading to false negatives. In case (1), disabling s may lead to false positives at runtime; because the transition is disabled, the monitor state remains hot and the monitor may therefore signal a violation that s would have prevented. In the inverse case (2), the transition moves the automaton from cold to hot. In this case, disabling s may yield a false negative; the monitor could fail to signal an actual violation.

We therefore can disable s in all other cases: if, for all continuations of s, all possible source states q and target states q' are either all hot or all cold. Such a transition would not change whether or not the automaton matches, and we declare that s is a nop shadow.

Note that our description discusses one shadow at a time. Because of dependencies between shadows, we iterate our algorithm until it reaches a fixed point, removing one shadow at a time, using a greedy algorithm. We discuss this issue further below.

We begin our description of the algorithm with a discussion of the forward analysis, which is simpler to compute than the backward analysis.

7.1 Forward analysis

The forward pass determines, for each statement s, the set of automaton states that the automaton could be in at statement s. The forward analysis works on a deterministic version of the input state machine[2]. CLARA obtains the deterministic automaton using the well-known subset-construction technique:

Definition 12 *Determinizing a non-deterministic state machine.* Let $\mathcal{L} \subseteq \Sigma^*$ be a regular Σ-language and let $\mathcal{M} = (Q, \Sigma, \Delta, Q_0, F)$ be a non-deterministic finite-state machine with $\mathcal{L}(\mathcal{M}) = \mathcal{L}$. Then we define the deterministic finite-state machine $det(\mathcal{M})$ as $det(\mathcal{M}) := (\mathcal{P}(Q), \Sigma, \delta, Q_0, \hat{F})$ by:

$$\begin{aligned} \delta &= \lambda Q_s.\lambda a.\ \{q_t \in Q \mid \exists q_s \in Q_s \text{ such that } \exists (q_s, a, q_t) \in \Delta\} \\ \hat{F} &= \{Q_F \in \mathcal{P}(Q) \mid \exists q \in Q_F \text{ such that } q \in F\} \end{aligned}$$

Figure 8a reproduces the non-deterministic finite-state machine for the ConnectionClosed example. Figure 8b shows the equivalent deterministic finite-state machine. We have assigned a fresh state number to each state in the deterministic automaton.

In Figure 7, next to the downwards-pointing arrow, we have annotated each statement with the states of the deterministic automaton just before and after executing that statement. In this example, the program has only a single control-flow path, and therefore our analysis will only associate a single state with each statement. However, if there are multiple control-flow paths reaching a statement

[2]Determinizing ensures that the state machine will be in only one state at a time. This simplifies the backward analysis: the backward pass partitions the states into equivalence classes. A non-deterministic state machine would require partitioning of sets of states, which is still possible but harder to describe.

s, and the execution along these paths yields different states q_1 and q_2, then our analysis will associate both q_1 and q_2 with s. In the sequel, we will denote the set of source states associated with s by $sources(s)$. Also, we will refer to the deterministic finite-state machine $det(\mathcal{M})$ as $\mathcal{M}_{fwd}$.

7.2 Backward analysis

The backward analysis determines, for every statement s, a set of sets of states; it finds one set for every possible continuation of the control flow after s which reaches the final state. Like the forward analysis, the backward analysis uses a determinized state machine. In particular, it uses a determinized state machine for the mirror language $\overline{\mathcal{L}}$.

Definition 13 *Mirror word.* Let $w = w_1 \dots w_n \in \Sigma^*$ be a Σ-word. We define the mirror word $\overline{w}$ to be $\overline{w} := w_n \dots w_1$.

Definition 14 *Mirror language.* Let $\mathcal{L} \subseteq \Sigma^*$ be a Σ-language. Then we define the mirror language $\overline{\mathcal{L}}$ to be:

$$\overline{\mathcal{L}} := \{\overline{w} \mid w \in \mathcal{L}\},$$

Given a non-deterministic finite-state machine $\mathcal{M}$ with $\mathcal{L}(\mathcal{M}) = \mathcal{L}$, one can easily obtain a non-deterministic finite-state machine accepting $\overline{\mathcal{L}}$ by reversing the transition function.

Definition 15. Reversed finite-state machine Let $\mathcal{M} = (Q, \Sigma, \Delta, Q_0, F)$ be a non-deterministic finite-state machine. Then we define the reversed finite-state machine $rev(\mathcal{M})$ as $rev(\mathcal{M}) := (Q, \Sigma, rev(\Delta), F, Q_0)$ with

$$rev(\Delta) := \{(q_t, a, q_s) \mid (q_s, a, q_t) \in \Delta\}.$$

For any non-deterministic finite-state machine $\mathcal{M}$,

$$\mathcal{L}(rev(\mathcal{M})) = \overline{\mathcal{L}(\mathcal{M})}.$$

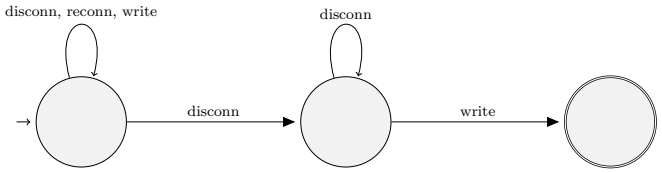
Our backward analysis operates on the state machine

$$\mathcal{M}_{bkwd} := det(rev(det(\mathcal{M}))) = det(rev(\mathcal{M}_{fwd})).$$

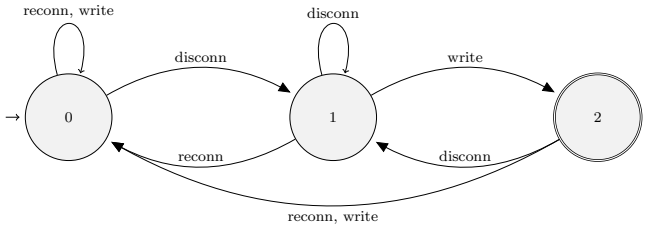
Note that $\mathcal{L}(\mathcal{M}_{bkwd}) = \overline{\mathcal{L}}$. Figure 8c shows the state machine that the backward analysis uses for the ConnectionClosed example. Note that the states of $\mathcal{M}_{bkwd}$ are actually subsets of the state set of $\mathcal{M}_{fwd}$; we labelled every state of $\mathcal{M}_{bkwd}$ with the corresponding state set of $rev(det(\mathcal{M}))$ (Figure 8b). Note that, for presentation purposes, we omitted the reject state from the Figure; the reject state represents the empty state set.

It would be sound to have the backward analysis operate on any state machine recognizing $\overline{\mathcal{L}}$, for instance on a determinized version of the reversed version of the non-deterministic $\mathcal{M}$. However, by determinizing, reversing, and determinizing $\mathcal{M}$, we automatically obtain a *minimal* deterministic finite-state machine for $\overline{\mathcal{L}}$. (See [Brzozowski 1962] for a proof.) Minimal deterministic finite-state machines yield additional optimization potential because they collapse together multiple equivalent states, yielding richer equivalence classes. (Section 7.3 elaborates on this point.)

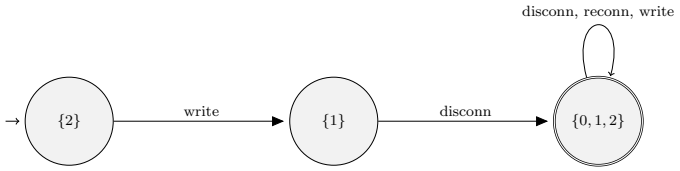
The forward analysis conceptually starts at the beginning of the program execution. The backward analysis, on the other hand, starts at every statement which



(a) Original non-deterministic finite-state machine for $\mathcal{L}$.



(b) $\mathcal{M}_{fwd}$, a deterministic finite-state machine for $\mathcal{L}$.



(c) $\mathcal{M}_{bkwd}$, a minimal deterministic finite-state machine for the mirror language $\overline{\mathcal{L}}$.

Fig. 8: Finite-state machines for Connection example.

potentially reaches a final state, i.e., at every shadow s such that $label(s) = l$ with an l-transition into a final state $q_F \in F$.

In Figure 7, we show how the states of $\mathcal{M}_{bkwd}$ evolve through the backward analysis. At first, the only label that can bring the ConnectionClosed monitor into a final state is a "write". The analysis therefore starts immediately before every write statement. The analysis then proceeds exactly as the forward analysis. For instance, starting in state $\{2\}$ and reading a "write" through the automaton in Figure 8c, the analysis infers that the next state, just before the write, is $\{1\}$. Due to the symmetries between the analyses, we have implemented the forward and backward analyses using a common code base; the only difference is in the inputs (state machines, control-flow graphs) that we provide to them.

7.3 Determining Nop shadows

Figure 7 presents our running example, now annotated with both the analysis information from the forward analysis (on the left) and from the backward analysis (on the right). The combined analysis information enables us to identify and disable nop shadows. Our notion of a nop shadow is related to the idea of *continuation-equivalent states*. We say that states q_1 and q_2 are *continuation-equivalent at a shadow* s, or simply *equivalent at* s, and write $q_1 \equiv_s q_2$, if, for all possible continuations of the control flow after s, the dependency state machine for s reaches its

final state at the same program points, whether we are in state q_1 or q_2 at s. We can formally define this equivalence relation as follows.

For every shadow s, call the sets of states computed by the backward analysis immediately after s the *futures* of s. Further, call the states computed by the forward analysis immediately before s the *sources* of s, and for every state q in $\mathit{sources}(s)$, let $\mathit{target}(q, s)$ be the target state reached after executing an s-transition from q. For instance, for the disconnect statement at line 5 of Figure 7 we have:

$$\begin{aligned} \mathit{futures}(\text{line 5}) &= \{\ \{\}, \{0,1,2\}\ \} \\ \mathit{sources}(\text{line 5}) &= \{0\} \\ \mathit{target}(0, \text{line 5}) &= 1 \end{aligned}$$

We further define continuation-equivalence for states as:

$$q_1 \equiv_s q_2 \quad := \quad \forall Q \in \mathit{futures}(s).\ q_1 \in Q \Leftrightarrow q_2 \in Q.$$

A shadow is a nop shadow when it transitions between states in the same equivalence class, unless the target state is an accepting state. (Because reaching a final state triggers the monitor, such transitions have an effect even though they switch between equivalent states.) Recall that F is the set of accepting, i.e., property-violating states of $\mathcal{M}_{fwd}$.

Definition 16. A shadow at a statement s is a *nop shadow* if:

$$\forall q \in \mathit{sources}(s).\ q \equiv_s \mathit{target}(q, s) \wedge \mathit{target}(q, s) \notin F.$$

The first conjunct states a nop-shadow transitions only between states that are in the same equivalence class. As mentioned above, however, if $\mathit{target}(q, s) \in F$, then the shadow triggers the runtime monitor. According to Clara's monitoring semantics, a monitor must signal repeated property violations every time the violation occurs—some monitors execute error-handling code. For instance, on "`c.close(); c.write(); c.write()`", the monitor should signal a violation after both "write" events. However, the second "write" event does not change the monitor's state; we have $2 = \mathit{target}(2, s) = 2$, so we need to explicitly handle such cases.

Examples. The disconnect statement at line 5 of Figure 7 has $\mathit{source}(s) = \{0\}$ and $\mathit{target}(0, s) = 1$. For both sets $Q_f \in \mathit{futures}(s) = \{\ \{\}, \{0,1,2\}\ \}$, $0 \in Q_f \Leftrightarrow 1 \in Q_f$. Consequently, we have $0 \equiv_s 1$. Because $1 \notin F$, s is a nop shadow.

The write statement at line 7 is different. Here, $\mathit{source}(s) = \{1\}$ and $\mathit{target}(1, s) = 2$. The set $Q_f = \{2\} \in \mathit{futures}(s)$ has $2 \in Q_f$ but $1 \notin Q_f$. Hence, $1 \not\equiv_s 2$, i.e., s is not a nop shadow and may therefore not be disabled.

Remarks. The condition for determining nop shadows subsumes a more restrictive condition: one can disable a shadow if it loops. When a shadow s loops then "$\forall q \in \mathit{source}(s).\ q = \mathit{target}(q, s)$", and hence $q \equiv_s \mathit{target}(q, s)$ obviously holds: identity implies equivalence.

The above definition of nop shadows also explains why it helps the backward analysis to use a minimal deterministic finite-state machine: in a minimal state machine, all (forward-)equivalent states are collapsed together. The collapsed state will be labeled with a larger set Q_f of $\mathcal{M}_{fwd}$ states than the un-collapsed sets would

have been. Hence, after collapsing equivalent states, more sets Q_f will contain both source and target states.

Need to re-iterate. Our example program contains several nop shadows. For instance, all shadows in lines 3–6 from Figure 7 are nop shadows, and indeed it is sound to disable any single shadow from that set. However, we can only remove shadows one-by-one: after disabling a shadow, we need to re-compute the analysis information for its containing method, because disabling a shadow changes the monitor's transition structure within the method. Note that in our example, disabling both disconn shadows at lines 5 and 6 is unsound: removing both shadows leads to the monitor not reaching its final state at line 7.

Algorithm 3 presents the main loop of the Nop-shadows Analysis. Our transformation proceeds greedily as follows: for each method m, it first computes forward and backward analysis results. Next, it searches for nop shadows in m, based on the analysis results. If m contains any nop shadows, we arbitrarily disable a nop shadow, re-run the flow-insensitive Orphan-shadows Analysis on shadows in m, and re-iterate the flow-sensitive Nop-shadows Analysis, disabling shadows until we find no more nop shadows. If we remove any shadows from m, we re-run the Orphan-shadows Analysis on all the shadows in the whole program before continuing with the next method. We re-iterate this entire process (over all shadow-bearing methods) until no further nop-shadows can be identified. On our benchmark set, we found that two iterations of the outer loop were always sufficient.

Algorithm 3 Main loop for Nop-shadows Analysis.

```
repeat
  for each method m still bearing enabled shadows do
    repeat
      compute forward and backward analysis results for m.
      if m contains nop shadows then
        arbitrarily choose and remove any one nop shadow.
        re-run Orphan-shadows Analysis on shadows from m.
      end if
    until no nop shadows remain in m.
    if we have removed any nop shadows from m then
      re-run Orphan-shadows Analysis on entire program.
    end if
  end for
until we failed to remove a nop shadow in the previous iteration
```

In our example, the algorithm would leave one of the shadows at lines 5–6, and the shadow at line 7—exactly the minimal set of shadows in this case.

Figure 9 shows how often we re-iterate the analysis of each method, summarizing results over all methods from our benchmark set where Nop-shadows Analysis applies. Observe that we iterate only a few times for the vast majority of cases—this number is bounded by the number of still-enabled shadows in the method—and there are only twelve cases in which we have to iterate more than

ten times. There was one method which required 78 re-iterations: `fillArray` in class `CompactArrayInitializer` of the bloat benchmark with the FailSafeIter tracematch. This method contains a large number of statements that modify a collection (an instruction stream).

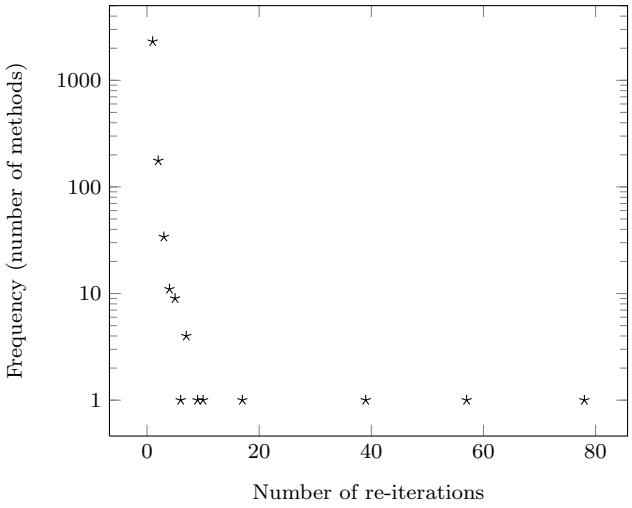


Fig. 9: Number of re-iterations per method (log scale).

Our simplified example ignored the following features of Java code:

(1) conditional control flow and loops,
(2) multiple methods with virtual dispatch,
(3) aliased objects, and
(4) more general specification patterns referring to more than one object.

In the following, we explain a general analysis that takes all of the above into account. As we will show, it is sound for any single-threaded Java program without reflection; in continuing work, we are investigating the use of dynamic information for circumscribing the potential impact of reflection on program behaviour.

7.4 Implementing the Nop-shadows Analysis

We next present a sound implementation of the function *necessaryTransition* for the Nop-shadows Analysis. We decided to implement the Nop-shadows Analysis using only intra-procedural analyses; each of our static analyses considers only a single method at a time. The reasons for this design choice are twofold. First, intra-procedural analyses almost always run more quickly than inter-procedural analyses, since they consider far less code. Our second reason is empirical. We manually investigated the still-active instrumentation points in our benchmarks following the application of the flow-insensitive Orphan-shadows Analysis, and found that, in most cases, intra-procedural analysis information suffices to rule out unnecessary instrumentation points, when combined with coarse-grained inter-procedural summary information available from the Orphan-shadows Analysis. It appears that

most procedures locally establish the conditions that they require to satisfy the type of conditions that we currently specify and verify with CLARA. The results presented in this paper confirm these findings.

The Nop-shadows Analysis rules out shadows on a per-method, per-state-machine basis. It first computes flow-sensitive alias information for each method. Next, it runs the forward and backward analyses seen earlier, which enable it to detect and remove nop shadows.

We continue by motivating the need for the flow-sensitive alias information. Recall that we defined the semantics of a dependency state machine over ground traces, which are projections of the single trace of parameterized events occurring at runtime. Our static analysis needs an analogue of projection to extract sub-traces for different variable bindings. We proceed by defining object representatives and then explain how they are used in binding representatives.

7.4.1 *Abstracting from objects with object representatives.* Runtime monitors associate automaton states with variable bindings, i.e., mappings from free variables declared in the dependency state machine to concrete runtime objects. However, concrete runtime objects are not available at compile time. We hence need to resort to a static abstraction of runtime objects.

We have developed a particular static abstraction, object representatives [Bodden et al. 2008b], which enable analyses to disambiguate object references at compile time by integrating different pointer analyses. At compile time, we model a runtime binding $x = o(\texttt{v1}) \wedge y = o(\texttt{v2})$ with a binding $x = r(\texttt{v1}) \wedge y = r(\texttt{v2})$, where $r(\texttt{vi})$ is the object representative of $o(\texttt{vi})$.

An object representative represents a (reference-typed) program variable at a specific program statement. Object representatives support must-not-alias and must-alias queries. A must-not-alias query between r_1 and r_2 returns **true** when r_1 and r_2 cannot represent the same runtime object; we write $r_1 \neq r_2$. We determine must-not-aliasing by combining flow-insensitive context-sensitive whole-program pointer information with an intra-procedural flow-sensitive must-not-alias analysis.

Similarly, a must-alias query between r_1 and r_2 returns **true** when r_1 and r_2 are guaranteed to always refer to the same run-time object; we write $r_1 = r_2$ in that case. We resolve must-alias queries using an intra-procedural flow-sensitive analysis only, and return **false** when r_1 and r_2 come from different methods or, more generally, when they are not known to must-alias.

Finally, when r_1 and r_2 neither must-alias nor must-not-alias, we say that they may-alias, and write $r_1 \approx r_2$. Table I gives an overview of our notation.

We call the set of all object representatives $\tilde{\mathcal{O}}$. For any subset $R \subseteq \tilde{\mathcal{O}}$ of object representatives, we define sets *mustAliases*(R) and *mustNotAliases*(R) as follows:

$$\begin{aligned} \textit{mustAliases}(R) &:= \{r' \in \tilde{\mathcal{O}} \mid \exists r \in R \text{ such that } r = r'\} \\ \textit{mustNotAliases}(R) &:= \{r' \in \tilde{\mathcal{O}} \mid \exists r \in R \text{ such that } r \neq r'\} \end{aligned}$$

7.4.2 *Abstracting from bindings with binding representatives.* At runtime, variable bindings map from variables to runtime objects. Object representatives do not suffice for representing variable bindings at compile time; we next describe our more informative compile time representation for bindings.

$r_1 \approx r_2$	Object representatives may-alias
$r_1 = r_2$	Object representatives must-alias
$r_1 \neq r_2$	Object representatives must-not-alias

Table I: Aliasing relations between object representatives

Binding representatives resemble the static bindings $\hat{\beta}$ that we used in the Orphan-shadows Analysis, but contain richer information: a binding representative contains both positive and negative information. The positive information records which objects a variable could possibly be bound to. The negative information, on the other hand, records which objects a variable cannot be bound to.

We define a binding representative $b \in \tilde{\mathcal{B}}$ as a pair (β^+, β^-) containing a positive binding β^+ and a negative binding β^-. Both binding functions map a Dependency State Machine's free variables to sets of object representatives. We naturally extend both partial binding functions to total functions by mapping a free variable v to $\emptyset$ when no other mapping for v is defined.

Consider the binding representative,

$$(\{x \mapsto \{r_1, r_2\}, y \mapsto \{r_3\}\}, \{x \mapsto \{r_4\}\}).$$

This binding representative states that x can only be bound to objects represented by both r_1 and r_2, and can certainly not be bound to objects represented by r_4. (We can deduce $r_1 \approx r_2$: if $r_1 \neq r_2$, then x could not simultaneously be bound to both r_1 and r_2, while if $r_1 = r_2$, then we would only store either r_1 or r_2.) Further, y can only be bound to objects represented by r_3.

We sometimes choose to write binding representatives as a conjunction of equations. For instance, one can write the above binding representative as:

$$x = r_1 \wedge x = r_2 \wedge y = r_3 \wedge x \neq r_4.$$

Such equations allow us to perform Boolean arithmetic on bindings. For instance, if we additionally know $r_1 \neq r_2$, i.e., r_1 and r_2 must-not-alias, then:

$$x = r_1 \wedge x = r_2 \equiv \textbf{false}.$$

Certain implications that hold in Boolean logic do not, however, hold in our three-valued logic. For instance, $\neg(r_1 = r_2)$ does not imply $r_1 \neq r_2$. Even if two object representatives do not must-alias, they need not must-not-alias: they could also may-alias.

Table II summarizes applicable simplifications for binding representatives of the form $(x = / \neq r_1) \wedge (x = / \neq r_2)$. Note that, if we know that $r_1 = r_2$ or $r_1 \neq r_2$, then we can always simplify the resulting binding representative, except for $(x \neq r_1) \wedge (x \neq r_2)$ (shown in gray), where we need to store the full binding representative. Also, if r_1 and r_2 may-alias (Table IIc), then we must always store the full binding representative.

The simplifications in Tables IIb and IIa allow us to recognize impossible bindings by reducing self-contradictory binding representatives to **false**. As we will see, this helps avoid false positives, thereby increasing precision.

$\mathbf{r_1 \neq r_2}$	$x = r_1$	$x \neq r_1$
$x = r_2$	**false**	$x = r_2$
$x \neq r_2$	$x = r_1$	$x \neq r_1 \wedge x \neq r_2$

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

$\mathbf{r_1 = r_2}$	$x = r_1$	$x \neq r_1$
$x = r_2$	$x = r_1 \equiv x = r_2$	**false**
$x \neq r_2$	**false**	$x \neq r_1 \equiv x \neq r_2$

(b) Resulting binding representative when r_1 and r_2 must-alias

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

(c) Resulting binding representative when r_1 and r_2 may-alias

Table II: Simplifications of binding representatives using alias information.

For any variable-to-object representative binding $\beta : \mathcal{V} \to \tilde{\mathcal{O}}$, and any binding representative $b = (\beta^+, \beta^-)$, we can determine whether β is compatible with b:

$$\begin{aligned} \mathit{compatible}(\beta, (\beta^+, \beta^-)) \;:=\; \nexists v \text{ such that } & \beta(v) \in \mathit{mustNotAliases}(\beta^+(v)) \\ & \vee\ \beta(v) \in \mathit{mustAliases}(\beta^-(v)). \end{aligned}$$

In other words, β is incompatible with b if β binds some variable v to an object representative that must-not-aliases some object representative in v's positive binding, or if some v must-aliases some object representative in v's negative binding. Note that when β is empty, i.e., binds no variables at all, β will be compatible with any binding representative.

Shadows may also be compatible with binding representatives. Every shadow s induces a variable binding $\beta_s = \mathit{shadowBinding}(s)$ of type $\mathcal{V} \to \tilde{\mathcal{O}}$. Hence, in the following we will often write $\mathit{compatible}(s, b)$ in place of $\mathit{compatible}(\beta_s, b)$.

Using the notion of compatibility, we can define an inclusion relation on binding representatives. Let b_1 and b_2 be two binding representatives. We say that b_2 is *at least as permissive* as b_1, or $b_1 \subseteq_{\tilde{\mathcal{B}}} b_2$, if the following holds:

$$b_1 \subseteq_{\tilde{\mathcal{B}}} b_2 \quad :\Longleftrightarrow \quad (\ \forall \beta.\ \mathit{compatible}(\beta, b_1) \rightarrow \mathit{compatible}(\beta, b_2)\ )\,.$$

That is, $b_1 \subseteq_{\tilde{\mathcal{B}}} b_2$ if for every variable v, every object o that can be bound to v according to b_1 can also be bound to o according to b_2.

We define *strictly more permissive*, or $\subset_{\tilde{\mathcal{B}}}$, in terms of $\subseteq_{\tilde{\mathcal{B}}}$, as follows:

$$b_1 \subset_{\tilde{\mathcal{B}}} b_2 \quad :\Longleftrightarrow \quad b_1 \neq b_2 \wedge b_1 \subseteq_{\tilde{\mathcal{B}}} b_2.$$

We will denote the empty binding representative, in which both binding functions β^+ and β^- are undefined for all variables, by $\top$. Note that

$$\forall b \in \tilde{\mathcal{B}} :\ b \subseteq_{\tilde{\mathcal{B}}} \top.$$

7.4.3 *The worklist algorithm.* We next describe the forward and backward analyses that, together, enable us to identify nop shadows. Algorithm 4 presents the worklist algorithm which implements our analyses. Our forward and backward analyses both compute sets of possible configurations before and after each statement.

A configuration (Q_c, b_c) is an element of $\mathcal{P}(Q) \times \tilde{\mathcal{B}}$, i.e., a configuration combines a set $Q_c \subseteq Q$ of automaton states with a binding representative b_c. The underlying state set Q is the state set of $\mathcal{M}_{fwd}$; the forward and backward analysis operate on the same state set, but use different (reversed) transition functions.

Algorithm 4 *worklist*$(initial, succ_{cfg}, succ_{ext}, \delta)$

The syntax $f[x \mapsto y]$ denotes the function that is equal to f on all values v, except for x, in which case it returns y:

$$f[x \mapsto y] := \lambda v. \begin{cases} y & \text{if } v = x \\ f(v) & \text{otherwise} \end{cases}$$

```
1:  wl := initial
2:  before := after := λstmt. ∅ // associate ∅ with every statement
3:  while wl non-empty do
4:    pop job (stmt, cs) from wl
5:    // reduce configurations so that only most permissive ones remain
6:    cs_temp := cs ∪ before(stmt)
7:    cs_new := {(Q_c, b_c) ∈ cs_temp | ∄(Q_c, b'_c) ∈ cs_temp. b_c ⊂_B̃ b'_c} \ before(stmt)
8:    cs' := { cs_new   if shadows(stmt) = ∅  // no changes
             { ∅         otherwise  // initialize to empty; we'll fill below
9:    for c ∈ cs_new, s ∈ shadows(stmt) do
10:     cs' := cs' ∪ transition(c, s, δ)
11:   end for
12:   before := before[stmt ↦ before(stmt) ∪ cs]
13:   cs'_new := cs' \ after(stmt) // filter out configurations already computed
14:   if cs'_new non-empty then
15:     after := after[stmt ↦ after(stmt) ∪ cs'_new]
16:     // add jobs for intra-procedural successor statements
17:     for stmt' ∈ succ_cfg(stmt) do
18:       wl := wl[stmt' ↦ wl(stmt') ∪ cs'_new]
19:     end for
20:     // add jobs for inter-procedural successor statements
21:     for stmt' ∈ succ_ext(stmt) do
22:       wl := wl[stmt' ↦ wl(stmt')
                    ∪ reachingStar (cs'_new, relevantShadows (stmt))]
23:     end for
24:   end if
25: end while
```

The algorithm first initializes a worklist *wl* as described below in Section 7.4.5. The worklist contains jobs $(stmt, cs)$, which map from statements to sets of configurations. For every statement *stmt*, *wl* contains a set of configurations reaching *stmt* which require successor configurations. The worklist is empty if it maps every statement to the empty set. The algorithm also initializes two mappings *before* and *after* to store previously-computed configurations. These mappings allow us to perform a terminating fixed point iteration.

Lines 6–7 implement an important optimization: they prune subsumed configurations. The algorithm first computes the union cs_{temp} of the old *before* set and the configurations that need to be computed at the current statement, according to the job popped from the worklist. Because the worklist maps statements *stmt* to jobs, the current job is the only job for *stmt*, so the set cs_{temp} holds all information computed so far for *stmt*. At line 7, the algorithm removes from cs_{temp} subsumed configurations (Q_c, b_c), where cs_{temp} also contains a configuration (Q_c, b'_c) with b'_c strictly more permissive than b_c. (Any shadow that is compatible with b_c will also be compatible with b'_c. Hence, if (Q_c, b_c) causes a shadow to be identified as a *necessary* shadow, then so will (Q_c, b'_c).) In our experiments, this optimization often reduced the number of configurations computed for a method by two to three orders of magnitude.

Finally, line 7 also removes from the resulting set all configurations contained in the *before* set and places the result into cs_{new}. This is sound because the algorithm would have already computed successor configurations for these configurations (at lines 9–11) and there is no need to compute the same information again.

Next, the algorithm computes, for every new configuration $c \in cs_{new}$ and every shadow s at *stmt*, successor configurations, using a function *transition* (described below). The algorithm then updates the statement's *before* set and checks (line 14) if any new configurations arose at *stmt*. If so, the algorithm propagates these configurations to appropriate successor statements. At lines 16–19, the algorithm adds new jobs containing the successor configurations cs'_{new} for any statement that is a successor of *stmt* in m's control-flow graph (as determined by $succ_{cfg}$). Lines 20–23 handle inter-procedural control flow which may transitively return to the current method m. We will explain the inter-procedural part of Algorithm 4 in detail in Section 7.4.4.

The transition function. Algorithm 5 implements our transition function. For a given configuration and shadow, the algorithm computes a set *cs* of successor configurations. Our implementation directly mirrors Avgustinov et al.'s implementation of the tracematch runtime [Allan et al. 2005]. The transition function δ computes the set Q_t of target states from the shadow's label l and the incoming states Q_c. For the forward analysis, δ is the transition function of $\mathcal{M}_{fwd}$, and for the backward analysis, it is the transition function of $\mathcal{M}_{bkwd}$.

The remaining part of Algorithm 5 handles variable bindings. At runtime, the event induced by shadow s changes the states of runtime monitors compatible with β_s from Q_c to Q_t. The resulting variable binding after a change is $b_c \wedge \beta_s$. The monitors for all variable bindings incompatible with β_s remain in Q_c. Hence, the variable bindings that remain in Q_c are $b_c \wedge \neg\beta_s$. Lines 4–7 compute successor configurations for all the variable bindings that move to Q_t, using the function *and*. In lines 9–10, the algorithm creates configurations for all these variable bindings that remain in Q_c, using the function *andNot*. We explain both functions below.

Note that the algorithm applies *andNot* for each bound variable v separately, following the tracematch runtime [Allan et al. 2005]. Consider a shadow s with a

Algorithm 5 *transition*$((Q_c, b_c), s, \delta)$

1: $cs := \emptyset$ // initialize result set
2: $l := \mathit{label}(s), \beta_s := \mathit{shadowBinding}(s)$ // extract label and bindings from s
3: $Q_t := \delta(Q_c, l)$ // compute target states
4: $\beta^+ := \underline{\mathit{and}}(b_c, \beta_s)$ // compute configurations for objects moving to Q_t
5: **if** $\beta^+ \neq \bot$ **then**
6: $\quad cs := cs \cup \{(Q_t, \beta^+)\}$
7: **end if**
8: // compute configurations for objects staying in Q_c
9: $B^- := \bigcup_{v \in dom(\beta_s)} \{\ \underline{\mathit{andNot}}(b_c, \beta_s, v)\ \} \setminus \{\bot\}$
10: $cs := cs \cup \{(Q_c, \beta^-) \mid \beta^- \in B^-\}$
11: **return** cs

variable binding β_s which binds two variables, e.g. $x = r(\texttt{v1}) \wedge y = r(\texttt{v2})$. Then:

$$\begin{aligned} \beta^- &\equiv b_c \wedge \neg\beta_s \\ &\equiv b_c \wedge \neg(x = r(\texttt{v1}) \wedge y = r(\texttt{v2})) \\ &\equiv (b_c \wedge \neg x = r(\texttt{v1})) \vee (b_c \wedge \neg x = r(\texttt{v2})). \end{aligned}$$

Since our abstraction stores all information in Disjunctive Normal Form, we must therefore return multiple configurations in this case, one for every disjunct.

Algorithms *and* and *andNot* use the simplification rules from Table II to (1) return $\bot$ whenever the abstraction allows us to conclude that b_c and β_s are incompatible, and (2) minimize the number of bound object representatives in the resulting binding representative, without losing soundness or precision. Returning $\bot$ means that the current configuration will not be propagated any further (see Algorithm 5, lines 5 and 9). This is an essential contribution to the precision of our analysis. Minimizing the number of bound object representatives leads to a smaller abstraction and to a smaller number of possible configurations, thus enabling earlier termination of the worklist algorithm, Algorithm 4.

We present our implementation of *and*, which adds bindings β_s to a binding representative (β^+, β^-), in Algorithm 6. For every variable v bound by β_s, the algorithm compares the existing positive and negative bindings for v with the object representative $\beta_s(v)$. The bindings are incompatible if $\beta_s(v)$ must-not-aliases some object representative in $\beta^+(v)$, or if it must-aliases some object representative in $\beta^-(v)$. The algorithm returns $\bot$ for incompatible bindings. Next, in line 7, the algorithm refines the positive binding by adding $\beta_s(v)$ to $\beta^+_{new}(v)$. We consider three cases. When $\beta_s(v) \in \mathit{mustAliases}(\beta^+_{new}(v))$ already, then it will not be added to the set again (by the design of the implementation). The case $\beta_s(v) \in \mathit{mustNotAliases}(\beta^+(v))$ was excluded above. Hence, $\beta_s(v)$ will only be added if it may-aliases all object representatives for v. Finally, in line 9, the algorithm prunes superfluous negative bindings. For instance, if $\beta_s = x \mapsto r(\texttt{v})$ was just added to $\beta^+_{new}(v)$, then the information $x = r(\texttt{v})$ implies that $x \neq r^-$ for all r^- where $r^- \neq r(\texttt{v})$. Hence we can remove such object representatives r^- from $\beta^-_{new}(v)$—such statements are implied by $x = r(\texttt{v}) \in \beta^+_{new}(v)$. (This corresponds

Algorithm 6 *and*$((\beta^+, \beta^-), \beta_s)$

```
1: β⁺_new := β⁺, β⁻_new := β⁻
2: for v ∈ dom(β_s) do
3:    if β_s(v) ∈ mustNotAliases(β⁺(v)) ∨ β_s(v) ∈ mustAliases(β⁻(v)) then
4:       return ⊥ // bindings were incompatible
5:    end if
6:    // add new positive binding
7:    β⁺_new := β⁺_new[v ↦ β⁺_new(v) ∪ {β_s(v)}]
8:    // prune superfluous negative bindings
9:    β⁻_new := β⁻_new[v ↦ β⁻_new(v) \ {r⁻ | r⁻ ≠ β_s(v)}]
10: end for
11: return (β⁺_new, β⁻_new)
```

to the top right and bottom left cells of Table IIa.)

We implemented *andNot*, which updates (β^+, β^-) with the fact that $v \in dom(\beta_s)$ no longer binds $\beta_s(v)$, as shown in Algorithm 7. First, if $\beta_s(v)$ must-aliases any object representative from v's positive binding, then the bindings are incompatible, and the algorithm returns $\bot$. Otherwise, the algorithm adds $\beta_s(v)$ to the negative bindings and returns the updated binding representative. However, as with *and*, we avoid adding redundant negative information when we know that $\beta_s(v)$ already must-not-aliases some positive binding for v.

This concludes the description of our transition function. We return to our explanation of the worklist algorithm, Algorithm 4.

7.4.4 *The external-successor function succ$_{ext}$.* Lines 20–23 of Algorithm 4 handle inter-procedural control flow. Figure 10 illustrates the analysis of a potentially-recursive method m, represented by the central dark rectangle. The dashed arrows denote the successor function *succ$_{cfg}$* given by m's control-flow graph. The solid arrows represent a second, inter-procedural, successor function *succ$_{ext}$*. We've assumed that m includes method calls, which can potentially recursively call back to m itself; the recursion may be indirect, through intermediaries in the call graph.

Call the set of potentially-recursive call sites C. Then, configurations that we computed for any $c \in C$ must propagate to m's entry statement (edge (1)) through recursion. We also need to propagate configurations coming into c to its control-flow

Algorithm 7 *andNot*$((\beta^+, \beta^-), \beta_s, v)$

```
1: if β_s(v) ∈ mustAliases(β⁺(v)) then
2:    return ⊥ // bindings were incompatible
3: end if
4: // negative binding redundant if it must-not-aliases a positive binding
5: if β_s(v) ∈ mustNotAliases(β⁺(v)) then
6:    return ( β⁺, β⁻ )
7: else // return updated binding
8:    return ( β⁺, β⁻[v ↦ β⁻(v) ∪ {β_s(v)}] )
9: end if
```

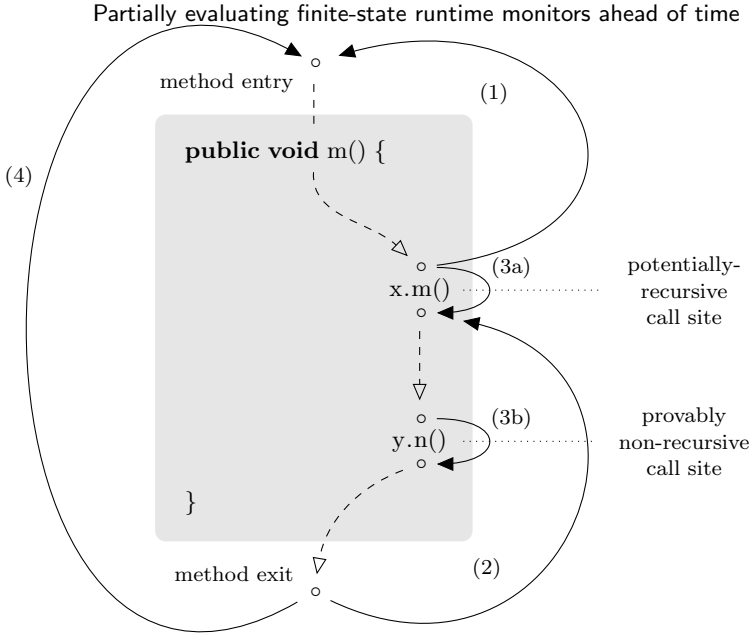


Fig. 10: Inter-procedural control-flow for the current method m.

successor (edge (3a)), accounting for the case where c does not infinitely recurse. Furthermore, configurations that we computed for any of m's exit statements must also propagate to all potentially-recursive call sites $c' \in C$ in m (edge (2)). For provably non-recursive call sites c'' (as determined by our call graph), we only propagate configurations from c'' to its control-flow graph successor, but not to m's entry statement (edge (3b)). Finally, we account for the case where multiple recursive calls to m occur within a recursive call by propagating configurations from m's exit statement(s) to its entry statement, if m has any potential recursion (edge (4)).

We formally define the function $succ_{ext}$ as follows. Let $heads(m)$ be the set of entry statements of m, and $tails(m)$ the set of exit statements of m[3]. Further, let $recCall(m)$ be the set of statements of m that contain an invoke expression through which m can potentially call itself recursively. Conversely, $nonRecCall(m)$ contains all statements that contain an invoke expression through which m can certainly not be called. Then:

$$succ_{ext} := \lambda stmt. \begin{cases} heads(m) \cup succ_{cfg}(stmt) & \text{if } stmt \in recCall(m) \\ succ_{cfg}(recCall(m)) \cup heads(m) & \text{if } stmt \in tails(m) \\ succ_{cfg}(stmt) & \text{if } stmt \in nonRecCall(m) \\ \emptyset & \text{otherwise} \end{cases}$$

Observe that we add some edges from $succ_{cfg}$ to $succ_{ext}$. When propagating configurations along an edge of $succ_{ext}$, we do more than just copy configurations from

[3]Because our backward analysis operates on a reversed control-flow graph, $heads(m)$ for that analysis—the tails of the input control-flow-graph—can contain more than one element.

the edge's source statement to its target statement: while executing an external successor edge of m, other methods may also execute and cause state transitions in the monitoring state machine. To model these potential state transitions through other methods, line 22 of Algorithm 4 adds $\mathit{reachingStar}(cs'_{new}, \mathit{relevantShadows}(stmt))$ as well as the ordinary set of configurations cs'_{new} for inter-procedural successors.

The functions relevantShadows, reachingPlus and reachingStar. We next define three helper functions. *reachingStar* computes successor configurations after 0 or more relevant shadow executions, using the flow-insensitive analysis information computed by the Orphan-shadows Analysis. *reachingPlus* computes successors after 1 or more shadow executions. Both of these functions summarize the effects of the shadows in *relevantShadows*.

We first define *relevantShadows*. If *stmt* contains an invoke expression, then *relevantShadows*(*stmt*) contains all shadow-bearing statements in all methods transitively reachable through the method invocation, except for statements from m itself. We exclude statements from m because we have explicitly accounted for the effects of the shadows in m by adding the $succ_{ext}$ edges to the heads and tails. If *stmt* is a head or tail of m, then *relevantShadows*(*stmt*) contains all shadow-bearing statements in the entire program, except for the ones in m. Unfortunately, we do not know which methods execute before or after m (even with the call graph), and hence we have no way of soundly reducing this set any further.

Given a statement *stmt* and a set cs of configurations just before *stmt*, the function *reachingPlus* computes the set of configurations after executing at least one shadow at a statement in *relevantShadows*(*stmt*). When a configuration $c = (Q_c, b_c)$ reaches an exit point or a recursive call site during the analysis of method m, all relevant shadows may perform transitions on c. However, only shadows that are compatible with b_c can actually cause the state set Q_c to change. Hence, for every binding representative $b \in \vec{\mathcal{B}}$ and shadow set $ss \subseteq \mathcal{S}$, we define the set $\mathit{compL}(b, ss)$ as:

$$\mathit{compL}(b, ss) := \{a \in \Sigma \mid \exists s \in ss \text{ such that } \mathit{compatible}(s, b) \wedge \mathit{label}(s) = a\}.$$

This set contains the labels of all shadows in ss compatible with b.

We then define $\mathit{reachingPlus}(cs, stmt)$ as the set of configurations reachable from cs by applying at least one compatible shadow which is relevant to *stmt*. Formally, $\mathit{reachingPlus}(cs, stmt)$ is the least fixed point satisfying:

- $(Q_c, b_c) \in cs \wedge l \in \mathit{compL}(b_c, ss) \implies (\delta(Q_c, l), b_c) \in \mathit{reachingPlus}(cs, stmt)$
- $(Q_c, b_c) \in \mathit{reachingPlus}(cs, s) \wedge l \in \mathit{compL}(b_c, ss) \implies (\delta(Q_c, l), b_c) \in \mathit{reachingPlus}(cs, stmt)$.

We further define *reachingStar* as the reflexive closure of *reachingPlus*:

$$\mathit{reachingStar}(cs, stmt) := \mathit{reachingPlus}(cs, stmt) \cup cs.$$

Hence, *reachingStar* computes the set of configurations that one can reach from cs by executing 0 or more shadows that are relevant at *stmt*.

7.4.5 *Initializing the worklist algorithm.* We next explain how we initialize Algorithm 4, which takes four parameters: *initial*, $succ_{cfg}$, $succ_{ext}$ and δ. In forward-

analysis mode, $succ_{cfg}$ is simply the successor function of m's control-flow graph, $succ_{ext}$ is the inter-procedural successor function defined above, and δ is the transition function of $\mathcal{M}_{fwd}$. For the backward analysis, we simply invert the two successor functions, and use the transition function of $\mathcal{M}_{bkwd}$ for δ.

We still need to define the set *initial* of initial configurations, which serves to initialize the worklist from Algorithm 4. We show the initialization, along with subsequent fixed-point iterations, for the forward analysis in Figure 11a. The "iteration" box is described in Algorithm 4; for now, consider only the "initialization" box.

We may assume, without loss of generality, that we are analyzing the first invocation of method m; Algorithm 4 accounts for subsequent executions of m with the loop in line 20–23. The first invocation of m enters the method through its first statement (its head). The configurations that can reach m's head are those that arise starting with the initial state set Q_0 and executing any shadows outside of m, in any order, for any variable binding (i.e., for $\top$). Hence, for the forward analysis we define:

$$initial := \{\ (h, reachingStar(\{(Q_0, \top)\}, relevantShadows(h))) \mid h \in heads(m)\ \}\,.$$

We now give *initial* for the backward analysis. Our goal is to create jobs associating statements *stmt* with configurations c from which the remainder of the execution (including the execution of *stmt* itself) could lead into a final state.

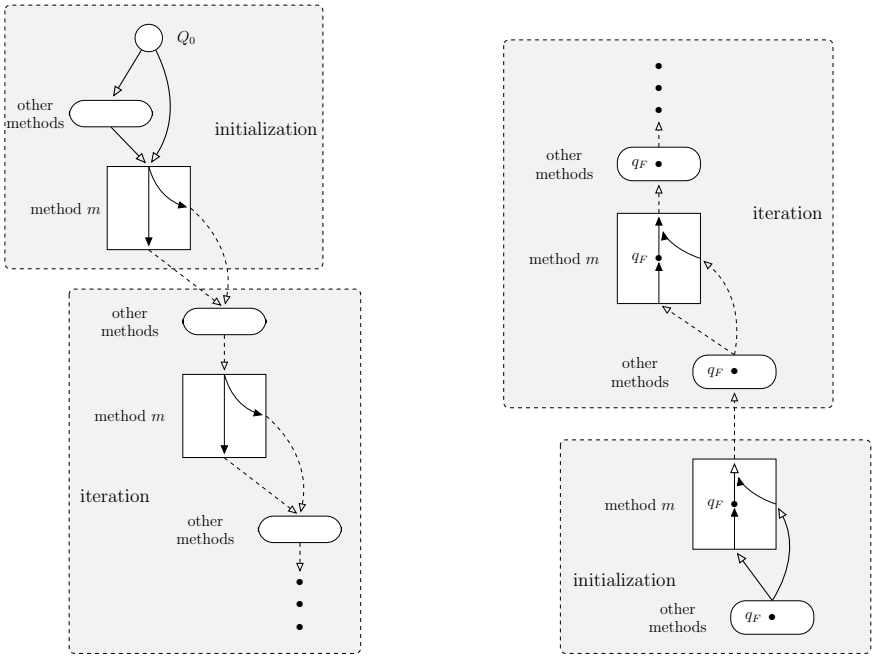
Figure 11b illustrates the initialization and iteration. Dually to the forward analysis, we assume that m will not be executed again *after* its current execution.

The simplest case leading to a final state is intraprocedural. We must initialize a job when there is a "final" shadow in m itself—a shadow s labeled with a label $l = label(s)$ such that there exists an l-transition into a final state $q_F \in F$.

However, we also need to consider the case where m executes completely and the remainder of the execution drives the configuration into a final state using shadows in methods other than m. We therefore create jobs associating any tail statement *stmt* of m with any configuration c in $reachingPlus(\{F\}, relevantShadows(stmt))$. This is almost dual to the initialization for the forward analysis, except that we use *reachingPlus*, not *reachingStar*: we cannot reach a final state in F from any of m's tail statements if there are no shadows in any other methods at all. *reachingPlus* only includes configurations that reach in at least one step, as required here.

The last case is where a callee of m reaches a final state. Let this callee be invoked at *stmt*. We again associate *stmt* with $reachingPlus(\{F\}, relevantShadows(stmt))$. Here, $relevantShadows(stmt)$ will contain all shadows reachable through the call site *stmt*, rather than all shadows in methods other than m.

We hence initialize the backward analysis with the union of two sets. One set holds configurations that lead into a final state within m, while the other set holds configurations that could into a final state outside of m:



(a) Configurations reaching a method m's first statement, and handling of inter-procedural information.

(b) Paths along which configurations can propagate a final state q_F to a statement in m.

Fig. 11: Initialization and iteration of forward and backward passes.

$$
\begin{aligned}
&\mathit{initial} := \\
&\quad \text{let } \mathit{reachingConfigs} = \mathit{reachingPlus}(\{(F, \top)\}, \mathit{relevantShadows}(\mathit{stmt})) \text{ in:} \\
&\qquad \{\ (\mathit{stmt}, \{(F, \top)\}) \mid \exists s \in \mathit{shadowsOf}(\mathit{stmt}) :\ \delta(F, \mathit{label}(s)) \neq \emptyset)\} \cup \\
&\qquad \{\ (\mathit{tail}, \mathit{reachingConfigs}) \mid \mathit{tail} \in \mathit{tails}(m) \cup \mathit{recCall}(m) \cup \mathit{nonRecCall}(m)\ \}
\end{aligned}
$$

Note that δ above denotes the transition function of $\mathcal{M}_{bkwd}$, while *tails* denotes the return statements of m.

7.5 Optimizations for faster analysis

In Section 7.4.3 we discussed one important optimization that eliminated configurations that were less permissive than other configurations at the same statement. That optimization often reduced the number of configurations computed for a method by two to three orders of magnitude. We now describe other optimizations which also decrease the analysis time.

Abstracted call graph. The call graph that we use to identify all shadows in the transitive closure of an outgoing method call abstracts the call graph computed by the points-to analysis in Spark [Lhoták and Hendren 2003]; we omit paths that never reach a shadow-bearing method. This accelerates graph look-ups. In particular, if a method invocation cannot transitively call any shadow-bearing methods, then

the call graph will not have any call edge for the invocation and the analysis can identify the call as harmless in constant time. In our benchmark set, abstracting the call graph was highly effective: on average, the abstracted call graph had only about 4.3% of the edges of the complete call graph, with 12.9% in the worst case (12984 remaining edges in bloat-FailSafeIter), and 0.02% (26 edges in fop-HasNext) in the best case.

Caching. We cache results extensively. For instance, we cache points-to sets, the results of the must-alias and must-not-alias analysis for every method, and the set of methods transitively reachable through a method call. We also cache the set of currently-enabled shadows in methods. Note that it could be imprecise to naïvely cache this set: when the analysis disables a shadow, it must be removed from the set; furthermore, the removal must be visible when analyzing other methods later on. To avoid having to re-compute the set of currently-enabled shadows, we store sets of enabled shadows using a class `EnabledShadowSet`. When a shadow s is added to such a set, the set automatically registers itself with the shadow as a listener. If the analysis disables s later on, s notifies all registered listeners, which then update themselves by removing s. In addition, if the analysis attempts to add a shadow s to an `EnabledShadowSet` but s is already disabled, the shadow is not actually added (nor registered).

Aborting overly long analysis runs. Despite these optimizations, the Nop-shadows Analysis still takes a long time to finish on a small number of methods. Figure 12 summarizes method `peelLoops(int)` from `EDU.purdue.cs.bloat.cfg.FlowGraph` of the benchmark bloat. For this benchmark, our context-sensitive points-to analysis fails to compute context information for the iterators and collections. Hence, when analyzing this method with respect to the FailSafeIter monitor, the Nop-shadows Analysis gets the imprecise information $r(\texttt{i1}) \approx r(\texttt{i2}) \approx r(\texttt{i3})$, i.e., it has to assume that `i1`, `i2` and `i3` could all point to the same iterator. This leads to a large number of possible configurations. Assume that we have a configuration with a binding representative $b := c = r(\texttt{c1}) \wedge i = r(\texttt{i1})$, and we want to compute $b \wedge i = r(\texttt{i2})$. If we did have precise points-to information then this would tell us that $r(\texttt{i1}) \neq r(\texttt{i2})$ (because we know that both iterators cannot be the same) and hence we would get:

$$\begin{aligned} b \wedge i = r(\texttt{i2}) &\equiv c = r(\texttt{c1}) \wedge i = r(\texttt{i1}) \wedge i = r(\texttt{i2}) \\ &\equiv c = r(\texttt{c1}) \wedge \mathbf{false} \\ &\equiv \mathbf{false} \end{aligned}$$

However, because we only know $r(\texttt{i1}) \approx r(\texttt{i2})$, the analysis fails to reduce "$c = r(\texttt{c1}) \wedge i = r(\texttt{i1}) \wedge i = r(\texttt{i2})$" further. With the many consecutive loops in `peelLoops(int)`, this drastically increases the size and number of configurations to be computed before the analysis reaches its fixed point. Worse yet, because of the imprecise pointer information, the analysis fails to identify any nop shadows.

We therefore recorded the maximal number of configurations computed on a successful (i.e. nop shadow-detecting) analysis run in any of our benchmarks. This occurred in `visitBlock(Block)` of class `EDU.purdue.cs.bloat.cfg.VerifyCFG`, not quite coincidentally in the same benchmark. The analysis computed 8828 con-

```
1  void foo(Collection c1, Collection c2, Collection c3)
2      Iterator i1 = c1.iterator();
3      while(i1.hasNext()) {
4          i1.next();
5          c2.add(..);
6      }
7      Iterator i2 = c2.iterator();
8      while(i2.hasNext()) {
9          i2.next();
10         c3.add(..);
11     }
12     Iterator i3 = c3.iterator();
13     while(i3.hasNext()) {
14         i3.next();
15     }
16 }
```

Fig. 12: Worst-case example for complexity of Nop-shadows Analysis (in the case of imprecise points-to sets).

figurations before it removed a shadow from this method. We then modified the Nop-shadows Analysis so that it would abort the analysis of a single method (thus continuing with the next method) whenever it computed more than a fixed quota of configurations. We defined this quota to be 15000, which still accommodates a significant number of additional configurations beyond the 8828 that we observed. We believe that this value is high enough to yield excellent precision in cases where pointer information is precise; our experiments also showed that it is low enough to significantly decrease the overall analysis time in the benchmarks bloat-FailSafeIter, bloat-FailSafeIterMap and pmd-FailSafeIterMap. (In all other cases, the quota never exceeded 15000 and CLARA aborted no analysis runs.)

Additionally, the benchmarks bloat, chart and pmd all use reflection in connection with collections. For instance, Figure 13 shows a simplified version of a `clone` method in chart. A shortcoming in the Java specification means that, even if an object implements the `Cloneable` interface, the object is not required to implement a publicly-accessible `clone` method. The chart developers work around this shortcoming by calling the `clone` method reflectively when it exists. Such reflection confuses the Spark points-to analysis: the analysis has no idea which class's clone method will be called—reflection is not modeled precisely enough in Spark. As a result, there are many possible `clone` implementations to consider and the demand-driven analysis (see Section 6.2) fails to compute context in its given quota.

7.6 Soundness of Nop-shadows Analysis

Recall that Section 4 defined the semantics of dependency state machines and provided soundness constraints for the predicates *necessaryTransition*. Any constraint-respecting implementation of *necessaryTransition* implies a sound analysis: if the predicate holds, the analysis will not affect the runtime behaviour of the specified monitors. We now show that the Nop-shadows Analysis respects *necessaryTransition*. To restate the soundness condition, any sound implementation of *necessaryTransition*

```
1  public static Object clone(final Object object) throws CloneNotSupportedException
   {
2      if (object == null) {
3          throw new IllegalArgumentException("Null 'object' argument.");
4      }
5      if (object instanceof PublicCloneable) {
6          final PublicCloneable pc = (PublicCloneable) object;
7          return pc.clone();
8      } else {
9          final Method method = object.getClass().getMethod("clone",(Class[]) null);
10         if (Modifier.isPublic(method.getModifiers())) {
11             return method.invoke(object, (Object[]) null);
12         }
13     }
14     throw new CloneNotSupportedException("Failed to clone.");
15 }
```

Fig. 13: Clone method in chart using reflection.

must respect:

$$\forall a \in \Sigma.\ \forall t = t_1 \dots t_i \dots t_n \in \Sigma^+.\ \forall i \in \mathbb{N}.\\ a = t_i \wedge \mathit{matches}(t_1 \dots t_n) \neq \mathit{matches}(t_1 \dots t_{i-1} t_{i+1} \dots t_n)\\ \implies \mathit{necessaryTransition}(a, t, i)$$

Helper definitions. We will denote the transitive closure of a transition function δ by $\widehat{\delta}$. Also, we define the variant finite-state machine $\mathcal{M}_q$ to be the state machine $\mathcal{M} = (Q, \Sigma, q_0, \delta, F)$ with an alternate initial state $q \in Q$, i.e. $\mathcal{M}_q := (Q, \Sigma, q, \delta, F)$.

Soundness of shadow removal for Nop-shadows Analysis. Assume that the Nop-shadows Analysis disables a shadow s that triggers the i-th event with $\mathit{label}(s) = t_i$. We will prove that, by construction,

$$\mathit{matches}(t_1 \dots t_{i-1} t_i t_{i+1} \dots t_n) = \mathit{matches}(t_1 \dots t_{i-1} t_{i+1} \dots t_n), \tag{1}$$

implying the soundness condition.

Consider a projected and therefore ground runtime trace $t = t_1 \dots t_i \dots t_n \in \Sigma^+$. For convenience, define $w_1 := t_1 \dots t_{i-1}$, $a := t_i$ and $w_2 := t_{i+1} \dots t_n$, i.e., we have that $t_1 \dots t_n = w_1 a w_2$. Let $\mathit{source} := \widehat{\delta}(q_0, w_1)$ and $\mathit{target} := \widehat{\delta}(q_0, w_1 a) = \delta(\mathit{source}, a)$. Because we assumed that the Nop-shadows Analysis declares shadow s to be a nop shadow, Definition 16 provides:

$$\forall Q_f \in \mathit{futures}(s) : \mathit{source} \in Q_f \iff \mathit{target} \in Q_f \wedge \mathit{target} \notin F.$$

From the definition of *matches* we know that:

$$\forall w \in \mathit{pref}(w_1) : w \in \mathit{matches}(w_1 w_2) \iff w \in \mathit{matches}(w_1 a w_2).$$

Therefore, prefixes of w_1 automatically satisfy Equation 1. We need only consider non-prefix words w where $w \notin \mathit{pref}(w_1)$.

But we need not consider prefixes of $w_1 a$ either. Because $target \notin F$ we know that $w_1 a \notin matches(w_1 a w_2)$. Hence,

$$\forall w \in pref(w_1 a) : w \in matches(w_1 w_2) \iff w \in matches(w_1 a w_2).$$

Therefore, without loss of generality, we consider only words w with $w \notin pref(w_1 a)$.

For such w, we need to show

$$w \in matches(w_1 w_2) \iff w \in matches(w_1 a w_2).$$

Since we have $w = w_1 a w' \in (pref(w_1 a w_2) \setminus pref(w_1 a))$ and

$$\forall Q_f \in futures(s) : source = \widehat{\delta}(q_0, w_1) \in Q_f \iff target = \widehat{\delta}(q_0, w_1 a) \in Q_f,$$

we know that $\mathcal{L}(\mathcal{M}_{source}) = \mathcal{L}(\mathcal{M}_{target})$. Hence w is a matching prefix of $w_1 w_2$ if and only if it is a matching prefix of $w_1 a w_2$.

Therefore, all that remains to be shown is that the set of states that we approximate in our forward and backward analysis is correct. In particular, our implementation must ensure that, for every set $Q_f \in futures(s)$, the states $q \in Q_f$ are indeed continuation-equivalent, i.e., that for all ground traces t, the continuation of the program execution after reading s satisfies

$$t \in \mathcal{L}(\mathcal{M}_{source}(s)) \iff t \in \mathcal{L}(\mathcal{M}_{target}(s)).$$

Our implementation must therefore never merge state sets: merging may cause the analysis to assume invalid equivalencies. Indeed, our worklist algorithm, Algorithm 4 (page 29), ensures that state sets are never merged: while the algorithm does over-approximate pointer information in various ways, it never merges configurations that have differing state sets Q_c. Every configuration $c = (Q_c, b_c)$ represents one element of the set *futures*. Algorithm 4 simply propagates these configurations but never merges them. In particular, note that the algorithm has no special treatment for control-flow merge points: when two different configurations reach the same statement along different paths, the algorithm simply propagates both configurations; it does not attempt to merge these configurations.

Further, the algorithm takes into account all possible continuations by propagating configurations along all possible intra-procedural and inter-procedural paths. To properly propagate configurations, the algorithm needs to conservatively handle binding representatives. When propagating intra-procedurally, the algorithm refines configurations' binding representatives using the simplification rules from Table II, which can trivially be seen to be sound. When propagating configurations inter-procedurally, along $succ_{ext}$, the algorithm does not refine binding representatives. Because the unrefined binding representatives are at least as permissive as refined representatives, this is a sound over-approximation.

The soundness of the forward and backward analysis then follows from the fact that we (1) initialize both the backward and forward analyses with configurations at all nodes where an initial (or final) state could be reached; (2) propagate configurations along all possible control-flow paths (or abstractions of these), taking into account all relevant shadows (as determined by *relevantShadow*), and (3) never merge any configurations with differing state sets. □

8. CERTAIN-MATCH ANALYSIS

The analysis information obtained during the Nop-shadows Analysis also enables us to identify shadows s which certainly drive a Dependency State Machine into its final state. Because such shadows imply that the program is definitely violating a stated property, developers could profitably use a list of certain matches. Formally, we define a predicate *certainMatch* on shadows $s \in \mathcal{S}$:

$$\mathit{certainMatch}(s) := \forall q \in \mathit{sources}(s) : \ \mathit{target}(q, s) \in F.$$

As an example, consider again the write shadows in the the two pieces of code from figures 4a and 4d (page 8). Let s be the write shadow in Figure 4a. In this figure, there is only one possible execution, and this execution yields $\mathit{sources}(s) = \{\text{disconnected}\}$. Because $\mathit{target}(\text{disconnected}, \text{write}) = \text{error} \in F$, the Certain-match Analysis will flag the write shadow as a certain match.

On the other hand, there are two possible executions paths leading to the write shadow in Figure 4d. One path closes the connection while the other one does not. Therefore, we have $\mathit{sources}(s) = \{\text{connected}, \text{disconnected}\}$. As a result, we obtain $\mathit{target}(\text{connected}, \text{write}) = \text{connected} \notin F$, so the Certain-match Analysis will not flag this write shadow as a certain match.

Because the Certain-match Analysis can operate on analysis information that the Nop-shadows Analysis already computed, it has negligible compile-time cost. Further, because the analysis only reports a certain match if the shadow in question completes the match from *all* its possible source states, the Certain-match Analysis for a method m can only yield false positives if m is actually dead, i.e., cannot be reached on any concrete program execution. The Certain-match Analysis may miss some certain matches; for instance, it may assume that certain control-flow paths are realizable, while the concrete program never actually realizes these paths. The Certain-match Analysis therefore satisfies opposite design goals from the other analyses that we presented: while the other analyses are sound over-approximations that may report false positives but never miss potential violations, the Certain-match Analysis is an unsound under-approximation that may miss actual violations but never reports false positives (except for dead code, as explained above).

9. PRESENTING ANALYSIS RESULTS FOR MANUAL CODE INSPECTION

Designing a static analysis that is both sound and precise has obvious benefits. However, we also quickly experienced the following drawback. Precise analyses are usually quite complex, and when they do fall short, it is in complex situations.

In the context of CLARA, some of the shadows remaining after our analyses—potential property violations—can be difficult to manually classify as certainly-violating or certainly-safe. Our hybrid approach enables the programmer to simply not care: she can test the program with the inserted residual runtime monitor and observe whether the monitor is actually triggered. Such an approach, however, depends on the availability of good test cases. We therefore sought to present the static-analysis results in an easily accessible way, easing the task of manual code inspection as much as possible.

9.1 Potential points of failure and Potential failure groups

To reduce the workload on the programmer during manual inspection, we first divide all still-enabled shadows into semantic groups. First, we select all still-enabled shadows from the program that can lead the runtime monitor directly to an error state (e.g. all "write" shadows for the ConnectionClosed property). These are the program points at which the runtime monitor may potentially trigger its error handler at runtime. In the following, we will call each such shadow a *potential point of failure* (PPF). Next, we use points-to sets to associate each PPF with all its *context shadows*, i.e., with all shadows that potentially may have driven the runtime monitor to a state from which executing the shadow at the PPF then makes the monitor reach the final state. For every PPF p the context shadows of p are exactly all those shadows compatible with p. The combination of a PPF with its context shadows is called a *potential failure group* (PFG). That way, each group represents one distinct error scenario. CLARA reports its analysis results as a list of PFGs. The first author's dissertation [Bodden 2009] shows that inspecting PFGs instead of individual shadows can reduce the number of items to inspect by about 70% on average. In previous work, we also demonstrated methods to rank the reported list of PFGs such that PFGs whose shadows likely remain enabled only due to analysis imprecision are ranked further to the bottom of the list [Bodden et al. 2008a]. This, in turn, causes actual property violations more likely to appear at the top of the list.

Programmers can inspect the list of PFGs in a textual format. However, such text files may still be awkward to use. Ideally, one would like to display the analysis results inline with the analyzed program's source code. We therefore developed a plugin for the Eclipse IDE that allows programmers to display the analysis results as an overlay to the program's code. Figure 14 shows a screenshot of this plugin on one of CLARA's test cases. Lines holding a shadow are highlighted in yellow and give information about the shadow's abstract symbol name and the Nop-shadows Analysis's analysis information on the right-hand side. (Future versions will also display the property's state machine inline.) Further, we use arrows to link relevant shadows with each other. An arrow exists from a shadow s_1 to a shadow s_2 if there is a PFG containing both shadows, both shadows are within the same method and the program's control may flow from s_1 to s_2.

Many of the potential failure groups that remain after analysis, however, are spread over different methods. This is because our intra-procedural analysis often successfully rules out PFGs that are confined to a single method (except, of course, for methods that actually cause a violation). We found the prospect of drawing arrows between multiple methods or even classes unappealing. Instead, we now offer users a new context menu for shadows which automatically links to all shadows in the same PFG but outside the clicked-on method.

We found this way of presenting our analysis a tremendous improvement over a textual output. In particular, the plugin helped us to quickly identify implementation errors in earlier versions of our analyses, and also allowed us to easily identify actual property violations in our benchmark set. Further, good tool support for manual code inspection allows programmers to use CLARA as a compile-time-only tool that allows them to identify possible property violations without ever run-

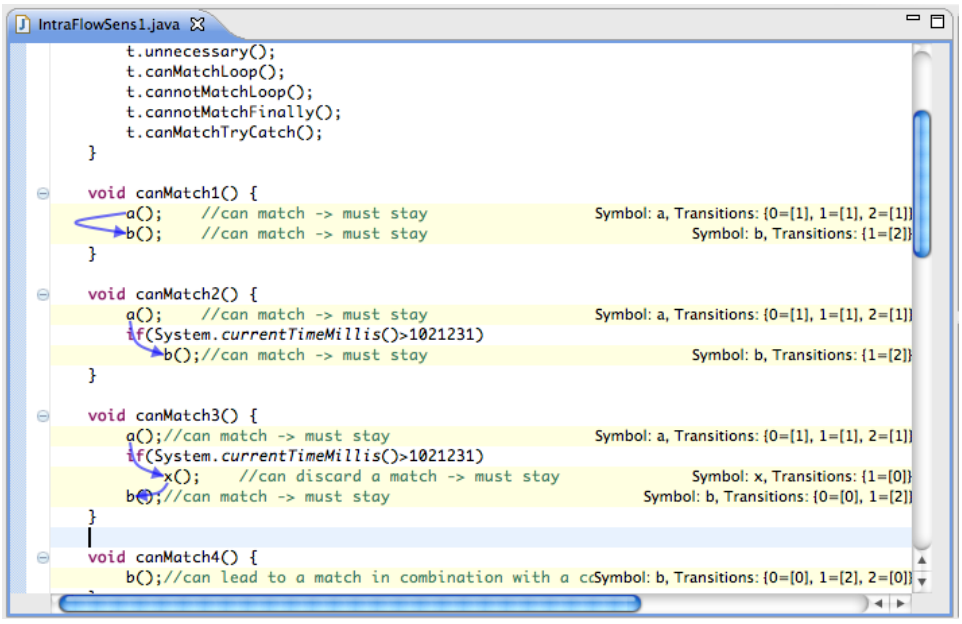


Fig. 14: Presentation of analysis results in the Eclipse IDE

ning a runtime monitor. For such an approach, it suffices to provide CLARA with a "skeleton" runtime monitor that consists only of advice definitions and a Dependency State Machine annotation—such monitors need not contain any code in advice bodies. This is particularly important as research has shown that writing correct and efficient code for parameterized runtime monitors is highly nontrivial [Avgustinov et al. 2006; Chen and Roşu 2007].

10. EXPERIMENTS

In this section we explain our empirical evaluation and present our experimental results. Due to space limitations, we can only give a summary of those results. The first author's dissertation [Bodden 2009] gives a full account.

This work presents experimental results for monitors generated from tracematch specifications [Allan et al. 2005]. Because CLARA abstracts from the implementation details of a runtime monitor through Dependency State Machines, CLARA supports all AspectJ-based runtime monitors that carry a Dependency State Machine annotation. Our earlier work [Bodden et al. 2009] showed that the efficacy of our static analyses is independent of the concrete monitoring formalism.

For our experiments, we wrote a set of twelve tracematch specifications for different properties of collections and streams in the Java Runtime Library. Table III gives brief descriptions for each of these properties. We selected properties of the Java Runtime Library due to the ubiquity of clients of this library. Our tracematch definitions are available at `http://www.bodden.de/clara/`.

We used CLARA to instrument the benchmarks of version 2006-10-MR2 of the DaCapo benchmark suite [Blackburn et al. 2006] with runtime monitors for the twelve properties that we defined. DaCapo contains eleven different workloads of

property name	description
ASyncContainsAll	synchronize on `d` when calling `c.containsAll(d))` for synchronized collections `c` and `d`
ASyncIterC	only iterate a synchronized collection `c` when owning a lock on `c`
ASyncIterM	only iterate a synchronized map `m` when owning a lock on `m`
FailSafeEnum	do not update a vector while iterating over it
FailSafeEnumHT	do not update a hash table while iterating over its elements or keys
FailSafeIter	do not update a collection while iterating over it
FailSafeIterMap	do not update a map while iterating over its keys or values
HasNextElem	always call hasMoreElements before calling nextElement on an Enumeration
HasNext	always call hasNext before calling next on an Iterator
LeakingSync	only access a synchronized collection using its synchronized wrapper
Reader	do not use a Reader after its InputStream was closed
Writer	do not use a Writer after its OutputStream was closed

Table III: Monitored specifications for classes of the Java Runtime Library

which we consider all but eclipse. Eclipse makes heavy use of reflection, which Clara still has trouble with. In current work [Bodden et al. 2010], we are studying advances on this issue, but these are outside the scope of this work. For our experiments, we used a machine with an AMD Athlon 64 X2 Dual Core Processor 3800+ running Ubuntu 7.10 with kernel version 2.6.22-14 and 4GB RAM. We ran the static analysis on IBM's J9 virtual machine, allowing for 3GB of heap space.

In the following we will discuss (1) the fraction of shadows that Clara can successfully identify as unnecessary for monitoring, (2) the positive impact of disabling these shadows on the runtime overhead of the monitoring process and (3) the effectiveness of our Certain-match Analysis.

10.1 Fraction of shadows identified as irrelevant

Table IV summarizes the analysis results for our 120 tracematch/property combinations. In 11 cases, the property did not apply to the benchmark, leaving 109 cases to consider. The table reports, as white slices, the fraction of shadows that the analysis identified as irrelevant. In red (or gray) we show the fraction of shadows that are known to trigger actual violations at runtime. No sound static analyses could disable these shadows: because the shadows trigger a property violation at runtime they need to remain enabled. The remaining black slices represent shadows which we are unsure about. These shadows remain active even after analysis, either due to analysis imprecision or due to actual property violations.

As the table shows, our analysis is very effective in most cases. Clara was able to prove for 74 out of these 109 cases (68%) that the program cannot violate the property on any execution (all-white circles). In the remaining cases, the analysis can often disable a large fraction of the instrumentation. Black slices due to imprecision remain mainly in bloat, jython and pmd. bloat is notorious for having very long-lived objects and a literally very bloated code base. This makes it hard for static analyses to handle this benchmark. In fact, bloat has been removed from the new version 9.12 of the DaCapo benchmark suite. jython and pmd both make heavy use of dynamic class loading and reflection. This confuses our pointer analysis, which makes very conservative approximations in such situations. Our pointer analysis therefore believes that certain iterators and enumerations in these benchmark might be aliased even though no aliasing exists in practice. We are currently

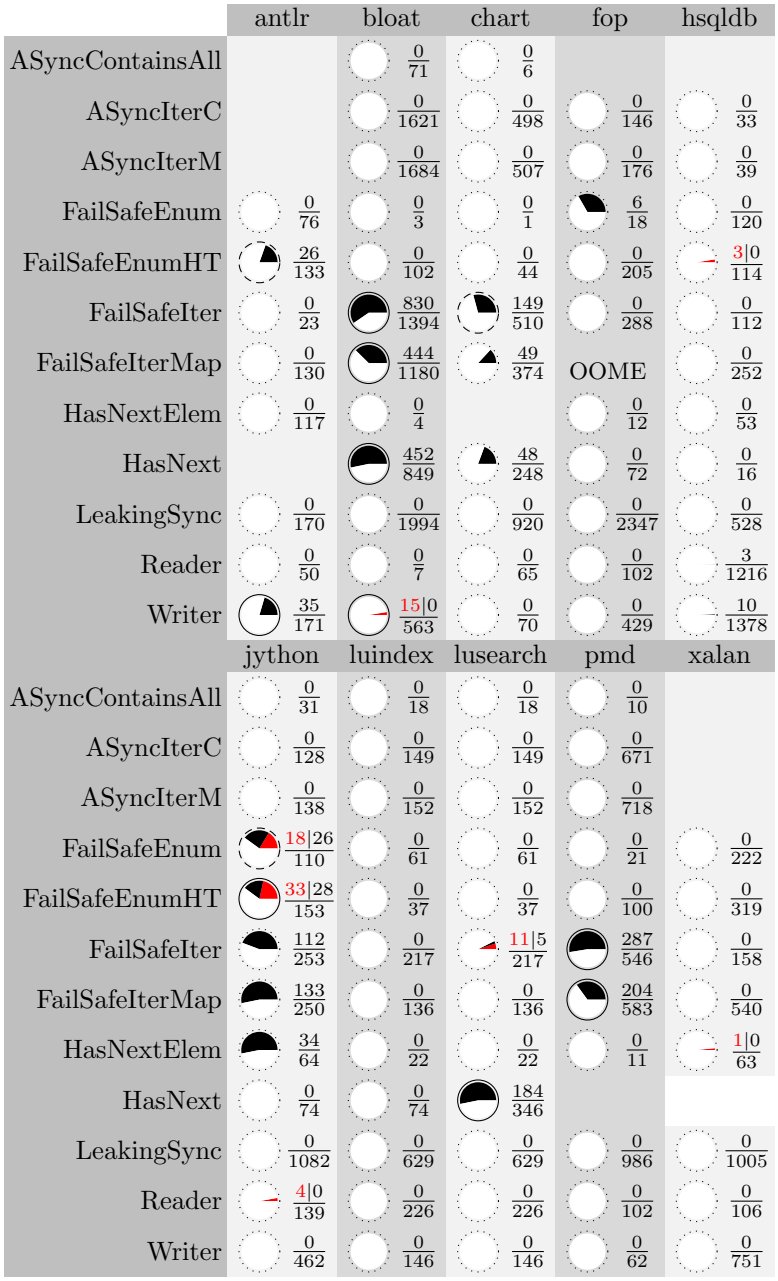


Table IV: Shadows identified as irrelevant, and therefore disabled. White slices represent shadows our analysis identified as irrelevant. Black slices represent shadows that we fail to identify as irrelevant, due to analysis imprecision or because the shadows may help trigger a property violation at runtime. Red (or gray) slices represent shadows that we confirmed relevant by manual inspection. Outer rings represent the monitor's runtime overhead after optimizing advice dispatch. Solid: overhead $\geq 15\%$, dashed: overhead $< 15\%$, dotted: no overhead. OOME = OutOfMemoryException during static analysis

trying to extend CLARA so that it can handle reflection with more fine-grained approximations. For fop/FailSafeIterMap, our analysis ran out of memory, despite the fact that we allowed the analysis 3GB of heap space.

10.2 Reduction of runtime overhead

To measure runtime overhead, we ran all benchmark/property combinations both before and after applying our static analysis. We used the HotSpot Client VM (build 1.4.2_12-b03, mixed mode) with default heap size settings. To execute the benchmarks, we used DaCapo's `-converge` switch, which repeatedly runs benchmarks until they reach a steady state before measuring runtime, yielding error margins usually below 3%. Table IV gives qualitative information about the residual monitor's runtime overhead through the ring that surrounds each circle. A solid ring denotes an overhead of at least 15%, a dashed ring an overhead of less than 15%, and a dotted ring means that no observable overhead remains. In Table V, we quantify the runtime overheads in more detail. We marked the 74 cases for which CLARA was able to prove that the program cannot violate the property on any execution with a "✓". In these cases, monitoring is unnecessary because CLARA removes all instrumentation. However, if we chose to test-run these combinations anyway, the runtime overhead would be zero, as the runtime monitor is never called. 37 of the original 109 combinations showed a measurable runtime overhead. After applying the static analysis, measurable overhead only remained in 13 cases (35% of 37). These cases often show significantly less overhead than without optimization.

10.3 Effectiveness of Certain-match Analysis

As our results show, the Certain-match Analysis was much less effective than the other analyses: among all our benchmark/property combinations, the analysis found only one certain match: line 218 of method `InductionVarAnalyzer.isMu(..)` of bloat, in combination with the HasNext property. The code looks like this:

```
216 Iterator iter = cfg.preds(phi.block()). iterator ();
217 Block pred1 = (Block) iter.next();
218 Block pred2 = (Block) iter.next();
```

This code certainly does violate the property: it calls `next()` twice without calling `hasNext()` in between. In our version of bloat and on our test inputs this causes no problems; however, the code could cause an exception if bloat violated its invariants and defines a phi block with zero or one successors.

It may be surprising that the Certain-match Analysis is so much less effective than the Nop-shadows Analysis, even though they are based on the same analysis information. We suggest two differences for this difference in effectiveness.

First, our benchmark programs have already been debugged and therefore rarely violate the correctness properties that we specify. This fact benefits shadow-disabling analyses (since already-debugged programs require almost no monitoring), but equally hinders the Certain-match Analysis: with few violations, there will be few certain matches.

Second, the Certain-match Analysis only reports matches known to be certain. For a match to be certain, the analysis has to know that (1) all property-violating events must occur on the same object, and (2) these events must execute in a

	antlr		bloat		chart		fop		hsqldb	
	pre	post	pre	post	pre	post	pre	post	pre	post
ASyncContainsAll	-	-	0	0 ✓	0	0 ✓	-	-	-	-
ASyncIterC	-	-	140	0 ✓	0	0 ✓	5	0 ✓	0	0 ✓
ASyncIterM	-	-	139	0 ✓	0	0 ✓	0	0 ✓	0	0 ✓
FailSafeEnumHT	10	4	0	0 ✓	0	0 ✓	0	0 ✓	0	0
FailSafeEnum	0	0 ✓	0	0 ✓	0	0 ✓	0	0	0	0 ✓
FailSafeIter	0	0 ✓	>1h	>1h	8	8	14	0 ✓	0	0 ✓
FailSafeIterMap	0	0 ✓	>1h	22027	0	0	7	OOM	0	0 ✓
HasNextElem	0	0 ✓	0	0 ✓	-	-	0	0 ✓	0	0 ✓
HasNext	-	-	329	258	0	0	0	0 ✓	0	0 ✓
LeakingSync	9	0 ✓	163	0 ✓	91	0 ✓	209	0 ✓	0	0 ✓
Reader	30218	0 ✓	0	0 ✓	0	0 ✓	0	0 ✓	0	0
Writer	37862	36	229	228	0	0 ✓	5	0 ✓	0	0

	jython		luindex		lusearch		pmd		xalan	
	pre	post	pre	post	pre	post	pre	post	pre	post
ASyncContainsAll	0	0	0	0 ✓	0	0 ✓	0	0 ✓	-	-
ASyncIterC	0	0	0	0 ✓	0	0 ✓	28	0 ✓	-	-
ASyncIterM	0	0	0	0 ✓	0	0 ✓	35	0 ✓	-	-
FailSafeEnumHT	>1h	>1h	32	0 ✓	0	0 ✓	0	0 ✓	0	0 ✓
FailSafeEnum	0	0	30	0 ✓	18	0 ✓	0	0	0	0 ✓
FailSafeIter	0	0	5	0 ✓	20	0	2811	524	0	0 ✓
FailSafeIterMap	13	13	5	0 ✓	0	0 ✓	>1h	>1h	0	0 ✓
HasNextElem	0	0	12	0 ✓	0	0 ✓	0	0	0	0
HasNext	0	0	0	0 ✓	0	0 ✓	70	64	-	-
LeakingSync	>1h	0	34	0 ✓	365	0 ✓	16	0 ✓	0	0 ✓
Reader	0	0	0	0 ✓	77	0 ✓	0	0	0	0 ✓
Writer	0	0	0	0 ✓	0	0 ✓	0	0	0	0 ✓

Table V: Effect of CLARA's static analyses on runtime overheads; numbers are runtime overheads in percent before and after applying the analyses; ✓: all instrumentation removed, proving that no violation can occur; >1h: run took over one hour

property-violating order. But our analyses use only intra-procedural must-alias information and control-flow information. The Certain-match Analysis can therefore only be effective for violations that (1) refer to objects all bound in the same method, and (2) are indeed violations on all possible executions of this method. Both these restrictions are seldom fulfilled, and they are even more rarely fulfilled in combination. Most of our properties refer to multiple objects; FailSafeIter, for instance, refers to a connection and an interator. For such properties, the Certain-match Analysis could only succeed if the monitored events on both objects are confined to the method being analyzed.

The above observations help explain why the Certain-match Analysis's success occurs with the HasNext pattern: this pattern only reasons about a single iterator object, and iterators are usually only used in a single method (rather than being passed to other methods). Moreover, in the one case in which the Certain-match Analysis did succeed, the match is indeed certain, i.e., will occur on all executions: there is only one execution path on this piece of code.

To summarize, we conclude that the Certain-match Analysis is not very effective because programs are usually correct, because matches are seldom certain, and because the analysis has must-information on an intra-procedural level only.

11. RELATED WORK

We next discuss three broad areas of related work. We first present related work in the area of runtime monitoring and hybrid static and dynamic approaches to verifying program properties similar to ours. We then discuss work on statically verifying typestate properties, because the properties that CLARA verifies can be seen as typestate properties. Finally, we explain how our work relates to previous work (by ourselves and others) on statically analyzing tracematches.

11.1 Runtime monitoring and hybrid approaches

We next discuss a number of runtime monitoring tools that influenced the design and implementation of CLARA. Many of these tools also implement hybrid static and dynamic approaches similar to those that we propose in this paper—statically analyze first, then monitor remaining cases at runtime. We also discuss the potential use of these tools in combination with CLARA.

The first author previously developed J-LO, the Java Logical Observer [Bodden 2005], a tool for checking temporal assertions at runtime in Java programs. The propositions in J-LO's temporal logic formulae use AspectJ pointcuts as propositions. The J-LO tool accepts linear temporal logic formulae over AspectJ pointcuts as input, and generates plain AspectJ code by manipulating an abstract syntax tree. Like in CLARA, pointcuts in J-LO specifications can be parameterized by variable-to-object bindings. While the implementation of J-LO is effective in finding seeded errors in small example programs, its runtime overhead renders J-LO unsuitable for use on larger programs. In principle, one could annotate the J-LO-generated aspects with dependency information and then use CLARA's static analyses to remove some of this overhead.

Tracematches. Tracematches were first proposed and implemented by Allan et al. [2005]. Like J-LO, tracematches generate a low-level AspectJ-based runtime monitor from a high-level specification which uses AspectJ pointcuts to denote events of interest. Tracematch implementations generate far more efficient runtime monitors than J-LO. Furthermore, Avgustinov et al. [2007] perform sophisticated static analyses of a tracematch's induced state machine to compute an optimized monitor implementation which is (1) correct, (2) supports garbage collection of parts of its internal state, and (3) allows indexing of partial matches. As Avgustinov et al. show, both garbage collection and indexing are necessary to achieve a low runtime overhead. Such monitor optimizations, however, are not always sufficient on their own. Our reported experimental results use optimized monitor implementations as a baseline, and illustrate that combining monitor optimizations with our analyses will yield low runtime overhead in most cases.

Note that the two above analyses only consider the state machine, while we analyze both the state machine and the program. This allows us to disable shadows at suitable program points, which aids static program inspections for potential property violations by reducing the size of the inspection task. Previous work focused on making tracematches run faster; it did not analyze or modify shadows.

Another, orthogonal, approach to reducing runtime monitoring overhead is by using collaborative runtime verification. We have explored collaborative runtime verification for tracematches [Bodden et al. 2010], where we partition the moni-

toring work both spatially and temporally. Spatial partitioning creates a number of copies of the program being monitored, where each program copy monitors a small subset of the full set of shadows. We found that most copies incurred no overhead at all over the un-instrumented program. Temporal partitioning switches the instrumentation off and on over time, thereby reducing the overhead of runtime monitoring (as well as the probability of catching a property violation). CLARA supports spatial partitioning independently of the monitor implementation. CLARA cannot, however, easily support temporal partitioning because this would require additional knowledge about the monitor implementation that Dependency State Machines do not encode.

JavaMOP. JavaMOP provides an extensible logic framework for specification formalisms [Chen and Roşu 2007]. JavaMOP has several specification formalisms built in, including extended regular expressions (ERE), past-time and future-time linear temporal logic (PTLTL/FTLTL), and context-free grammars, and supports the addition of new formalisms via logic plugins. JavaMOP translates specifications into AspectJ aspects using the rewriting logic Maude [Clavel et al. 1996]. JavaMOP aims to be a generic framework which supports multiple specification languages; in particular, it makes few assumptions about any particular specification language. However, this generality makes it difficult, if not impossible, to analyze JavaMOP specifications analogously to how Avgustinov et al. have analyzed tracematch specifications. Feng Chen extended [Bodden et al. 2009] the JavaMOP implementation to perform a limited specification analysis, which enabled JavaMOP to annotate generated monitors with dependency information. CLARA can use this information to partially evaluate JavaMOP monitors at compile time.

Chen and Roşu report [Chen and Roşu 2007] that their implementation outperforms the tracematch implementation in terms of runtime overhead while also being memory-safe. However, their claim only holds when the validation or violation handler does not reference any of the objects that are involved in the match. As soon as the handler references such objects, JavaMOP must resort to strong references that keep the internal state of the objects' monitors alive indefinitely, thereby causing high memory consumption and runtime overhead.

Another shortcoming of JavaMOP is that it currently only supports specifications that bind all the objects of a complete match at the first transition. Patterns like FailSafeIter conform to this limitation: the create symbol leading out of the initial state binds both variables, c and i. JavaMOP can then store the state for this variable binding in a cascaded hash map that maps from c to a map that in turn maps from i to the state in question. Our benchmark set, however, contains specifications which do not respect this limitation. ASyncContainsAll describes a situation with calls to c_1`.containsAll(`c_2`)` for two synchronized collections c_1 and c_2. The first symbol binds c_1 when it is returned from a call to `synchronizedCollection`. The second symbol similarly binds c_2. Only the final symbol, at calls to `containsAll()`, binds both values together. In recent work [Chen and Roşu 2009], Chen and Roşu present an algorithm that circumvents this problem by using a different indexing structure. However, the authors do not discuss whether they allow a program to reclaim monitoring state by using weak references, nor, more importantly, why their algorithm would still be sound if they did. The analysis by Avgustinov et al.

combines both weak references and efficient indexing in a provably sound way, but requires an analysis of the specification pattern.

Query Languages. Like tracematches, the Program Query Language [Martin et al. 2005] enables developers to specify properties of Java programs; each property may bind free variables to runtime heap objects. PQL supports a richer specification language than tracematches: it uses stack automata augmented with intersection, rather than finite state machines. Martin et al. propose a flow-insensitive static-analysis to reduce the runtime overhead of monitoring programs with PQL. This approach inspired our Orphan-shadows Analysis. As the authors show and as we confirm in our work, such an analysis can be effective in ruling out impossible matches. However, we also showed that a flow-sensitive analysis enables additional optimizations. PQL instruments the program under test using the BCEL bytecode engineering toolkit. If PQL used AspectJ instead, then it should be possible to optimize the generated monitor with Clara.

The Program Trace Query Language, PTQL [Goldsmith et al. 2005], provides an SQL-like language for querying properties of program traces at runtime, along with a compiler for the query language. Its "partiqle" compiler modifies the source program to notify a monitor about relevant events at runtime. The compiler attempts to partially evaluate program queries at compile time, just like AspectJ compilers only insert runtime checks when they cannot fully evaluate a pointcut at compile time. Because PTQL uses its own compiler, and is not based on AspectJ, Clara cannot currently evaluate PTQL queries ahead of time. Even if PTQL did generate aspects for monitoring, the PTQL language is very expressive—probably Turing complete. Hence it remains unclear whether one could effectively determine dependencies within a query at compile time so that Clara could exploit these dependencies to optimize PTQL monitors.

Static checkers. We also discuss two fully-static checkers, PMD and FindBugs, and compare them to our Certain-match Analysis. PMD [Copeland 2005] is a static checker that aims to find violations of "best practices" or programming styles, rather than finding actual programming errors. An example is a rule that says "A class that has private constructors and does not have any static methods or fields cannot be used.". PMD has no support for data-flow analyses. Hence it cannot, for instance, easily determine whether a pointer may be null or whether a variable will be initialized before it is used. FindBugs [Hovemeyer and Pugh 2004] is a popular static rule checker developed and maintained by the University of Maryland. FindBugs comes with a rich set of checkers that identify programming problems with respect to common libraries like the Java Runtime Library. FindBugs rules usually favour false negatives over false positives: they will often miss programming errors, but emitted warnings often indicate an actual programming problem. The Certain-match Analysis is more similar in spirit to FindBugs than PMD, since it uses analysis results to identify definite problems in a program under analysis. However, it leverages pointer analysis information and carries out a more detailed program analysis than FindBugs. Its domain of applicability, however, is more restricted than that of FindBugs, since it only finds violations of Dependency State Machine properties.

11.2 Typestate

The target class of Dependency State Machine program properties was inspired by typestate systems. Typestate systems track the conceptual states that each object goes through during its lifetime in the computation [Strom and Yemini 1986; Fink et al. 2006; Drossopoulou et al. 2002]. They generalize standard type systems by allowing the typestate of an object to change during the computation. In their initial paper on typestate [Strom and Yemini 1986], Strom and Yemini proposed the idea of having a value's type depend on an internal state—its typestate. Operations can change a value's type by changing that value's typestate.

Fink et al. present a static analysis of typestate properties [Fink et al. 2006] for Java programs. Their approach, like ours, uses a staged analysis which starts with a flow-insensitive pointer-based analysis, followed by flow-sensitive checkers. The authors' analyses allow only for specifications that reason about a single object at a time. This prevents programmers from expressing properties spanning multiple objects. Like us, Fink et al. aim to verify properties fully statically. However, our approach nevertheless allows developers to provide specialized instrumentation and recovery code, while their approach only emits a compile-time warning. Also, our Clara framework supports a range of property languages so that developers can conveniently specify the properties to be verified; Fink et al. do not describe how developers might specify their properties.

Bierhoff and Aldrich [Bierhoff and Aldrich 2007] present an approach to enable the checking of typestate properties in the presence of aliasing. The authors' approach aims to be modular, and therefore abstains from potentially expensive whole-program analyses such as the points-to analyses used by Clara. Bierhoff and Aldrich instead associate references with access permissions, creating an abstraction based on linear logic. The access permissions enable their approach to relate the states of one object (e.g. an iterator) with the state of another object (e.g. a collection which is being iterated upon). These permissions classify how many other references to the same object may exist and define the allowed operations on references. Their approach requires every method to be annotated with information about how access permissions and typestates change when a method is executed. Of course, this need not imply that the programmer must add these annotations: many approaches can infer typestate properties.

Bierhoff and Aldrich's approach has the advantage of being modular: given appropriate annotations it can analyze any method, class or package independent of context. Our approach, on the other hand, must analyze the whole program, and expects a complete call graph, with sufficient precision to avoid unnecessary false positives. When the whole program is available, and can be analyzed, then Clara has the advantage of not requiring program annotations. Clara only requires annotations to describe error situations, rather than normal program behaviour, and automatically analyzes the program to see whether such error situations can occur. We have found that worst-case assumptions coupled with coarse-grained side-effect information are surprisingly effective.

Because Bierhoff and Aldrich's work defines a type system and not a static checker like Clara, the programmer's workflow differs slightly between Bierhoff and Aldrich's approach and the Clara approach. Clara allows the programmer

to write and compile a program that may violate the given safety property. If CLARA's verification fails, it generates runtime checks. Bierhoff and Aldrich's approach instead defines a type checker; potentially property-violating programs do not compile. The CLARA approach allows more programs to compile, but these programs may violate the property at runtime. CLARA does, however, notify the developer that his or her program is definitely free of violations, or definitely violates the property.

DeLine and Fähndrich's approach [DeLine and Fähndrich 2004] is similar in flavour to Bierhoff and Aldrich's. The authors implemented their approach in the Fugue tool for specifying and checking typestates in .NET-based programs. Fugue checks typestate specifications statically, in the presence of aliasing. The authors present a programming model of typestates for objects with a sound modular checking algorithm. The programming model handles typical features of object-oriented programs such as down-casting, virtual dispatch, direct calls, and sub-classing. The model also permits subclasses to extend the interpretation of typestates and to introduce additional typestates, similar to the statecharts-based approach by Strom and Yemini. As in Bierhoff and Aldrich's approach, DeLine and Fähndrich assume that a programmer (or tool) has annotated the program under test with information about how calls to a method change the typestate of the objects that the method references. One fundamental difference between the two approaches is the treatment of aliasing. While Bierhoff and Aldrich used access permissions to reason about aliases, Fugue's type system tracks objects merely as "not aliased" or "maybe aliased". Objects typically remain "not aliased" as long as they are only referenced by the stack. The respective objects can change state only during this period. Once they become "maybe aliased", Fugue forbids any state-changing operations on these objects. This makes Fugue's type system less permissive than Bierhoff and Aldrich's system, where even aliased objects can change states.

Like us, Dwyer and Purandare use typestate analyses to specialize runtime monitors [Dwyer and Purandare 2007]. Their work identifies "safe regions" in the code using a simple static typestate analysis. Safe regions can be methods, single statements or compound statements (e.g. loops). A region is safe if its deterministic transition function does not drive the typestate automaton into a final state. A special case of a safe region would be a region that does not change the automaton's state at all. The authors call such a region an identity region. For regions that are safe, but not identity regions, the authors summarize the effect of the region and modify the program under test to update the typestate with the region's effects all at once when the region is entered, instead of one-by-one during the region's execution. The optimized program will therefore execute faster because it will execute fewer transitions at runtime. However, one disadvantage of such summary transitions is a decoupling between the code locations that perform a state transition and the locations that actually cause these transitions. This decoupling may impede the manual understanding and verification of code behaviour with respect to the monitored properties. Our static analysis does not attempt to determine regions; we instead decide, for each single shadow, whether it is a nop-shadow. Dwyer and Purandare's analysis should be easily implementable in CLARA and we encourage such an implementation.

11.3 Other static analyses for tracematches

This paper summarizes the results of a large body of research which took place over the past four years. In the course of this research, we strove to find analysis implementations that were correct, precise and efficient. We comment below on some of our earlier attempts and their weaknesses. We also describe how CLARA improves on these earlier algorithm versions, yielding an implementation that is indeed correct, precise and efficient. Also, earlier analyses applied to tracematches only; CLARA instead provides these implementations as instantiations of a unified framework that supports a wide variety of AspectJ-based runtime-monitoring tools.

Naeem and Lhoták have concurrently developed a static analysis to analyze tracematches at compile time. Their analysis differs from our analyses mostly in the chosen abstraction. We discuss the impact of the choice of abstraction below.

Two of our previous papers presented earlier versions of techniques evaluating tracematches ahead of time. CLARA integrates this early work with small but significant improvements, and it generalizes the previous work to apply to an entire class of runtime monitoring tools, rather than just tracematches. In the first paper [Bodden et al. 2007], we presented a three-staged analysis consisting of a Quick Check, a flow-insensitive Consistent-shadows analysis and a flow-sensitive Active-shadows analysis. The Quick Check is essentially the same as the one that we present here, except for one minor difference: the quick check in our earlier work would consider an entire state machine, and simply disable checking of the whole state machine, if the program cannot reach any final state along any path. The Quick Check that we present here acts on every path separately: when the state machine cannot reach a final state any more along a path p then the check disables monitoring of the events on p, even if one can reach the final state along other paths. This improvement helps with properties that yield state machines with multiple accepting paths, e.g. for Reader, where there is one path that reports a violation when writing to an InputStream whose Reader was closed, and a second path which reports a violation when writing to a Reader whose InputStream was closed.

The Consistent-shadows analysis is similar to the Orphan-shadows Analysis that we present here. In fact, both analyses yield identical results. However, the Orphan-shadows Analysis has smaller analysis time and memory requirements than the Consistent-shadows analysis. The Consistent-shadows analysis is a generate-and-test algorithm, which in principle takes time exponential in the number of shadows. We found that, for most cases, the Consistent-shadows analysis was efficient enough to be usable, but that analysis suffered from long analysis times and large memory consumption in cases like bloat-FailSafeIter. The execution time of the Orphan-shadows Analysis is polynomial in the number of shadows, and it uses a quadratic amount of memory to cache its results.

Impact of analysis abstractions. The third stage from our earlier work, the Active-shadows analysis, was a first attempt at a flow-sensitive analysis of tracematches. While CLARA's third analysis stage, the Nop-shadows Analysis, is flow-sensitive only on an intra-procedural level, the Active-shadows analysis from our earlier work was a flow-sensitive, context-insensitive analysis of the entire program. Unfortunately, the abstraction that we chose for this Active-shadows analysis only allowed for weak updates because it did not encode must-alias information. Furthermore,

we chose a flow-insensitive pointer abstraction, and the computation of typestate information was context-insensitive too. These choices made the earlier analysis so imprecise that it was unable to identify any nop-shadows at all in a benchmark set quite similar to the benchmark set that we consider here. These results, in combination with the work that we present in this thesis, show that choosing the right abstractions is key to obtaining good precision. CLARA uses precise intra-procedural flow-sensitive pointer information and context-sensitivity. As we have showed here, this information can yield much optimization potential and therefore significantly improves over the earlier Active-shadows analysis.

In a second paper [Bodden et al. 2008a] we presented an analysis similar to the Nop-shadows Analysis that we present in this work, except for the following points. Firstly, the earlier analysis recognizes "necessary shadows" using shadow histories. Unfortunately, this is unsound (see [Bodden 2010] for details). The earlier analysis is also optimistic: it assumes that a shadow s is unnecessary and can be removed, unless it drives a shadow history containing s into a final state. The Nop-shadows Analysis that we present here instead detects "unnecessary shadows", i.e., nop shadows, using a backwards analysis. This analysis is pessimistic: it assumes that a shadow is necessary until we prove that it is a nop shadow. Because pessimistic analyses make a pessimistic base assumption, implementation errors in such analyses are less likely to produce unsound results: if our analyses misses any corner cases, it would likely keep shadows alive that are actually nop shadows rather than accidentally disabling shadows that are not nop shadows.

A second difference between the two analyses is the treatment of inter-procedural control-flow. In our earlier work, we assumed that any state machine instance could be in any state when entering the current method m. This is sound but also quite imprecise. In the Nop-shadows Analysis that we present here, we instead use the function *reachingStar* to compute a better approximation. Also, in [Bodden et al. 2008a], we did not use the inter-procedural successor function $succ_{ext}$. Instead, whenever we recognized an outgoing method call that could (potentially transitively) call a shadow-bearing method, then we simply "tainted" successor configurations and refused to remove shadows with tainted configurations. The solution that we present in this paper is not only more elegant, it is also more precise. Tainting makes a worst-case assumption about outgoing method calls, while our current implementation considers such method calls more precisely, by considering the potential actions of the rest of the program.

Note also that all of the earlier analyses were designed and implemented to work for tracematches only. In this paper, we present for the first time a set of algorithms that is applicable to any runtime monitor written in the form of an AspectJ aspect. The only restriction is that this aspect must carry an annotation in the form of a Dependency State Machine.

The Nop-shadows Analysis presented in this paper was originally published in [Bodden 2010], in much less detail: the earlier publication, due to space restrictions, did not discuss our treatment of pointers and inter-procedural control flow.

Context-sensitive, flow-sensitive whole-program analysis of tracematches. Naeem and Lhoták [2008] present a context-sensitive flow-sensitive inter-procedural whole-program analysis to analyze typestate-like properties of multiple interacting objects

at compile time. One could easily integrate their analysis into CLARA.

The analysis that Naeem and Lhoták present can be seen as a generalized version of our Nop-shadows Analysis. Our Nop-shadows Analysis is mostly intra-procedural and uses only flow-insensitive information to model inter-procedural control flow. Naeem and Lhoták's analysis, on the other hand, propagates configurations along call edges and then further through the bodies of called methods. This can potentially lead to enhanced precision when multiple methods use combinations of objects that are relevant to a given specification.

Naeem and Lhoták also use a different pointer abstraction from ours. Our pointer abstraction, object representatives, is flow-sensitive only on an intra-procedural level, and at outgoing method calls we resort to context-sensitive but flow-insensitive pointer information. Naeem and Lhoták instead use a special "binding lattice" that models each object by the variables that may or must point to the object. This representation encodes must-aliasing and must-not-aliasing at the same time.

Their analysis updates the aliasing information at the same time as the state information. This differs from our approach, where we compute the inter-procedural points-to information once per program, and the intra-procedural flow-sensitive must-alias analysis and must-not-alias analysis once per method, before applying our main worklist algorithm to each method. In theory, one can gain additional precision by computing typestate information at the same time as the pointer information like Naeem and Lhoták do. We have constructed examples where such precision gives better results than our analysis. However, our benchmark set includes no instances where we failed to identify a nop-shadow because we did not use a combined pointer and typestate abstraction; it appears that our separate abstractions do not actually cause much imprecision in practice.

As Naeem and Lhoták note [Naeem and Lhoták 2008, Section 5], there are also examples in which our analyses succeed in identifying a nop shadow, but Naeem and Lhoták's fails on that shadow. This is due to the context-sensitive points-to analysis that we use. Generally, this points-to analysis is very precise, due to its large amount of context information. When this context information matters, the pointer abstraction that Naeem and Lhoták chose may be less precise.

Naeem and Lhoták are currently re-implementing their analysis to increase precision and performance. We therefore have not yet compared our analysis to Naeem and Lhoták directly. In the future, we plan to create a joint comparative study in which we consistently use the same tracematch specifications and analyze the same benchmark versions with the same runtime library, taking into account the exact same set of potentially dynamically loaded classes.

Naeem and Lhoták's analysis builds on our Active-shadows analysis from earlier work, and therefore the analysis suffers from the same unsoundness problem that we described above: Naeem and Lhoták's analysis also uses shadow histories. It should be possible to fix Naeem and Lhoták's analysis by making it use an analysis scheme similar to CLARA's, where one conducts both a forward and a backward analysis while maintaining the pointer abstractions that the authors are using now. However, our algorithms recompute all analysis information after disabling any shadow, because disabling a shadow may change the program's transition structure. As we showed, this can be done efficiently on an intra-procedural level. However, it

would be unrealistic to re-iterate an inter-procedural analysis every time a shadow gets disabled: every inter-procedural iteration typically takes several minutes to compute. Augmenting the analysis information by including dependencies could limit the need for complete re-iterations. Such an approach would be applicable to both CLARA and Naeem and Lhoták's analysis, and we plan to consider such an approach in future work.

12. CONCLUSION

We have presented four static whole-program analyses which partially evaluate parametrized finite-state runtime monitors at compile time. We implemented the analyses in CLARA, a novel framework for the partial evaluation of AspectJ-based runtime monitors. Our evaluation of CLARA on several large-scale Java programs demonstrated that most of our benchmark programs fulfill our example properties. For the remaining programs, CLARA reduced the monitoring overhead to below 10%. We also found multiple property violations in our benchmark suite.

Our results show that CLARA provides push-button technology to statically approximate and optimize expressive runtime monitors. CLARA's mechanism is largely independent of the runtime monitor's concrete implementation strategy and can therefore be used with a wide range of current runtime monitoring tools. A direct application of our static analysis techniques enables runtime monitor optimization. We also advocate a compile-time-only approach where programmers use our analyses to identify code locations where a program could potentially violate a given finite-state property. We have explained that an effective integration into an integrated development environment allows programmers to tell apart actual property violations from false positives relatively easily.

In ongoing work, we are extending the CLARA static-analysis framework to better cope with native calls [Bodden et al. 2010], for instance those induced by reflection, and to analyze and optimize runtime monitors not only for individual programs but instead for entire software product lines [Kim et al. 2010]. In future work, we plan to design typestate analyses that do not require the entire closed program but instead operate on individual program modules, e.g. software services. Furthermore, we plan to investigate how to combine data-flow analyses like the one presented here with model checking. For instance, Rungta et al. Rungta et al. [2009] have recently shown how to effectively guide a model checker to problem points previously determined through a (generic) static analysis. Such an approach would allow programmers to rule out even more (if not all) false positives than CLARA does.

Acknowledgements. This work would not have been the same without the support of many people, including Pavel Avgustinov, Julian Tibble, Oege de Moor, Torbjörn Ekman and other members of the Programming Tools Group at Oxford University, Grigore Roşu, Feng Chen, Matthew Dwyer, Rahul Purandare, Kevin Bierhoff, Ciera Jaspan, Ondřej Lhoták, Nomair Naeem and Manu Sridharan. Thank you all for your support and for the lively discussion!

REFERENCES

ALLAN, C., AVGUSTINOV, P., CHRISTENSEN, A. S., HENDREN, L., KUZINS, S., LHOTÁK, O., DE MOOR, O., SERENI, D., SITTAMPALAM, G., AND TIBBLE, J. 2005. Adding Trace Matching

with Free Variables to AspectJ. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM Press, 345–364.

aspectj 2003. The AspectJ home page.

Avgustinov, P., Christensen, A. S., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., and Tibble, J. 2005. abc: An extensible AspectJ compiler. In *International Conference on Aspect-oriented Software Development (AOSD)*. ACM Press, 87–98.

Avgustinov, P., Tibble, J., Bodden, E., Lhoták, O., Hendren, L., de Moor, O., Ongkingco, N., and Sittampalam, G. 2006. Efficient trace monitoring. Tech. Rep. abc-2006-1. March.

Avgustinov, P., Tibble, J., and de Moor, O. 2007. Making trace monitors feasible. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM Press, 589–608.

Bierhoff, K. and Aldrich, J. 2007. Modular typestate checking of aliased objects. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 301–320.

Blackburn, S. M., Garner, R., Hoffman, C., Khan, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Phansalkar, A., Stefanovic, D., VanDrunen, T., von Dincklage, D., and Wiedermann, B. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM Press, 169–190.

Bodden, E. 2005. J-LO - A tool for runtime-checking temporal assertions. M.S. thesis, RWTH Aachen University.

Bodden, E. 2009. Verifying finite-state properties of large-scale programs. Ph.D. thesis, McGill University.

Bodden, E. 2010. Efficient hybrid typestate analysis by determining continuation-equivalent states. In *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*. ACM, New York, NY, USA, 5–14.

Bodden, E., Chen, F., and Roşu, G. 2009. Dependent advice: A general approach to optimizing history-based aspects. In *International Conference on Aspect-oriented Software Development (AOSD)*. ACM Press, 3–14.

Bodden, E., Hendren, L., Lam, P., Lhoták, O., and Naeem, N. A. 2010. Collaborative runtime verification with tracematches. *Journal of Logic and Computation 20,* 3, 707–723.

Bodden, E., Hendren, L. J., and Lhoták, O. 2007. A staged static program analysis to improve the performance of runtime monitoring. In *European Conference on Object-Oriented Programming (ECOOP)*. Lecture Notes in Computer Science (LNCS), vol. 4609. Springer, 525–549.

Bodden, E., Lam, P., and Hendren, L. 2008a. Finding Programming Errors Earlier by Evaluating Runtime Monitors Ahead-of-Time. In *Symposium on the Foundations of Software Engineering (FSE)*. ACM Press, 36–47.

Bodden, E., Lam, P., and Hendren, L. 2008b. Object representatives: a uniform abstraction for pointer information. In *Visions of Computer Science - BCS International Academic Conference*. British Computing Society.

Bodden, E., Lam, P., and Hendren, L. 2010. Clara: a framework for statically evaluating finite-state runtime monitors. In *1st International Conference on Runtime Verification (RV)*. LNCS, vol. 6418. Springer, 74–88.

Bodden, E., Sewe, A., Sinschek, J., and Mezini, M. 2010. Taming Reflection (Extended version). Tech. Rep. TUD-CS-2010-0066, CASED. Mar. `http://cased.de/`.

Brzozowski, J. A. 1962. Canonical regular expressions and minimal state graphs for definite events. In *Symposium on Mathematical Theory of Automata*. Polytechnic Institute of Brooklyn, 529–561.

Chen, F. and Roşu, G. 2007. MOP: an efficient and generic runtime verification framework. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM Press, 569–588.

Chen, F. and Roşu, G. 2009. Parametric trace slicing and monitoring. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Lecture Notes in Computer Science (LNCS), vol. 5505. Springer, 246–261.

Clavel, M., Eker, S., Lincoln, P., and Meseguer, J. 1996. Principles of maude. *Electronic Notes in Theoretical Computer Science (ENTCS) 4*.

Copeland, T. 2005. *PMD Applied*. Centennial Books.

DeLine, R. and Fähndrich, M. 2004. Typestates for objects. In *European Conference on Object-Oriented Programming (ECOOP)*. Lecture Notes in Computer Science (LNCS), vol. 3086. Springer, 465–490.

Drossopoulou, S., Damiani, F., Dezani-Ciancaglini, M., and Giannini, P. 2002. More dynamic object reclassification: Fickle ii. *24,* 2, 153–191.

Dwyer, M. B. and Purandare, R. 2007. Residual dynamic typestate analysis: Exploiting static analysis results to reformulate and reduce the cost of dynamic analysis. In *International Conference on Automated Software Engineering (ASE)*. ACM Press, 124–133.

Fink, S., Yahav, E., Dor, N., Ramalingam, G., and Geay, E. 2006. Effective typestate verification in the presence of aliasing. In *International Symposium on Software Testing and Analysis (ISSTA)*. ACM Press, 133–144.

Goldsmith, S., O'Callahan, R., and Aiken, A. 2005. Relational queries over program traces. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM Press, 385–402.

Hovemeyer, D. and Pugh, W. 2004. Finding bugs is easy. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM Press, 132–136.

Kim, C. H. P., Batory, D., Bodden, E., and Khurshid, S. 2010. Reducing Configurations to Monitor in a Software Product Line. In *1st International Conference on Runtime Verification (RV)*. LNCS. Springer. To appear.

Krüger, I. H., Lee, G., and Meisinger, M. 2006. Automating software architecture exploration with M2Aspects. In *Workshop on Scenarios and state machines: models, algorithms, and tools (SCESM)*. ACM Press, 51–58.

Lhoták, O. and Hendren, L. 2003. Scaling Java points-to analysis using Spark. In *International Conference on Compiler Construction (CC)*. Lecture Notes in Computer Science (LNCS), vol. 2622. Springer, 153–169.

Maoz, S. and Harel, D. 2006. From multi-modal scenarios to code: compiling LSCs into AspectJ. In *Symposium on the Foundations of Software Engineering (FSE)*. ACM Press, 219–230.

Martin, M., Livshits, B., and Lam, M. S. 2005. Finding application errors using PQL: a program query language. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM Press, 365–383.

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

Naeem, N. A. and Lhoták, O. 2008. Typestate-like analysis of multiple interacting objects. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM Press, 347–366.

Rungta, N., Mercer, E. G., and Visser, W. 2009. Efficient testing of concurrent programs with abstraction-guided symbolic execution. In *Proceedings of the 16th International SPIN Workshop on Model Checking Software*. Springer-Verlag, Berlin, Heidelberg, 174–191.

Sridharan, M. and Bodík, R. 2006. Refinement-based context-sensitive points-to analysis for Java. In *Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 387–400.

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