

Partially evaluating finite-state runtime monitors ahead of time

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

Finite-state properties account for an important class of program properties, typically related to the order 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 completely 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. 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: tpestate analysis, static analysis, runtime monitoring

1. INTRODUCTION AND CONTRIBUTIONS

Finite-state properties constrain acceptable operations on a single object or a group of inter-related objects, depending on the object's or group's history. Tpestate systems [Strom and Yemini 1986], an instantiation of the idea of finite-state properties, enable the specification and (potentially static) verification of finite-state properties for program understanding and verification. One can define type systems [Bierhoff and Aldrich 2007; DeLine and Fähndrich 2004] that prevent programmers from writing code with tpestate errors. Unfortunately, current tpestate systems require elaborate program annotations, essentially to identify statements that may access an object (and hence modify its tpestate) and variables that may or must point to the same objects (i.e., may or must alias). Such annotations are hard to maintain, possibly explaining in part why such type systems have not been adopted.

Portions of this work were published in [Bodden et al. 2007; Bodden et al. 2008a; Bodden 2010; Bodden et al. 2010].

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© YYYY ACM 0164-0925/YYYY/01-ARTA \$10.00

DOI 10.1145/0000000.0000000 <http://doi.acm.org/10.1145/0000000.0000000>

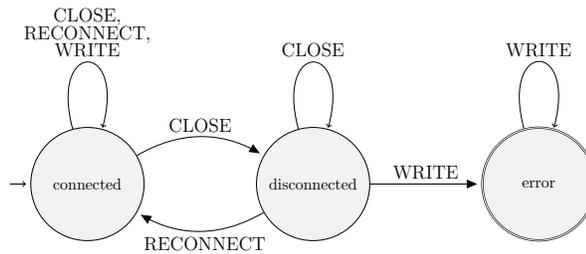


Fig. 1: “ConnectionClosed” finite-state property: no write after close.

A more 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]. The tracematches formalism combines regular expressions with AspectJ [Kiczales et al. 2001] pointcuts to provide a high-level specification language for runtime monitors. JavaMOP [Chen and Roşu 2007] is an open framework for notations that 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. The 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. Program annotations for state changes are replaced by non-intrusive AspectJ pointcuts. Program annotations for aliasing constraints are not necessary in our approach because we infer aliasing constraints through a combination of precise pointer analyses.

Static analyses for optimizing monitors have different requirements from fully-static approaches. Consider our running example, in which programmers must not write to a connection handle that is currently in its “disconnected” 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. (The looping transitions on the initial state implement our matching semantics, where the runtime monitor signals an error each time it reaches a final state. Because the machine is non-deterministic, the self-loops shown in the figure enable the monitor to report more than one violation for a single runtime trace.) 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, so our approach guarantees that executions of the generated monitor have no false positives or false negatives—it may

do so by deferring some decisions until runtime. Sound static approaches that only attempt to prove the absence of property violations but have no runtime monitoring component, e.g. [Fink et al. 2006], have no such option; deferring decisions is not acceptable. They can only declare false positives.

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]. The term “shadows” 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. statements causing events of type 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 removing transitions from that location will 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 that 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. We ourselves have successfully used CLARA in combination with tracematches and JavaMOP [Bodden et al. 2009; Bodden 2009].

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 free of program locations that could drive the monitor into an error state. In these cases, CLARA statically guarantees that the program can never violate the stated

property, eliminating the need for runtime monitoring of that program. In other cases, the residual runtime monitor will require less instrumentation than the original monitor, therefore yielding a 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. Due to the design of our analyses and the CLARA framework, our analyses are equally effective on any runtime monitor for a given property, independent of the source of the 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.

The relationship of this article to previous publications is as follows. Two of the contributions in this article are entirely novel: the design, implementation, and evaluation of the Certain-match Analysis (Section 8), and the Eclipse plugin for presenting analysis results to developers (Section 9). This article also presents improved—more precise or faster—versions of two previously-presented static analyses for eliminating unnecessary monitoring points (see Section 11.3 for details), and presents a complete description of the Nop-shadows Analysis (Section 7), for the first time, including its handling of pointers and inter-procedural effects. We have also improved the presentation of the soundness proofs for the static analyses from Bodden’s PhD thesis [2009]. In summary, this article presents a consolidated view of the entire CLARA framework, notably including all of its static analyses, the relevant correctness proofs, and a full evaluation, in a single publication.

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.

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 `disconnect`, `reconnect` and `write` events. When disconnecting a connection, the `CLOSE` advice places the closed connection object into the set `closed`. When the connection is re-connected, the `RECONNECT` 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 Ma-*

```

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 RECONNECT(Connection c) returning:
8     call(* Connection.reconnect()) && target(c) { closed.remove(c); }
9
10  dependent before WRITE(Connection c):
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, RECONNECT;
17    initial connected:  CLOSE -> connected,
18                       WRITE -> connected,
19                       RECONNECT -> connected,
20                       CLOSE -> disconnected;
21    disconnected: RECONNECT -> connected,
22                CLOSE -> disconnected,
23                WRITE -> error;
24    final error:      WRITE -> error;
25  }
26 }

```

Fig. 2: “ConnectionClosed” aspect with Dependency State Machine.

chine. 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. [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 tracematch [Allan et al. 2005] specification for ConnectionClosed. Tracematches

```

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

```

Fig. 3: “ConnectionClosed” tracematch.

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 (hence the keyword `sym`), and line 7 uses this alphabet to define the regular expression “CLOSE+ 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, a tool can easily make its generated aspects available for CLARA to analyze and optimize. In our experiments, we will focus on optimizing runtime monitors from tracematches. However, in previous work [Bodden et al. 2009], we have shown that our analyses are equally effective on monitors generated from other types of high-level specifications. The only requirement for CLARA’s clients is that the provided Dependency State Machine annotations must be semantically equivalent to the annotated runtime monitor. In Section 4.1 we provide a full dynamic semantics of Dependency State Machines, which clients can use to prove that the provided Dependency State Machine annotations are indeed correct.

```

1 public static void main(String args[]) {
2     Connection c = new Connection(args[0]);
3     c.disconnect(); // 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.disconnect(); // 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.disconnect(); // 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.disconnect(); // CLOSE(c)
5     c.write(args [0]); // WRITE(c): violation—write after close possible
6 }

```

(d) Example program which matches on some inputs

Fig. 4: Example programs

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 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 is triggered, 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 [Chen and Roşu 2009] over the pieces of advice applicable to that run. (The traces are parameterized by object identities.) 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 “ $\{\text{CLOSE}(c \mapsto o(c1))\}\{\text{WRITE}(c \mapsto o(c2))\}$ ”: the “disconnect” method call is only matched by the `CLOSE` piece of advice, and this piece of advice binds the aspect’s variable `c` to `o(c1)`, the object referenced by `c1`. Similarly, the “write” method call is only matched by the `WRITE` piece of advice and binds `c` to `o(c2)`. Parameterized traces give rise to “ground” traces by projection onto compatible variable bindings. In the above example, we can project onto $c \mapsto o(c1)$ and $c \mapsto o(c2)$. Projection onto $c \mapsto o(c1)$ yields the trace

“close”, while $c \mapsto o(c2)$ yields the trace “write”. Neither projected trace belongs to the language that the Dependency State Machine in Figure 2 accepts.

The program from Figure 4a yields the trace “ $\{\text{CLOSE}(c \mapsto o(c))\} \{\text{WRITE}(c \mapsto o(c))\}$ ”. Here, projection onto $c \mapsto o(c)$ 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 author’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), which comes with usage requirements expressed as finite-state properties. 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 programmer 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 many other runtime verification tools will likely support Dependency State Machines in the 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 monitor into the program under test, i.e., instruments the program with code that notifies the monitor about state transitions that the program performs. (CLARA uses the AspectBench Compiler [Avgustinov et al. 2005] for weaving.) CLARA then invokes its static analysis engine, which may include third-party static analyses. These analyses collect information about the finite-state property to approximate the set of relevant instrumentation points. Whenever an analysis declares that an instrumentation point does not affect whether or not the program violates the property, CLARA 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.

CLARA outputs a list of *potential failure groups*. Each 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, along with a set of context shadows, which trigger events on the same objects as the potential point of failure, and hence may contribute to the property violation. This presentation of our analysis results was 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.

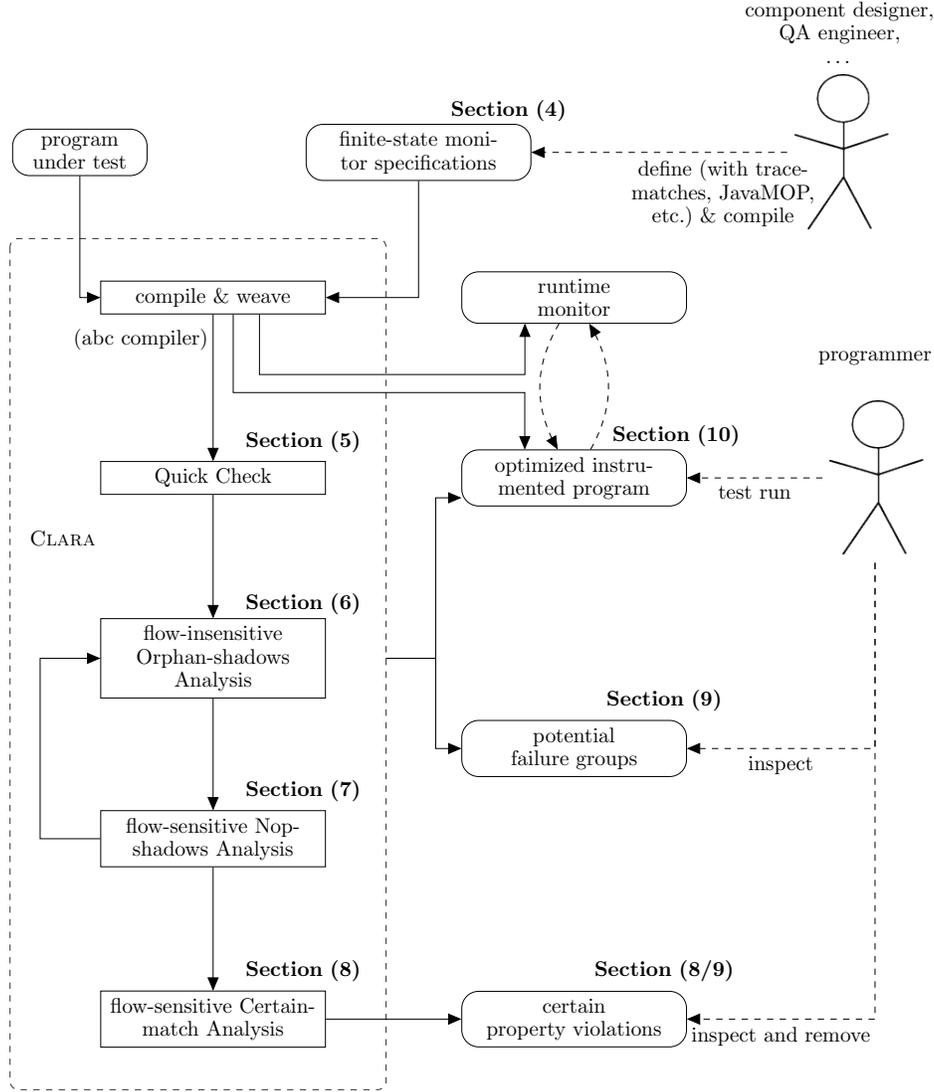


Fig. 5: Overview of CLARA

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 (final) states.

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 $w = (a_1, \dots, a_n)$ 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 $(q_0, q_{i_1}, \dots, q_{i_n})$ 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 $\text{pref}(w)$ as:

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

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} :

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

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

CLARA uses 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 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 `ConnectionClosed` from Section 1. Programs typically use many simultaneously-active connection objects, where each connection object could be in a different state. Modern runtime monitoring systems therefore allow the user to define *parametric* runtime monitors [Chen and Roşu 2009]. A parametric monitor effectively comprises a set of monitors, one monitor for every variable binding. CLARA's semantics is 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 $\beta : \mathcal{V} \rightarrow \mathcal{O}$. 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 $(a, \beta) \in \Sigma \times \mathcal{B}$. $\hat{\Sigma}$ is the set of all parametrized events. 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:

$$\{(\text{CLOSE}, c \mapsto o(c1))\} \{(\text{WRITE}, c \mapsto o(c2))\}$$

Here, $o(v)$ 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 if they agree on the objects that they jointly bind:

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

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

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

Definition 9 (Parametrized and projected event trace). Any finite program run induces a parametrized event trace $\hat{t} = \hat{e}_1 \dots \hat{e}_n \in \hat{\Sigma}^*$. 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 $e_{f(i)} := \hat{e}_i \downarrow \beta$ for all $i \in \{1, \dots, n\}$, with $f : [1, n] \rightarrow [1, m]$ order-preserving and $m \leq n$, then

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

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 $\text{groundTraces}(\hat{t})$ of non-empty ground traces of \hat{t} is:

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

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 execution. We say that \hat{t} violates the property described by \mathcal{M} at position i when:

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

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.

Correct definitions of Dependency State Machines. In the future we expect runtime verification tools that currently generate runtime monitors as AspectJ aspects to instead generate aspects annotated with Dependency State Machines. That way, the monitors become automatically analyzable by CLARA. The only requirement on CLARA's clients, i.e., on the runtime verification tools, is that the semantics of the generated Dependency State Machine

(as defined by Definition 11) must coincide with the semantics of the annotated monitor. Such proofs are simple to conduct for the two tools we tried, tracematches and JavaMOP, and we expect them to also be simple for other finite-state runtime monitoring tools.

4.2. Statically optimizing parametrized monitors

Our definition of the matching semantics for finite-state runtime monitors states 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.

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

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

The *mustMonitor* predicate depends on the predicate *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). Any sound implementation of *necessaryTransition* must satisfy:

$$\begin{aligned} \forall a \in \Sigma. \forall t = a_1 \dots a_i \dots a_n \in \Sigma^+. \forall i \in \mathbb{N}. \\ a = a_i \wedge \text{matches}_{\mathcal{L}}(a_1 \dots a_n) \neq \text{matches}_{\mathcal{L}}(a_1 \dots a_{i-1} a_{i+1} \dots a_n) \\ \implies \text{necessaryTransition}(a, t, i). \end{aligned}$$

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 prove that these approximations imply the soundness condition.

5. SYNTACTIC QUICK CHECK

The Quick Check is, as the name suggests, a simple analysis that can execute within milliseconds. This analysis rules out transitions that 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 available to the compiler after it inserts runtime monitoring instrumentation; it runs in time polynomial in runtime monitor size and linear in program size.

Examples and Discussion

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 first removes all WRITE-transitions from the finite-state machine. Next, the algo-

rithm finds that all states have become unproductive.¹ This way, the Quick Check correctly determines 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 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; on the remaining productive states 0, 2, and 3, a -transitions always loop. The Quick Check would report that b must be monitored, but that is moot, since b events cannot occur in our program.

Similarly, consider the case where the program can produce all events but d events. 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. This point applies to our subsequent analyses as well.

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. 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.

Algorithm

Algorithm 1 computes a set of symbols that need monitoring, given a set L of labels that occur. First, the set Δ_L filters out transitions from the monitor carrying labels that do not occur in the program. Next, the set Q_p retains only productive states from Q —states that are reachable through Δ_L from the initial state q_0 and which can reach some final state in Q_F . The set of productive transitions Δ_p then retains from Δ_L only those transitions that lead from and to productive states. Finally, the algorithm returns the set of symbols either appearing in non-looping productive transitions or for which a productive state has

¹A productive state is a state that is reachable from an initial state and from which some final state can be reached.

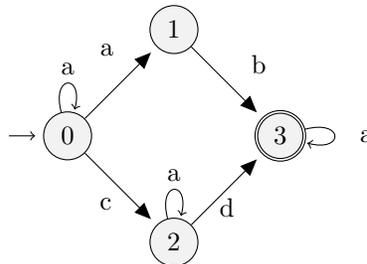


Fig. 6: Example automaton to illustrate Quick Check.

no outgoing transition at all. The latter class of symbols must remain, as they cause input words to be rejected.

Algorithm 1 Compute symbols required under Quick Check.

Input: set L of labels of all transition statements in the program

Output: set *symbolsThatNeedMonitoring*

```

1: let  $\Delta_L = \{(q_s, a, q_t) \in \Delta \mid a \in L\}$  in
2: let  $Q_p = \{q \in Q \mid \exists \text{ path in } \Delta_L \text{ from } q_0 \text{ to } q \wedge \exists \text{ path in } \Delta_L \text{ from } q \text{ to } q_f \in Q_F\}$  in
3: let  $\Delta_p = \{(q_s, a, q_t) \in \Delta_L \mid q_s \in Q_p \wedge q_t \in Q_p\}$  in
4: return  $\{a \in \Sigma \mid \exists q_s \in Q_p. ((\exists q_t \in Q_p. (q_s, a, q_t) \in \Delta_p \wedge q_s \neq q_t) \vee$ 
    $(\neg \exists q_t \in Q_p. (q_s, a, q_t) \in \Delta_p))\}$ 

```

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

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

Soundness of the Quick Check

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

$$\begin{aligned} \forall a \in \Sigma \quad \forall t = a_1 \dots a_i \dots a_n \in \Sigma^+ \quad \forall i \in \mathbb{N} : \\ a = a_i \wedge \text{matches}_{\mathcal{L}}(a_1 \dots a_n) \neq \text{matches}_{\mathcal{L}}(a_1 \dots a_{i-1} a_{i+1} \dots a_n) \\ \implies \text{necessaryTransitionQC}(a, t, i) \end{aligned}$$

PROOF. Follows immediately from Algorithm 1. Assume $\text{matches}_{\mathcal{L}}(a_1 \dots a_n) \neq \text{matches}_{\mathcal{L}}(a_1 \dots a_{i-1} a_{i+1} \dots a_n)$. Because the *matches* sets differ, we know that after having read the prefix $a_1 \dots a_{i-1}$, the automaton must either move from one productive state to another (ensured by the first disjunct of the return value and the definition of Δ_p) or it must move to no state at all (ensured by the second disjunct). In either case, $a \in \text{symbolsThatNeedMonitoring}$, so *necessaryTransitionQC*(a, t, i) holds. \square

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: it removes transitions which are ineffective because they are bound to objects that never match. The Orphan-shadows Analysis runs in time polynomial in runtime monitor size and in shadow count; it incurs setup overhead beyond the Quick Check, though, because it relies on points-to information. In particular, the Orphan-shadows Analysis models runtime objects using the static abstraction of points-to sets. For any program variable p , the points-to set $\text{pointsTo}(p)$ is the set of all allocation sites, as represented by **new** statements, that can reach p through a chain of assignments.

Example and Motivation

Recall the example from Figure 4b, which never matched because the CLOSE and WRITE events occurred on different objects: 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 $\text{pointsTo}(\mathbf{c1})$ and $\text{pointsTo}(\mathbf{c2})$ would be disjoint: it is impossible for **c1** and **c2** to point to the same object.

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 $\tilde{\beta}$. 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 “compatible 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, $compatible(\beta_i, \beta_j)$.

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:

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

Shadows. Programs contain *shadows*, which are static program points causing finite-state machine transitions. Pointcuts in the declaration of the dependency state machine induce shadows. Each shadow binds some number of variables. At runtime, shadows cause events, and their variables become bound to heap objects. 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 $stCompatible(s_1, s_2)$, if their bindings are statically compatible. Like most points-to analysis clients, our static analysis exploits the negation of $stCompatible$: it disregards transitions or events that can, in combination, only lead to incompatible variable bindings. Such events clearly cannot drive the runtime monitor into an error state.

Algorithm

Algorithm 2 presents the algorithm for the Orphan-shadows Analysis. This algorithm runs the Quick Check once for each shadow s . Each invocation of the Quick Check is told that only the labels for shadows compatible with s exist. Then s is necessary for Orphan-shadows Analysis iff, considering only the set of shadows compatible with s , the Quick Check declares that s is necessary.

To connect the Orphan-shadows Analysis to CLARA’s optimization engine, we proceed analogously to the Quick Check, and define the predicate $necessaryTransitionOSA$, as a second instantiation of $necessaryTransition$:

$$necessaryTransitionOSA(a, t, i) := shadow(a_i) \in necessaryShadows$$

6.1. Soundness of the Orphan-shadows Analysis

To show soundness for the Orphan-shadows Analysis (as per Section 4) we must show:

$$\begin{aligned} \forall a \in \Sigma \quad \forall t = a_1 \dots a_i \dots a_n \in \Sigma^+ \quad \forall i \in \mathbb{N} : \\ a = a_i \wedge matches_{\mathcal{L}}(a_1 \dots a_n) \neq matches_{\mathcal{L}}(a_1 \dots a_{i-1} a_{i+1} \dots a_n) \\ \implies necessaryTransitionOSA(a, t, i) \end{aligned}$$

PROOF. Assume $matches_{\mathcal{L}}(a_1 \dots a_n) \neq matches_{\mathcal{L}}(a_1 \dots a_{i-1} a_{i+1} \dots a_n)$. As with the Quick Check, after having read the prefix $a_1 \dots a_{i-1}$, 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 a_i -transition). In either case, the disequality implies that the transition at position i

Algorithm 2 Compute symbols required under Orphan-shadows Analysis

Output: set $necessaryShadows$

- 1: **function** compSyms(s) = $\{ l \in L \mid \exists s' \in \mathcal{S}. stCompatible(s, s') \wedge l = label(s') \}$
 - 2: **return** $\{ s \in \mathcal{S} \mid label(s) \in QuickCheck(compSyms(s)) \}$
-

must have a variable binding compatible with all bindings of all transitions at positions 1 through n . Therefore, by construction, it must hold that $shadow(a_i) \in necessaryShadows$ and hence $necessaryTransitionOSA(a, t, i)$. \square

6.2. 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 [2006]. Context-sensitive analyses distinguish objects that are allocated in different calling 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 an emptiness-of-intersection query for points-to sets $pointsTo(c1)$ and $pointsTo(c2)$, the wrappers compute a first approximation of $pointsTo(c1)$ and $pointsTo(c2)$. If this approximation is sufficient to determine that $pointsTo(c1) \cap pointsTo(c2) = \emptyset$, then the wrappers immediately return **false**. 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

The key technical contribution in this work is our novel Nop-shadows Analysis. This analysis is a flow-sensitive intra-procedural analysis, which incorporates inter-procedural information from the flow-insensitive Orphan-shadows Analysis. Its abstractions track, for each program statement, (1) the set of heap objects that could be in each state of the Dependency State Machine, and (2) sets of states from which objects could reach a final state. Using this information, our analysis identifies *nop shadows*, which are shadows that do not affect whether the Dependency State Machine can reach a final state.

Two main reasons motivated our design decision to use an intra-procedural analysis. 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. Most procedures appear to 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.

We next explain the need for flow-sensitivity by presenting 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 7.6, we discuss how our analysis handles loops, multiple methods, and events on arbitrary combinations of aliased objects.

7.1. Example

Figure 7 presents example code. It is annotated with a simplified version of the analysis information that our analyses 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. Because our example code only manipulates one connection object, we can defer our discussion of aliasing to Section 7.6.1.

While this example is contrived, it demonstrates the possibilities for optimization by taking control flow into account. (The flow-insensitive Orphan-shadows Analysis does not suffice: both disconnect and reconnect events occur on the same object.) 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 without monitoring this prefix. Conversely, the operations at lines 8 to 10 do not lead to a pattern violation and hence need no monitoring either. Soundness requires monitoring at least one of the two disconnects at lines 5 and 6, but not both.

Using the Nop-shadows Analysis results, the compiler need only retain instrumentation at lines 5 and 7, or 6 and 7—the minimal set of instrumentation points guaranteeing an optimized instrumented program will report an error if and only if the un-optimized program would have reported an error.

7.2. Analysis Overview

The Nop-shadows Analysis identifies nop shadows one at a time, by combining results from two analysis phases. The first phase uses a backward dataflow analysis to tell apart, for every statement in a method, (1) states which may possibly lead (in the rest of the program) to a final state (“hot states”) and (2) states that will never give rise to a final state (“cold states”). The second phase uses a forward dataflow analysis to compute possible monitor states at each statement s . The analysis can then identify nop shadows by combining results from the two phases.

Let shadow s lead from q to q' . Then s must be monitored, i.e., is not a nop shadow, if there is some continuation for which q and q' are not 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, leading to false positives if disabled; or (2) a transition from a cold state to a hot state, leading to false negatives if disabled. In (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 (2),

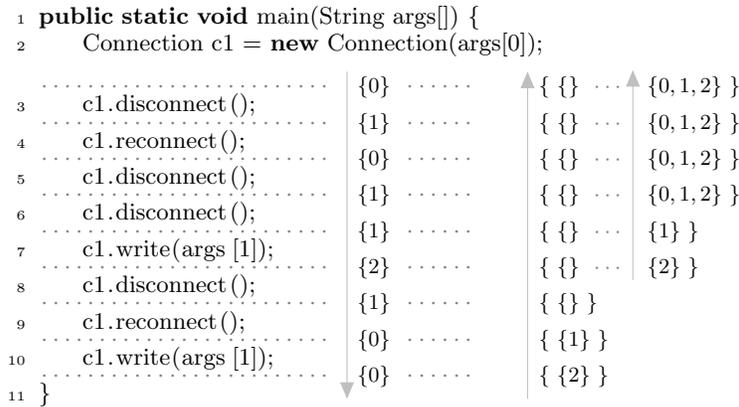


Fig. 7: Example program, annotated with combined analysis information.

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 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.

Dependencies between shadows require us to iterate our algorithm until it reaches a fixed point, removing one shadow at a time.

We describe the algorithm by first discussing the forward and backward phases, as they apply to a single set of variable bindings; Section 7.6.1 presents our full abstraction, which maps Dependency State Machine states to logical formulas over binding representatives.

7.3. Forward pass

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 determinized input state machine². CLARA builds the deterministic automaton using subset-construction:

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 $\text{det}(\mathcal{M})$ as $\text{det}(\mathcal{M}) := (\mathcal{P}(Q), \Sigma, \delta, Q_0, \hat{F})$ 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 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 —it does not merge states. In the sequel, we will denote the set of source states associated with s by $\text{sources}(s)$. Also, we will refer to the deterministic finite-state machine $\text{det}(\mathcal{M})$ as \mathcal{M}_{fwd} .

7.4. Backward pass

The backward analysis determines, for every statement s , sets of states that are “hot” at s ; 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 $\bar{\mathcal{L}}$, which consists of the reverse of every word 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 $\text{rev}(\mathcal{M})$ accepting $\bar{\mathcal{L}}$ by reversing the transition function.

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

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

Our backward analysis operates on the state machine

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

²Determinizing ensures that the state machine will be in only one state at a time. This simplifies the backward pass, which partitions the states into equivalence classes.

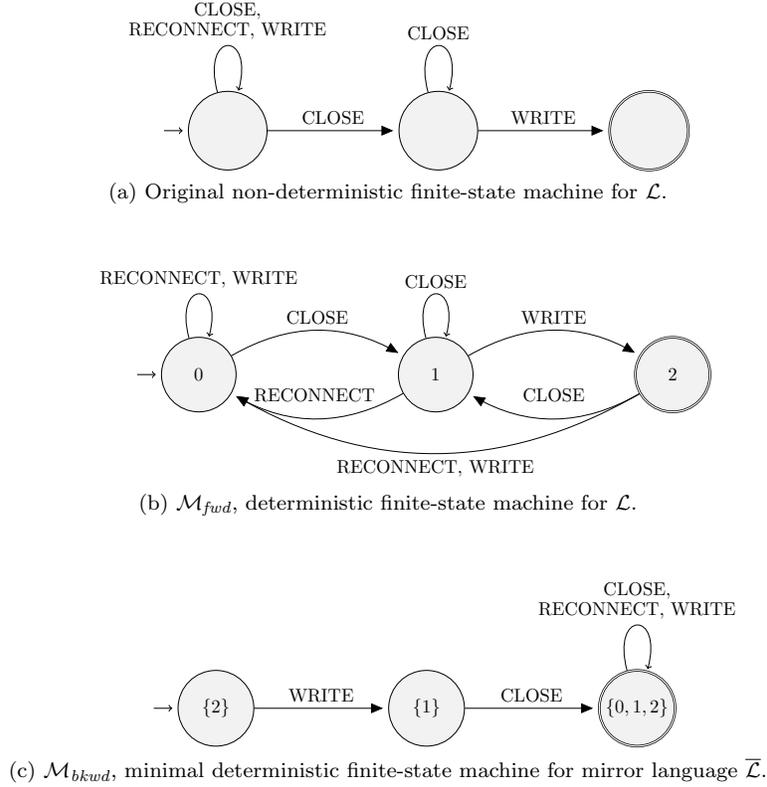


Fig. 8: Finite-state machines for Connection example.

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. 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 $det(rev(\mathcal{M}))$ (Figure 8b). For presentation purposes, we omitted the reject state from the Figure; the reject state represents the empty state set. By construction, \mathcal{M}_{bkwd} is minimal. (See [Brzozowski 1962] for a proof.)

The forward analysis conceptually starts at the beginning of the program execution. (Since our analyses are intra-procedural, we use Orphan-shadows Analysis to summarize caller effects). The backward analysis, on the other hand, starts at every statement which 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.

7.5. Determining Nop shadows

The analysis information in Figure 7 enables us to identify and disable nop shadows. Our notion of a nop shadow is related to the novel 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 reaches its final state at the same program points, whether we are in state q_1 or q_2 at s . We 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 $sources(s)$, let $target(q, s)$ 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:

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

We further define continuation-equivalence for states as:

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

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

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

The first conjunct states a nop-shadow transitions only between states that are in the same equivalence class. However, if $target(q, s) \in F$, 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.disconnect(); 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 $2 = target(2, s) = 2$, so we must explicitly handle such cases.

Examples. The disconnect statement at line 5 of Figure 7 has $source(s) = \{0\}$ and $target(0, s) = 1$. For both sets $Q_{fut} \in futures(s) = \{ \{\}, \{0, 1, 2\} \}$, $0 \in Q_{fut} \Leftrightarrow 1 \in Q_{fut}$. 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, $source(s) = \{1\}$ and $target(1, s) = 2$. The set $Q_{fut} = \{2\} \in futures(s)$ has $2 \in Q_{fut}$ but $1 \notin Q_{fut}$. Hence, $1 \not\equiv_s 2$, i.e., s is not a nop shadow, and must therefore remain enabled.

Remark on minimization. Although any machine accepting the mirror language would work for the backward analysis, the definition of nop shadows explains why we 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_{fut} of \mathcal{M}_{fwd} states than the un-collapsed sets would have been. Hence, after collapsing equivalent states, more sets Q_{fut} 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. For instance, in our example, disabling both CLOSE transitions 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. For each shadow-bearing method in the program, this algorithm iterates the forward and backwards passes, along with the Orphan-shadows Analysis (when necessary), greedily removing shadows until it reaches a fixed point. On our benchmarks, two iterations of the outer loop always sufficed.

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 the 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 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).

Our simplified discussion 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 an analysis that takes all of the above into account. We show that 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 [Bodden et al. 2011].

7.6. Full description of the Nop-shadows Analysis

We next present a sound implementation of the function *necessaryTransition* for the Nop-shadows Analysis. First, we motivate the need for 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

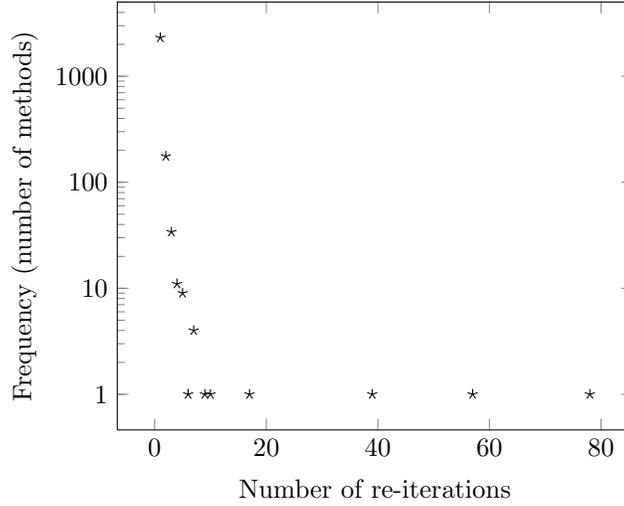


Fig. 9: Number of re-iterations per method (log scale).

analysis needs an analogue of projection to extract sub-traces for different variable bindings. We next define object representatives and explain how they are used in our abstraction.

7.6.1. Abstractions: object and binding 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. Because concrete runtime objects are not available at compile time, we use a static abstraction. We model a runtime binding $x = o(v1) \wedge y = o(v2)$ with a binding $x = r(v1) \wedge y = r(v2)$, where $r(vi)$ is the “object representative” of $o(vi)$. Object representatives are a static abstraction that uniformly incorporate aliasing information from multiple alias analyses. A full description of object representatives is beyond the scope of this paper; see [Bodden et al. 2008b] for details. In brief, object representatives almost transparently substitute for runtime objects by supporting both may-alias and must-alias queries. The must-alias relation is the equality relation for object representatives. That is, when our implementation generates a set of two object representatives $\{r_1, r_2\}$, then if r_1 and r_2 must-alias, we have $r_1 = r_2$, so the set reduces to $\{r_1\} = \{r_2\}$. This smaller representation saves time and memory during analysis.

At compile time, we implement object representatives as objects that are instantiated with a local variable v and statement s as parameter. An object representative hence represents the object pointed to by v at s . We omit s when it is clear from the context, or when v has a single assignment, and write $o(v)$. (Note that storing s is necessary because our internal representation is not in Static Single Assignment Form [Cytron et al. 1991]. If it were, s could be inferred from v .) Each object representative r_1 also has access to a flow-insensitive context-sensitive whole-program pointer analysis and to intra-procedural flow-sensitive must- and must-not-alias analyses, allowing the representative to decide aliasing relations to other object representatives r_2 on a best-effort basis (while defaulting to “may alias”, denoted $r_1 \approx r_2$). Table I summarizes the relations between object representatives.

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:

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

$r_1 \approx r_2$	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, while the negative information records which objects a variable cannot be bound to.

Binding representatives are key to our abstraction. Our analyses maintain, for each program statement and Dependency State Machine state, a set of binding representatives representing objects that could be in a given state (or that are hot, for the backward pass).

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 extend both partial functions to total functions by mapping a free variable v to \emptyset when no other mapping for v is defined.

For example, the binding representative

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

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 .) Also, 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:

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

This representation allows us to easily identify and perform simplifications on binding representatives; see [Bodden 2009] for details. These simplifications can reduce self-contradictory binding representatives to **false**, increasing the precision of our analysis.

Recall the important concept of compatibility, which we use to determine when a given shadow affects an abstract state. For any variable-to-object representative binding $\beta : \mathcal{V} \rightarrow \tilde{\mathcal{O}}$, and any binding representative $b = (\beta^+, \beta^-)$, we define compatibility between β and b :

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

That is, 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 $\beta_s = \text{shadowBinding}(s)$ of type $\mathcal{V} \rightarrow \tilde{\mathcal{O}}$. We will often write $\text{compatible}(s, b)$ in place of $\text{compatible}(\beta_s, b)$.

Using the notion of compatibility, we can define an inclusion relation on binding representatives, which we will use to accelerate our analysis passes: we don't propagate representatives that are subsumed by other representatives. Let b_1 and b_2 be binding representatives. Then b_2 is *at least as permissive* as b_1 , or $b_1 \subseteq_{\tilde{\mathcal{B}}} b_2$, if:

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

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:

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

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

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

7.6.2. 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 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 reversed transition functions.

Algorithm 4 *worklist(initial, succ_{cfg}, succ_{ext}, δ)*

```

1: wl := initial
2: before := after :=  $\lambda stmt. \emptyset$  // associate  $\emptyset$  with every statement
3: while wl non-empty do
4:   let (stmt, cs) = wl.pop() in
5:   // reduce configurations so that only most permissive ones remain
6:   let cstemp = cs  $\cup$  before(stmt) in
7:   let csnew =  $\{(Q_c, b_c) \in cs_{temp} \mid \nexists (Q_c, b'_c) \in cs_{temp}. b_c \subset_{\tilde{\mathcal{B}}} b'_c\} \setminus before(stmt)$  in
8:   let cs' = if shadows(stmt) =  $\emptyset$  then csnew
           else  $\{c' \mid \forall c \in cs_{new}. \forall s \in shadows(stmt). c' = \underline{transition}(c, s, \delta)\}$  in
9:   before.put(stmt, before(stmt)  $\cup$  cs)
10:  let cs'new = cs'  $\setminus after(stmt)$  in // filter out configurations already computed
11:  if cs'new non-empty then
12:    after.put(stmt, after(stmt)  $\cup$  cs'new)
13:    // add jobs for intra-procedural successor statements
14:    for stmt'  $\in succ_{cfg}(stmt)$  do
15:      wl.put(stmt', wl(stmt')  $\cup$  cs'new)
16:    end for
17:    // add jobs for inter-procedural successor statements
18:    for stmt'  $\in succ_{ext}(stmt)$  do
19:      wl.put(stmt', wl(stmt')  $\cup$  reachingStar(cs'new, relevantShadows(stmt)))
20:    end for
21:  end if
22: end while

```

While this algorithm builds on the worklist algorithm used in standard dataflow analyses, it differs in a number of ways: 1) it integrates the results of a flow-insensitive inter-procedural analysis at call sites and returns; 2) it does not merge dataflow facts at control-flow merges (and is hence intra-procedurally path-sensitive); 3) it processes individual configurations, not the entire in set for each statement at each iteration; and 4) it prunes subsumed configurations on-the-fly. Section 7.6.4 describes how our algorithm initializes the worklist. 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 $stmt$ whose successor configurations must be computed. The worklist is empty if it maps every statement to the empty

set. The algorithm also initializes mappings *before* and *after* to store previously-computed configurations. These mappings allow us to detect the fixed point.

Lines 6–7 implement an optimization. Effectively we conduct a merge operation on each statement’s *before* set. However, unlike most other merge operations, ours is lossless: it only removes redundant information, i.e., configurations that are subsumed by others also present in the set. 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 subsumed configurations (Q_c, b_c) where cs_{temp} 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) .) This optimization reduced the number of configurations computed for many methods 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 final result into cs_{new} . Since the algorithm previously computed successor configurations for these configurations (line 8), it need not recompute this information.

The remainder of the algorithm implements the standard work-list dataflow analysis algorithm, using *transition* as described below. Lines 17–20 handle inter-procedural control-flow by adding intra-procedural edges to the beginning or end of the current method (see Section 7.6.3 for details.)

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]. Because object representatives can approximately stand in for runtime objects at compile time, static analysis algorithms based on object representatives can closely resemble their corresponding runtime algorithms.

The transition function δ computes the set 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}_{bkw} .

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 3–6 compute successor configurations for all the variable bindings that move to Q_t , using the function *and*. In lines 8–9, 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 variable binding β_s which binds two variables, e.g. $x = r(\mathbf{v1}) \wedge y = r(\mathbf{v2})$. Then:

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

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 simplification rules to (1) return \perp 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 \perp means that the current configuration will not be propagated

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

```

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

```

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

```

1:  $\beta_{new}^+ := \beta^+, \beta_{new}^- := \beta^-$ 
2: for  $v \in dom(\beta_s)$  do
3:   if  $\beta_s(v) \in \underline{mustNotAliases}(\beta^+(v)) \vee \beta_s(v) \in \underline{mustAliases}(\beta^-(v))$  then
4:     return  $\perp$  // bindings were incompatible
5:   end if
6:   // add new positive binding
7:    $\beta_{new}^+ := \beta_{new}^+[v \mapsto \beta_{new}^+(v) \cup \{\beta_s(v)\}]$ 
8:   // prune superfluous negative bindings
9:    $\beta_{new}^- := \beta_{new}^-[v \mapsto \beta_{new}^-(v) \setminus \{r^- \mid r^- \neq \beta_s(v)\}]$ 
10: end for
11: return  $(\beta_{new}^+, \beta_{new}^-)$ 

```

any further (see Algorithm 5, lines 4 and 8). 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.

We explain our implementation of *and* (Algorithm 6), which adds bindings β_s to a binding representative (β^+, β^-) . 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 \perp 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 $\beta_s(v) \in \underline{mustAliases}(\beta_{new}^+(v))$ already, then it will not be added to the set again (by the design of the implementation). The case $\beta_s(v) \in \underline{mustNotAliases}(\beta^+(v))$ 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 $\beta_s = x \mapsto r(v)$ was just added to $\beta_{new}^+(v)$, then $x = r(v)$ implies that $x \neq r^-$ for all r^- where $r^- \neq r(v)$. Hence we can remove such object representatives r^- from $\beta_{new}^-(v)$ —such statements are implied by $x = r(v) \in \beta_{new}^+(v)$.

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 \perp . 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 .

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

```

1: if  $\beta_s(v) \in mustAliases(\beta^+(v))$  then
2:   return  $\perp$  // bindings were incompatible
3: end if
4: // negative binding redundant if it must-not-aliases a positive binding
5: if  $\beta_s(v) \in mustNotAliases(\beta^+(v))$  then
6:   return  $(\beta^+, \beta^-)$ 
7: else // return updated binding
8:   return  $(\beta^+, \beta^- [v \mapsto \beta^-(v) \cup \{\beta_s(v)\}])$ 
9: end if

```

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

7.6.3. The external-successor function $succ_{ext}$. One of the contributions of this article is a discussion of how we account for inter-procedural control-flow in our intra-procedural algorithm. These are handled in lines 17–20 of Algorithm 4. 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 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 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 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 19 of Algorithm 4 adds

³Because 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.

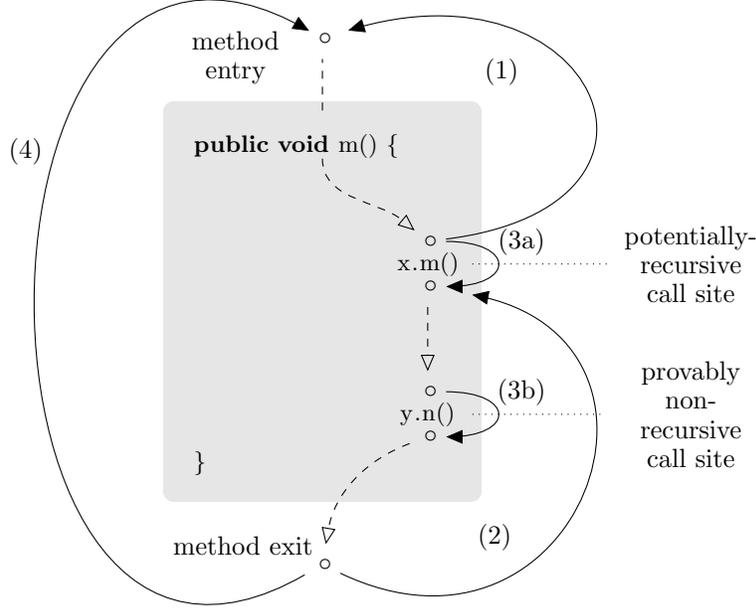


Fig. 10: Inter-procedural control-flow for the current method m .

$reachingStar(cs'_{new}, relevantShadows(stmt))$ as well as the ordinary set of configurations cs'_{new} for inter-procedural successors.

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 invoke, except for statements from m itself. We exclude m as we have explicitly accounted for its effects 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 (excluding m). We do not know which methods execute before or after m (even with the call graph), so we cannot reduce this set any further.

Given a statement $stmt$ and a set cs of configurations just before $stmt$, $reachingPlus$ 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 compatible with b_c can change Q_c . Hence, for every binding representative $b \in \tilde{\mathcal{B}}$ and shadow set $ss \subseteq \mathcal{S}$, we define $compL(b, ss)$ as:

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

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

We then define $reachingPlus(cs, stmt)$ as the set of configurations reachable from cs by applying at least one compatible shadow which is relevant to $stmt$. Formally,

$reachingPlus(cs, stmt)$ is the least fixed point satisfying:

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

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

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

Hence, $reachingStar$ computes the set of configurations reachable from cs by executing 0 or more shadows that are relevant at $stmt$.

7.6.4. 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 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 pass, 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 17–20. 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 intra-procedural. 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, m may also return to its caller, and then the remainder of the execution could drive the configuration into a final state using shadows in other methods. We therefore create jobs associating tail statements $stmt$ of m with configurations c in $reachingPlus(\{F\}, relevantShadows(stmt))$. This is almost dual to the initialization for the forward analysis, except 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.

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 :

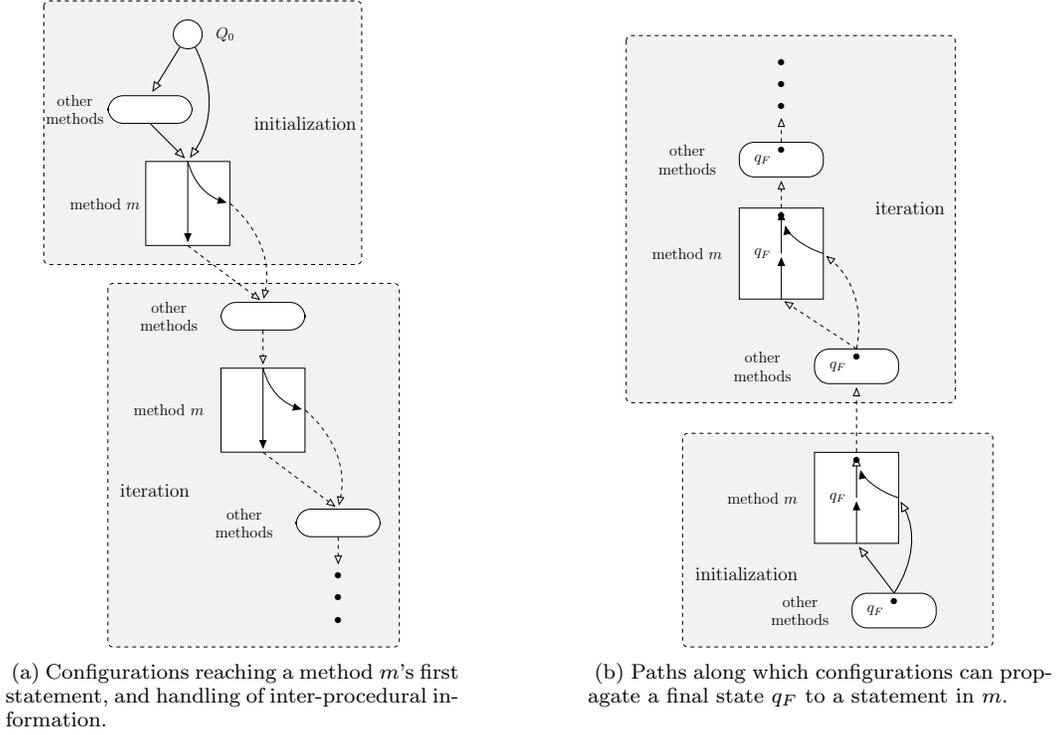


Fig. 11: Initialization and iteration of forward and backward passes.

$initial :=$

let $reachingConfigs = reachingPlus(\{(F, \top)\}, relevantShadows(stmt))$ **in**
 $\{ (stmt, \{(F, \top)\}) \mid \exists s \in shadowsOf(stmt) : \delta(F, label(s)) \neq \emptyset \} \cup$
 $\{ (tail, reachingConfigs) \mid tail \in tails(m) \cup recCall(m) \cup nonRecCall(m) \}.$

Here, δ denotes the transition function of \mathcal{M}_{bkwd} , $tails$ the return statements of m .

7.7. Optimizations for faster analysis

In Section 7.6.2 we discussed one important optimization that eliminated configurations that were less permissive than other configurations at the same statement. We now describe other optimizations that 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, accelerating graph look-ups. 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 benchmarks, abstracting the call graph was highly effective: on average, the abstracted graph had only 4.3% of the edges of the complete 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, which generally speeds up the analysis, but seems to cause an out-of-memory error on one of our benchmarks. We cache points-to sets, must-alias and must-not-alias analysis results for every method, and the set of methods transitively reachable through method calls. We also cache the set of currently-enabled shadows for each method. Naïvely caching enabled shadows does not work: a disabled shadow must be removed from its containing method, but the removal must also be visible when analyzing other methods later on—the cached list of shadows for that method is no longer valid.

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 imprecise information: `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 $b := c = r(c1) \wedge i = r(i1)$, and we want to compute $b \wedge i = r(i2)$. Precise points-to information would tell us that $r(i1) \neq r(i2)$ (as the iterators cannot be the same), so that:

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

However, since we only know $r(i1) \approx r(i2)$, the analysis cannot reduce “ $c = r(c1) \wedge i = r(i1) \wedge i = r(i2)$ ”. The many consecutive loops in `peelLoops(int)` greatly increase the size and number of configurations to be computed before the analysis reaches its fixed point. Worse yet, due to the imprecise pointer information, the analysis fails to find 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 configurations 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, comfortably exceeding the 8828 observed configurations. We believe that this value is high enough to yield excellent precision given precise pointer information; 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`. (`CLARA` did not abort analysis runs on any other benchmarks.)

Additionally, the benchmarks `bloat`, `chart` and `pmd` all use reflection in connection with collections. 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.

7.8. 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,

```

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).

```

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.

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* must respect:

$$\begin{aligned}
 \forall a \in \Sigma \forall t = a_1 \dots a_i \dots a_n \in \Sigma^+ \forall i \in \mathbb{N} : \\
 a = a_i \wedge \text{matches}(a_1 \dots a_n) \neq \text{matches}(a_1 \dots a_{i-1} a_{i+1} \dots a_n) \\
 \implies \text{necessaryTransition}(a, t, i).
 \end{aligned}$$

Helper definitions. We denote the transitive closure of a transition function δ by δ^* . 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. Assume that the Nop-shadows Analysis disables a shadow s that triggers the i -th event with $\text{label}(s) = a_i$. We will prove that, by construction,

$$\text{matches}(a_1 \dots a_{i-1} a_i a_{i+1} \dots a_n) = \text{matches}(a_1 \dots a_{i-1} a_{i+1} \dots a_n), \quad (1)$$

implying the soundness condition.

Consider a projected and therefore ground runtime trace $t = a_1 \dots a_i \dots a_n \in \Sigma^+$. For convenience, define $w_1 := a_1 \dots a_{i-1}$, $a := a_i$ and $w_2 := a_{i+1} \dots a_n$, i.e., we have that $a_1 \dots a_n = w_1 a w_2$. Let $\text{source} := \delta^*(q_0, w_1)$ and $\text{target} := \delta^*(q_0, w_1 a) = \delta(\text{source}, a)$. Since the Nop-shadows Analysis declares shadow s to be a nop shadow, Definition 13 provides:

$$\forall Q_{\text{fut}} \in \text{futures}(s). \text{source} \in Q_{\text{fut}} \iff \text{target} \in Q_{\text{fut}} \wedge \text{target} \notin F.$$

From the definition of *matches* we know that:

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

Therefore, prefixes of w_1 automatically satisfy Equation 1. We need only consider non-prefix words w where $w \notin \text{pref}(w_1)$. But we need not consider prefixes of $w_1 a$ either. Because $\text{target} \notin F$ we know that $w_1 a \notin \text{matches}(w_1 a w_2)$. Hence,

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

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

For such w , we need to show

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

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

$$\forall Q_{\text{fut}} \in \text{futures}(s). \text{source} = \delta^*(q_0, w_1) \in Q_{\text{fut}} \iff \text{target} = \delta^*(q_0, w_1 a) \in Q_{\text{fut}},$$

we know that $\mathcal{L}(\mathcal{M}_{\text{source}}) = \mathcal{L}(\mathcal{M}_{\text{target}})$. 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 these set of states that we approximate in our forward and backward passes are correct. In particular, our implementation must ensure that, for every set $Q_{\text{fut}} \in \text{futures}(s)$, the states $q \in Q_{\text{fut}}$ are continuation-equivalent, i.e., for all ground traces t , the continuation of the program execution after reading s satisfies,

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

Our implementation must therefore never merge state sets: merging may cause the analysis to assume invalid equivalencies. Our worklist algorithm, Algorithm 4 (page 24), ensures that state sets are never merged: while the algorithm does over-approximate pointer information in various ways, it never merges configurations with differing state sets Q_c . Every configuration $c = (Q_c, b_c)$ represents one element of the set *futures*. Algorithm 4 propagates these configurations but never merges them. In particular, 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- 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 simplification rules. When propagating configurations inter-procedurally along succ_{ext} , the algorithm does not refine binding representatives. Since unrefined binding representatives are at least as permissive as refined representatives, this is a sound over-approximation.

The soundness of the forward and backward analyses then follow from the fact that we (1) initialize both 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 them), taking into account all relevant shadows (as determined by *relevantShadow*), and (3) never merge configurations with differing state sets. \square

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 $s \in \mathcal{S}$:

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

As an example, consider again the write shadows in the the two pieces of code from figures 4a and 4d (page 6). Let s be the write shadow in Figure 4a. In this figure, there is only one possible execution, and this execution yields $\text{sources}(s) = \{\text{disconnected}\}$. Because $\text{target}(\text{disconnected}, \text{WRITE}) = \text{error} \in F$, the Certain-match Analysis will flag the write shadow as a certain match.

On the other hand, there are two possible execution paths leading to the write shadow in Figure 4d. One path closes the connection while the other one does not. Hence $\text{sources}(s) = \{\text{connected}, \text{disconnected}\}$. As a result, we obtain $\text{target}(\text{connected}, \text{WRITE}) = \text{connected} \notin F$, 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 complex, so 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. 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 classes unappealing. Instead, we offer users a 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 running 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.

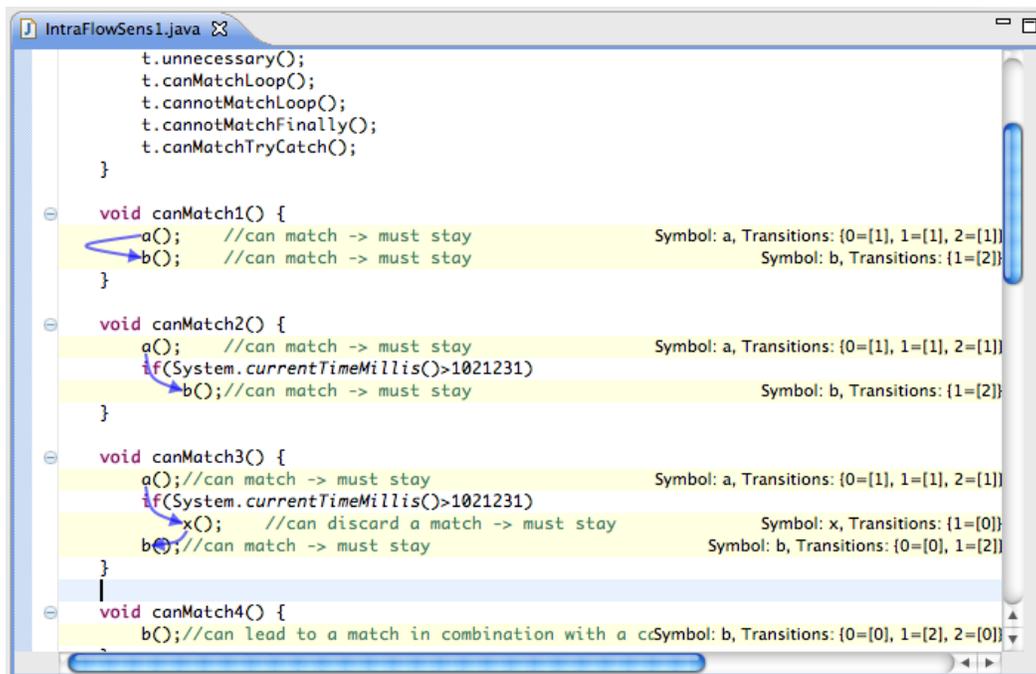


Fig. 14: Presentation of analysis results in the Eclipse IDE

property name	description
ASyncContainsAll	synchronize on <code>d</code> when calling <code>c.containsAll(d)</code> for synchronized collections <code>c</code> and <code>d</code>
ASyncIterC	only iterate a synchronized collection <code>c</code> when owning a lock on <code>c</code>
ASyncIterM	only iterate a synchronized map <code>m</code> when owning a lock on <code>m</code>
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 <code>hasMoreElements</code> before calling <code>nextElement</code> on an Enumeration
HasNext	always call <code>hasNext</code> before calling <code>next</code> on an Iterator
LeakingSync	only access a synchronized collection using its synchronized wrapper
Reader	do not use a Reader after its <code>InputStream</code> was closed
Writer	do not use a Writer after its <code>OutputStream</code> was closed

Table II: Monitored properties for classes of the Java Runtime Library

For our experiments, we wrote a set of twelve tracematch specifications for different properties of collections and streams in the Java Runtime Library. Table II 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, all our benchmarks, scripts, and CLARA itself are available at <http://www.bodden.de/clara/>. We maintain CLARA as an open-source project.

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 we defined. DaCapo contains eleven different workloads. We consider all but eclipse. Eclipse makes heavy use of reflection, which CLARA still has trouble with (see [Bodden et al. 2011]

for a possible strategy—beyond the scope of this work—for dealing with reflection). 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 3GB of heap space.

We will now 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 monitoring, and (3) the effectiveness of our Certain-match Analysis.

10.1. Fraction of shadows identified as irrelevant

Table III 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 must 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 trying to extend CLARA so that it can handle reflection using finer-grained approximations.

For fop/FailSafeIterMap, our analysis ran out of memory, despite having 3GB of heap space available. fop uses many maps and iterators, and the FailSafeIterMap tracematch induces 263 shadows in one of fop’s methods. We believe that this method is the reason that the analysis runs out of memory on fop. The fact that fop is the largest of our benchmarks seems to be coincidental; however, it does induce a total of 1116 relevant shadows with the FailSafeIter tracematch. Also, the points-to analysis does not seem to work well on fop, so that the Orphan-shadows Analysis is only able to eliminate 2 shadows for FailSafeIterMap. The Nop-shadows Analysis must therefore cope with 1114 shadows, a number much higher than for all other benchmark/property combinations (see [Bodden 2009] for details).

Note that we did not optimize our analysis implementation for memory consumption. Indeed, we cache information where possible. Hence, it may well be possible that other implementations of our algorithms could cope with fop/FailSafeIterMap, even with less than 3GB. We certainly do not believe that this limitation is fundamental.

10.2. Reduction of runtime overhead

To measure runtime overhead, we ran all benchmark/property combinations 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 III 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. Table IV quantifies

	antlr	bloat	chart	fop	hsqldb
ASyncContainsAll		0/71	0/6		
ASyncIterC		0/1621	0/498	0/146	0/33
ASyncIterM		0/1684	0/507	0/176	0/39
FailSafeEnum	0/76	0/3	0/1	6/18	0/120
FailSafeEnumHT	26/133	0/102	0/44	0/205	3/0/114
FailSafeIter	0/23	830/1394	149/510	0/288	0/112
FailSafeIterMap	0/130	444/1180	49/374	OOM	0/252
HasNextElem	0/117	0/4		0/12	0/53
HasNext		452/849	48/248	0/72	0/16
LeakingSync	0/170	0/1994	0/920	0/2347	0/528
Reader	0/50	0/7	0/65	0/102	3/1216
Writer	35/171	15/0/563	0/70	0/429	10/1378
	jython	luindex	lusearch	pmd	xalan
ASyncContainsAll	0/31	0/18	0/18	0/10	
ASyncIterC	0/128	0/149	0/149	0/671	
ASyncIterM	0/138	0/152	0/152	0/718	
FailSafeEnum	18/26/110	0/61	0/61	0/21	0/222
FailSafeEnumHT	33/28/153	0/37	0/37	0/100	0/319
FailSafeIter	112/253	0/217	11/5/217	287/546	0/158
FailSafeIterMap	133/250	0/136	0/136	204/583	0/540
HasNextElem	34/64	0/22	0/22	0/11	1/0/63
HasNext	0/74	0/74	184/346		
LeakingSync	0/1082	0/629	0/629	0/986	0/1005
Reader	4/0/139	0/226	0/226	0/102	0/106
Writer	0/462	0/146	0/146	0/62	0/751

Table III: 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. OOM = OutOfMemoryException during static analysis.

	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 IV: 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

the runtime overheads in more detail. We marked the 74 cases for which CLARA could prove that the program cannot violate the property on any execution with “✓”. 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

Our results show that the Certain-match Analysis was much less effective than the other analyses. Despite several runtime matches, the analysis found only one certain match: line 218 of method `InductionVarAnalyzer.isMu(..)` of `bloat`, with `HasNext`. The code is:

```

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. Hence, Certain-match Analysis did find what it was looking for. However, the finding also indicates a problem with our `HasNext` specification: it is fine for a client to call `next()` twice in a row, as long as the underlying collection has at least two elements. `bloat` maintains the internal invariant that `phi` nodes should always point to two blocks. Hence, the above situation does not depict a bug in `bloat`.

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 explanations 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 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 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 iterator. 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 (not 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.

10.4. Analysis time

We designed our analysis to be particularly efficient by separating it into three different stages. Earlier stages are cheaper to compute and reduce the load on more complex later stages. Moreover, while the first two stages are inter-procedural, the Nop-shadows Analysis and Certain-match Analysis are mostly intra-procedural: any information they use from the inter-procedural level is flow-insensitive and has already been computed by the Orphan-shadows Analysis once these analysis stages execute.

The Quick Check never took longer than one second to execute on any of our benchmarks. The Orphan-shadows Analysis took never longer than 91 seconds, where this time includes the time for points-to set and call-graph computation. The analysis itself never took longer than 17 seconds, with an average of 1.4 seconds. This second analysis state is fast because, despite being inter-procedural, it is flow-insensitive, and because we compute context information on demand, only for such variables that we care about and only for such variables that cannot already be determined to not alias without context information.

The third stage, the Nop-shadows Analysis, took under 50 seconds on average. This time includes all re-iterations of the Orphan-shadows Analysis and Nop-shadows Analysis that CLARA performs. In 90% of the cases, the analysis finished in under one minute. By far the worst case is `bloat-FailSafeIter` with just over 19 minutes of analysis time for this stage. The `bloat` benchmark contains methods with thousands of bytecode instructions, many of which use iterators and collections. This explains these extraordinarily high analysis times. The average analysis time for a single shadow-bearing method was about half a second.

In all but two cases, the total compilation and analysis time, including all three analysis stages, was under ten minutes. The combination `bloat-FailSafeIterMap` took almost 18 minutes in total, and `bloat-FailSafeIter` took just about 25 minutes in total.

11. RELATED WORK

We 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 like ours. We then discuss static verification of tpestate properties: the properties that CLARA verifies can be seen as tpestate 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 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.

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 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. CLARA’s specification language supports the annotation of J-LO-generated aspects with dependency information; CLARA’s static analyses could then remove some of J-LO’s 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. Our reported experimental results use optimized monitor implementations as a baseline, and show that combining monitor optimizations with our analyses yields low runtime overhead in most cases.

Another, orthogonal, approach to reducing runtime monitoring overhead is by using collaborative runtime verification. We previously explored collaborative runtime verification for tracematches [Bodden et al. 2010], partitioning the monitoring both spatially and temporally. Spatial partitioning creates many copies of the program being monitored; each program copy monitors a small subset of the full set of shadows. We found that most copies incurred no overhead over the un-instrumented program. Temporal partitioning switches the instrumentation off and on over time, 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, which is unavailable from Dependency State Machines.

JavaMOP. JavaMOP provides an extensible logic framework for specification formalisms [Chen and Roşu 2007]. JavaMOP accepts specifications in various formalisms and translates them into AspectJ aspects. Due to its generality, it makes few assumptions about any particular specification language. This generality makes it difficult, if not impossible, to analyze JavaMOP specifications. Feng Chen extended the JavaMOP implementation [Bodden et al. 2009] 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, and we have successfully used CLARA with monitors generated by JavaMOP [Bodden 2009].

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 ma-

chines. 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 “partique” 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 compare two fully-static checkers, PMD and FindBugs, to our Certain-match Analysis. PMD [Copeland 2005] aims to find violations of “best practices” or programming styles, rather than actual programming errors. For example: “A class that has private constructors and does not have any static methods or fields is unusable.” PMD has no support for data-flow analyses, so it cannot (in general) evaluate pointer nullness or variable initialization-before-use. FindBugs [Hovemeyer and Pugh 2004] is a static rule checker from the University of Maryland. FindBugs comes with a rich set of checkers that identify common problems when using certain popular libraries (e.g. 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 closer 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. Tpestate

The target class of Dependency State Machine program properties was inspired by tpestate systems. Tpestate 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 tpestate of an object to change during the computation. Strom and Yemini [1986] first proposed the idea of having a value’s type depend on an internal state—its tpestate. Operations can change a value’s type by changing that value’s tpestate.

Two critical differences between our work and the related work on tpestate are: 1) the treatment of false positives; and 2) how our (fundamentally intra-procedural) approach accounts for effects in other procedures.

False positives. Our work differs from much of the work below in its treatment of potential false positives. When our static analysis succeeds completely, it provides the same guarantee as classical tpestate: the program under analysis never violates the specified tpestate properties. However, our work originated in runtime monitoring, and thus supports a hybrid monitoring/recovery situation. In particular, CLARA can generate a program which evaluates residual monitors at runtime, with much less overhead than the full runtime monitor.

Our approach thus allows developers to provide specialized instrumentation and recovery code in the event of an actual run-time violation detected by a monitor. Fully-static approaches, on the other hand, typically emit a compile-time warning when they fail to verify typestate specifications. In the fully-static context, it is much more critical to eliminate false positives, so these analyses go to greater lengths to accurately analyze programs.

Inter-procedural analyses. Naeem and Lhoták [2008], Fink et al. [2006] and Das et al. [2002] all implement inter-procedural typestate analyses using IFDS [Horwitz et al. 1995]; a major challenge in developing a usable flow-sensitive inter-procedural analysis is ensuring scalability, especially for path-sensitive analyses. Our analyses, by contrast, are scalable because they are flow-insensitive at the inter-procedural level. Only our Nop-shadows Analysis is flow-sensitive, but only at the intra-procedural level. It otherwise uses summaries of caller and callee effects obtained from a flow-insensitive whole-program analysis.

Staged approach. Fink et al. [2006] present a static analysis of typestate properties for Java programs. Their approach resembles ours—we both use a staged analysis which starts with a flow-insensitive pointer-based analysis, followed by flow-sensitive checkers. However, the analysis in [Fink et al. 2006] allows only for specifications that reason about a single object at a time. This prevents programmers from expressing properties spanning multiple objects. Furthermore, Fink et al. aim to verify properties fully statically, with the implications discussed above. Finally, 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.

Integrating pointer information; path sensitivity. The ESP tool by Dor et al. [2004] implements inter-procedural typestate checking for large C programs using value flow simulation. Value flow simulation enables the use of a path-sensitive analysis on an inter-procedural level. A key feature of this approach is that, like ours, it takes may- and must- pointer information into account. Furthermore, both approaches separate the alias analysis from the analysis of the property in question.

Our memory abstraction differs from ESP’s in two ways: 1) while ESP tracks one object at a time, we track multi-object properties using binding representatives (Section 7.6.1), which store information about all relevant objects; and 2) within binding representatives, we track individual objects using object representatives, while ESP uses value alias sets. Value alias sets contain program expressions (e.g. variable names or heap access paths). Both memory abstractions separately track may- and must- information to enable strong updates; the key difference is our support for multi-object properties. It should be possible to use value alias sets instead of object representatives as a basis for binding representatives.

Our Nop-shadows Analysis also tracks a richer domain-specific property atop the memory abstraction. ESP only tracks the current state of the machine (like the forward pass of Nop-shadows Analysis), while we track both the current state and the set of states that are continuation-equivalent to a final state. This enables our Nop-shadows Analysis to identify shadows that do not meaningfully change the automaton state.

ESP was one of the first practical *path-sensitive* inter-procedural analyses; it correlates branches guarded by (simple variants of) the same predicate. To guard against combinatorial explosion, their analysis only distinguishes the effects of program paths which have different typestates. To do so, the analysis abstraction tracks constant propagation (or, potentially, other) information along with typestate information. The analysis then performs branch correlation—using propagated constants, it prunes infeasible paths.

ESP successfully scaled to run in hundreds of seconds on a 140,000 LOC C program. Unfortunately, the authors did not evaluate how well their approach would perform without branch correlation. It is therefore hard to judge how the additional information in the analysis abstraction impacts analysis speed and precision.

Like ESP, our Nop-shadows Analysis is also path-sensitive—it does not merge information from different branches. However, it does not track information about the values of scalar variables, and hence does not perform branch correlation. Evidence about the efficacy of branch correlation in ESP would help compare our analysis to ESP. Our benchmark results suggest that the lack of branch correlation does not appear to pose a significant problem on our benchmark set; as we discussed in Section 10.1, reflection seems to be more important than analysis precision. Because Nop-shadows Analysis is flow-sensitive only on an intra-procedural level, our analysis completes in a reasonable time despite being path-sensitive.

Annotation-based approaches. Bierhoff and Aldrich [2007] present an approach to enable the checking of typestate properties in the presence of aliasing. Their Plural tool 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 (potentially inferred) information about how access permissions and typestates change when a method is executed.

The modularity of the Plural approach implies that, 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. Furthermore, CLARA itself does not carry out any flow-sensitive whole-program annotations; scalability is mostly limited by the scalability of the pointer and call graph analysis. In any case, we have found that worst-case assumptions coupled with coarse-grained side-effect information are surprisingly effective for quickly analyzing the whole program.

DeLine and Fähndrich [2004]’s approach is similar in flavour to Bierhoff and Aldrich’s approach. Their Fugue tool checks .NET programs for conformance to typestate specifications statically, in the presence of aliasing. The authors present a programming model of typestates for objects with a sound modular checking algorithm. 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. Fugue forbids any state-changing operations on possibly-aliased objects. This makes Fugue’s type system less permissive than Bierhoff and Aldrich’s system, where even aliased objects can change states.

Safe regions. Like us, Dwyer and Purandare [2007] use typestate analyses to specialize runtime monitors. Their work identifies “safe regions” in the code using a 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. The authors summarize the effect of safe regions 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.

Their “safe region” analysis has the same goal as our work—both approaches enable a compiler to emit an optimized monitor, which will execute faster than the full monitor because it will execute fewer transitions at runtime. However, summary transitions decouple the link between the code locations that perform a state transition and the locations that actually cause these transitions. This decoupling may impede 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 whether each shadow in the program is a nop-shadow. It is difficult to directly compare the efficacy of the two approaches, due to the different design decisions taken for each approach. In any case, Dwyer and Purandare’s analysis is complementary to ours and should be easily implementable in CLARA. We encourage such an implementation.

11.3. Other static analyses for tracematches

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 staged analysis consisting of a Quick Check, a flow-insensitive Consistent-shadows analysis and a flow-sensitive Active-shadows analysis. The earlier Quick Check considered an entire state machine; it disabled checking of the whole state machine if the program could not reach any final state along any path. The present Quick Check acts on each path separately: when the state machine cannot reach a final state along 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 one path flags writes to an InputStream whose Reader was closed, and a second path flags writes to a Reader whose InputStream was closed.

The Consistent-shadows analysis yields identical results to the Orphan-shadows Analysis that we present here. However, the Orphan-shadows Analysis runs faster and uses less memory than the Consistent-shadows analysis. The Consistent-shadows analysis is a generate-and-test algorithm, which takes time exponential in the number of shadows. The Consistent-shadows analysis usually ran quickly enough to be usable, but 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 Active-shadows analysis abstraction only allowed 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. These choices made the earlier analysis so imprecise that it found no nop-shadows at all in largely the same benchmark set as the one in the present paper. The previous results, in combination with the work that we present here, show that choosing the right abstractions is key to obtaining good precision. CLARA uses precise intra-procedural flow-sensitive pointer information and context-sensitivity. This information can yield much optimization potential and therefore significantly improves over the earlier Active-shadows analysis.

In [Bodden et al. 2008a], we presented an analysis similar to the Nop-shadows Analysis that we present here, 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 in how we approximate inter-procedural control-flow. In earlier work, we assumed that any state machine instance could be in any state when entering the current method m . This is sound but imprecise. In the Nop-shadows Analysis that we present here, we instead use the function *reachingStar* to compute a better approximation. Also, we previously 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 here is not only more elegant, it is also more precise. Tainting makes a worst-case assumption about outgoing method calls. Our current implementation considers such method calls more precisely—it considers 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 CLARA distribution comes with an extensive set of test cases containing trace-matches, hand-written monitors, and monitors generated by JavaMOP that we annotated by hand. We found that annotating JavaMOP monitors with Dependency State Machines is easy. In particular, no other changes to the generated code are required.

The Nop-shadows Analysis presented in this paper was first published in [Bodden 2010], in much less detail: due to space restrictions, we did not discuss our treatment of pointers, nor our flow-insensitive handling of effects from 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 analysis to analyze typestate-like properties of multiple interacting objects at compile time. The main difference between their work and ours is that we have no flow-sensitive inter-procedural analysis, while Naeem and Lhoták do. One could integrate their analysis into CLARA.

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 through the bodies of called methods. This can lead to enhanced precision when multiple methods use combinations of objects that are relevant to a given specification, at the cost of analysis time.

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 “binding lattice”, which 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.

Naeem and Lhoták are 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 same set of potentially dynamically loaded classes.

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: programmers can 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 and reflective calls [Bodden et al. 2011], 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 tpestate 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. [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!

This work was supported in part by Canada's Natural Science and Engineering Research Council, the German Federal Ministry of Education and Research (BMBF) within EC SPRIDE, and by the Hessian LOEWE excellence initiative within CASED.

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.
- 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 tpestate 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 tpestate 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., SINSCHKE, J., OUESLATI, H., AND MEZINI, M. 2011. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *ICSE '11: International Conference on Software Engineering*. ACM, 241–250.
- BRZOWSKI, 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.
- COPELAND, T. 2005. *PMD Applied*. Centennial Books.
- CYTRON, R., FERRANTE, J., ROSEN, B., WEGMAN, M., AND ZADECK, F. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 4, 451–490.
- DAS, M., LERNER, S., AND SEIGLE, M. 2002. ESP: Path-sensitive program verification in polynomial time. In *Conference on Programming Language Design and Implementation (PLDI)*. Berlin, Germany, 57–68.
- DELINE, R. AND FÄHNDRICH, M. 2004. Tpestates for objects. In *European Conference on Object-Oriented Programming (ECOOP)*. Lecture Notes in Computer Science (LNCS), vol. 3086. Springer, 465–490.
- DOR, N., ADAMS, S., DAS, M., AND YANG, Z. 2004. Software validation via scalable path-sensitive value flow analysis. In *International Symposium on Software Testing and Analysis (ISSTA)*. Boston, MA, 12–22.
- DROSSOPOULOU, S., DAMIANI, F., DEZANI-CIANCAGLINI, M., AND GIANNINI, P. 2002. More dynamic object reclassification: Fickle II. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 24, 2, 153–191.
- DWYER, M. B. AND PURANDARE, R. 2007. Residual dynamic tpestate 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 tpestate 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.

- HORWITZ, S., REPS, T., AND SAGIV, M. 1995. Demand interprocedural dataflow analysis. In *Symposium on the Foundations of Software Engineering (FSE)*. 104–115.
- 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.
- KICZALES, G., HILSDALE, E., HUGUNIN, J., KERSTEN, M., PALM, J., AND GRISWOLD, W. G. 2001. An overview of AspectJ. In *European Conference on Object Oriented Programming (ECOOP'01)*. LNCS, vol. 2072. Springer, 327–353.
- 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.
- 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.