Int J Softw Tools Technol Transfer
DOI 10.1007/s10009-010-0183-5

RUNTIME VERIFICATION

The Clara framework for hybrid typestate analysis

Eric Bodden · Laurie Hendren

Abstract A typestate property describes which operations are available on an object or a group of inter-related objects, depending on this object's or group's internal state, the typestate. Researchers in the field of static analysis have devised static program analyses to prove the absence of typestate-property violations on all possible executions of a given program under test. Researchers in runtime verification, on the other hand, have developed powerful monitoring approaches that guarantee to capture property violations on actual executions. Although static analysis can greatly benefit runtime monitoring, up until now, most static analyses are incompatible with most monitoring tools. We present CLARA, a novel framework that makes these approaches compatible. With CLARA, researchers in static analysis can easily implement powerful typestate analyses. Runtime-verification researchers, on the other hand, can use CLARA to specialize AspectJ-based runtime monitors to a particular target program. To make aspects compatible to CLARA, the monitoring tool annotates them with so-called dependency state machines. CLARA uses the static analyses to automatically convert an annotated monitoring aspect into a residual runtime monitor that is triggered by fewer program locations. If the static analysis succeeds on all locations, this proves that the program fulfills the stated typestate properties, making runtime monitoring entirely obsolete. If not, the residual runtime monitor is at least optimized. We instantiated CLARA with three static typestate analyses and applied these analyses to monitoring aspects generated from tracematches. In two-thirds of the cases in our experiments, the static analysis succeeds on all locations, proving that the program fulfills the stated properties, and completely obviating the need for runtime monitoring. In the remaining cases, the runtime monitor is often significantly optimized.

Keywords Runtime monitoring · Static analysis · Aspect-oriented programming · Finite-state machines · Typestate

1 Introduction

A typestate property [34] describes which operations are available on a group of inter-related objects, depending on the group's internal state, the typestate. Software engineers use typestate properties to describe important properties of the programs they develop. For instance, a program should not write to a connection handle while the handle is in state "closed". Figure 1 shows a finite-state machine that expresses the language of all program executions that violate this property. At the beginning, a connection is in the "connected" state, but it can move to the "error" state when a "disconnect" event is followed by a "write". A "reconnect" event moves the connection back into the "connected" state, in which "write" operations are allowed.

Researchers in the static-analysis and programming-languages community have developed type systems [5,18] and static program analyses [9,12,13,20] that attempt to prove the absence of typestate-property violations for all possible executions of a given program. Type systems have the advantage of strong static guarantees: they typically prevent a programmer from writing a program that may contain typestate violations. On the other hand, they require many program annotations, which are hard to come up with and

E. Bodden (✉)
Software Technology Group, Technische Universität Darmstadt, Darmstadt, Germany
e-mail: eric.bodden@gmail.com

L. Hendren
Sable Research Group, McGill University, Montréal, Québec, Canada

Published online: 05 December 2010

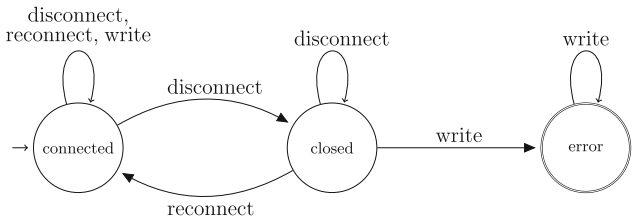


Fig. 1 "ConnectionClosed" example typestate property

```
1  aspect ConnectionClosed {
2      Set closed = new WeakIdentityHashSet();
3
4      after /*disconn*/ (Connection c) returning:
5          call(* Connection.disconnect()) && target(c) {
6          closed.add(c);
7      }
8
9      after /*reconn*/ (Connection c) returning:
10         call(* Connection.reconnect()) && target(c) {
11         closed.remove(c);
12     }
13
14     after /*write*/ (Connection c) returning:
15         call(* Connection.write(..)) && target(c) {
16         if(closed.contains(c))
17             error("May not write to "+c+": it is closed!");
18     }
19 }
```

Fig. 2 Monitoring aspect for "ConnectionClosed"

hard to maintain. Static typestate analyses require no such annotations, but have the disadvantage of executing a lot longer because they are not modular: type systems analyze one method or class at a time, while typestate analyses must typically analyze the entire program.

Researchers in runtime verification, on the other hand, have developed powerful runtime-monitoring tools [1,7,16,25,27]. These tools augment the program under test with a runtime monitor that is guaranteed to notify the programmer about typestate violations at runtime. Many of these dynamic-analysis tools use a two-staged approach to instrument the program under test: the tools generate instrumentation code in the form of AspectJ aspects. The user can then enable runtime checks for the program under test by weaving these aspects into the program. That way, AspectJ has become a popular intermediate representation for runtime monitors.

Figure 2 shows an AspectJ aspect that implements a runtime monitor for the "ConnectionClosed" example that we mentioned above. The aspect contains three pieces of advice, i.e., three code snippets that execute when certain events occur in the execution of the program under test. The first piece of advice, at lines 4–7, monitors calls to the `disconnect` method. When such a call is encountered, the piece of advice adds the target object of the call, i.e., the connection, to the set `closed`. The second piece of advice, in lines 9–12, processes calls to `reconnect` accordingly, removing the connection from the set again. The third piece of advice, in lines 14–18, first checks whether the connection that is written to is currently in the `closed` set. If it is, then the advice issues an error message.

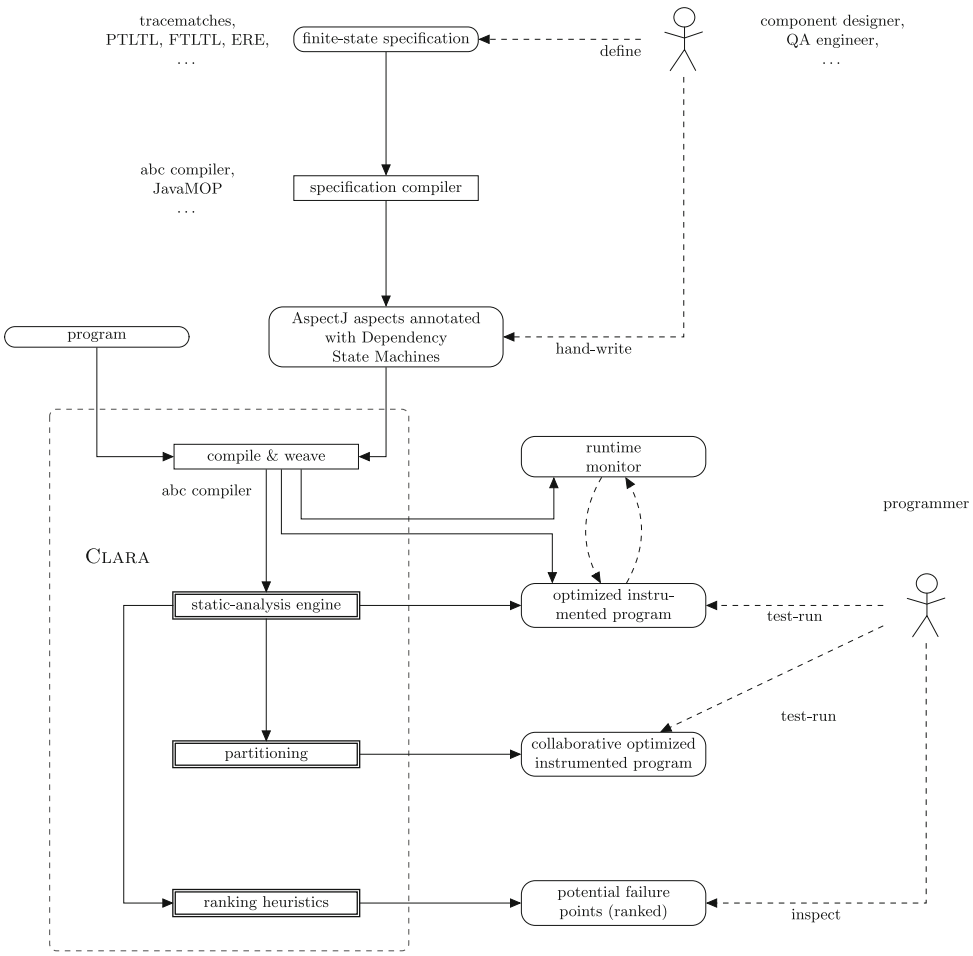
Runtime-verification, especially using such aspects, has several desirable properties. For instance, because finite-state specifications are evaluated at runtime, these specifications can be very expressive: they can refer to runtime events, compare runtime values and can evaluate predicates over the current heap. Also, when a runtime monitor detects a property violation, it can react to this violation in many different ways. A simple monitor could issue an error message, while a more involved monitor could try to work around the effects of the detected violation or revert the program to a safe state. Another positive property is that a runtime monitor can give the following guarantee: if a program run violates the property that the monitor describes, then the monitor will detect this violation.

On the other hand, runtime verification yields several drawbacks. One important drawback is that the instrumentation that is added to the program under test can yield a significant runtime overhead when test-running the program. After all, if the runtime monitor needs to monitor many events on a program run, the monitor has to consume a certain amount of execution time to update its internal state based on those events. Certain optimizations can be done and have to be done on the level of the runtime monitor itself: if the runtime monitor can compute every single state transition faster, then the instrumented program will run faster too. Avgustinov et al. [3] showed which optimizations are necessary to make runtime monitoring feasible. However, as we will show in our experiments, in some cases, these optimizations may not be sufficient.

A second important drawback of runtime verification is that it does not give any static guarantees. Test-runs and runtime monitoring can only show the absence of property violations on any single execution, but cannot prove their absence on all executions. To gain confidence in the program under test, the programmer has to achieve adequate test coverage, i.e., she has to test-run the instrumented program potentially many times. On the one hand this is time consuming, and on the other hand this may yield the problem that the programmer cannot say for certain when the instrumented program was tested enough. While an increasing number of different test runs can strengthen the confidence that the program will not violate the stated property on any execution, these test runs still do not constitute a proof. Therefore, it would be desirable to conduct a static analysis that can prove a program safe with respect to the finite-state property already at compile-time.

In this work, we present CLARA, a novel framework that aids researchers in implementing hybrid typestate analy-

Fig. 3 Overview of CLARA



ses, i.e., static analyses with a dynamic monitoring component [15]. Figure 3 gives an overview of CLARA. The major design goal of CLARA was to allow researchers to combine a wide range of static typestate analyses with a wide range of runtime monitors: CLARA takes monitoring aspects as input and should pose no special syntactic restrictions on the aspects that it processes. On the one hand, this gives researchers in runtime verification much flexibility when generating their aspects, but on the other hand, this decision poses problems: AspectJ aspects are, in full generality, Turing-complete, which makes it impossible for CLARA to extract the original finite-state property from the raw monitoring aspect. CLARA's static analyses, however, cannot function without a high-level description of these properties. To solve this problem, CLARA expects all provided monitoring aspects to be annotated with auxiliary information in the form of a "dependency state machine". We show a possible Dependency-State-Machine annotation for the ConnectionClosed example in lines 2–11 of Fig. 4. This annotation captures the typestate property directly as a textual representation of the underlying finite-state machine (Fig. 1). To allow for seamless integration with existing tools, we designed dependency state machines as a language extension to AspectJ. Section 2 will explain dependency state machines further.

As Fig. 3 shows, some software engineers first define finite-state properties of interest, in some finite-state formalism for runtime monitoring, such as tracematches [1], Extended Regular Expressions (ERE) or Future-time or Past-time Linear-Temporal Logic [30] (FTLTL/PTLTL). The engineer then uses some specification compiler, such as JavaMOP [16] or the AspectBench Compiler [2] (abc) to automatically translate these finite-state-property definitions into AspectJ monitoring aspects. Depending on the tool being used, these aspects may then already be annotated with appropriate dependency state machines: we implemented extensions to abc that generate these annotations automatically when transforming tracematches into AspectJ aspects. For other tools, such as JavaMOP, it should be easy to extend these tools so that they generate these annotations as well. In case the specification compiler does not support dependency state machines, the programmer can easily add the appropriate annotations to the generated aspects manually.

```
1  aspect ConnectionClosed {
2      dependency {
3          disconn, write, reconn;
4          initial connected:    disconn -> connected,
5                                write -> connected,
6                                reconn -> connected,
7                                disconn -> disconnected;
8                  disconnected: disconn -> disconnected,
9                                write -> error;
10         final    error;
11     }
12
13     Set closed = new WeakIdentityHashSet();
14
15     dependent after disconn(Connection c) returning:
16         call(* Connection.disconnect()) && target(c) {
17         closed.add(c);
18     }
19
20     dependent after reconn(Connection c) returning:
21         call(* Connection.reconnect()) && target(c) {
22         closed.remove(c);
23     }
24
25     dependent after write(Connection c) returning:
26         call(* Connection.write(..)) && target(c) {
27         if(closed.contains(c))
28             error("May not write to "+c+": it is closed!");
29     }
30 }
```

Fig. 4 Aspect with dependency state machine

CLARA then takes the resulting, annotated monitoring aspects as input, along with the program under test (either as Java source code or bytecode). CLARA first weaves the monitoring aspect into the program. The dependency state machine defined in the annotation provides CLARA with enough domain-specific knowledge to analyze the woven program. Researchers can add a number of static analyses to CLARA and have them applied in any order. When any of these analyses determine that an instrumentation point is irrelevant to all stated properties, i.e., can neither lead to a violation of this property, nor can prevent a property violation, then CLARA disables the instrumentation at this point. The result is an optimized instrumented program that updates the runtime monitor only at locations that remain enabled.

In addition, users can instruct CLARA to modify the advice dispatch of the monitoring aspect in such a way that the program under test can be used for Collaborative Runtime Verification [11]. In Collaborative Runtime Verification, different users are sent differently configured versions of the program under test, where each version only contains partial monitoring code. This usually helps to keep the monitoring overhead low. By design, this partitioning of instrumentation points is orthogonal to the static-analysis engine, i.e., it can be used in combination with any static analysis (or all of them).

Last but not least, CLARA comes with a set of built-in ranking heuristics. These heuristics rank all program points that CLARA reports as "potential point of failure" according to some confidence value. As a result, CLARA will often show first those program points at which the program certainly violates the stated typestate property. Program points at which a violation is still possible, but not likely, will show up further to the bottom of the ranked list. In addition, CLARA associates with each potentially property-violating program point all other program points that may have led up to this violation. This allows programmers to inspect the context of the violation easily.

To validate our approach, we first generated AspectJ-based runtime monitors from tracematches [1] (a form of regular expression). We modified the AspectBench Compiler [2], which implements tracematches by transforming them into regular AspectJ aspects, to automatically annotate the resulting aspects with dependency state machines, making the aspects compatible with CLARA. We then used CLARA to apply three different static analyses to all annotated monitor definitions.

Our results show that CLARA's analyses can effectively support programmers in two ways. First, the analyses reduce the number of program points that require instrumentation by large amounts. This makes it easier for programmer to investigate these program points to determine where a program could possibly violate the stated typestate properties. In many cases, CLARA even managed to eliminate all instrumentation, proving that the program cannot violate the stated properties on any execution. Second, the results show that CLARA can effectively reduce the runtime overhead that the runtime monitors induce, speeding up test runs and therefore easing the task of validating the typestate properties at runtime.

Our results further show that CLARA is sufficiently flexible to support typestate analyses of various levels of details and various AspectJ-based runtime-monitoring techniques. CLARA succeeds in decoupling the static typestate analyses from the runtime-monitoring code.

CLARA is freely available as open source at http://bodden.de/clara/, along with extensive documentation, the author's dissertation, which describes CLARA in detail, and with benchmark results.

We structured the remainder of this paper as follows. In Sect. 2, we explain the syntax and semantics of CLARA's main abstraction, dependency state machines. In Sect. 3 we briefly outline the three static example typestate analyses that we provide with CLARA. A discussion of our special code generation for collaborative runtime verification follows in Sect. 4. In Sect. 5 we comment on some experiments which show that the static analyses can indeed help to improve the performance of the supplied runtime monitors. We discuss related work in Sect. 6 and conclude in Sect. 7.

Fig. 5 Syntax of dependency state machines, as extension (shown in boldface) to the syntax of AspectJ

Modifier ::= "public" | "synchronized" | ... | "dependent".

AdviceDecl ::= *Modifier** [*RetType*] *BefAftAround* ***AdviceName***
"(" [*ParamList*] ")" [*AftRetThrow*] ":" *Pointcut Block*.

AdviceName ::= ID

AspectMemberDecl ::= *AdviceDecl* | ... | *DependencyDecl* | ***DependencySMDecl***.

DependencySMDecl ::= "dependency" "{" ***AdviceRefList*** ";" ***StateList*** ";" "}".

AdviceRefList ::= ***AdviceRef*** | ***AdviceRef*** "," ***AdviceRefList***.

AdviceRef ::= ***AdviceName*** | ***AdviceName*** "(" ***VarList*** ")".

VarList ::= ***VarName*** | ***VarName*** "," ***VarList***.

VarName ::= ID | "*".

StateList ::= ***State*** | ***State StateList***.

State ::= ***StateModifier**** *Identifier* [":" ***TransitionList***] ";".

StateModifier ::= "initial" | "final".

TransitionList ::= ***Transition*** | ***Transition*** "," ***TransitionList***.

Transition ::= *Identifier* "->" *Identifier*.

2 Dependency state machines

2.1 Syntax

Figure 4 already demonstrated our language extension using the ConnectionClosed example. Line 3 establishes the alphabet that the state machine is evaluated over. In our example, the alphabet of the regular language that the state machine accepts is {`disconn`, `write`, `reconn`}. Every symbol in the alphabet refers to a named "dependent" piece of advice in the same aspect. Note that with our language extension, we can assign the pieces of advice in lines 15, 20 and 25 of Fig. 4 proper names. In Fig. 2, which showed the aspect in plain AspectJ, this was not possible, and we therefore wrote the names as comments. In CLARA, only pieces of advice that are declared as "dependent" can have names. Other pieces of advice have no names and execute with AspectJ's standard semantics. Lines 4–10 enumerate all states in the state machine in question, and for each state enumerate further a (potentially empty) list of outgoing transitions. An entry "`s1: l → s2`" reads as "there exists an `l`-transition from `s1` to `s2`". In addition, a programmer can mark states as `initial` or `final`, i.e., accepting. We give the complete syntax for dependency state machines in Fig. 5, as a syntactic extension to AspectJ.

According to the semantics that we will give to dependency state machines, the dependency declaration in the ConnectionClosed example states that any piece of `disconn`, `write` or `reconn` advice must execute on a connection `c` whenever not executing this piece of advice on `c` would change the set of events at which the dependency state machine reaches its final state on `c`. (More on the semantics later). Note, however, that the symbol declarations in line 3 omit the variable name `c` of the connection: we just wrote `disconn, write, reconn`. We can do so because, by default, a dependency annotation infers variable names from the formal parameters of the advice declarations that it references (lines 15, 20 and 25 in the example). This means that the alphabet declaration in line 3 is actually a short hand for the more verbose form `disconn(c), write(c), reconn(c)`.

The semantics of variables in dependency declarations is similar to unification semantics in logic programming languages like Prolog [17]: The same variable at multiple locations in the same dependency refers to the same object. For each advice name, the dependency infers variable names in the order in which the parameters for this advice are given at the site of the advice declaration. Variables for return values from `after returning` and `after throwing` advice are appended to the end. For instance, the following advice declaration would yield the symbol createIter(c, i).

```
dependent after createIter(Collection c)
  returning(Iterator i): call(* Collection.iterator()) {}
```

We decided to allow for this kind of automatic inference of variable names because both code-generation tools and programmers frequently seem to follow the convention that equally named advice parameters are meant to refer to the same objects. That way, programmers or code generators can use the simpler short-form as long as they follow this convention. Nevertheless the verbose form can be useful in rare cases. Assume the following piece of advice:

```
dependent before detectLoops(Node n, Node m):
 call(Edge.new(..)) && args(n,m) {
  if(n==m) { System.out.println("No loops allowed!"); }}
```

This advice only has an effect when both `n` and `m` refer to the same object. However, due to the semantics of AspectJ,

the advice cannot use the same name for both parameters, which means that the inferred annotation would be detectLoops(n,m). The verbose syntax for dependent advice allows us to state nevertheless that for the advice to have an effect, both parameters actually have to refer to the same object, for instance `k`: **dependency**{ detectLoops(k,k); ... }.

2.2 Type-checking dependency state machines

After parsing, we impose the following semantic checks:

- A piece of advice carries a name if and only if it carries also a `dependent` modifier.
- Every advice must be referenced only by a single declaration of a dependency state machine.
- The state machine must have at least one initial and at least one final state.
- The listed alphabet may contain every advice name only once, i.e., declares a set.
- The names of states must be unique within the dependency declaration.
- Transitions may only refer to the names of advice that are named in the alphabet of the dependency declaration, and to the names of states that are also declared in the same dependency declaration.
- Every state must be reachable from an initial state.
- If the verbose form for advice references is used:
  - The number of variables for an advice name equals the number of parameters of the unique advice with that name, including the after-returning or after-throwing variable. (inference ensures this)
  - Advice parameters that are assigned equal names have compatible types: For two advice declarations `a(A x)` and `b(B y)`, with `a(p)` and `b(p)` in the same dependency declaration, `A` is cast-convertible [22, Sect. 5.5] to `B` and vice versa.
  - Each variable should be mentioned at least twice inside a dependency declaration. If a variable v is only mentioned once we give a warning because in this case the declaration states it no dependency with respect to v. The warning suggests to use the wildcard "*" instead. Semantically, * also generates a fresh variable name. However, by stating * instead of a variable name, the programmer acknowledges explicitly that the parameter at this position should be ignored when resolving dependencies.

Note that these checks are very minimal and allow for a large variety of state machines to be supplied. For instance, we do allow multiple initial and final states. We also allow the state machine to be non-deterministic. The state machine can have unproductive states from which no final state can be reached, and the state machine even does not have to be connected, i.e., it may consist of multiple components which are not connected by transitions. In this case, the state machine essentially consists of multiple state machines that share a common alphabet. Note that we forbid multiple dependency declarations to reference the same piece of advice: because these dependency declarations could use different alphabets the semantics of which would be unclear.

2.3 Semantics

We define the semantics of a dependency state machine as an extension to the usual advice-matching semantics of AspectJ [24]. In AspectJ, pieces of advice execute at "joinpoints", which are effectively periods of program execution. Programmers can determine the set of joinpoints at which a piece of advice should apply through "pointcuts", which are predicates over joinpoints. In the example from Fig. 2, the expression **call**(∗ Connection.disconnect()) && **target**(c) is a pointcut that picks out all method calls to the `disconnect` method of class `Connection`. At the same time, the pointcut binds the `target` object of the call to the variable `c`.

Let $\mathcal{A}$ be the set of all pieces of advice and $\mathcal{J}$ be the set of all joinpoints that occur on a given program run. We model advice matching in AspectJ as a function *match* that we regard as given by the underlying AspectJ compiler:

$$match : \mathcal{A} \times \mathcal{J} \to \{\beta \mid \beta : \mathcal{V} \rightharpoonup \mathcal{O}\} \cup \{\bot\}.$$

Let $\mathcal{V}$ be the set of all variable names and $\mathcal{O}$ the set of all runtime objects. For each pair of advice $a \in \mathcal{A}$ and joinpoint $j \in \mathcal{J}$, *match* returns $\bot$ in case a does not execute at j. If a does execute then *match* returns a variable binding β, a mapping from a's parameters to objects ({ } for parameter-less advice).

Based on this definition, we informally demand for any *dependent* piece of advice a, that a only has to execute when it would execute under AspectJ's semantics it and when it not executing a at j would change the set of joinpoints at which the dependency state machine reaches its final state for a binding "compatible" with β. (We define this term later).

Semantics by example. Figure 6 contains a small example program that we use to explain the intuition behind this semantics. The program contains several lines that trigger joinpoints that the ConnectionClosed aspects monitor. In AspectJ terminology, researchers frequently call a program point that triggers a joinpoint j the "joinpoint shadow" of j, or just "shadow" for short. The example program violates the ConnectionClosed property in lines 5 and 7 by first disconnecting the connection o(`c1`) and then writing to o(`c1`). (For any variable `v`, we use o(`v`) to refer to the object that `v` references). The joinpoint shadows [28] at these two lines are also the only two shadows in the program that the

```
1  public static void main(String args[]) {
2      Connection  c1 = new Connection(args[0]),
3                  c2 = new Connection(args[1]);
4      c1.write(args[2]);    //write(c1)
5      c1.disconnect();      //disconnect(c1)
6      c1.disconnect();      //disconnect(c1)
7      c1.write(args[2]);    //write(c1)
8      c1.disconnect();      //disconnect(c1)
9      c2.write(args[2]);    //write(c2)
10 }
```

Fig. 6 Example program

ConnectionClosed monitoring aspect from Fig. 2 must monitor so that this aspect correctly issues its error message at runtime. In particular, since the connection starts off in its initial state "connected", the `write` event at line 4 has no impact on the connection's state: the monitor loops on state "connected", and hence we call the `write` shadow at this line "irrelevant". Similarly, at line 6, the monitor is guaranteed to be in the "closed" state. Monitoring further `disconn` events does not change the automaton state in this situation either. Hence, the `disconn` shadows at this line is irrelevant as well. The `disconn` event at line 8 does cause a state change (from "connected" to "closed"), but this state change does not matter: because no `write` event ever follows on $o(\texttt{c1})$, this state change cannot impact the set of future joinpoints at which the dependency state machine reaches its final state (because there are none), and hence cannot impact the set of joinpoints at which the runtime monitor will have a visible effect, i.e., will issue its error message. This is true even though another `write` event which follows at line 9. This latter `write` event occurs on `c2` and not on `c1`. Because we know that `c2` cannot possibly reference the same object as `c1`, i.e., $o(\texttt{c1}) \neq o(\texttt{c2})$, this `write` event is not what we call "compatible" with the `disconn` event at line 8.

Formal semantics. In our view of AspectJ, pieces of advice are matched against "parameterized traces", i.e., traces that are parameterized through variable bindings. The semantics of state machines are usually defined using words over a finite alphabet Σ. In particular, state machines as such have no notion of variable bindings. In the following, we will call traces over Σ, which are given as input to a dependency state machine "ground traces", as opposed to the parameterized trace that the program execution generates. We will define the semantics of dependency state machines over ground traces. We obtain these ground traces from the parameterized execution trace by projecting each parameterized event onto a set of ground events. This yields a set of ground traces—one ground trace for every variable binding.

Further, we will define the semantics of dependency state machines in terms of "events", not joinpoints. Joinpoints differ from events in that joinpoints describe regions in time, while events describe atomic points. A joinpoint has a beginning and an end, and code can be executed before or after the joinpoint (i.e., at its beginning or end) or instead of the joinpoint. In particular, joinpoints can be nested. For instance, a field-modification joinpoint can be nested in a method-execution joinpoint. Pieces of advice, even "around advice", execute at atomic events before or after a joinpoint. Because these events are atomic, they cannot be nested. Joinpoints merely induce these events.

Event. Let j be an AspectJ joinpoint. Then j induces two events, j_{before} and j_{after} which occur at the beginning, respectively, end of j. For any set $\mathcal{J}$ of joinpoints we define the set $\mathcal{E}(\mathcal{J})$ of all events of $\mathcal{J}$ as:

$$\mathcal{E}(\mathcal{J}) := \bigcup_{j \in \mathcal{J}} \{j_{\text{before}}, j_{\text{after}}\}.$$

In the following, we will often just write $\mathcal{E}$ instead of $\mathcal{E}(\mathcal{J})$, if $\mathcal{J}$ is clear from the context.

For any declaration of a dependency state machine, the set of dependent-advice names mentioned in the declaration of the dependency state machine induces an alphabet Σ, where every element of Σ is the name of one of these dependent pieces of advice. For instance, the alphabet for the ConnectionClosed dependency state machine from Fig. 2 would be $\Sigma = \{\text{disconn}, \text{write}, \text{reconn}\}$. Matching these pieces of advice against a runtime event e results in a (possibly empty) set of matches for this event, where each match has a binding attached. We call this set of matches the parameterized event $\hat{e}$.

Parameterized event. Let $e \in \mathcal{E}$ be an event and Σ be the alphabet of advice references in the declaration of a dependency state machine. We define the parameterized event $\hat{e}$ to be the following set:

$$\hat{e} := \bigcup_{a \in \Sigma} \{(a, \beta) \mid \beta = \mathit{match}(e, a) \wedge \beta \neq \bot\}.$$

Here, $\mathit{match}(e, a)$ is the "usual" matching function that the original AspectJ semantics provide, overloaded for events.

We call the set of all parameterized events $\hat{\mathcal{E}}$:

$$\hat{\mathcal{E}} := \bigcup_{e \in \mathcal{E}} \{\hat{e}\}$$

It is necessary to consider sets of matches because multiple pieces of advice can match the same event. While this is not usually the case, we decided to cater for the unusual cases, too. As an example, consider the dependency state machine in the UnusualMonitor aspect in Fig. 7a. The aspect defines a dependency between two pieces of advice `a` and `b`. Note that the pointcut definitions of `a` and `b` overlap, i.e. describe non-disjoint sets of program events. The advice `b` executes before all non-static calls to methods named `foo`. The advice `a` executes before these events too, because, by its definition, it executes before any non-static method call.

```
1  aspect UnusualMonitor {
2      dependency{
3          a, b;
4          //transitions omitted from example
5      }
6
7      dependent before a(Object x):
8        call(* *(..)) && target(x) { ... }
9
10     dependent before b(Object x):
11       call(* foo(..)) && target(x) { ... }
12 }
```

(a)

```
1 SomeClass v1 = new SomeClass();
2 SomeClass v2 = new SomeClass();
3 v1.foo(); v1.bar(); v2.foo();
```

(b)

Fig. 7 UnusualMonitor aspect and example program. (**a**) UnusualMonitor aspect with overlapping pointcuts. (**b**) Example program

Next, assume that we apply this aspect to the small example program in Fig. 7b. We show the program's execution trace in the first row of Fig. 8 (to be read from left to right). This execution trace naturally induces the parameterized event trace that we show in the second row of the figure: this trace is obtained by matching at any event every piece of advice against this event.

Next we explain how we use projection to obtain "ground traces", i.e. Σ-words, from this parameterized event trace.

Projected events. For every $\hat{e} \in \hat{\mathcal{E}}$ and binding β we define a projection of $\hat{e}$ with respect to β. Projection yields a set of ground events from Σ which an ordinary finite-state machine can process. The projection with respect to β contains all ground events a that are, in $\hat{e}$, associated with a binding β_a that is "compatible" with β. In other words, the projection contains all ground events at $\hat{e}$ that may refer to the same objects as β:

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

Here, *compatible* is a relation over binding as follows:

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

In this equation, $\mathit{dom}(\beta_i)$ denotes the domain of β_i, i.e., the set of all variables that β_i assigns a value. This means that β_1 and β_2 are compatible as long as they do not assign different objects to the same variable.

Parameterized and projected event trace. Any finite program run induces a parameterized event trace $\hat{t} = \hat{e}_1, \ldots, \hat{e}_n \in \hat{\mathcal{E}}^*$. For any variable binding β we define a set of projected traces $\hat{t} \downarrow \beta \subseteq \Sigma^*$ as follows. $\hat{t} \downarrow \beta$ is the smallest subset of Σ^* for which holds:

$$\forall t = e_1, \ldots, e_n \in \Sigma^* : \text{ if } \forall i \in \mathbb{N} \text{ with } 1 \leq i \leq n : e_i \in \hat{e}_i \downarrow \beta \text{ then } t \in \hat{t} \downarrow \beta$$

We call traces like t, which are elements of Σ^*, "ground" traces, as opposed to parameterized traces, which are elements of $\hat{\mathcal{E}}^*$.

For our example, the third and fourth row of Fig. 8 show the four ground traces that result when projecting this parameterized event trace onto the variable bindings $x = o(\mathtt{v1})$ and $x = o(\mathtt{v2})$. For $x = o(\mathtt{v1})$ we obtain the two traces "aa" and "ba", for $x = o(\mathtt{v2})$ we obtain the two traces "a" and "b".

A dependency state machine will reach its final state (and the related aspect will have an observable effect, e.g., will issue an error message) whenever a prefix of one of the ground traces of any variable binding is in the language described by the state machine. This yields the following definition.

Set of non-empty ground traces of a run. Let $\hat{t} \in \hat{\mathcal{E}}^*$ be the parameterized event trace of a program run. Then we define the set $\mathit{groundTraces}(\hat{t})$ of non-empty ground traces of $\hat{t}$ as:

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

We intersect with Σ^+ to exclude the empty trace. This is because the empty trace cannot possibly cause the monitoring aspect to have an observable effect.

The semantics of a dependency state machine

We define the semantics of dependency state machines as a specialization of the predicate $\mathit{match}(a, e)$, which models the decision of whether or not the dependent advice $a \in \mathcal{A}$

execution trace	`v1.foo();`	`v1.bar();`	`v2.foo();`
parameterized trace $\hat{t}$	$\{(a, x = o(\mathtt{v1})), (b, x = o(\mathtt{v1}))\}$	$\{(a, x = o(\mathtt{v1}))\}$	$\{(a, x = o(\mathtt{v2})), (b, x = o(\mathtt{v2}))\}$
projected ground traces for $\hat{t} \downarrow x = o(\mathtt{v1})$	a b	a a	
projected ground traces for $\hat{t} \downarrow x = o(\mathtt{v2})$			a b

Fig. 8 Traces resulting from code in Fig. 7; note that $o(\mathtt{v1}) \neq o(\mathtt{v2})$

matches at event $e \in \mathcal{E}$, and if so, under which variable binding. As noted earlier, this predicate *match* is given through the semantics of plain AspectJ. We call our specialization *stateMatch* and define it as follows:

$$\begin{aligned}
&\mathit{stateMatch}: \ \mathcal{A} \times \hat{\mathcal{E}}^* \times \mathbb{N} \ \rightarrow \ \{\beta \mid \beta : \mathcal{V} \rightharpoonup \mathcal{O}\} \cup \{\bot\} \\
&\mathit{stateMatch}(a, \hat{t}, i) \\
&\quad = \text{let } \beta = \mathit{match}(a, e) \text{ in} \\
&\qquad \begin{cases} \beta & \text{if } \beta \neq \bot \wedge \exists t \in \mathit{groundTraces}(\hat{t}) \\ & \quad \text{such that } \mathit{necessaryShadow}(a, t, i) \\ \bot & \text{else} \end{cases}
\end{aligned}$$

As we can see, *stateMatch* takes as arguments not only the piece of advice for which we want to determine whether it should execute at the current event, but also the entire parameterized event trace $\hat{t}$, and the current position i in that event trace. Note that $\hat{t}$ contains also future events that are yet to come. This makes the function *stateMatch* undecidable. This is intentional. Even though there can be no algorithm that decides *stateMatch* precisely, we can derive static analyses that approximate all possible future traces. The function *necessaryShadow* mentioned above is a parameter to the semantics that can be freely chosen, as long as it adheres to a certain soundness condition that we define next. We say that a static optimization for dependency state machines is sound if it adheres to this condition.

Soundness condition. The soundness condition will demand that an event needs to be monitored if we would miss a match or obtain a spurious match by not monitoring the event. A dependency state machine $\mathcal{M}$ matches, i.e., causes an externally observable effect after every prefix of the complete execution trace that is in $\mathcal{L}(\mathcal{M})$, the language that $\mathcal{M}$ accepts.

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

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

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 in $\mathcal{L}$:

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

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

As before, the predicate *necessaryShadow* can be freely chosen, as long as it adheres to the following soundness condition:

Soundness condition. Let $\mathcal{L} := \mathcal{L}(\mathcal{M})$. For any sound implementation of *necessaryShadow* we demand:

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

Hence the soundness condition states that, if we are about to read a symbol a, then we can skip a if the monitoring aspect would have an observable effect when processing the complete trace t just as often (and at the same points in time) as it would when processing the partial trace where $t_i = a$ is omitted.

2.4 Approximations of dependency state machines

It may be interesting to know whether it is possible to provide CLARA with approximate dependency state machines as input that do not fully reflect the actual property that is monitored. This may be of particular concern when the runtime monitor is supposed to monitor a language $\mathcal{L}$ that cannot be fully expressed with a finite-state machine monitoring a regular language $\mathcal{L}_{\text{DSM}}$. In such cases, we can give different guarantees for the following two situations:

1. $\mathcal{L} \subseteq \mathcal{L}_{\text{DSM}}$: In this case, CLARA guarantees that the static optimizations will not introduce any false negatives. In other words, every trace that would cause the un-optimized monitor to reach a final state and execute its error handler will also do so for the optimized monitor. However, there may be false positives: the optimized monitor may miss some events that would otherwise reset the monitor into a "safe state" and therefore may signal some errors that did not actually occur.
2. $\mathcal{L} \supseteq \mathcal{L}_{\text{DSM}}$: In this case, the situation is just inverted. CLARA guarantees the absence of false positives, but cannot guarantee that the optimized monitor will actually signal all property violations.

If neither of the above inclusions hold between the specified and the monitored language then CLARA cannot give any guarantees: there may be false positives as well as false negatives.

3 Included analyses

In version 1.0, CLARA comes pre-equipped with three static analyses. As shown in Fig. 3, CLARA executes these analyses right after weaving, in its static-analysis engine. All analyses have direct access to all declared dependency state machines and to the woven program. Further, the analyses have access

to a list of joinpoint shadows, i.e., a list of those program points that may trigger events that the monitoring aspects react to. For every such shadow, CLARA exposes the following pieces of information:

- The dependent piece of advice that this shadow invokes, along with the name of this piece of advice and an ordered list of variable names that the advice binds.
- The shadow's source-code position. If the program is given in the form of bytecode, CLARA attempts to extract the shadow's line number from debug information in the bytecode.
- The dynamic residue of the shadow, i.e., an abstract representation of the dynamic check that determines, at runtime, whether the advice will actually execute at this program point. Static analyses can replace this residue according to their analysis information. For instance, if an analysis determines that a shadow never needs to execute, then the analysis can set the residue to the constant `NeverMatch`.
- A mapping from the variables that the advice binds at this shadow to a points-to set that models all objects that these variables could possibly point to. Computing this mapping may be expensive. Therefore, CLARA computes the mapping only when it is requested.

In the following, we will describe how the three static analyses use the above information to determine "irrelevant shadows", or "nop shadows", as we often call them. Those are shadows that we can safely refrain from monitoring at runtime, without jeopardizing the correctness of the resulting residual monitor: when disabling a nop shadow s, then the fact that this shadow is a nop shadow guarantees that the resulting reduced event trace, which misses events that would otherwise be triggered by s, causes the runtime monitor to reach its final state at exactly the same events as a complete trace, i.e., with s enabled, would have caused.

3.1 Quick Check

The Quick Check executes, as the name suggests, very quickly, usually within milliseconds. The check only accesses syntactic information that is readily available after the weaving process. For every given dependency state machine $\mathcal{M}$, the Quick Check first determines for every symbol a in the alphabet of $\mathcal{M}$ how many shadows exist in the program that are labeled with a. If for some symbol a there exist no shadows at all, then the Quick Check removes all edges from $\mathcal{M}$ that carry a as label. Next, the Quick Check reduces the dependency state machine by removing all states and edges that have become unreachable, or from which final state cannot be reached any more. Let us call the resulting state machine $\mathcal{M}_q$. Then, in a last step, the Quick Check disables all shadows that trigger events the symbol of which is not in the alphabet of $\mathcal{M}_q$. For further details about the Quick Check, we refer the interested readers to [10,12].

As an example, let us assume that we wish to verify that some program adheres to the ConnectionClosed, and that the Quick Check discovers that there exists no joinpoint shadow for `write` in the entire program: the program disconnects and reconnects connection objects, but it never writes to any connection. In this case, the Quick Check would remove the `write`-edge from the state machine in Fig. 1. This, in turn, means that the resulting state machine contains no path from an initial to a final state. Hence, the reduced alphabet is empty. As a result, the Quick Check can disable all shadows for this program: the Quick Check just proved that the program can never violate the ConnectionClosed property on any execution.

Correctness. The Quick Check is easy to prove correct. Discarding transitions that are labeled with a symbol a that matches nowhere in the program cannot change $matches(t)$ for any trace t because no such trace can contain a. Discarding unreachable states also does not impact $matches(t)$, and neither does disabling symbols that the state machine does not refer to after all such states have been removed. □

3.2 Orphan-shadows analysis

The second analysis stage, the orphan-shadows analysis, performs the same check, but on a per-object basis. This stage uses a flow-insensitive, context-sensitive points-to analysis [31] to disambiguate pointer references. This allows the analysis to decide which joinpoint shadows could potentially refer to the same objects. The analysis then uses this information as follows. In our example, if the program disconnects a particular connection `c`, but never writes to `c`, then for this `c` the dependency is not fulfilled and therefore one does not need to monitor any `disconnect`, `reconnect` or `write` events on this connection. In [10,12], we also give further details about the orphan-shadows analysis.

Correctness. The orphan-shadows analysis distinguishes two objects referred to by variables x and y only if the points-to sets of x and y have an empty intersection. In this case, it is known that x and y cannot refer to the same objects and therefore events that bind the same specification variable to x, respectively y, cannot possibly yield the same ground trace. Otherwise the orphan-shadows analysis behaves just like the Quick Check and is correct for the same reasons. □

3.3 Nop-shadows analysis

The nop-shadows analysis is the analysis that is most involved. It is the only one of the three analyses stages that

is flow-sensitive, i.e., that takes into account the order in which the program under test may trigger events of interest. Like the orphan-shadows analysis, the nop-shadows analysis also uses pointer information to disambiguate pointer references. Unlike the orphan-shadows analysis, however, the nop-shadows analysis does not only use flow-insensitive points-to information for this purpose, but instead uses object representatives [14], a special pointer abstraction that combines points-to information with flow-sensitive must-alias and must-not-alias information. This abstraction yields enhanced precision when the order of events matters. In particular, the must-alias information allows us to determine that a particular event must have occurred before another, on the same object.

We based the nop-shadows analysis entirely on our semantics of dependency state machines. This semantics states that a dependent advice must be dispatched on some variable binding β if not dispatching the advice would alter the set of events (or joinpoints) at which the monitor reaches its final state for a binding that is compatible with β. The nop-shadows analysis exploits this definition by computing an equivalence relation between states of the dependency state machines. This relation allows the analysis to identify "nop shadows" as shadows that only switch between equivalent states. We say that two states q_1 and q_2 are "continuation-equivalent" at a joinpoint shadow s (or "equivalent" for short), and write $q_1 \equiv_s q_2$ if, given all possible continuations of the execution after s, the fact whether the monitor is in state q_1 or in state q_2 at s does it not impact when the dependency state machines reach its final state on these possible continuations. The analysis uses alias information to disambiguate states for different variable bindings.

Given this equivalence relation, we can then identify shadows s that only switch between "equivalent" states on all possible executions that lead through s. Using a forward analysis, we first compute all possible states just before s. Then, if for every such state q the target state after executing s is equivalent to q, by definition of our semantics of dependency state machines we know that dispatching a piece of advice a at such a shadow s would have no effect. We exploit this fact in two different ways. First, we filter the shadow from the list of shadows that are displayed to the user after weaving. This aids the programmer in reasoning about the effects that the aspect may have. Second, we remove all advice-dispatch code from this shadow, potentially speeding up the execution of the woven program.

Consider again the example that is given in Fig. 6. We first focus on the `write` shadow at line 4. Given the only possible execution path that leads up to this line, we know that the dependency state machine must be in state "connected" when reaching the line. We also know that a `write` transition leads from "connected" back to "connected" only, i.e., the transition loops. State "connected" is obviously equivalent to itself: $q_1 = q_2$ implies $q_1 \equiv_s q_2$. Therefore, the nop-shadows analysis can safely disable the advice dispatch at the shadow at line 4. When identifying such a "nop shadow" and disabling the advice dispatch at this shadow, we re-iterate the nop-shadows analysis, this time under the new assumption that no advice will be dispatched at the shadow. During this re-iteration, the analysis will disable the `write` shadow at line 9, and either of the `disconnect` shadows at line 5 or 6, depending on which one is analyzed first, and the `disconnect` shadow at line 8. This last shadow at line 8 is interesting in the sense that it switches between equivalent states that are not equal, i.e., we have $q_1 \equiv_s q_2$ although $q_1 \neq q_2$. At this shadow, the non-deterministic dependency state machine is simultaneously in states "connected" and "error". From these states, the `disconnect` transition moves into state "disconnected". Although this is definitely not the same internal state, the state "disconnected" is equivalent to both other states it given all possible continuations, i.e., given all executions that could follow line 8: after this line, we only see a `write` event, but this event occurs on another object and is therefore not relevant to the `disconnect` transition in line 8.

Computing the appropriate equivalence relation requires both a forward and a backward-analysis component: the forward component computes equivalencies between states "with respect to the past", while the backward analysis computes equivalencies "with respect to the future", i.e., with respect to the possible continuations. The forward-analysis component works by propagating through the program the states of a determinized version of the original dependency state machine $\mathcal{M}$. The backward-analysis component is an exact dual of the forward one: it propagates backward through the program the states of a determinized version of the inverted state machine of $\mathcal{M}$. To obtain an efficient implementation, our analysis uses flow-sensitive information on an intra-procedural, i.e., per-method level only, and uses a coarse grain flow-insensitive abstraction at method boundaries. Space limitations prevent us from explaining the nop-shadows analysis any further. We give further details on this analysis in previous work [9].

Correctness. The nop-shadows analysis is correct by construction. If at a statement s transitions only between states that are provably continuation-equivalent with respect to all possible continuations then

$$matches_{\mathcal{L}}(t_1, \ldots, t_{i-1} t_{i+1}, \ldots, t_n)$$

cannot hold if s triggers the ith event. Therefore, it is sound to define $necessaryShadow(a, t, i) = \textbf{false}$. □

3.4 Adding analyses to CLARA

Researchers in the field of static analysis can easily integrate their own analyses into CLARA by scheduling a new `ReweavingPass`. CLARA executes all such analysis passes right after weaving, and before later-on re-weaving the program based on the analysis information (hence the name). CLARA executes all passes in sequence. For instance, CLARA executes the three default analyses in the order in which we presented them here: simple and quick ones first, more involved ones later. This makes sense because in many cases even simple analyses, like the Quick Check, are already powerful enough to recognize all shadows as irrelevant. When this happens, the remaining analysis stages need to do nothing at all.

Programmers can insert their own analysis at any point in the sequence. Every pass uses a unique pass identifier. Users can refer to these identifiers to enable or disable analysis passes on CLARA's command line. For instance, when given the command-line parameters "`-static-analyses quick-osa`", CLARA will enable the Quick Check and the orphan-shadows analysis, however disable the nop-shadows analysis. It may be the case that analysis stages are inter-dependent. For instance, our current implementation of the nop-shadows analysis assumes that the orphan-shadows analysis has executed previously, because the nop-shadows analysis reuses the points-to information that the orphan-shadows analysis computed. Programmers can explicitly declare such dependencies when instantiating a `ReweavingPass`. CLARA then checks the command-line parameters and reports a helpful error message when the user attempts to run CLARA with an combination of analyses that would violate these dependencies.

4 Collaborative runtime verification

In the last section, we have presented a set of static program analyses and optimizations that evaluate runtime monitors ahead of time. These analyses can often reduce and sometimes completely eliminate the performance penalties that runtime monitors induce. However, even the most sophisticated static-analysis techniques will fail in some cases: for some programs instrumentation remains even after applying all our static-analysis stages. When this instrumentation happens to reside within a hot loop, the performance penalty can be large. This may be acceptable in a pre-deployment setting, where developers can produce a large number of slow test runs on dedicated machines. But even then the runtime overhead that the instrumentation induces may be too large.

The situation is even worse when considering a setting in which instrumentation-carrying programs are deployed. In the verification community it is now widely accepted that, especially for large programs, verification is incomplete, and hence bugs may arise in deployed code on the machines of end users. If deployed code carried instrumentation for runtime verification, developers could track down the causes of observed failures more easily.

In such a setting it is mandatory that the monitoring code only induces a low runtime overhead. According to researchers in industry [23], companies would likely be willing to accept runtime verification in deployed code if the verification overhead was below 5%. Hence in this chapter, we show how to reduce the runtime verification-induced overhead further, using methods from remote sampling [26]. Because companies that produce large pieces of software (which are usually hard to analyze) often have access to a large user base, one can leverage the size of the user base to deploy different partial instrumentation ("probes") for each user. A centralized server can then combine runtime-verification results from runs with different probes. Sampling-based approaches have many different applications. We are most interested in using sampling to reduce instrumentation overhead for individual end users. We have developed an approach to partition this overhead, called it spatial partitioning.

Spatial partitioning works by partitioning the set of instrumentation points into subsets. We call each subset of instrumentation points a it probe. Each user is given a program instrumented with only a few probes. This works very well in many cases.

Spatial partitioning reduces the overhead of runtime verification by only leaving a subset of a program's shadows enabled. However, choosing an arbitrary subset of shadows is more than unsatisfactory; in particular, arbitrarily disabling shadows for weakly referenced symbols may lead to false positives. Consider the example from Fig. 9, in combination with the HasNext property [12]. This property states that it is an error to call `next()` twice in a row on some iterator without calling `hasNext()` in between (to check if the iterator actually has a next element). The example program in Fig. 9 was taken from an earlier version of our own implementation of CLARA. The program uses two different iterators, "`entryIter`" and "`iterator`", to print the contents of a map to a StringBuffer. We have underlined the shadows at which events occur that are of interest to the HasNext pattern. In this example, one safe spatial partitioning would be to disable all shadows in the program except for those referring to `entryIter` (lines 3, 4 and 17). However, many partitionings are unsafe; for instance, disabling the `hasNext` shadow on line 3, but enabling the `next` shadow on line 4 on a map with two or more entries gives a false positive, since the monitor "sees" two calls to `next()` and not the call to `hasNext()` that would prevent the match.

Fig. 9 Example program using two iterators

```
1  private void mapToString(Map<String, List<String>> map, StringBuffer sb) {
2      for (Iterator<Map.Entry<String, List<String>>> entryIter =
3              map.entrySet().iterator(); entryIter.hasNext();) {
4          Map.Entry<String, List<String>> entry = entryIter.next();
5          sb.append(entry.getKey());
6          List<String> args = entry.getValue();
7          if (!args.isEmpty()) {
8              sb.append("(");
9              for (Iterator<String> iterator = args.iterator(); iterator.hasNext();) {
10                 String varName = iterator.next();
11                 sb.append(varName);
12             if(iterator.hasNext())
13                     sb.append(",");
14             }
15             sb.append(")");
16         }
17         if(entryIter.hasNext()) {
18             sb.append(",");
19         }
20         sb.append(" ");
21     }
22     sb.append("\n");
23 }
```

Enabling arbitrary subsets of shadows can also lead to wasted work. Disabling the `next` shadow in the above example and keeping the `hasNext` shadow enabled would, of course, lead to overhead from the `hasNext` shadow. But `hasNext` shadows can never lead to a complete match without any `next` shadows: `hasNext` shadows can only force the runtime monitor to exit a state; `hasNext` shadows are no progress shadows. We therefore need a more principled way of determining sensible groups of shadows to enable or disable.

Spatial partitioning uses information from dependency state machines to compute a set of "probes". Each probe is a set of shadows that is both "consistent" and "complete". The probe is consistent because we require that all shadows that are part of the probe must be compatible with each other (with respect to the dependency state machine that defines the probe). In other words, there must be no two shadows in the probe that assign the same variable v to two disjoint points-to sets. (If this occurred then we knew that the shadows would assign v to different objects). The probe also has to be complete: no shadow that is compatible may be left out.

The concrete algorithms to compute consistent and complete probes go beyond the scope of this paper. We refer the interested readers to previous work [8,11].

It is interesting to note that by the way in which we construct probes, spatial partitioning can guarantee complete coverage in the following sense. Assume a program run r that leads to a property violation in the fully instrumented program. Further assume that we obtain k different probes through spatial partitioning, and that we distribute differently instrumented versions of the program to a number of users, such that every single one of the k probes is enabled in at least one user's program version. When all these users now re-execute the same program run r then there will be at least one user for which r causes the runtime monitor to notify this user about the property violation.

Further, we wish to note that spatial partitioning in CLARA is compatible with any of the other static analyses: at any point in time, users of CLARA can opt to have all remaining shadows partitioned, no matter which analysis stages executed earlier.

5 Experimental results

In this section, we will both empirically motivate the need for the static analyses and optimizations that we presented, and we show that the static analyses that CLARA provide can indeed benefit runtime monitoring. Throughout this paper, we consider two possibilities of verifying that a program satisfies a finite-state property: statically and through runtime verification. When a programmer weaves a runtime monitor into her program under test, the AspectJ compiler emits a list of source code locations at which runtime events of interest could occur—the joinpoint shadows. After the compilation has finished, the programmer could in principle inspect all joinpoint shadows manually to see whether the shadows may indeed contribute to a property violation. However, this approach is only viable if just a handful of shadows exist. The second approach is to not consider the joinpoint shadows at all, but to instead run the woven program and see if it reports a property violation at runtime. To give meaningful results, this approach requires good test coverage and,

Table 1 Monitored specifications for classes of the Java Runtime Library (available at http://www.bodden.de/clara/benchmarks/)

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

equally important, requires the monitoring instrumentation to induce a low enough runtime overhead so that test runs can be completed in a reasonable amount of time. In our experiments, we applied a number of runtime monitors to a number of programs and then sought to determine whether either approach, static inspection or runtime verification would be possible for any combination of aspect and program.

For our experiments, we therefore wrote a set of 12 tracematch [1] specifications for different properties regarding collections and streams in the Java Runtime Library. Table 1 gives brief descriptions for each of these properties. The author's dissertation [8, Sect. 2] explains all of these properties in great detail. We selected properties of the Java Runtime Library because this allowed us to find a large set of programs to which these properties are of interest. Although this selection of properties induces a bias to our results, we believe that this bias is minor: we have no reason to believe that the properties of other application interfaces would be in any way more complex than those of the Java Runtime Library. This is especially true in light of a study by Dwyer et al. [19], which showed that most specification patterns mentioned in the scientific literature are indeed quite simple.

Although one can apply CLARA to any AspectJ-based runtime monitor, we decided to restrict our experiments to monitors generated from tracematch specifications. In particular, we will not consider JavaMOP-based specifications. In previous work [10], we showed that CLARA always produces equivalent analysis results for equivalent runtime monitors, independent of the specification formalism that the programmer used to define these monitors. Hence, there is little benefit from re-doing all the experiments again with equivalent JavaMOP-based specifications. Note that the monitors that abc generates from tracematches are already heavily optimized [3] to induce a minimal runtime overhead. Therefore, our baseline is by no means a naïve baseline. All our tracematch definitions are available for download at http://www.bodden.de/clara/benchmarks/.

Note that our tracematches contain no recovery code or even notification code of any kind: the bodies of the tracematches are empty. This is for two reasons. First, the content of the bodies would only impact our static-analysis results if the body called back into the program under test. This is unlikely for a monitoring aspect. Hence, we can just as well assume that the body is empty. The content of the body does, however, have an impact on the runtime overhead of the runtime monitor. We are interested in measuring the time that the monitor has to consume to update its internal state based on the events that it monitor. By using empty tracematch bodies we can make sure that we measure only this overhead, and not any additional overhead that error-reporting or recovery code would cause.

For our benchmarks, we used version 2006-10-MR2 of the DaCapo benchmark suite [6]. The suite contains 11 different workloads that exercise 10 different programs. In Table 2, we give brief descriptions of the benchmarks (taken from [6]) and also state the number of methods that they contain. Note that, on average, a DaCapo benchmark has about four times as many methods and about four times as much code as one of the well-known SPEC benchmarks [32,33]. The benchmarks hsqldb, lusearch and xalan are multi-threaded. The benchmarks luindex and lusearch are two different workloads that produce two different program runs of the same program lucene. All benchmarks use dynamic class loading, and the benchmark jython even generates classes dynamically which it then executes.

Table 2 The DaCapo benchmarks, version 2006-10-MR2 (taken from http://dacapobench.org/)

Benchmark	Classes	Methods	Description
antlr	224	2,972	Parses one or more grammar files and generates a parser and lexical analyzer for each
bloat	263	3,986	Performs a number of optimizations and analysis on Java bytecode files
chart	512	7,187	Uses JFreeChart to plot a number of complex line graphs and renders them as PDF
eclipse	344	3,978	Executes some of the (non-gui) JDT performance tests for the Eclipse IDE
fop	967	6,889	Takes an XSL-FO file, parses it and formats it, generating a PDF file
hsqldb	385	5,859	Executes a JDBCbench-like in-memory benchmark, executing a number of transactions against a model of a banking application
jython	508	7,240	Interprets a the pybench Python benchmark
luindex	311	3,013	Uses lucene to index a set of documents; the works of Shakespeare and the King James Bible
lusearch	311	3,013	Uses lucene to do a text search of keywords over a corpus of data comprising the works of Shakespeare and the King James Bible
pmd	530	4,785	Analyzes a set of Java classes for a range of source code problems
xalan	562	6,463	Transforms XML documents into HTML

5.1 Monitoring without static analysis

We used CLARA to weave any of the 12 monitoring aspects separately into each one of the 10 DaCapo programs. By default, CLARA weaves only into the application itself, not into the Java Runtime Library. Hence, the runtime monitors that we use do, for example, monitor events where the benchmark program uses collections and streams of the Java Runtime Library, but it does not monitor events where these objects are used inside the Java Runtime Library. As a baseline, we also compiled every benchmark program with the CLARA, but with no aspects present. This results in an un-instrumented program that has a bytecode layout similar to the instrumented programs that CLARA produces when aspects are present.

5.1.1 Number of shadows after weaving

Table 3 shows for every tracematch/benchmark combination the number of shadows that the woven program for this combination contains. Note that the benchmarks luindex and lusearch share the same code base. Therefore, these benchmarks produce the same number of shadows in all cases. As the table shows, the compilation process produced some shadows in all, but 11 out of the 120 combinations. Moreover, the number of shadows is usually quite large. A total of 100 cases result in more than 10 shadows, and 88 cases result in even more than 50 shadows; on average the compilation results in 326 shadows spread over 82 methods. Therefore, in all but a few lucky cases it would be impractical for a programmer to investigate all these program points manually to see if these points contribute to a property violation.

Table 3 Number of shadows right after weaving

	antlr	bloat	chart	eclipse	fop
ASyncContainsAll	0	71	6	10	0
ASyncIterC	0	1,621	498	214	146
ASyncIterM	0	1,684	507	236	176
FailSafeEnumHT	133	102	44	217	205
FailSafeEnum	76	3	1	117	18
FailSafeIter	23	1,394	510	391	288
FailSafeIterMap	130	1,180	374	548	1,374
HasNextElem	117	2	0	89	10
HasNext	0	849	248	109	72
LeakingSync	170	1,994	920	1,325	2,347
Reader	50	7	65	218	102
Writer	171	563	70	1,045	429

	luindex				
	hsqldb	jython	lusearch	pmd	xalan
ASyncContainsAll	0	31	18	10	0
ASyncIterC	33	128	149	671	0
ASyncIterM	39	138	152	718	0
FailSafeEnumHT	114	153	37	100	319
FailSafeEnum	120	110	61	21	222
FailSafeIter	112	253	217	546	158
FailSafeIterMap	252	250	136	583	540
HasNextElem	53	64	22	6	63
HasNext	16	63	74	346	0
LeakingSync	528	1,082	629	986	1,005
Reader	1,216	139	226	102	106
Writer	1,378	462	146	62	751

It may, however, be possible to monitor these programs for violations at runtime.

5.1.2 Runtime overhead through monitoring code

To determine the runtime overhead that the monitoring aspects cause, we first executed the un-instrumented programs to establish a baseline. We used the standard workload size of the DaCapo harness. The DaCapo benchmark suite comes with a `-converge` option, that tries to make the determined runtime values better comparable. When the option is enabled, then DaCapo runs the benchmark in question multiple times until the relative standard deviation of the determined runtimes drops below 3%. DaCapo then assumes that the benchmark has reached a "stable state", e.g., that the virtual machine has loaded and just-in-time compiled all of the benchmark's methods. DaCapo then runs the benchmark one more time and reports the runtime of this last run. In the past, we have experienced problems with this approach: if, coincidentally, the last run is extraordinarily better or worse than the previous runs then the runtime that DaCapo reports will deviate from the "normal" runtime. We hence modified the harness so that it would instead proceed as follows.

DaCapo first runs the benchmark once without collecting any timing information—a warm-up run. Then DaCapo re-runs the benchmark multiple times, again until the relative standard deviation of the determined runtimes drops below 3% (but at least 5 times and at most 20 times). Then we report the mean of these runs. This gives us the advantage that the number that we report originates from a sample of runs from which we know that this sample deviated no more than 3%.

We executed the benchmarks using the HotSpot Client VM (build 1.4.2_12-b03, mixed mode), with its standard heap size on a machine with an AMD Athlon 64 X2 Dual Core Processor 3800+ running Ubuntu 7.10 with kernel version 2.6.22-14 and 4 GB RAM. Then we executed, in the very same way, the programs that have had the monitoring aspects woven into them.

Table 4 shows for every benchmark the baseline execution time (with no aspect present, in milliseconds), and for every monitoring aspect the relative runtime overhead that this aspect causes, in percent, both before and after having applied our static optimizations. For instance, antlr showed a 378.62-times overhead when instrumented with the Writer tracematch and with optimizations disabled, but only 36% overhead when optimizations were enabled. We do not show values that are within the 3% error margin. The value ">1h" means that a single benchmark run took longer than 1 h. In this case, we aborted the run after this first hour. The benchmarks eclipse, hsqldb and xalan showed no overheads.

As the results show, runtime verification is indeed a viable solution for many of the benchmark/property configurations that we consider. The vast majority of test runs do not expose any perceivable overhead. This is true even for benchmark/property combinations that have a high number of shadows. For instance, chart-ASyncIterM has 507 shadows, but shows no perceivable runtime overhead. This is because the specific test run does not exercise the instrumented collections and iterators a lot. Nevertheless, there are some cases for which the overhead is high. In about one quarter of our test runs the overhead was at least 10%, and in five cases the overhead was so high that even a single execution of the benchmark took longer than 1 h. Such overheads clearly prevent programmers from using runtime verification in these particular cases. In the following sections we will show that our static analyses can significantly lower and sometimes completely eliminate the runtime overhead in the majority of cases.

5.2 Monitoring with all three static analyses enabled

We next re-compiled all 120 benchmark/property combinations, but this time with all three analyses stages, i.e., Quick Check, orphan-shadows analysis and nop-shadows analysis enabled. Note that, because the nop-shadows analysis is flow-sensitive, it is not sound to apply the analysis to multi-threaded programs. When a program is multi-threaded, then this means that the program's threads could execute shadows in an order that the nop-shadows analysis did not anticipate. After all, the nop-shadows analysis assumes that its intra-procedural control-flow graphs soundly model all possible control flow. In future work, we plan to make our analysis thread safe by using a may-happen-in-parallel analysis [4] to determine which methods could potentially execute in parallel. The benchmarks hsqldb, lusearch and xalan are multi-threaded. For now, we just analyzed these three benchmarks like all other benchmarks, i.e., we made the un-safe assumption that these programs do not execute dependent-advice shadows in parallel.

DaCapo's benchmarks load classes using reflection. Static analyses like ours have to be aware of these classes so that they can construct a sound call graph. We wrote an AspectJ aspect that would print at every call to `forName` and a few other reflective calls the name of the class that this call loads and the location from which it is loaded. We further double-checked with Ondřej Lhoták, who compiled such lists of dynamic classes earlier. We then provided the abc-internal call-graph analysis with this information. The resulting call graph is sound for the program runs that DaCapo performs. A limitation of our approach is that obtaining a call graph that is sound for all runs may be challenging for programs that use reflection.

For eclipse we were unable to determine where dynamic classes are loaded from. Eclipse loads classes not from JAR files, but from "resource URLs", which eclipse resolves internally, usually to JAR files within other JAR files. CLARA currently cannot load classes from such URLs and that is why we omit eclipse in our experiments. The jython benchmark generates code at runtime, which it then loads. We did not

Table 4 Runtime overheads before and after static analyses

	antlr		bloat		chart		fop	
	Before	After	Before	After	Before	After	Before	After
Baseline	4,079		9,276		14,666		2,562	
ASyncContainsAll								
ASyncIterC			**140**				5	
ASyncIterM			**139**					
FailSafeEnumHT	**10**	4						
FailSafeEnum								
FailSafeIter			**>1 h**	**>1 h**	8	8	**14**	
FailSafeIterMap			**>1 h**	**22027**			7	OOME
HasNextElem								
HasNext			**329**	**258**				
LeakingSync	9		**163**		**91**		**209**	
Reader	**30,218**							
Writer	**37,862**	**36**	**>1 h**	**228**			5	
	jython		luindex		lusearch		pmd	
	Before	After	Before	After	Before	After	Before	After
baseline	11,105		17,144		13,940		13,052	
ASyncContainsAll								
ASyncIterC							**28**	
ASyncIterM							**35**	
FailSafeEnumHT	**>1h**	**>1h**	**32**					
FailSafeEnum			**30**		**18**			
FailSafeIter			5		**20**		**2,811**	**524**
FailSafeIterMap	**13**	**13**	5				**>1h**	**>1h**
HasNextElem			**12**					
HasNext							**70**	**64**
LeakingSync	**>1h**		**34**		**365**		**16**	
Reader					**77**			
Writer								

Baseline in milliseconds, rest in percent, values within the 3% error margin omitted, values of at least 10% in boldface
eclipse, hsqldb and xalan show no overheads
OOME OutOfMemoryException during static analysis

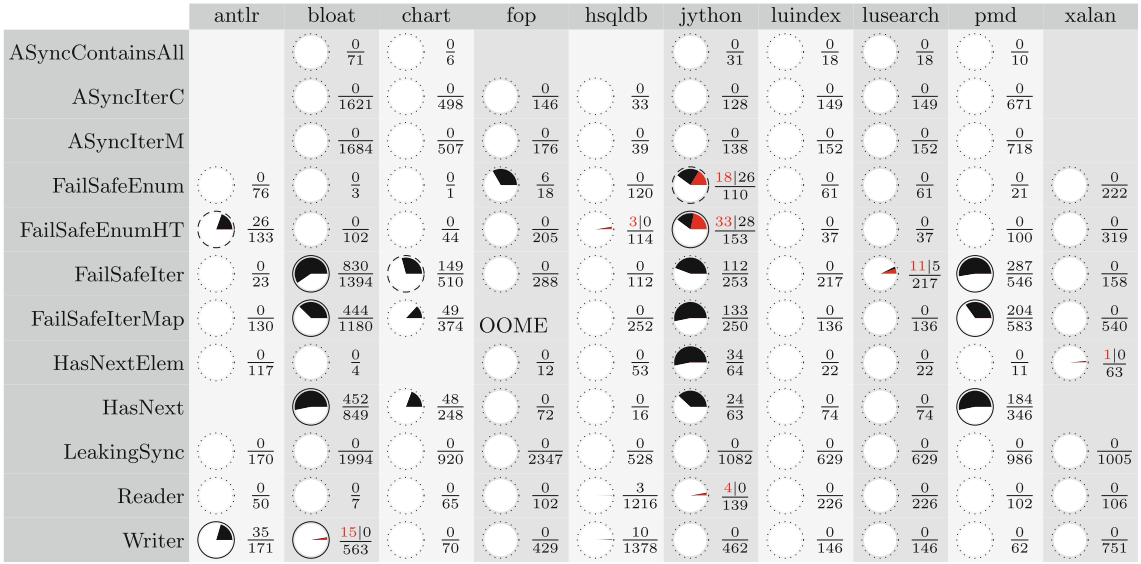
analyze this code and so made the unsound assumption that this code would not invoke any dependent advice.

5.2.1 Shadows remaining after all three analyses stages

Table 5 summarizes our analyses result. The table reports, as white slices, the fraction of shadows that the analysis identified as irrelevant. In gray we show the fraction of shadows which are known to trigger actual violations at runtime. No sound static analyses could disable these shadows: because the shadows trigger a property violation at runtime they need to remain enabled. The remaining black slice 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. Black slices due to imprecision almost only remain for 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 current version 9.12. jython and pmd both make heavy use of dynamic class loading and reflection. This confuses our pointer analysis, which has to make very conservative approximations in such situations. As a result, our pointer analysis believes that certain iterators and enumerations in these benchmark could be aliased although, in fact, this cannot be the case. We are

Table 5 Shadows identified as irrelevant, and therefore disabled



White slices represent shadows that our three analyses stages managed to identify as irrelevant. Black slices represent shadows that we fail to identify as irrelevant, due to analysis imprecision or because the shadows actually are relevant to triggering a property violation at runtime. Red (or gray) slices represent shadows that we confirmed to be relevant, through manual inspection. The outer rings represent the aspect's runtime overhead after optimizing the advice dispatch. Solid: overhead $\geq 15\%$, dashed: overhead $< 15\%$, dotted: no overhead
OOME OutOfMemoryException during static analysis

currently trying to extend CLARA so that it can handle reflection with more fine-grained approximations.

5.2.2 Runtime overhead after all analyses

Table 5 shows the runtime overhead that remains after applying our optimizations, but only qualitatively, in the form of rings. Solid rings mean an overhead of more than 15%, dashed rings mean an overhead of under 15% and a dotted ring means no observable overhead. Note that in cases where the analyses manage to disable all shadows, the overhead will naturally be zero because in these cases CLARA effectively emits an un-instrumented program. Table 4 shows the overhead numbers in its "after" columns.

The author's dissertation [8] contains more information about the experiments. In particular, it presents all raw data. The dissertation also discusses the relative impacts of the three analyses stage, and presents results on the effectiveness of our approach for collaborative runtime verification.

6 Related work

CLARA's static analyses belong to the family of typestate analyses. Strom and Yemini [34] were the first to suggest the concept of typestate analysis. In the last few years, researchers have presented several new approaches with varying cost/precision trade-offs. In the following, we describe the approaches that are most relevant to our work. We distinguish type-system based approaches, static verification approaches and hybrid verification approaches.

Type-system based approaches. Type-system based approaches define a type system and implement a type checker. This is to prevent programmers from compiling a potentially property-violating program in the first place and gives the advantage of strong static guarantees. On the other hand, the type checker may reject useful programs that statically appear to violate the stated property, but will not actually violate the property at runtime. Our approach allows the programmer to define a program that may violate the given safety property. Our analysis then tries to verify that the program is correct, and when this verification fails it delays further checks until runtime.

Bierhoff and Aldrich [5] present an intra-procedural type-system based approach that enables the checking of typestate properties in the presence of aliasing. The author's approach aims at being modular, and therefore abstains from potentially expensive whole-program analyses like ours. To be able to reason about aliases nevertheless, Bierhoff and Aldrich associate special access permissions with references. Access permissions allow the type checker to reason about a reference locally. The author's current approach assumes that a program contains information about access permissions and also typestate changes in the form of special program annotations. Our approach does not require any program annotations; it is fully automatic.

DeLine and Fähndrich's approach [18] is similar in flavor to Bierhoff and Aldrich's, but uses a more restrictive abstraction of aliases that allows for less flexible calling conventions for typestate-changing methods. The authors implemented their approach in the Fugue tool for specifying and checking typestates in .NET-based programs. 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 this method references.

Static analysis approaches Unlike type systems, static analysis approaches perform a whole-program analysis and, unlike hybrid approaches, they have no runtime component.

Fink et al. [21] present a static analysis of typestate properties. Their approach, like ours, uses a staged analysis which starts with a flow-insensitive pointer-based analysis, followed by flow-sensitive checkers. The authors' analyses allow only for specifications that reason about a single object at a time, while we allow for the analysis of multiple interacting objects. Fink et al.'s algorithms only determine "final shadows" that complete a property violation (like "write" in our example), but not shadows that initially contribute to a property violation (e.g. "close") or can prevent a property violation (e.g. "reconnect"). Therefore, these algorithms are unsuitable for generating residual runtime monitors.

Hybrid analysis approaches Naeem and Lhoták present a fully context-sensitive, flow-sensitive, inter-procedural whole-program analysis for typestate-like properties of multiple interacting objects [29]. Naeem and Lhoták's analysis is fully inter-procedural. This can yield enhanced precision in cases where combinations of objects that are relevant to a given specification are used by multiple methods. Our benchmark set showed some instances where this additional information would have been helpful, but not many. It even holds that, although our analysis is mostly intra-procedural, there are some instances where our combination of analyses is more precise than Naeem and Lhoták's. This is due to the highly context-sensitive points-to sets that we compute. Unfortunately, as we showed in earlier work [9], their analysis therefore suffers from an unsoundness problem. All analyses that CLARA provides were proven sound [8].

Dwyer and Purandare use existing typestate analyses to specialize runtime monitors [20]. 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. A special case of a safe region would be a region that does not change the automaton's state at all—an "identity region". For regions that are safe, but no identity regions, the authors summarize the effect of this region and change the program under test to update the typestate with the region's effects all at once when the region is entered. This has the advantage that the analyzed program will execute faster because it will execute fewer transitions at runtime. However, unlike our approach, the author's analysis does not aid programmers who wish to inspect their code manually. The fact that the author's transformation changes the points at which transitions occur makes it even harder for programmers to manually inspect these program points. Dwyer and Purandare's approach is, although hybrid, not based on shadow histories and hence we have no reason to believe that it is unsound. The approach cannot generally handle groups of multiple interacting object, but ours can.

7 Conclusion

We have presented CLARA, a framework for evaluating finite-state runtime monitors ahead of time, through static analysis. CLARA is compatible with any runtime monitor that is expressed as an AspectJ aspect. To make such an aspect analyzable by CLARA, one just needs to assure that the aspect is annotated with a dependency state machine, a textual finite-state machine representation of the property at hand. We have presented the syntax and semantics of dependency state machines. We further presented CLARA's extensible static-analysis engine, along with three example analyses that we provide with CLARA. Through experiments with the DaCapo benchmark suite, we have shown that the static-analysis approach that CLARA provides can greatly reduce the amount of instrumentation necessary for runtime monitoring in most Java programs. Our experiments further revealed that this reduced amount of instrumentation yields a largely reduced runtime overhead in many cases.

CLARA is available as open source. We hope that other researchers will soon be joining us in using CLARA, and that this will foster progress in the field of typestate analysis.

References

1. Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Adding Trace Matching with Free Variables to AspectJ. In: OOPSLA, pp. 345–364. ACM Press (October 2005)
2. Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: abc: an extensible AspectJ compiler. In: AOSD, pp. 87–98. ACM Press (March 2005)
3. Avgustinov, P., Tibble, J., de Moor, O.: Making trace monitors feasible. In: OOPSLA, pp. 589–608. ACM Press (October 2007)
4. Barik, R.: Efficient computation of may-happen-in-parallel information for concurrent Java programs. In: International Workshop on Languages and Compilers for Parallel Computing (LCPC), volume 4339 of LNCS, pp. 152–169. Springer (October 2005)

5. Bierhoff, K., Aldrich, J.: Modular typestate checking of aliased objects. In: OOPSLA, pp. 301–320 (October 2007)
6. 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., Wiedermann, B.: The DaCapo benchmarks: Java benchmarking development and analysis. In: OOPSLA, pp. 169–190. ACM Press (October 2006)
7. Bodden, E.: J-LO—a tool for runtime-checking temporal assertions. Master's thesis, RWTH Aachen University (November 2005)
8. Bodden, E.: Verifying finite-state properties of large-scale programs. PhD thesis, McGill University (June 2009). Available through ProQuest
9. Bodden, E.: Efficient hybrid typestate analysis by determining continuation-equivalent states. In: ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, pp. 5–14. ACM, New York, NY, USA (2010)
10. Bodden, E., Chen, F., Roşu, G.: Dependent advice: a general approach to optimizing history-based aspects. In: AOSD, pp. 3–14. ACM Press (March 2009)
11. Bodden, E., Hendren, L., Lam, P., Lhoták, O., Naeem, N.A.: Collaborative runtime verification with tracematches. J. Log. Comput. (November 2008). doi:10.1093/logcom/exn077
12. Bodden, E., Hendren, L.J., Lhoták, O.:. A staged static program analysis to improve the performance of runtime monitoring. In: European Conference on Object-Oriented Programming (ECOOP), volume 4609 of LNCS, pp. 525–549. Springer (2007)
13. Bodden, E., Lam, P., Hendren, L.: Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In: Symposium on the Foundations of Software Engineering (FSE), pp. 36–47. ACM Press (November 2008)
14. Bodden, E., Lam, P., Hendren, L.: Object representatives: a uniform abstraction for pointer information. In: Visions of Computer Science—BCS International Academic Conference. British Computing Society (September 2008)
15. Bodden, E., Lam, P., Hendren, L.: Clara: a framework for statically evaluating finite-state runtime monitors. In: 1st International Conference on Runtime Verification (RV), volume 6418 of LNCS, pp. 74–88. Springer (November 2010)
16. Chen, F., Roşu, G.: MOP: an efficient and generic runtime verification framework. In: OOPSLA, pp. 569–588. ACM Press (October 2007)
17. Clocksin, W.F., Mellish, C.: Programming in Prolog, 5th edn. Springer, New York (2003)
18. DeLine, R., Fähndrich, M.: Typestates for objects. In: European Conference on Object-Oriented Programming (ECOOP), Volume 3086 of *LNCS*, pp. 465–490. Springer (June 2004)
19. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: International Conference on Software Engineering (ICSE), pp. 411–420. ACM Press (May 1999)
20. Dwyer, M.B., Purandare, R.: Residual dynamic typestate analysis: Exploiting static analysis results to reformulate and reduce the cost of dynamic analysis. In: International Conference on Automated Software Engineering (ASE), pp. 124–133. ACM Press (May 2007)
21. Fink, S., Yahav, E., Dor, N., Ramalingam, G., Geay, E.: Effective typestate verification in the presence of aliasing. In: International Symposium on Software Testing and Analysis (ISSTA), pp. 133–144. ACM Press (July 2006)
22. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java(TM) Language Specification. 3rd edn. Addison-Wesley Professional, Reading (2005)
23. Grieskamp, W.: (Microsoft Research): Personal communication (January 2007)
24. Hilsdale, E., Hugunin, J.: Advice weaving in AspectJ. In: AOSD, pp. 26–35. ACM Press (March 2004)
25. Krüger, I.H., Lee, G., Meisinger, M.: Automating software architecture exploration with M2Aspects. In: Workshop on Scenarios and State Machines: Models, Algorithms, and Tools (SCESM), pp. 51–58. ACM Press (May 2006)
26. Liblit, B., Aiken, A., Zheng, A.X., Jordan, M.I.: Bug isolation via remote program sampling. In: Conference on Programming Language Design and Implementation (PLDI), pp. 141–154. ACM Press (June 2003)
27. Maoz, S., Harel, D.: From multi-modal scenarios to code: compiling LSCs into AspectJ. In: Symposium on the Foundations of Software Engineering (FSE), pp. 219–230. ACM Press (November 2006)
28. Masuhara, H., Kiczales, G., Dutchyn, C.: A compilation and optimization model for aspect-oriented programs. In: International Conference on Compiler Construction (CC), Volume 2622 of LNCS, pp. 46–60. Springer (April 2003)
29. Naeem, N.A., Lhoták, O.: Typestate-like analysis of multiple interacting objects. In: OOPSLA, pp. 347–366. ACM Press (October 2008)
30. Pnueli, A.: The temporal logic of programs. In: IEEE Symposium on the Foundations of Computer Science (FOCS), pp. 46–57. IEEE Computer Society, (October 1977)
31. Sridharan, M., Bodík, R.: Refinement-based context-sensitive points-to analysis for Java. In: Conference on Programming Language Design and Implementation (PLDI), pp. 387–400. ACM Press (June 2006)
32. Standard Performance Evaluation Coorperation. SPECjvm98 Documentation (March 1999). Release 1.03 edition
33. Standard Performance Evaluation Coorperation. SPECjbb2000 (Java Business Benchmark) Documentation (2001). Release 1.01 edition
34. Strom, R.E., Yemini, S.: Typestate: a programming language concept for enhancing software reliability. IEEE Trans Softw Eng (TSE) **12**(1), 157–171 (1986)