Clara: a Framework for Partially Evaluating Finite-state Runtime Monitors Ahead of Time*

Eric Bodden[1], Patrick Lam[2], and Laurie Hendren[3]

[1] Technische Universität Darmstadt
[2] University of Waterloo
[3] McGill University

Abstract. Researchers have developed a number of runtime verification tools that generate runtime monitors in the form of AspectJ aspects. In this work, we present Clara, a novel framework to statically optimize such monitoring aspects with respect to a given program under test. Clara uses a sequence of increasingly precise static analyses to automatically convert a monitoring aspect into a residual runtime monitor. The residual monitor only watches events triggered by program locations that the analyses failed to prove safe at compile time. 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 residual runtime monitor is usually much more efficient than a full monitor, yet still captures all property violations at runtime.

1 Introduction

Finite-state properties, also known as typestate [1] properties, constrain the set of acceptable operations on a single object or a group of objects, depending on the object's or group's history. Many formalisms allow programmers to easily express typestate properties, including linear temporal logic, regular expressions, message sequence charts and live sequence charts [2, Chapter 2]. Potential applications of runtime monitoring include the evaluation of arbitrary queries over the runtime program state and the enforcement of stated properties. For instance, a monitor could detect attempts to circumvent an access-control policy and then either log the attempt or stop the detected unauthorized access. Researchers have proposed and implemented runtime monitoring tools [3–7] which compile high-level temporal specifications into monitor implementations.

While runtime monitoring could be useful for finding violations in practice, it is subject to the same problems as software testing. Runtime monitoring gives no static guarantees: a particular program run can only prove the presence of property violations, not their absence. Hence, developers and testers must exercise judgment in deciding when to stop monitoring program runs, since exhaustive testing is generally infeasible. Furthermore, although significant advances have

* This work was supported by NSERC and CASED (www.cased.de).

been made [8–10], runtime monitors can still slow down monitored programs significantly, sometimes by several orders of magnitude.

In this paper we therefore propose CLARA, a framework for partially evaluating runtime monitors at compile time. Partial ahead-of-time evaluation addresses all of the problems mentioned above. CLARA specializes a given runtime monitor to a program under test. The result is a residual runtime monitor that only monitors events triggered by program locations that the analyses failed to prove safe at compile time. In our experiments, CLARA's analyses can prove that the program is free of program locations that could drive the monitor into an error state in 68% of all cases. In these cases, CLARA gives the strong static guarantee that the program can never violate the stated property, eliminating the need for runtime monitoring of that program. In many other cases, the residual runtime monitor will require much less instrumentation than the original monitor, therefore yielding a greatly reduced runtime overhead. In 65% of all cases that showed overhead originally, no overhead remains after applying the analyses.

CLARA's principal design goal is to provide a maximally general framework for statically analyzing runtime monitors. We anticipate that CLARA will appeal to researchers in runtime verification, as it supports a large variety of runtime monitoring tools. Researchers in static analysis, on the other hand, can easily extend CLARA with novel static analyses to understand and optimize runtime monitors even further. How do we achieve this generality? CLARA's design is based on the crucial observation that most current runtime-verification tools for Java share two common properties: (1) internally, they use a finite-state-machine model of the property, and (2) they generate runtime monitors in the form of AspectJ aspects [11]. Figure 1 shows a state-machine model for the "ConnectionClosed" property: a disconnected connection should not be written to, unless the connection is potentially reconnected at some later point. Figure 2 shows a monitoring aspect for this property. The remainder of the paper explains this aspect and its analysis in more detail. CLARA takes such monitoring aspects as input and weaves the aspects into the program under test. While weaving, CLARA conducts static analyses, suppressing calls to the monitoring aspect when it can statically prove that these calls are unnecessary.

To perform its static analysis, CLARA must understand the monitoring aspect's internal transition structure. Because every aspect-generating monitoring tool uses a different code-generation strategy, and we wish to be independent of that strategy, CLARA expects the monitoring aspect to carry an annotation

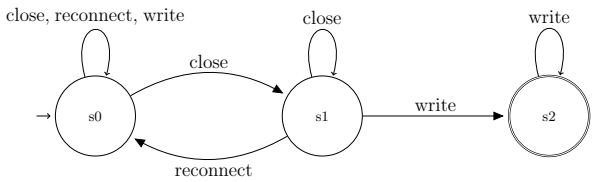

Fig. 1: "ConnectionClosed" typestate property: no write after close.

```
1  aspect ConnectionClosed {
2      Set closed = new WeakIdentityHashSet();
3
4      dependent after close(Connection c) returning:
5          call(* Connection.disconnect()) && target(c) { closed.add(c); }
6
7      dependent after reconn(Connection c) returning:
8          call(* Connection.reconnect()) && target(c) { closed.remove(c); }
9
10     dependent after write(Connection c) returning:
11         call(* Connection.write(..)) && target(c) {
12         if(closed.contains(c))
13             error("May not write to "+c+": it is closed!"); }
14
15     dependency {
16         close, write, reconn;
17         initial s0: close -> s0, write -> s0,  reconn -> s0, close -> s1;
18                 s1: reconn -> s0, close -> s1, write -> s2;
19         final   s2: write -> s2;
20     } }
```

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

encoding the monitor's transition structure explicitly—a Dependency State Machine. Figure 2 shows this annotation in lines 15–20. Most runtime verification tools can easily generate such a state-machine annotation because they internally use a state-machine model of the monitored property. For our experiments, we extended the implementation of tracematches [3] to generate the annotations automatically; we are currently talking to the developers of JavaMOP about extending their tool to generate annotations, too.

In this paper we present the following original contributions:

- We present Clara, an extensible open framework to evaluate AspectJ-based finite-state runtime monitors ahead of time.
- We explain the syntax and semantics of Dependency State Machines, Clara's mechanism to interface with existing runtime-monitoring tools.

Further, we summarize Clara's three predefined static analyses and show through a large set of experiments that, in many cases, these analyses can evaluate runtime monitors ahead of time, either largely reducing runtime overhead or entirely obviating the need for monitoring at runtime.

2 The Clara framework

Clara targets two audiences: researchers in (1) runtime verification and (2) static typestate analysis. Clara defines clear interfaces to allow the two communities to productively interact. Developers of runtime verification tools simply

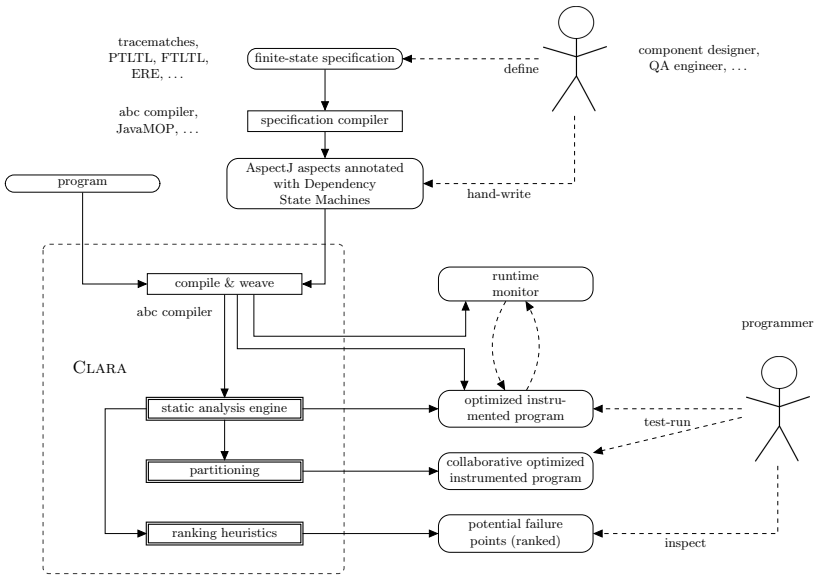

Fig. 3: Overview of CLARA

generate AspectJ aspects annotated with semantic meaning, in the form of Dependency State Machines. Static analysis designers can then create techniques to reason about the annotated aspects, independent of implementation strategy.

Figure 3 gives an overview of CLARA. A software engineer first defines (top right of figure) finite-state properties of interest, in some finite-state formalism for runtime monitoring, such as Extended Regular Expressions or Linear-Temporal Logic, e.g. using JavaMOP or tracematches. The engineer then uses some specification compiler such as JavaMOP or the AspectBench Compiler [12] (abc) to automatically translate these finite-state-property definitions into AspectJ monitoring aspects. These aspects may already be annotated with appropriate Dependency State Machines: we extended abc to generate annotations automatically when transforming tracematches into AspectJ aspects. Other tools, such as JavaMOP, should also be easy to extend to generate these annotations. If the specification compiler does not yet support Dependency State Machines, the programmer can easily annotate the generated aspects by hand.

CLARA then takes the resulting annotated monitoring aspects and a program as input. 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. We will further explain CLARA's predefined analyses in Section 4. The result is an optimized instrumented program that updates the runtime monitor at fewer locations. Sometimes, CLARA optimizes away all updates, which proves that the program cannot violate the monitored property.

CLARA also supports Collaborative Runtime Verification, which distributes instrumentation overhead among multiple users; and ranking heuristics, which aid programmers in inspecting remaining instrumentation manually [**?**] [2, Ch. 6 & 7]. Space limitations preclude us from discussing ranking and Collaborative Runtime Verification here.

CLARA is freely available as free software at `http://bodden.de/clara/`, along with extensive documentation, the first author's dissertation [2], which describes CLARA in detail, and benchmarks and benchmark results.

We next describe the syntax and semantics of Dependency State Machines, the key abstraction of CLARA. This abstraction allows CLARA to decouple runtime monitor implementations from static analyses.

3 Syntax and Semantics of Dependency State Machines

Dependency State Machines extend the AspectJ language to include semantic information about relationships between different pieces of advice. Runtime verification tools which generate AspectJ aspects can use this extension to produce augmented aspects. CLARA can reason about the augmented aspects to prove that programs never violate monitored properties or to generate optimized code.

3.1 Syntax

Our extensions modify the AspectJ grammar in two ways: they add syntax for defining Dependent Advice [16] and Dependency State Machines. The idea of Dependent Advice is that pieces of monitoring advice are often inter-dependent in the sense that the execution of one piece of advice only has an effect when executing before or after another piece of advice, on the same objects. Dependency State Machines allow programmers to make these dependencies explicit so that static analyses can exploit them. Our explanations below refer to the ConnectionClosed example in Figure 2.

The `dependent` modifier flags advice to CLARA for potential optimization; such advice may be omitted from program locations at which it provably has no effect on the state of the runtime monitor. Dependent advice must be named. Lines 4, 7 and 10 all define dependent advice.

The Dependency State Machines extension enables users to specify state machines which relate different pieces of dependent advice. Dependency State Machine declarations define state machines by including a list of edges between states and an alphabet; each edge is labelled with a member of the alphabet. CLARA infers the set of states from the declared edges. Line 16 declares the state machine's alphabet: {`disconn`, `write`, `reconn`}. Every symbol in the alphabet references dependent advice from the same aspect. Lines 17–19 enumerate, for each state, a (potentially empty) list of outgoing transitions. An entry "`s1: t -> s2`" means "there exists a `t`-transition from `s1` to `s2`". Users can also mark states as `initial` or `final` (error states). Final states denote states

in which the monitoring aspect "matches", i.e., produces an externally visible side effect like the error message in our example (line 13, Figure 2).

The first author's dissertation [2, page 134] gives the complete syntax for Dependency State Machines and also explains sanity checks for these annotations; e.g., each state machine must have initial and final states. Note that these checks are minimal and support a large variety of state machines so that CLARA can support many different runtime verification tools. For instance, we allow multiple initial and final states and we allow the state machine to be non-deterministic.

3.2 Semantics

The semantics of a Dependency State Machine refine the usual advice-matching semantics of AspectJ [13]. In AspectJ, pieces of advice execute at "joinpoints", or intervals of program execution. Programmers use "pointcuts", predicates over joinpoints, to specify the joinpoints where advice should apply. In Figure 2, the expression **call**(∗ Connection.disconnect()) && **target**(c) is a pointcut that picks out all method calls to the `disconnect` method of class `Connection`. When the pointcut applies, it binds the `target` object of the call to 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 follows:

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

Given advice $a \in \mathcal{A}$ and a joinpoint $j \in \mathcal{J}$, $match(a, j)$ is $\bot$ when a does not execute at j. If a does execute, then $match(a, j)$ yields a variable binding β, which maps a's formal parameters to objects.

Our formal semantics for Dependency State Machines will provide a replacement for *match*, called *stateMatch*, that determines the cases in which a dependent piece of advice needs to execute: informally, a dependent advice a must execute when (1) AspectJ would execute a and (2) when not executing a at j would change the set of joinpoints for which the Dependency State Machine reaches its final state for a binding compatible with β. (We define "compatible" later.) An optimal implementation, which determines exactly all cases in which a dependent advice does not need to execute, is un-computable, as it would have to anticipate the future behaviour (and inputs) of the program. The trick is therefore to implement statically computable approximations to *stateMatch*. At the end of this section, we will present a soundness condition for *stateMatch*. This condition uses the set of possible future behaviours to describe the permissible (sound) implementations of *stateMatch*.

Semantics by example. Figure 4 contains a small example program that helps explain the intuition behind our semantics. The program triggers joinpoints which the ConnectionClosed aspect monitors. AspectJ calls a program point that triggers a joinpoint j the "joinpoint shadow" of j, or just "shadow" [14] for short.

```
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): irrelevant shadow−stays in same state
5    c1.close ();       // close(c1)
6    c1.close ();       // close(c1): also irrelevant
7    c1.write(args[2]); // write(c1): violation−write after close on c1
8    c1.close ();       // close(c1): irrelevant−no subsequent writes on c1
9    c2.write(args[2]); // write(c2): write, but on c2, hence incompatible with 8
10 }
```

Fig. 4: Example program

Formal semantics. Our semantics of Dependency State Machines describe the set of program traces which cause the state machines to reach their final states. Note, however, that there is a mismatch between the usual semantics for 1) state machines and 2) program traces: state machines are not aware of variable bindings. We will call the traces that arise from program executions *parameterized traces* [15]. To apply Dependency State Machines to parameterized traces, we project a parameterized trace onto a set of ground traces, which the Dependency State Machine can process, obtaining one ground trace for every variable binding.

We will also define the semantics of Dependency State Machines in terms of "events", not joinpoints. A joinpoint describes a time interval, while an event is an atomic point in time. Events simplify reasoning by prohibiting nesting.

Event. Let j be an AspectJ joinpoint. Then j induces the pair of events j_{before} and j_{after}, which occur at the beginning and 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: $\mathcal{E}(\mathcal{J}) := \bigcup_{j \in \mathcal{J}} \{j_{\text{before}}, j_{\text{after}}\}$. We write $\mathcal{E}$ instead of $\mathcal{E}(\mathcal{J})$ when $\mathcal{J}$ is clear from context.

For any declaration of a Dependency State Machine, the list of dependent advice names forms an alphabet Σ. For instance, the alphabet for ConnectionClosed from Figure 2 is $\Sigma = \{\text{disconn}, \text{write}, \text{reconn}\}$.

Parameterized events. 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 $\hat{e}$ as follows:

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

Here, $\mathit{match}(a, e)$ is AspectJ's matching function, lifted to events; it therefore maps advice/event pairs to variable bindings, returning parameterized events. We label the set of all possible parameterized events $\hat{\mathcal{E}}$. Projection maps parameterized event traces ($\hat{\mathcal{E}}^*$) to "ground traces" (Σ^*).

Projected event. For every parameterized event $\hat{e} \in \hat{\mathcal{E}}$ and binding β we may project $\hat{e}$ with respect to β:

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

where *compatible* means that β_1 and β_2 agree on their joint domains:

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

In this predicate, $\mathit{dom}(\beta_i)$ denotes the domain of β_i, i.e., the set of variables where β_i is defined.

Parameterized and projected event trace. Any finite program run induces a finite parameterized event trace $\hat{t} = \hat{e}_1 \dots \hat{e}_n \in \hat{\mathcal{E}}^*$. 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:

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

We call such traces t, which are elements of Σ^*, "ground" traces; parameterized traces are instead elements of $\hat{\mathcal{E}}^*$.

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, which contains no events and hence cannot 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 AspectJ-inspired predicate $\mathit{match}(a, e)$, which models the decision of whether or not the dependent advice $a \in \mathcal{A}$ matches at event $e \in \mathcal{E}$, and if so, with which variable binding. We call our specialization *stateMatch* and define it as follows:

$$\mathit{stateMatch} : \ \mathcal{A} \times \hat{\mathcal{E}}^* \times \mathbb{N} \ \rightarrow \ \{\beta \mid \beta : \mathcal{V} \rightharpoonup \mathcal{O}\} \cup \{\bot\}$$

$$\mathit{stateMatch}(a, \hat{t}, i) :=$$
$$\text{let } \beta = \mathit{match}(a, e) \text{ in}$$
$$\begin{cases} \beta & \text{if } \beta \neq \bot \wedge \exists t \in \mathit{groundTraces}(\hat{t}) \text{ such that } \mathit{necessaryShadow}(a, t, i) \\ \bot & \text{otherwise} \end{cases}$$

Note that *stateMatch* considers the entire parameterized event trace $\hat{t}$, plus the current position i in that event trace. In particular, the trace $\hat{t}$ contains future events. The function *stateMatch* is therefore under-determined. This is intentional. Even though it is impossible to pass *stateMatch* all of its arguments, static analyses can approximate all possible future traces.

We have left a parameter *necessaryShadow* in the definition of *stateMatch*. This parameter may be freely chosen, as long as it meets the soundness condition defined below. A static optimization for Dependency State Machines is *sound* if it meets the soundness condition.

Soundness condition. The soundness condition requires that an event 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.

Matching prefixes of a word. Let $w \in \Sigma^*$ and $\mathcal{L} \subseteq \Sigma^*$. Then the matching prefixes of w (with respect to $\mathcal{L}$) are the set of prefixes of w in $\mathcal{L}$:

$$matches_{\mathcal{L}}(w) := \{p \in \Sigma^* \mid \exists s \in \Sigma^* \text{ such that } w = ps\} \cap \mathcal{L}$$

Soundness condition. For any sound implementation of *necessaryShadow* we require:

$$\begin{aligned}&\forall t = t_1 \dots t_i \dots t_n \in \Sigma^+.\ \forall i \leq n \in \mathbb{N}.\\ &\quad matches_{\mathcal{L}(\mathcal{M})}(t_1 \dots t_{i-1} t_i t_{i+1} \dots t_n) \neq matches_{\mathcal{L}(\mathcal{M})}(t_1 \dots t_{i-1} t_{i+1} \dots t_n)\\ &\qquad \Longrightarrow necessaryShadow(t_i, t, i)\end{aligned}$$

The soundness condition hence states that, if we are about to read a symbol t_i, and the monitoring aspect hits the final state when processing the complete trace t but not when processing the partial trace which omits t_i, or the other way around, then we must monitor t_i.

Note that CLARA's semantics assume that the advice associated with Dependency State Machines implement the monitor's transition structure. In particular, any dependent advice which does anything beyond computing a state transition must be marked final. Tools which generate Dependency State Machines, or programmers who write them, must take this semantics into account.

4 Clara as a framework

Version 1.0 of CLARA includes three sound static analyses which eliminate irrelevant shadows. Recall from Figure 3 that CLARA executes these analyses immediately after weaving; the analyses plug into its static analysis engine. Analyses may access all declared Dependency State Machines and the woven program. The analyses also receive a list of joinpoint shadows.

For every shadow s, CLARA exposes the following pieces of information:

- The dependent piece of advice a that s invokes, along with the name of a and a list of variables that a binds.
- The source code position of s.
- The dynamic residue of s, which abstractly represents the runtime check that determines whether a will actually execute. A static analysis can disable s by setting its residue to the constant "NeverMatch".
- A mapping from the variables that a binds at s to a points-to set [18] that models all objects that these variables could possibly point to.

CLARA comes pre-equipped with three analyses that all aim to determine "irrelevant shadows". Such shadows must return `false` for *necessaryShadow*; in other words, disabling an irrelevant shadow must preserve the behaviour of the runtime monitor. An analysis disables a shadow by modifying its dynamic residue to never match.

The **Quick Check** [16], CLARA's first analysis stage, quickly computes whether all shadows for a particular property are irrelevant because they do not suffice to reach a final state; if so, it removes all of the shadows for that property. The second analysis stage, the **Orphan Shadows Analysis** [16] takes pointer information into account to find more specific sets of shadows, related by pointer information, which can all be disabled. CLARA uses a flow-insensitive and context-sensitive, demand-driven, refinement-based pointer analysis [18] to determine which events may occur on which groups of compatible variable bindings. The third stage, the **Nop Shadows Analysis** [17], explicitly takes the program's control flow into account. Using a backwards pass, the Nop Shadows Analysis first determines for every shadow s a tri-partitioning of automaton states: states from which the remainder of the program execution will, may, or won't reach the final state. Next, the Nop Shadows Analysis uses a forward pass to determine the possible automaton states at s. If s may only transition between states in the same equivalence class, then the analysis can soundly disable s.

We described all of three analyses in earlier work in [2,16,17]; the dissertation also includes soundness proofs. In this paper, however, we describe for the first time the common framework that makes these analyses accessible to various AspectJ-based runtime monitoring tools.

Adding analyses to CLARA

CLARA allows researchers to add their own static analyses to the static analysis engine at any point. The CLARA website provides an empty skeleton extension for researchers to fill in. Analyses execute, in sequence, immediately after weaving. CLARA executes the three default analyses in the order in which we described them above: quick ones first, more complex ones later. In many cases, even simple analyses like the Quick Check are already powerful enough to recognize all shadows as irrelevant, which obviously simplifies the task of the more complicated analyses.

Programmers can insert their own analysis at any point in the sequence, as a so-called re-weaving pass. As the name suggests, a pass participates in a process called re-weaving [**?**]: just after having woven the monitoring aspects into the program, the AspectBench Compiler that underlies CLARA executes the given sequence of passes. Each pass may modify the so-called "weaving plan", e.g., by modifying the residues of joinpoint shadows. After all passes have finished, the compiler then restores the original un-woven program version and re-weaves the program using this new plan, this time then with fewer joinpoint shadows when the analysis passes succeeded.

5 Experimental results

In this section we explain our empirical evaluation and our experimental results. Due to space limitations, we can only give a summary of those results. The first author's dissertation [2] gives a full account.

Although one can apply CLARA to any AspectJ-based runtime monitor, we decided to restrict our experiments to monitors generated from tracematch specifications. This does not limit the generality of our results: in earlier work [16] we showed that the relative optimization effects of our static analyses are largely independent of the concrete monitoring formalism.

For our experiments we wrote a set of twelve tracematch [3] specifications for different properties of collections and streams in the Java Runtime Library. Table 1 gives brief descriptions for each of these properties. We selected properties of the Java Runtime Library due to the ubiquity of clients of this library. Our tracematch definitions are available at `http://bodden.de/clara/benchmarks/`.

property name	description
ASyncContainsAll	synchronize on `d` at calls to `c.containsAll(d)` for synchronized collections `c`, `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 is closed
Writer	do not use a Writer after its OutputStream is closed

Table 1: Monitored specifications for classes of the Java Runtime Library

We used CLARA to instrument the benchmarks of version 2006-10-MR2 of the DaCapo benchmark suite [20] with these runtime monitors. DaCapo contains eleven different workloads of which we consider all but eclipse. Eclipse uses reflection heavily, which Clara still has trouble dealing with. For our experiments, we used 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 4GB RAM. We summarize our results in Table 2.

As the table shows, instrumenting 109 out of the 120 cases require at least one instrumentation point for runtime monitoring. (We mark other cases with "-".) CLARA was able to prove (✓) for 74 out of these 109 cases (68%) that the program cannot violate the property on any execution. In these cases, monitoring is unnecessary because CLARA removes all instrumentation. 37 of the original 109 combinations showed a measurable runtime overhead. After applying the

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

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

Table 2: Effect of CLARA's static analyses; numbers are runtime overheads in percent before and after applying the analyses; ✓: all instrumentation removed, proving that no violation can occur; >1h: run took over one hour

static analysis, measurable overhead only remained in 13 cases (35%). These cases often show significantly less overhead than without optimization.

Jython causes trouble for CLARA because of its heavy use of reflection and dynamic class loading. Due to these features, the pointer analysis that CLARA uses has to make conservative assumptions, yielding imprecise results. CLARA also performs less well on Iterator-based properties than on others. Because Java programs usually create all iterator objects through the same `new` statement, CLARA requires context information to distinguish different iterators statically. Our pointer analysis sometimes fails to generate enough context information, leading to imprecision. For fop/FailSafeIterMap, our analysis ran out of memory, despite the fact that we allowed 3GB of heap space.

The first author's dissertation [2] presents detailed experiments and results.

6 Related Work

CLARA's static analyses can be considered to be typestate analyses. Strom and Yemini [1] were the first to suggest the concept of typestate analysis. Recently,

researchers have presented several new approaches with varying cost/precision trade-offs. We next describe the approaches most relevant to our work. We distinguish work in type systems, static verification and hybrid verification.

Type-system based approaches. Type-system based approaches define a type system and implement a checker for that system. The checker prevents programmers from compiling potentially property-violating programs and gives strong static guarantees. However, the type checker may reject useful programs which statically appear to violate the stated property but never actually violate the property at runtime.

DeLine and Fähndrich [21] as well as Bierhoff and Aldrich [22] present type systems for object-oriented languages with aliasing. Bierhoff and Aldrich's type system is generally more permissive than DeLine and Fähndrich's. To enable modular analyses, both of these approaches require annotations in the target program indicating state transitions and aliasing relationships. We do not require annotations in the program; our approach infers state changes from advice.

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

Fink et al. present a static analysis of typestate properties [23]. 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, their algorithms cannot generate 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 [24]. Naeem and Lhoták's analysis is fully inter-procedural. Unfortunately, Naeem and Lhoták based parts of their analysis on earlier work of ours [25] that turned out be unsound [17]. All of Clara's analyses provides have been proven sound [2].

Dwyer and Purandare use existing typestate analyses to specialize runtime monitors [26]. Their work identifies "safe regions" in the code using a static typestate analysis. Safe regions can be single statements, compound statements (e.g. loops), or methods. A region is safe if its deterministic transition function does not drive the typestate automaton into a final state. For such regions, their analyses 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. Because these specializations change the points at which transitions occur, they can make it harder for programmers to understand monitor behaviour. Further, their approach cannot generally handle groups of multiple interacting objects, while ours can.

7 Conclusion

We have presented CLARA, a framework for partially evaluating finite-state runtime monitors ahead-of-time using static analysis. CLARA is compatible with any runtime monitor that is expressed as an AspectJ aspect. To make any such aspect analyzable by CLARA, users need only ensure that the aspect is annotated with a Dependency State Machine, a textual finite-state-machine representation of the property being verified. Dependency State Machines function as an abstract interface, allowing researchers in runtime verification to implement monitor optimizations on one side of this interface and static-analysis researchers to implement static analyses on the other side. This way, CLARA allows researchers from two communities to integrate their approaches with each other.We have presented the syntax and semantics of Dependency State Machines and CLARA's extensible static analysis engine, along with three analyses that we provide with CLARA. Through experiments with the DaCapo benchmark suite, we have shown that CLARA's static analysis approach 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 free, open-source software. We hope that other researchers will soon be joining us in using CLARA, and that its availability will foster progress in the field of typestate analysis.

References

1. Strom, R.E., Yemini, S.: Typestate: A programming language concept for enhancing software reliability. IEEE Transactions on Software Engineering (TSE) **12**(1) (January 1986) 157–171
2. Bodden, E.: Verifying finite-state properties of large-scale programs. PhD thesis, McGill University (June 2009) Available through ProQuest.
3. 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. (October 2005) 345–364
4. Bodden, E.: J-LO - A tool for runtime-checking temporal assertions. Master's thesis, RWTH Aachen University (November 2005)
5. Chen, F., Roşu, G.: MOP: an efficient and generic runtime verification framework. In: OOPSLA. (October 2007) 569–588
6. Maoz, S., Harel, D.: From multi-modal scenarios to code: compiling LSCs into AspectJ. In: Symposium on the Foundations of Software Engineering (FSE). (November 2006) 219–230
7. 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). (May 2006) 51–58
8. Avgustinov, P., Tibble, J., de Moor, O.: Making trace monitors feasible. In: OOPSLA. (October 2007) 589–608
9. Chen, F., Meredith, P., Jin, D., Roşu, G.: Efficient formalism-independent monitoring of parametric properties. In: ASE. (2009) 383–394

10. Dwyer, M.B., Diep, M., Elbaum, S.: Reducing the cost of path property monitoring through sampling. In: ASE, Washington, DC, USA (2008) 228–237
11. AspectJ team: The AspectJ home page, `http://eclipse.org/aspectj/` (2003)
12. 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. (March 2005) 87–98
13. Bodden, E., Hendren, L., Lam, P., Lhoták, O., Naeem, N.A.: Collaborative runtime verification with tracematches. Journal of Logics and Computation (November 2008) doi:10.1093/logcom/exn077.
14. Bodden, E., Chen, F., Roşu, G.: Dependent advice: A general approach to optimizing history-based aspects. In: AOSD. (March 2009) 3–14
15. Hilsdale, E., Hugunin, J.: Advice weaving in AspectJ. In: AOSD. (March 2004) 26–35
16. 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., Springer (April 2003) 46–60
17. Chen, F., Roşu, G.: Parametric trace slicing and monitoring. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Volume 5505 of LNCS., Springer (March 2009) 246–261
18. Sridharan, M., Bodík, R.: Refinement-based context-sensitive points-to analysis for Java. In: Conference on Programming Language Design and Implementation (PLDI). (June 2006) 387–400
19. 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, New York, NY, USA, ACM (2010) 5–14
20. Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Optimising AspectJ. In: Conference on Programming Language Design and Implementation (PLDI). (June 2005) 117–128
21. 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. (October 2006) 169–190
22. DeLine, R., Fähndrich, M.: Typestates for objects. In: ECOOP. Volume 3086 of LNCS., Springer (June 2004) 465–490
23. Bierhoff, K., Aldrich, J.: Modular typestate checking of aliased objects. In: OOPSLA. (October 2007) 301–320
24. 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). (July 2006) 133–144
25. Naeem, N.A., Lhoták, O.: Typestate-like analysis of multiple interacting objects. In: OOPSLA. (October 2008) 347–366
26. 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). (November 2008) 36–47
27. Dwyer, M.B., Purandare, R.: Residual dynamic typestate analysis: Exploiting static analysis results to reformulate and reduce the cost of dynamic analysis. In: ASE. (May 2007) 124–133