Efficient Hybrid Typestate Analysis by Determining Continuation-Equivalent States

Eric Bodden*
Software Technology Group
Department of Computer Science
Technische Universität Darmstadt, Germany
bodden@acm.org

ABSTRACT

Typestate analysis determines whether a program violates a set of finite-state properties. Because the typestate-analysis problem is statically undecidable, researchers have proposed a hybrid approach that uses residual monitors to signal property violations at runtime.

We present an efficient novel static typestate analysis that is flow-sensitive, partially context-sensitive, and that generates residual runtime monitors. To gain efficiency, our analysis uses precise, flow-sensitive information on an intra-procedural level only, and models the remainder of the program using a flow-insensitive pointer abstraction. Unlike previous flow-sensitive analyses, our analysis uses an additional backward analysis to partition states into equivalence classes. Code locations that transition between equivalent states are irrelevant and require no monitoring. As we show in this work, this notion of equivalent states is crucial to obtaining sound runtime monitors.

We proved our analysis correct, implemented the analysis in the Clara framework for typestate analysis, and applied it to the DaCapo benchmark suite. In half of the cases, our analysis determined exactly the property-violating program points. In many other cases, the analysis reduced the number of instrumentation points by large amounts, yielding significant speed-ups during runtime monitoring.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Validation*

General Terms

Algorithms, Experimentation, Performance, Verification

Keywords

typestate analysis, static analysis, runtime monitoring

*Eric conducted most of this research as a Ph.D. student at McGill University, under supervision of Laurie Hendren. This work was supported by NSERC and CASED (www.cased.de).

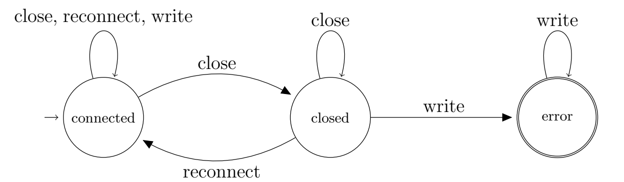


Figure 1: Finite-state machine for "Connection" property

1. INTRODUCTION

A typestate property [22] describes which operations are available on an object or even a group of inter-related objects, depending on this object's or group's internal state, the typestate. For instance, programmers must not write to a connection handle that is currently in its "closed" state. Figure 1 shows a non-deterministic finite-state machine for this property. It monitors a connection's "close", "reconnect" and "write" events and signals an error at its accepting state.

Typestate properties aid program understanding, and one can even define type systems [5, 14] that prevent programmers from causing typestate errors, or derive static typestate analyses [17] that try to determine whether a given program violates typestate properties. Unfortunately, the typestate-analysis problem is generally undecidable. Researchers have therefore proposed a hybrid approach [9, 10, 16] that uses static-analysis results to generate a residual runtime monitor. This monitor captures actual property violations as they occur, but only updates its internal state at relevant statements, as determined through static analysis.

A correct runtime monitor must observe events like "close" and "write" that can cause a property violation, but also events like "reconnect" that may prevent the violation from occurring. Missing the former causes false negatives while missing the latter causes false positives, i.e., false warnings. Either is unacceptable, as runtime monitors must handle property violations exactly when they occur. A correct static analysis must therefore determine program locations that can trigger either kind of such "relevant" events.

In this work we present an efficient novel static typestate-analysis algorithm called Nop-shadows Analysis[1] that uses a forward and a backward pass to identify provably irrelevant code locations. For every program statement s of interest, the forward analysis determines the possible typestates that can reach s. The additional backward analysis partitions these states into equivalence classes. A program location

[1]The aspect-oriented-programming community uses the term "shadow" [18] to refer to instrumentation points.

that can only transition between equivalent states is irrelevant. This eases the burden on the programmer, as the programmer does not need to consider such irrelevant locations during manual code inspections. Moreover, we eliminate the monitoring instrumentation from these program locations, speeding up the residual runtime monitor. Our novel analysis is not only simpler than earlier approaches, it is also both precise and efficient.

Any precise typestate analysis has to be flow-sensitive and requires must-alias information: to determine that the code "c1.reconnect(); c2=c1; c2.write();" correctly uses the connection that c1 and c2 refer to, the analysis needs to know that c1 and c2 must point to the same object, i.e. that c1 and c2 must-alias. Such information is expensive to compute. To gain efficiency, our analysis computes flow information and must-alias information on an intra-procedural level only, and models the remaining program using a carefully designed flow-insensitive pointer abstraction.

To evaluate our approach, we have implemented our analysis in the Clara framework for typestate analysis [7] and applied the analysis to the DaCapo benchmark suite [6]. Our results show that our lightweight abstractions are precise enough to exactly tell apart property-violating program points from irrelevant program points in half of the cases. For these cases, the analysis determines exactly the property-violating program locations. In many other cases the analysis identifies and disables large amounts of irrelevant program points. This eases manual code inspection and, as we show, speeds up the resulting residual monitor significantly. Our modest abstractions restrict the analysis time to a few minutes in most cases.

We proved our analysis correct. As we found out during this process, two analyses that we [10] and others [20] published previously are unsound. They fail to identify certain program locations that trigger violation-preventing events like "reconnect" above. As a consequence, the resulting runtime monitors may cause false warnings at runtime. This unsoundness is caused by the fact that traditional typestate analyses use a forward-analysis pass only and have no notion of equivalent states like our novel analysis does.

To summarize, this paper presents the following original contributions:

- a novel flow-sensitive static typestate analysis, called "Nop-shadows Analysis", that detects equivalent typestates to determine all statements that are relevant to causing or preventing a property violation,
- an implementation of the Nop-shadows Analysis in the Clara framework that generates efficient residual runtime monitors,
- a set of experiments that shows that the analysis is both precise and efficient, and
- an explanation of why some other static analyses that use a forward-analysis pass only, generate unsound runtime monitors.

We structured the remainder of this paper as follows. We start off by giving a brief overview of the Clara framework, in which we implemented our analysis. In Section 3 we give an example to illustrate the Nop-shadows Analysis itself. We explain the full analysis in Section 4, followed by our experiments in Section 5. In Section 6 we discuss how we improve over the state of the art. We conclude in Section 7.

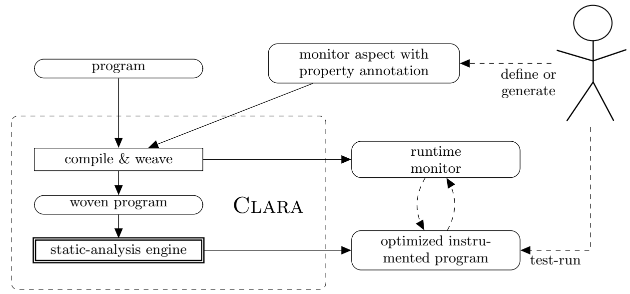


Figure 2: Overview of Clara

2. THE CLARA FRAMEWORK

Clara (CompiLe-time Approximation of Runtime Analyses) is a novel research framework for the implementation of hybrid typestate analyses. We developed Clara to support easy implementation of the analysis that we present in this paper, among others. Clara's major design goal is to de-couple the code-generation for efficient runtime monitors from the static analyses that convert these monitors into faster, residual monitors. In this work, we can only give an overview of Clara. The author's dissertation [7] gives a more detailed account. Clara is available at:

`http://bodden.de/clara/`

Figure 2 gives an overview of Clara. With Clara, the researcher first defines a set of typestate properties, denoting each property as an annotation to an AspectJ [3] aspect that implements a runtime monitor for the same property. Annotations directly encode a non-deterministic finite-state machine, just as the one in Figure 1. In the author's dissertation [7] we show how researchers can even use runtime-monitoring approaches like tracematches [2], JavaMOP [13], and others, to generate these annotated aspects automatically from high-level monitor specifications.

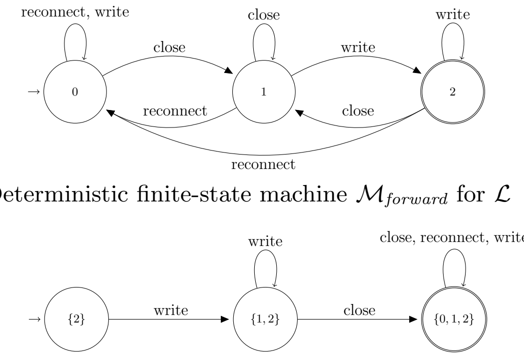
Clara weaves the monitoring aspect into the program under test and emits helper classes that implement the runtime monitor as defined by the aspect. Clara extends the AspectBench Compiler [4] for this purpose. Clara then invokes its static-typestate-analysis engine. Researchers can add a number of static analyses to Clara and have them applied in any order. These analyses obtain, through the finite-state machine defined in the aspect's annotation, enough information about the typestate property to precisely approximate the set of relevant instrumentation points. When an analysis determines that an instrumentation point is irrelevant to a property, i.e., the program can neither violate the property nor prevent a property violation at this point, then Clara automatically disables the instrumentation for this property at this point. The result is an optimized instrumented program that updates the runtime monitor only at program points at which instrumentation remains enabled. The approach that we present in the following instantiates Clara's static-analysis engine with a combination of two previously published supporting analyses and our novel Nop-shadows Analysis.

3. ANALYSIS BY EXAMPLE

We motivate our analysis using our running example: it is an error to write to a connection object that was closed, unless the connection was re-connected in between. Figure 1 shows the appropriate non-deterministic finite-state

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

Figure 3: Simple example program using a single connection



(a) Deterministic finite-state machine $\mathcal{M}_{forward}$ for $\mathcal{L}$

(b) Deterministic finite-state machine $\mathcal{M}_{backward}$ for $\overline{\mathcal{L}}$

Figure 4: Finite-state machines for Connection example

machine that CLARA obtains from parsing the annotation in the specified aspect. The state machine accepts events of type "close", "reconnect" and "write". When a "write" follows a "close", then the state machine moves into its error state. CLARA represents error states as accepting states. Let us call the language that this state machine accepts $\mathcal{L}$.

Figure 3 shows a simple example program that uses a single connection, along with the analysis information that we compute (explanation follows). To keep the example simple, this program contains only straight-line code, no outgoing method calls that may change the connection's type state, and no aliasing. Our implementation, however, handles complete Java programs, including method calls, recursion, loops, exceptions and aliasing (see Section 4).

Our example program violates the connection property by closing the connection (even twice, at lines 5 and 6), and then writing to the connection (line 7). Note, though, that all other statements in this program are irrelevant to the property violation. In particular, one does not need to monitor the "close" and "reconnect" operations at lines 3 and 4 because they precede the violating fragment of the run. Conversely, the operations at lines 8 to 10 follow this fragment, and hence do not need to be monitored either. A little more subtle, it is even correct to omit monitoring one of the two "close" events at lines 5 and 6. (But not both!) The static analysis that we present in the following will eliminate the instrumentation at exactly those shadows that we just identified as irrelevant—"nop shadows", as we call them. Instrumentation will only remain in lines 5 and 7, or 6 and 7—an optimal result for this program.

Example application of analysis algorithm.

For a method containing n shadows, the Nop-shadows Analysis consists of up to n "rounds", where each round identifies a single nop shadow until no further nop shadows can be identified. Each round consists of a forward and a backward pass. The forward pass computes for every statement s the typestates that can reach s. The backwards pass conversely computes classes of states from which the property state machine can reach a violating state using the remainder of the program execution that follows s.

The forward pass uses a determinized version of the finite-state machine from the property specification. Figure 4a shows this state machine for our example. In the following, we will call this state machine $\mathcal{M}_{forward}$. We number $\mathcal{M}_{forward}$'s states for presentation purposes. The forward analysis starts off in this state machine's initial state 0 and then updates the state according to the shadows that it encounters during analysis. In Figure 3, we show the states that the forward analysis computes before and after each statement, next to the downward arrow. For instance, the "close" statement at line 3 changes the typestate from 0 to 1. Importantly, at property violations, e.g. at line 7, the analysis will reach the violating state 2.

The backward pass, on the other hand, uses a deterministic finite-state machine for $\mathcal{L}$'s mirror language $\overline{\mathcal{L}}$. For any word $w = w_1 \dots w_n \in \mathcal{L}$, we define the mirror word $\overline{w}$ as $\overline{w} := w_n \dots w_1$. The mirror language $\overline{\mathcal{L}}$ is defined by $\overline{\mathcal{L}} := \{\overline{w} \mid w \in \mathcal{L}\}$. While it would be sound for our backwards pass to use any finite-state machine that accepts the language $\overline{\mathcal{L}}$, we specifically use the state machine that we obtain by (1) reversing $\mathcal{M}_{forward}$ (by flipping all edges and swapping initial and accepting states), and then (2) determinizing this finite-state machine again. The resulting state machine is minimal for $\overline{\mathcal{L}}$ [12]. As we will explain in Section 4, a minimal finite-state machine yields a more precise analysis result because in this state machine equivalent states are collapsed into a single state. For our example, Figure 4b shows the finite-state machine that we obtain this way. (We omit the "sink" state that represents the empty state set.) We call this automaton $\mathcal{M}_{backward}$. Note that we labeled each of this automaton's states with the set of states of $\mathcal{M}_{forward}$ that this state represents. These labels are important: our analysis will compare states from the forward analysis with states from the backward analysis, and labeling $\mathcal{M}_{backward}$'s states with their equivalent states of $\mathcal{M}_{forward}$ eases this comparison.

According to the semantics of CLARA's state-machine notation, a single program run can cause multiple property violations. Each violating sub-path of the execution path starts at one of the program's possible entry points and ends at what we call a "final shadow". A final shadow is a shadow that is labeled with a symbol that leads into an error state, like "write" in our example. Therefore, in the example, we apply the backwards analysis starting at both write statements (lines 10 and 7). In Figure 3, we show the analysis result for both backwards-analysis runs on the right-hand side. For instance, the close statement at line 6 changes the typestate from the state labeled with $\{1, 2\}$ to the state labeled with $\{0, 1, 2\}$. The same statement further loops on the sink state $\{\}$: as we show next to the left upward arrow, the statement transitions from $\{\}$ again to $\{\}$.

Nop-shadow condition. We now explain how we combine the forward and backward-analysis information to identify nop shadows. Let *source*(s) be the state that the forward analysis computed just before a statement s, *target*(s) the state for the location just after s, and *futures*(s) the set of state sets that the backwards analysis computed for just after s. For instance, for the close statement at line 5 of Figure 3 we have:

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

The following property is crucial to our approach: because we compute *futures*(s) using a deterministic state machine for $\overline{\mathcal{L}}$, the sets in *futures*(s) represent equivalence classes. For instance, the set $\{0, 1, 2\}$ represents the fact that, using the remainder of the program execution, one will reach a property violation from $\mathcal{M}_{forward}$'s states 0, 1 or 2 *either way*. By using a minimal state machine we assure optimality: when two states q_1 and q_2 are equivalent, then in *futures*(s) both q_1 and q_2 will be members of the same class.

In the following, for two states q_1 and q_2 we say that q_1 and q_2 are equivalent and write $q_1 \equiv q_2$ if the following holds:

$$\forall Q \in \mathit{futures}(s).\ q_1 \in Q \leftrightarrow q_2 \in Q$$

A shadow is a nop shadow when it transitions between states in the same equivalence class. Let us denote by F the set of accepting, i.e., violating states of $\mathcal{M}_{forward}$. Then we call a shadow at a statement s a "nop shadow" if:

1. $\mathit{source}(s) \equiv \mathit{target}(s)$, and
2. $\mathit{target}(s) \notin F$.

The second case is necessary, because according to CLARA's monitoring semantics, a monitor must signal repeated property violations every time the violation occurs. This is useful when the monitor executes error-handling code. For instance, on "c.close();c.write();c.write()" the monitor should signal a violation after both "write" events. This is although the second "write" event does not change the typestate: we have $\mathit{source}(s) = \mathit{target}(s) = 2$.

Need for re-iteration. It is important to note that, by our above definition, a nop shadow is only a nop shadow in its current context, i.e., if all other shadows remain enabled. This is because after identifying any particular single shadow as a nop shadow, by disabling this shadow, we change the program's transition structure, and previous nop shadows may therefore not be nop shadows any more. This is exactly the case at lines 5 and 6 of our example: According to the nop-shadow condition, both shadows are nop shadows. However, disabling one of these shadows will render the other one necessary: one needs to observe one of these two shadows; otherwise the runtime monitor will not reach its violating state at line 7. Conversely, our nop-shadow condition does not initially identify the "reconnect" shadow at line 9 as a nop shadow. This is because disabling this shadow, while keeping the shadows at lines 8 and 10 enabled, would cause a false positive at runtime, at line 10. However, after any of these latter two shadows have been enabled, the reconnect shadow will become a nop shadow too, and can be disabled as well. To be both sound and effective, we therefore programmed our analysis to disable a single shadow at a time, and then re-compute the analysis information, re-iterating until no further nop shadow can be identified. The number of iterations is bounded by the number of shadows that the current method holds. In our example, we reach a fixed point in the 7th iteration; instrumentation will only remain in lines 5 and 7, or 6 and 7.

While this result is optimal for this particular program, note that because our algorithm disables shadows one by one, it works in a "greedy" fashion. In theory, this may cause the algorithm to reach its fixed point in a local optimum instead of the global optimum. The author's dissertation [7, Appendix C] gives a constructed example that demonstrates this behavior. As we will show, however, our simple "greedy" solution performs very well in practice. We conjecture that this is due to the usually relatively simple structure [15] that typical finite-state properties have.

4. NOP-SHADOWS ANALYSIS

We next explain how we handle the general analysis problem, involving loops, outgoing method calls, recursion, exceptions and aliasing. For every shadow s in the instrumented program, to determine whether s is a nop shadow, our analysis needs to compute *source*(s), *target*(s) and the set *futures*(s). The analysis needs to fulfil multiple conditions to compute these pieces of analysis information in a sound way. We (1) need to consider all possible control-flow paths through the program that could potentially lead up to the execution of s and that are completed by the continuation of the control flow after s. Then (2), along every such path, we must assure that we never merge *futures*-sets at any point: merging sets would compute incorrect equivalencies. Lastly, (3) the analysis must be able to distinguish the typestates of multiple different (combinations of) objects. In particular, the analysis must be able to handle aliasing.

To cope with (3), the analysis distinguishes the analysis information for different objects by propagating "configurations" instead of bare state sets: a configuration $c = (Q, b)$ combines an automaton-state set Q with a variable binding b. When c is associated with a statement s, then the state set Q holds all possible typestates just before executing s. The binding b describes the object(s) which this state set is associated with. In our connection example, for any given statement s there could exist multiple connections that are in different typestates when s executes. The variable bindings help distinguish these different typestates. A variable binding maps one or more variables from the typestate specification to object representatives[2] that model the runtime objects that these variables are bound to. The treatment of variable bindings is quite intricate, but has been presented in previous work [10]. Hence, we ease our presentation by abstracting from variable bindings and instead assuming that we perform typestate analysis only for one single object representative. For the remainder of this paper we therefore assume that a configuration is just a set of automaton states, without any binding. The author's dissertation [7] gives a complete treatment including variable bindings, with proof.

[2]Object representatives [11] transparently model runtime objects at compile time. Two representatives are equal when they must-alias, i.e., represent the same runtime objects. In addition, they support a must-not-alias operation. Object representatives combine flow-insensitive whole-program points-to sets with intra-procedural flow-sensitive alias information. We compute points-to sets with Sridharan and Bodík's context-sensitive points-to analysis [21].

We ensure condition (2), i.e., not merging state sets, by simply not merging configurations at any time. In particular, we do not merge configurations at control-flow merge points: if a conditional execution leads to a configuration c_1 on one branch and to c_2 on another branch then we propagate both c_1 and c_2 after both branches have merged.

For efficiency, we designed the Nop-shadows Analysis to compute flow-sensitive analysis information only on an intra-procedural level. In particular, our must-alias analysis is only intra-procedural. One may think that such an analysis would have to be quite imprecise. However, before we designed our analysis, we manually investigated the instrumentation points that our typestate instrumentation causes and found that, in most cases, intra-procedural analysis information was sufficient to determine nop shadows. To take care of condition (1), i.e., maintain soundness in the presence of inter-procedural control flow, we pair this precise intra-procedural information with coarse-grained inter-procedural summary information that can be computed relatively efficiently. The results that we present in Section 5 confirm that this solution is both precise and efficient.

In line with our example from Section 3, the Nop-shadows Analysis computes for every shadow-bearing method both a forward- and a backward-analysis pass. The forward and backward analysis are both instances of a general worklist algorithm that we show as Algorithm 1. In this algorithm, the syntax $f[x \mapsto y]$ denotes the function that is equal to f on all values v, except for x, in which case it returns y:

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

We will explain the internal workings of this algorithm in Section 4.2. First we will explain how we initialize the algorithm's parameters.

Algorithm 1 $\mathit{worklist}(\mathit{initial}, \mathit{succ}_{cfg}, \mathit{succ}_{ext}, \delta)$

$\mathit{wl} := \mathit{initial}$
$\mathit{before} := \mathit{after} := \lambda \mathit{stmt}.\ \emptyset$
while wl non-empty **do**
  pop job $(\mathit{stmt}, \mathit{cs})$ from wl
  $\mathit{before} := \mathit{before}[\mathit{stmt} \mapsto \mathit{before}(\mathit{stmt}) \cup \mathit{cs}]$
  $\mathit{cs}' := \begin{cases} \mathit{cs} & \text{if } \mathit{shadows}(\mathit{stmt}) = \emptyset \\ \emptyset & \text{otherwise} \end{cases}$
  for $c \in \mathit{cs}, \quad \mathit{shadow} \in \mathit{shadows}(\mathit{stmt})$ **do**
    $c' := \bigcup_{q \in c} \{\ \delta(q, \mathit{label}(\mathit{shadow}))\ \}$
    $\mathit{cs}' := \mathit{cs}' \cup \{c'\}$
  end for
  $\mathit{cs}_{new} := \mathit{cs}' - \mathit{after}(\mathit{stmt})$
  if cs_{new} non-empty **then**
    $\mathit{after} := \mathit{after}[\mathit{stmt} \mapsto \mathit{after}(\mathit{stmt}) \cup \mathit{cs}_{new}]$
    for $\mathit{stmt}' \in \mathit{succ}_{cfg}(\mathit{stmt})$ **do**
      $\mathit{wl} := \mathit{wl} \cup \{(\mathit{stmt}', \mathit{cs}_{new})\}$
    end for
    for $\mathit{stmt}' \in \mathit{succ}_{ext}(\mathit{stmt})$ **do**
      $\mathit{wl} := \mathit{wl}[\mathit{stmt}' \mapsto \mathit{wl}(\mathit{stmt}') \cup$
        $\mathit{reaching}(\mathit{cs}_{new}, \mathit{relatedShadows}(\mathit{stmt}))]$
    end for
  end if
end while

4.1 Initializing the worklist algorithm

The initialization depends on whether we perform a forward or backwards analysis. The first argument to the algorithm is a set *initial* of initial "jobs". A job is a pair $(\mathit{stmt}, \mathit{cs})$, associating a statement *stmt* with a set *cs* of configurations, i.e., with a set of sets of automaton states.

For the forward analysis we must initialize the algorithm with configurations that model all possible control flow that could have occured before entering the current method m, but without having executed m already. (Later in this section, we explain how we soundly handle re-executions of m as well, trough a special successor function succ_{ext}.)

If q_0 is the initial state of $\mathcal{M}_{forward}$ and $\mathit{shadowsNotIn}(m)$ is the set of shadows outside of m, then the set of initial jobs associates with m's entry statement e all configurations that are reachable from the initial configuration $\{q_0\}$ by executing any of the $\mathit{shadowsNotIn}(m)$:

$$\mathit{initial} := \big\{ (\ e,\ \mathit{reaching}(\{q_0\}, \mathit{shadowsNotIn}(m))\ ) \big\}$$

Here we define for every set cs of configurations and ss of shadows $\mathit{reaching}(\mathit{cs}, \mathit{ss})$ as the smallest set of configurations for which holds:

- $\mathit{cs} \subseteq \mathit{reaching}(\mathit{cs}, \mathit{ss})$, and
- $\forall c \in \mathit{reaching}(\mathit{cs}, \mathit{ss})\ \forall a \in l :$
  $\{\delta(q, a) \mid q \in c\} \in \mathit{reaching}(\mathit{cs}, \mathit{ss})$.

Note that this fixed-point computation is flow-insensitive: we do not consider the order in which the shadows in ss could execute. This allows us to compute *reaching* very efficiently.

For the backwards analysis, we generate initial configurations in a similar but not identical way. A violating trace can only start at the beginning of the program, but it can end (causing a violation) in the current method m itself, or in another method (either with m on the call stack or not). We generate initial jobs to cover these three cases. Due to space limitations we give a formal definition in the accompanying dissertation [7, Section 5.2.3.4].

The second and third arguments to the algorithm, succ_{cfg} and succ_{ext}, are successor functions that model the possible control flow within the current method m, respectively outside of m. Figure 5 visualizes both successor functions. We show the current method m, here `foo`, as a box. The method contains two invoke statements. The first statement resembles a potentially-recursive call (including mutually-recursive calls), the second one a provably non-recursive call. The dashed arrows denote the successor function succ_{cfg}, which is given by m's control-flow graph. This graph allows us to soundly handle control flow caused by conditionals, loops and exceptions. Solid arrows show the second, inter-procedural-successor function, succ_{ext}. During the execution of m, invoke expressions within m may cause methods to be called. These calls either can or cannot transitively perform a recursive call back into m. When the call may be recursive, then configurations that we computed for this call site can reach m's entry statement, see arrow (1). Conversely, for configurations that we computed for any of m's exit statements, we need to propagate these configurations back to any potentially-recursive call site within m, see (2). At compile time, we can usually only determine that a method call may be recursive, not that it must be. Hence, we also need to propagate configurations from the call site to after itself, see (3a). For calls that are provably not recursive (as

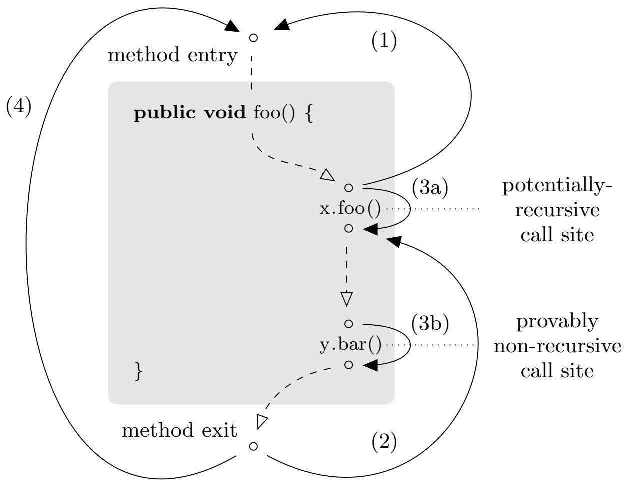


Figure 5: Functions $succ_{cfg}$ (dashed) & $succ_{ext}$ (solid)

determined by a call graph), it suffices to propagate configurations past the call site itself, see (3b). Lastly, we need to take into account the case in which method m returns (either normally or by throwing an exception—$succ_{cfg}$ handles both cases), and then re-executes. To model this case we propagate configurations from m's exit(s) to its entry, (4).

In line with Figure 5, we define $succ_{ext}$ as follows. Let $heads(m)$ be the set of entry statements of m, and $tails(m)$ the set of m's exit statements[3]. Further, let $recCall(m)$ be the set of potentially-recursive invoke statements of m, and $nonRecCall(m)$ the set of provably non-recursive invoke statements respectively. Then:

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

For the backward analysis we revert both successor functions the $succ_{cfg}$ and $succ_{ext}$, i.e., we flip all their edges, before passing them to our worklist algorithm. This causes the backward analysis to actually compute backwards.

The fourth and final parameter to our worklist algorithm is δ, the automaton's transition function. For the forward analysis we use the transition function of $\mathcal{M}_{forward}$. For the backward analysis we revert this function, flipping all edges. Effectively, this yields a non-deterministic version of $\mathcal{M}_{backward}$. This is sufficient because Algorithm 1 determinizes states on-the-fly (see line 8). By not determinizing $\mathcal{M}_{backward}$ ahead-of-time, we ensure that both the forward and the backward analyis act on the same state sets; just the transition function differs. This, in turn, makes it easier to determine nop shadows.

4.2 Actual worklist algorithm

Algorithm 1 first initializes its worklist wl with the set of initial jobs, as explained above. The algorithm further initializes two mappings *before* and *after* that store the configurations that have been computed so far before, respectively after each statement. These sets allow us to perform a terminating fixed point iteration.

Next, the algorithm iterates through its worklist. For every job ($stmt$, cs), the algorithm first updates $stmt$'s before-set. Then, when a statement holds no shadow, we just leave the configurations unchanged (line 6 in Algorithm 1). Otherwise, we compute (line 7), for every new configuration $c \in cs$ and *shadow* at *stmt*, successor configurations using the supplied transition function δ. (In Clara, programmers describe events through AspectJ [3] pointcuts. Although seldom the case, pointcuts can overlap, thereby causing one single statement to be associated with multiple shadows.) To compute the transition, the algorithm accesses the shadows's unique label $label(shadow)$. (In our running example, this label could be "close", "reconnect" or "write".) To allow the analysis to later-on compare state labels of $\mathcal{M}_{forward}$ with the state-set labels that $\mathcal{M}_{backward}$ uses, we determinize state machines on-the-fly: line 8 computes the unique set of successor states.

The algorithm then updates $stmt$'s after-set and associates new jobs with two different kinds of successor statements. First, in lines 14–16, the algorithm adds new jobs containing the successor configurations cs_{new} for any statement that is a successor of $stmt$ in m's control-flow graph, as determined by $succ_{cfg}$. Lines 17–19 use the the external-successor function $succ_{ext}$ to handle inter-procedural control flow.

When propagating configurations along a $succ_{ext}$-edge, it is not correct to just copy the configurations from the edge's source to it's target. Note that between any two executions of m, other methods may execute and cause transitions in the monitoring state machine. To soundly model these potential transitions by "other methods", Algorithm 1 associates in line 18 with any inter-procedural successor not just the set of new configurations cs_{new} but instead the set of configurations $reaching(cs_{new}, relatedShadows(stmt))$. We defined the function $reaching$ already above. The function $relatedShadows(stmt)$ computes the set of all shadows related to $stmt$. We define this set as follows. For any invoke statement $stmt$ (potentially recursive or not), the set $relatedShadows(stmt)$ contains all shadows in all methods transitively reachable through $stmt$, except for the ones in m itself. After all, these are all the shadows that one can reach before reaching m's entry statement again. Otherwise, i.e., if $stmt$ is a tail of m, then $relatedShadows(stmt)$ contains all shadows in the program, except for the ones in m. (Our implementation further narrows down related shadows by comparing each shadow's variable binding to the binding stored in the configuration cs_{new}.)

It is worthwhile noting that, because we compute the expression $reaching(cs_{new}, relatedShadows(stmt))$ for each statement separately, we gain a certain amount of context-sensitivity. While shadows inside a certain method m' (with $m' \neq m$) may be relevant to one statement of m they may be irrelevant to other statements in m, and by recomputing the above function we properly distinguish such cases.

4.3 Removing nop shadows

In addition to the Nop-shadows Analysis, Clara also contains implementation of a syntactic Quick Check and a flow-insensitive, pointer-based Orphan-shadows Analysis [7, 9]. Because these analyses do not take control-flow into account,

[3]Usually, $heads(m)$ will be a singleton set but because our backwards analysis operates on a reversed control-flow graph where heads become tails and tails become heads, $heads(m)$ can contain more than one element in this setting.

they take only milliseconds once points-to information has been computed. Therefore, before executing the more expensive Nop-shadows Analysis, we execute the Quick Check and the Orphan-shadows Analysis first. Then, if shadows remain enabled, we compute the Nop-shadows Analysis for every shadow-bearing method. The analysis information directly provides us with *source*, *target* and *futures* for every statement. We use this information to identify and disable a nop shadow if possible, and then re-iterate until we can find no further nop-shadows for this method. In our benchmark set we had to iterate ten times or less for all but four methods. When we reach the fixed point, we proceed with the next method. When all methods are processed, we apply the flow-insensitive Orphan-shadows Analysis again and then re-iterate the whole Nop-shadows Analysis. This is because disabling a shadow in one method may render shadows in other methods irrelevant. It seems to be always sufficient to iterate this outer loop two to three times. When this loop reaches a fixed point we stop.

5. EXPERIMENTS

To validate our approach, we verified a set of twelve typestate properties over ten benchmark programs of the DaCapo benchmark suite [6]. This lead to 120 property/benchmark combinations. 43 of these combinations (36%) were "interesting" to us in the sense that shadows remained after applying the first two, previously published, analysis stages. The 43 combinations comprised eight out of the original twelve properties. Table 1 explains these properties. We applied the Nop-shadows Analysis to these 43 combinations.

First, we were interested in comparing the Nop-shadows Analysis to a traditional typestate analysis that only has a forward component. As we will explain in Section 6, one cannot use the latter to identify nop shadows and optimize runtime monitors, but of course one can use both analyses to determine potentially property-violating program points, i.e., program points at which the abstraction reaches a final state. We therefore modified our analysis to run the forward pass only, recording all potentially property-violating program points. Then we ran the Nop-shadows Analysis as we described it earlier, with all re-iterations, disabling nop shadows as they are identified, and recording as well all potentially property-violating program points. Interestingly, we discovered that, for our benchmark set, the program points completely coincided. Hence, we can state that the backwards pass and the reiteration do not help with identifying these program points. (However, they are crucial for identifying nop shadows on the preceding execution.)

Next we were interested in seeing what fraction of shadows the Nop-shadows Analysis manages to identify as nop shadows. Table 2 summarizes our analysis results. The fraction of shadows that our analysis identified as nop shadows appears in white. In gray we show the fraction of shadows which are known to trigger actual violations at runtime. The remaining black slice represent shadows that remain active even after analysis, either due to analysis imprecision or due to actual property violations.

For 18 out of these 43 combinations (41%), our novel Nop-shadows Analysis was able to identify all shadows as irrelevant and therefore proved that the program cannot violate the stated property. These cases appear as all white circles. In four other cases, shadows remained enabled, but only because they do trigger a property violation. These cases contain gray but no black fragments. In other words, the analysis gave exactly the correct result, with no false positives, in half of the cases. In three cases, the analysis failed to identify any nop shadow (black circles).

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 nextElement on an Enumeration
HasNext	always call hasNext before calling next on an Iterator
Reader	do not use a Reader after its InputStream was closed
Writer	do not use a Writer after its OutputStream was closed

Table 1: Relevant typestate properties and their names

In the remaining 18 cases, the analysis removed a sometimes significant amount of shadows. This may speed up runtime monitoring for these cases, depending on whether the test run exercises these shadows a lot. For our experiments we used monitoring aspects generated from tracematches [2]. Table 2 gives qualitative information about the residual monitor's runtime overhead through the ring that surround each circle. (The dissertation [7] gives the full data.) Interestingly, while there is some correlation between the number of remaining shadows and the runtime overhead that these shadows cause, the correlation is not one-to-one. For instance, in jython-FailSafeIter, there remain 112 shadows, opposed to just 16 in the case of lusearch-FailSafeIter. Nevertheless, lusearch does show a perceivable runtime overhead but jython does not. We conclude that the positive effect that the removal of a shadow s may have on the monitoring overhead depends to some extend on how often the program under test would execute s at runtime.

Our analysis works well on the antlr, fop, hsqldb, luindex, lusearch and xalan benchmarks. Most of the potential false positives (black in the figure) appear only because the benchmarks use reflection. Due to a known deficiency [1], Java's `Cloneable` interface contains no public declaration of a `clone()` method. Therefore, Java's type system may prevent clients from calling `clone()` even on `Cloneable` objects. chart uses reflection to call the `clone()` method on objects that implement the `Cloneable` interface. Because chart clones collections, our points-to analysis has to safely assume that the collections could be of any type, including `EmptySet`, which, as a singleton object, is stored in a static field, causing our analysis to loose all context information. bloat, jython and pmd cause similar problems.

There appear to be few cases where our analysis is too imprecise because of its design. For example, two actually irrelevant final shadows remain enabled in hsqldb with Reader and Writer. These false positives occur because xalan uses different methods to open, close and write to streams. A fully inter-procedural analysis could rule out possible violations in these cases. However, we found that, due to its intraprocedural nature, the Nop-shadows Analysis has an interesting property: the analysis revealed missing pre-conditions on xalan's methods. For instance, the write-calling method is missing the pre-condition that the argument file should

	antlr	bloat	chart	fop	hsqldb	jython	luindex	lusearch	pmd	xalan
FailSafeEnum	0/3			6/7		44/47		0/5	0/10	
FailSafeEnumHT	26/30				3/3	61/76	0/15	0/5		
FailSafeIter		830/922	149/160			112/116	0/27	16/36	287/302	
FailSafeIterMap		444/446	49/49	OOME		133/151			204/314	
HasNextElem	0/86			0/8	0/6	34/47	0/16	0/6	0/6	1/3
HasNext		452/565	48/82	0/8		24/31	0/12	0/22	184/250	
Reader	0/14				3/3	4/4			0/24	
Writer	35/44	15/19			10/10				0/7	

Table 2: Shadows classified by our analysis. White slices: shadows that the Nop-shadows Analysis identified as irrelevant. Black slices: shadows that we fail to identify as irrelevant, due to analysis imprecision or an actual violation. Gray slices: actual property violations that we found through manual inspection. The outer rings represent the residual monitor's runtime overhead. Solid: overhead ≥ 15%, dashed: overhead < 15%, dotted: no overhead. OOME=OutOfMemoryError

not be in state "closed". In future work we plan to use this information to support program understanding and to further enhance precision.

Detected property violations. When manually inspecting the remaining shadows, we found several actual property violations. bloat violates Writer because it contains a method that writes to a file handle that it then closes. When called multiple times, this method will violate the property. We could not confirm whether certain runs of bloat may actually call this method multiple times. jython sometimes violates Reader by closing a stream prematurely. jython then catches the resulting exception and returns null. xalan violates HasNextElem by calling `nextElement` without a preceding call to `hasMoreElements`. Nevertheless, the program is safe because it checks the size of the underlying vector to assure that the calls are legal. The property specification is too simplistic in this setting. Several benchmarks violate FailSafeEnum and FailSafeEnumHT. These benchmarks do indeed modify vectors (or hash tables) while iterating over them. This can lead to unexpected behavior. With iterators this does not usually happen because iterators have fast-fail semantics and will throw a `ConcurrentModificationException` in such situations. luindex violates the FailSafeIter pattern. This error probably remained undetected because it only occurs on quite unlikely execution paths. The author's dissertation [7] provides more details.

Analysis time. Our analysis time is clearly dominated by the time it takes to compute the supporting analyses that the Nop-shadows Analysis requires. Constructing a call graph and context-sensitive points-to sets took about two and a half minutes on average. The Nop-shadows Analysis itself took under 50 seconds on average. This time includes all re-iterations of the Orphan-shadows Analysis and Nop-shadows Analysis that Clara performs. In 90% of the cases, the analysis finished in under one minute. By far the worst case was bloat-FailSafeIter, for which this analysis stage took 19 minutes. bloat is notoriously hard to analyze [8, 9, 20].

Limitations and threats to validity.

We identified the following limitations of our approach. All DaCapo 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 Soot [23] (which is part of Clara) with this information. The resulting call graph is sound for the program runs that DaCapo performs. 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. Soot 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 analyze this code and so made the unsafe assumption that this code would not cause any typestate changes.

Otherwise, the internal validity of our experiments is high because we directly measure the number of final shadows before and after the analysis. The final shadows are exactly the points that programmers would first inspect when checking possible property violations. Hence, reducing the number of final shadows will reduce the burden on the programmer. This is especially true when eliminating all final shadows, thus proving that the program cannot violate the property.

To measure the runtime overheads precisely, we extended the DaCapo harness with a custom driver class. With this driver class, DaCapo first executes a warm-up run and then re-runs the benchmark multiple times 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 arithmetic mean of these runs. DaCapo's standard driver only measures a single benchmark run, which has caused misleading results for us in the past.

The external validity is limited by our choice of benchmarks. However, the DaCapo benchmarks are a realistic, representative set of medium-sized to large-scale applications. The suite contains both "well-behaved" benchmarks that are free of reflection and benchmarks that are harder to analyze due to reflection. Our analysis excels on the former, however, further work is required to handle the latter more effectively. We plan to address these problems by simulating reflection in a more fine-grained manner.

6. RELATED WORK

Strom and Yemini [22] 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.

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 a 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.

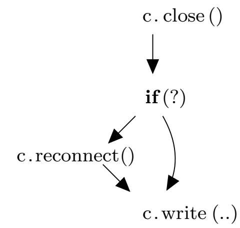
DeLine and Fähndrich's approach [14] is similar in flavor to Bierhoff and Aldrich's but uses a more restrictive abstraction of aliases that allows for less flexible calling conventions for typestate-changing methods. The authors implemented their approach in the Fugue tool for specifying and checking 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.

Fink et al. present a static analysis of typestate properties [17]. 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, i.e., they can be used for a purely static analysis only.

```
1 void foo(Connection c) {
2     c.close();
3     if(?) {
4         c.reconnect();
5     }
6     c.write(..);
7 }
```

(a)



(b)

Figure 6: Example exposing unsoundness in earlier hybrid typestate analyses

Hybrid analysis approaches.

In own previous work [10] we presented a hybrid typestate analysis that was, like the Nop-shadows Analysis, also flow-sensitive on an intra-procedural level only, and used a flow-insensitive abstraction of the remainder of the program. However, unlike the Nop-shadows Analysis, the earlier analysis used a forward-analysis only. A forward analysis can only approximate the possible *source* and *target* states of a statement s but not the *futures*.

Consider the property-violating program in Figure 6a. Remember that a correct runtime monitor must not only observe events at property-violating shadows like the "close" shadow at line 2 and the "write" shadow at line 6, but also at shadows that may prevent a violation, like the "reconnect" shadow at line 4. A design flaw caused our earlier forward analysis to mistakenly disable certain violation-preventing shadows like the reconnect shadow in this example. To determine relevant shadows, this earlier analysis used what we called a "shadow history". The shadow history at a statement s is the set of all shadows on the control flow that reaches s. When determining that the program may reach an error state at s, the analysis would then commit the shadow history at s to a global set of "relevant shadows" that need to be monitored at runtime.

In Figure 6b we show the control-flow graph for the example program. As the graph shows, due to the if-statement, the analysis will reach the write statement along two different branches. Along the left branch, the analysis determines that the connection is in its "connected" state when being written to, and therefore no shadow history is committed, as no violation can occur. Along the right branch, the analysis determines that the connection will be in state "closed" when reaching the write at line 6. Hence, the analysis will commit the shadow history. However, the shadow history along this branch contains the disconnect and write statements only, because the reconnect occurs on the other branch! The residual runtime monitor for this property will therefore miss any possible reconnect events, and may therefore signal false runtime warnings. We proved that our novel Nop-shadows Analysis causes neither false warnings nor missed violations.

Naeem and Lhoták present a fully context-sensitive and flow-sensitive inter-procedural whole-program analysis for typestate-like properties of multiple interacting objects [20]. This analysis can be seen as a generalized version of our own earlier analysis [10]. 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 intra-procedural only, there are some instances where our analysis is more precise than Naeem and Lhoták's. This is due to the highly context-sensitive points-to sets that we compute. It is important to note that Naeem and Lhoták also use shadow histories to determine relevant instrumentation points. Unfortunately, their analysis therefore suffers from the same unsoundness problem that we described above. Naeem and Lhoták had proven their entire analysis sound except for the use of shadow histories [19].

Dwyer and Purandare use existing typestate analyses to specialize runtime monitors [16]. 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 objects.

7. CONCLUSION

We have presented a precise flow-sensitive and partially context-sensitive typestate analysis that can handle typestate specifications that refer to multiple interacting objects, and which generates residual runtime monitors. Using an additional backwards pass, the analysis computes classes of equivalent states and disables transitions between such states. This two-pass approach allows for precise local reasoning directly at relevant program points. Although the analysis is lightweight and efficient, it is also precise, exactly telling apart property-violating program locations from irrelevant locations in more than half of the cases.

Acknowledgements. We thank Patrick Lam for inspiring this research and for many fruitful discussions. Andreas Sewe helped to improve the quality of a draft of this paper.

8. REFERENCES

[1] Bug-database entry regarding "Cloneable". `http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4098033`.

[2] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding Trace Matching with Free Variables to AspectJ. In *OOPSLA*, pages 345–364, October 2005.

[3] AspectJ. `http://eclipse.org/aspectj/`.

[4] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: An extensible AspectJ compiler. In *AOSD*, pages 87–98, March 2005.

[5] Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *OOPSLA*, pages 301–320, October 2007.

[6] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, pages 169–190, October 2006.

[7] Eric Bodden. *Verifying finite-state properties of large-scale programs*. PhD thesis, McGill University, June 2009.

[8] Eric Bodden, Feng Chen, and Grigore Roşu. Dependent advice: A general approach to optimizing history-based aspects. In *AOSD*, pages 3–14, March 2009.

[9] Eric Bodden, Laurie J. Hendren, and Ondřej Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *ECOOP*, volume 4609 of *LNCS*, pages 525–549, 2007.

[10] Eric Bodden, Patrick Lam, and Laurie Hendren. Finding Programming Errors Earlier by Evaluating Runtime Monitors Ahead-of-Time. In *FSE*, pages 36–47, November 2008.

[11] Eric Bodden, Patrick Lam, and Laurie Hendren. Object representatives: a uniform abstraction for pointer information. In *Visions of Computer Science - BCS International Academic Conference*. British Computing Society, September 2008.

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

[13] Feng Chen and Grigore Roşu. MOP: an efficient and generic runtime verification framework. In *OOPSLA*, pages 569–588, October 2007.

[14] Robert DeLine and Manuel Fähndrich. Typestates for objects. In *ECOOP*, volume 3086 of *LNCS*, pages 465–490, June 2004.

[15] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE*, pages 411–420, May 1999.

[16] Matthew B. Dwyer and Rahul Purandare. Residual dynamic typestate analysis: Exploiting static analysis results to reformulate and reduce the cost of dynamic analysis. In *ASE*, pages 124–133, May 2007.

[17] Stephen Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanual Geay. Effective typestate verification in the presence of aliasing. In *ISSTA*, pages 133–144, July 2006.

[18] Hidehiko Masuhara, Gregor Kiczales, and Chris Dutchyn. A compilation and optimization model for aspect-oriented programs. In *CC*, volume 2622 of *LNCS*, pages 46–60, April 2003.

[19] Nomair A. Naeem and Ondřej Lhoták. Extending typestate analysis to multiple interacting objects. Technical report, University of Waterloo, 04 2008. CS-2008-04.

[20] Nomair A. Naeem and Ondřej Lhoták. Typestate-like analysis of multiple interacting objects. In *OOPSLA*, pages 347–366, October 2008.

[21] Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for Java. In *PLDI*, pages 387–400, June 2006.

[22] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *TSE*, 12(1):157–171, January 1986.

[23] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java bytecode optimization framework. In *CASCON*, page 13. IBM Press, 1999.