Specifying and Exploiting Advice-Execution Ordering using Dependency State Machines[*]

Eric Bodden
Software Technology Group
Technische Universität Darmstadt, Germany
bodden@acm.org

ABSTRACT

In this paper we present Dependency State Machines, an annotation language that extends AspectJ with finite-state machines that define the order in which pieces of advice must execute to have a visible effect. Dependency State Machines facilitate the automatic verification and optimization of aspects, but also program understanding.

In this work we present the syntax and semantics of Dependency State Machines and one possible use case of Dependency State Machines: program understanding. We explain how a set of three static program analyses can exploit the information that Dependency State Machines carry to remove advice-dispatch code from program locations at which dispatching the advice would have no effect. Dependency State Machines hereby help to abstract from the concrete implementation of the aspect, making the approach compatible with a wide range of aspect-generating monitoring tools.

Our extensive evaluation using the DaCapo benchmark suite shows that our approach can pinpoint to the user exactly the program locations at which the aspect's execution matters in many cases. This is particularly useful when the aspect's purpose is to identify erroneous execution sequences: in these cases, the program locations that our analysis pinpoints resemble possible points of program failure.

Categories and Subject Descriptors

D.3.4 [**Programming Lang.**]: Processors—*Optimization*

General Terms

Experimentation, Languages, Performance

Keywords

Domain-specific aspect languages, compilation and static program analysis, runtime verification

[*]The author conducted this work as a PhD student at McGill University, under supervision of Laurie Hendren.

1. INTRODUCTION

Pieces of advice are often inter-dependent in the sense that the execution of one piece of advice will only have an effect before or after the execution of another. This is especially true when the aspect that declares these pieces of advice expresses a state-based runtime monitor. For instance, consider the example aspect in Figure 1, which issues an error message when writing to a disconnected connection. The pieces of advice in this aspect (lines 4–18) monitor `disconnect`, `reconnect` and `write` events on a connection object. The aspect issues an error message when a connection is disconnected and then written to without an intervening reconnect. Figure 2 shows the monitor that this aspect implements in the form of a finite-state machine that issues the error message when reaching its accepting state. It is important to realize that the three pieces of advice in this aspect are inter-dependent: the effect of executing

```
 1 aspect ConnectionClosed {
 2     Set closed = new WeakIdentityHashSet();
 3
 4     dependent after disconnect(Connection c) returning:
 5         call(* Connection.disconnect()) && target(c) {
 6         closed.add(c);
 7     }
 8
 9     dependent after reconnect(Connection c) returning:
10         call(* Connection.reconnect()) && target(c) {
11         closed.remove(c);
12     }
13
14     dependent after write(Connection c) returning:
15         call(* Connection.write(..)) && target(c) {
16         if(closed.contains(c))
17             error("May not write to "+c+", as it is closed!");
18     }
19
20
21     dependency{
22         disconnect, write, reconnect;
23          initial  connected: disconnect -> connected,
24                          write -> connected,
25                          reconnect -> connected,
26                          disconnect -> disconnected;
27                  disconnected: disconnect -> disconnected,
28                          write -> error;
29         final     error: write -> error;
30     }
31 }
```

Figure 1: Monitoring aspect "ConnectionClosed", annotated with Dependency State Machine

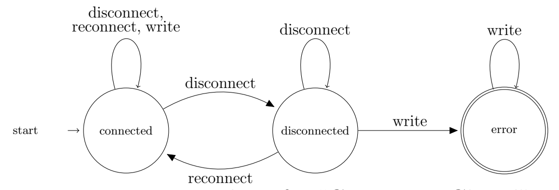


Figure 2: Finite-state machine for "ConnectionClosed" expl.

one piece of advice depends on whether or not other pieces of advice executed already. For instance, the `write` advice will issue an error message for a connection `c` if and only if `disconnect` executed on `c` already, and `reconnect` was not executed in between. Further, line 17 is the only line that has an effect that is visible outside the aspect.

When weaving such aspects into a program with modern aspect compilers like ajc [1] and abc [4], the compilers will report to the user *all* the joinpoint shadows at which the individual pointcuts that the pieces of advice in this aspect refer to could potentially match a joinpoint at runtime. As our experiments show, the sheer number of shadows that such compilers report makes it very hard for programmers to reason about the effect of these aspects. In the ConnectionClosed example, a programmer would have a hard job trying to determine through manual inspection whether or not the program can violate the ConnectionClosed property.

Fortunately, as we show in this work, many of these shadows are "irrelevant" in the sense that, when an irrelevant shadow matches a joinpoint at runtime, then the dispatch of the piece of advice that induced this shadow will never have any effect at this program point. For instance, in the ConnectionClosed example, assume a `write` shadow at a program point at which it is known that the connection that the shadow refers to must be in state "connected". As the state machine in Figure 2 and the aspect code show, the `write` shadow will have no effect in this state. In the state machine, the `write` transition loops, in the aspect code, executing the `write` advice will do nothing because the `if`-check that the body contains must evaluate to `false`.

In this work, we present Dependency State Machines, an annotation language that extends AspectJ and which makes such inter-advice dependencies explicit. In particular, a Dependency State Machine describes the order in which pieces of advice have to execute so that the execution of these pieces of advice, in combination, has an effect that is visible outside the aspect itself. Lines 21–30 in Figure 1 show the appropriate Dependency-State-Machine annotation for the ConnectionClosed example. As the reader can see, we deliberately kept the syntax simple: the annotation directly encodes the appropriate finite-state-machine representation (Figure 2) in a textual format. Line 22 enumerates the alphabet which this state machine is defined over. Every symbol name in this line refers to a named "dependent" piece of advice in the same aspect. Lines 23–29 enumerate all states, along with their outgoing transitions.

If a programmer knows the transition structure of their monitoring aspect, then the programmer can write these annotations by hand. However, many programmers use runtime-monitoring tools to generate such aspects automatically from formal property specifications. In this case, the formal specifications often contain enough information already such that the aspect-generating monitoring tool can automatically generate the appropriate dependency annotations, too. Many monitoring tools use finite-state machines as an internal monitor representation [3, 13, 7, 21], which makes generating the annotations even easier. In the future, researchers could also develop tools that generate Dependency State Machines directly from aspects. However, note that this would involve analyzing Turing-complete aspect code. Therefore, such approaches would always only be able to generate Dependency State Machines for a subset of well-structured aspects.

Once an aspect has been enriched with dependency annotations, tools can exploit the annotations for different purposes. We believe that Dependency State Machines potentially enable a wide range of static analyses and optimizations. In this work, however, we focus on using Dependency State Machines to improve program understanding. We present a flow-sensitive static whole-program analysis, called "Nop-shadows Analysis", that can tell apart irrelevant shadows ("nop shadows") from relevant shadows in many cases. The Nop-shadows Analysis builds on two flow-insensitive analyses that we published previously. These earlier analyses had no information about the order in which pieces of advice must execute, which makes them less precise. In this paper, we show that the Nop-shadows Analysis significantly improves over the results of the earlier analyses and that the combination of all three analyses can significantly reduce the number of shadows that a programmer has to consider when attempting to reason about the aspect's effect.

We validated our approach by applying all three analyses to the 120 combinations of twelve AspectJ aspects (annotated with Dependency State Machines) with ten benchmark programs of the DaCapo Benchmark Suite [6]. As our results show, our analysis successfully identifies a large fraction of shadows as irrelevant. In combination with our two previously published analyses, our novel analysis successfully identified 36020 of 39194 shadows in our benchmark set as irrelevant, i.e., a fraction of 92%. In other words, after applying our analyses, on average, a user would only have to consider about 8% of all shadows to determine where the aspect may have a visible effect. For more than half of the combinations, our analyses were able to show that the aspect has no effect at all on the program's execution. Because our aspects detect erroneous executions, we expect them to have no effect for correct programs. To summarize, this paper presents the following original contributions:

- The syntax and semantics of Dependency State Machines, a novel AspectJ language extension that encodes the order in which pieces of advice must execute to have a visible effect.
- The idea of using Dependency State Machines to improve program understanding by identifying and eliminating irrelevant joinpoint shadows, and a static program analysis that implements these concepts.
- A set of experiments that show that this program analysis can significantly reduce the number of joinpoint shadows that programmers need to consider when trying to reason about their aspects' effects.

Section 2 explains the syntax of Dependency State Machines. In Section 3 we explain our semantics of Dependency State Machines. Section 4 outlines three static analyses that exploit these semantics to identify "irrelevant" joinpoint shadows. We present benchmark results in Section 5, discuss related work in Section 6 and conclude in Section 7.

2. SYNTAX OF DEPENDENCY STATE MACHINES

Figure 1 already demonstrated our language extension using the ConnectionClosed example. Line 22 establishes the alphabet that the state machine is evaluated over. Every symbol in the alphabet refers to a named "dependent" piece of advice in the same aspect. In our language extension, only pieces of advice that are declared as "dependent" can have names. Other pieces of advice have no names and execute with AspectJ's standard semantics. Lines 23–29 enumerate all states in the state machine in question, and for each state enumerate further a (potentially empty) list of outgoing transitions. An entry "`s1: l -> s2`" reads as "there exists an `l`-transition from `s1` to `s2`". In addition, a programmer can mark states as `initial` or `final`, i.e., accepting. We give the complete syntax for Dependency State Machines in Figure 3, as a syntactic extension to AspectJ.

According to the semantics that we will give to Dependency State Machines, the dependency declaration in the ConnectionClosed example states that any piece of `disconnect`, `write` or `reconnect` advice must execute on a connection `c` whenever not executing this piece of advice on `c` would change the set of joinpoints at which the Dependency State Machine reaches its final state on `c`. (More on the semantics later.) Note, however, that the advice references in line 22 omit the variable name `c` of the connection: we just wrote `disconnect, write, reconnect`. We can do so because, by default, a dependency annotation infers variable names from the formal parameters of the advice declarations that it references (lines 4, 9 and 14 in the example). This means that the alphabet declaration in line 22 is actually a short hand for the more verbose form `disconnect(c), write(c), reconnect(c)`.

The semantics of variables in dependency declarations is similar to unification semantics in logic programming languages like Prolog [14]: The same variable at multiple locations in the same dependency refers to the same object. For each advice name, the dependency infers variable names in the order in which the parameters for this advice are given at the site of the advice declaration. Variables for return values from `after returning` and `after throwing` advice are appended to the end. For instance, the following advice declaration would yield the advice reference createIter (c, i).

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

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

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

This advice only has an effect when `n` and `m` both refer to the same object. However, due to the semantics of AspectJ, the advice cannot use the same name for both parameters—the inferred annotation would be detectLoops(n,m). The verbose syntax for dependent advice allows us to state nevertheless that for the advice to have an effect, both parameters actually have to refer to the same object, say `k`: **dependency**{detectLoops(k,k); ... }.

2.1 Type-checking Dependency State Machines

After parsing, we impose the following semantic checks:

- A piece of advice carries a name if and only if it carries also a `dependent` modifier.
- Every advice must be referenced only by a single declaration of a Dependency State Machine.
- The state machine must have at least one initial and at least one final state.
- The listed alphabet may contain every advice name only once, i.e., declares a set.
- The names of states must be unique within the dependency declaration.
- Transitions may only refer to the names of advice that are named in the alphabet of the dependency declaration, and to the names of states that are also declared in the same dependency declaration.
- Every state must be reachable from an initial state.
- If the verbose form for advice references is used:
    - The number of variables for an advice name equals the number of parameters of the unique advice with that name, including the after-returning or after-throwing variable. (inference ensures this)
    - Advice parameters that are assigned equal names have compatible types: For two advice declarations `a(A x)` and `b(B y)`, with `a(p)` and `b(p)` in the same dependency declaration, `A` is cast-convertible [18, §5.5] to `B` and vice versa.
    - Each variable should be mentioned at least twice inside a dependency declaration. If a variable v is only mentioned once we give a warning because in this case the declaration states *no* dependency with respect to v. The warning suggests to use the wildcard "`*`" instead. Semantically, `*` also generates a fresh variable name. However, by stating `*` instead of a variable name, the programmer acknowledges explicitly that the parameter at this position should be ignored when resolving dependencies.

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

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

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

AdviceName ::= ID

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

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

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

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

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

***VarName* ::=** ID | **"*"**

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

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

***StateModifier* ::=** "initial" | "final"

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

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

Figure 3: Syntax of Dependency State Machines, as extension (shown in boldface) to the syntax of AspectJ

3. SEMANTICS OF DEPENDENCY STATE MACHINES

We define the semantics of a Dependency State Machine as an extension to the usual advice-matching semantics of AspectJ [19]. 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. Consistent with our previous work on Dependent Advice [9], we model advice matching in AspectJ as a function *match* that we regard as given by the underlying AspectJ compiler:

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

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

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

3.1 Semantics by example

Figure 4 contains a small example program that we use to explain the intuition behind this semantics. The example program violates the ConnectionClosed property in lines 5 and 7 by first disconnecting the connection o(`c1`) and then writing to o(`c1`). (For any variable `v`, we use o(`v`) to refer to the object that `v` references.) The joinpoint shadows [23] at these two lines are also the only two shadows in the program that the ConnectionClosed monitoring aspect from Figure 1 must monitor so that this aspect correctly issues its error message at runtime. In particular, since the monitor starts off in its initial state "connected", the `write` event at line 4 has no impact on the monitor's state: the monitor loops on state "connected", and hence we call the `write` shadow at this line "irrelevant". Similarly, at line 6, the monitor is guaranteed to be in the "closed" state. Monitoring further

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

Figure 4: Example program

`disconnect` events does not change the automaton state in this situation either. Hence, the `disconnect` shadows at this line is irrelevant as well. The `disconnect` event at line 8 does cause a state change (from "connected" to "closed"), but this state change does not matter: because no `write` event ever follows on o(`c1`), this state change cannot impact the set of future joinpoints at which the Dependency State Machine reaches its final state (because there are none), and hence cannot impact the set of joinpoints at which the runtime monitor will have a visible effect, i.e., will issue its error message. This is true even though another `write` event follows at line 9. This latter `write` event occurs on `c2` and not on `c1`. Because we know that `c2` cannot possibly reference the same object as `c1`, i.e., o(`c1`) $\neq$ o(`c2`), this `write` event is not what we call "compatible" with the `disconnect` event at line 8.

3.2 Formal semantics

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

Further, we will define the semantics of Dependency State Machines in terms of "events", not joinpoints. Joinpoints differ from events in that joinpoints describe regions in time while events describe atomic points. A joinpoint has a beginning and an end, and code can execute before or after the joinpoint (i.e., at its beginning or end) or instead of the joinpoint. In particular, joinpoints can be nested. For instance, a field-modification joinpoint can be nested in a method-execution joinpoint. Pieces of advice, even "around advice", execute at atomic events before or after a joinpoint. Because these events are atomic, they cannot be nested. Joinpoints merely induce these events[1].

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

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

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

For any declaration of a Dependency State Machine, the set of dependent-advice names mentioned in the declaration of the Dependency State Machine induces an alphabet Σ, where every element of Σ is the name of one of these dependent pieces of advice. For instance, the alphabet for the ConnectionClosed dependency state machine from Figure 1 would be $\Sigma = \{\text{disconnect}, \text{write}, \text{reconnect}\}$. Matching these pieces of advice against a runtime event e results in a (possibly empty) set of matches for this event, where each match has a binding attached. We call this set of matches the parameterized event $\hat{e}$.

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

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

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

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

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

It is necessary to consider sets of matches because multiple pieces of advice can match the same event. While this is not usually the case, we decided to cater for the unusual cases, too. As an example, consider the Dependency State Machine in the UnusualMonitor aspect in Figure 5a. The aspect defines a dependency between two pieces of advice `a` and `b`. Note that the pointcut definitions of `a` and `b` overlap, i.e. describe non-disjoint sets of program events. The advice

[1]Our notion of events is essentially the same as the notion of joinpoints in the point-in-time joinpoint model that Masuhara, Endoh and Yonezawa proposed earlier [22].

```
aspect UnusualMonitor {
    dependency{
        a, b;
        //transitions omitted from example
    }

    dependent before a(Object x):
      call(* *(..)) && target(x) { ... }

    dependent before b(Object x):
      call(* foo (..)) && target(x) { ... }
}
```

(a) UnusualMonitor aspect with overlapping pointcuts

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

(b) Example program

Figure 5: UnusualMonitor aspect and example program

`b` executes before all non-static calls to methods named `foo`. The advice `a` executes before these events too, because, by its definition, it executes before any non-static method call.

Next, assume that we apply this aspect to the little example program in Figure 5b. We show the program's execution trace in the first row of Figure 6 (to be read from left to right). This execution trace naturally induces the parameterized event trace that we show in the second row of the figure: this trace is obtained by matching at any event every piece of advice against this event.

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

Projected event. For every $\hat{e} \in \hat{\mathcal{E}}$ and binding β we define a projection of $\hat{e}$ with respect to β:

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

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

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

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

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

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

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

For our example, the third and fourth row of Figure 6 show the four ground traces that result when projecting this parameterized event trace onto the variable bindings $x = o(\texttt{v1})$ and $x = o(\texttt{v2})$. For $x = o(\texttt{v1})$ we obtain the two traces "aa" and "ba", for $x = o(\texttt{v2})$ we obtain the two traces "a" and "b".

A Dependency State Machine will reach its final state (and the related aspect will have an observable effect, e.g., will is-

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

Figure 6: Traces resulting from code in Figure 5; note that $o(\texttt{v1}) \neq o(\texttt{v2})$

sue 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 *groundTraces*$(\hat{t})$ of non-empty ground traces of $\hat{t}$ as:

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

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

The semantics of a Dependency State Machine

We define the semantics of Dependency State Machines as a specialization of the predicate *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, under which variable binding. As noted earlier, this predicate *match* is given through the semantics of plain AspectJ. We call our specialization *stateMatch* and define it as follows:

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

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

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

Soundness condition.

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

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

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

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

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

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

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

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

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

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

4. IDENTIFYING RELEVANT JOINPOINT SHADOWS

In this section, we outline how we use Dependency State Machines to identify "relevant joinpoint shadows", i.e., shadows that may cause the aspect to have a visible effect at runtime. We first explain how our novel flow-sensitive analysis, the Nop-shadows Analysis, identifies "nop shadows": shadows that have no such effect. The "relevant" shadows are then all shadows that the analysis does not classify as "nop shadows". In Section 4.2 we then describe how the Nop-shadows Analysis improves over two analyses that we published previously, and we explain the added benefit of Dependency State Machines over our earlier approach. Section 4.3 gives the most important implementation details.

4.1 Nop-shadows Analysis

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

Given this equivalence relation, we can then identify shadows s that only switch between "equivalent" states on all possible executions that lead through s. By definition of our semantics of Dependency State Machines we know that dispatching a piece of advice a at such a shadow s would have no effect. We exploit this fact in two different ways. Firstly, we filter the shadow from the list of shadows that is displayed to the user after weaving. This aids the programmer in reasoning about the effects that the aspect may have. Secondly, we remove all advice-dispatch code from this shadow, potentially speeding up the execution of the woven program.

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

Computing the appropriate equivalence relation requires both a forward and a backward-analysis component: the forward component computes equivalencies between states "with respect to the past", while the backward analysis computes equivalencies "with respect to the future", i.e., with respect to the possible continuations. The forward-analysis component works by propagating through the program the states of a determinized version of the original Dependency State Machine $\mathcal{M}$. The backward-analysis component is an exact dual of the forward one: it propagates backwards through the program the states of a determinized version of the inverted state machine of $\mathcal{M}$. To obtain an efficient implementation, our analysis uses flow-sensitive information on an intra-procedural, i.e., per-method level only, and uses a coarse grain flow-insensitive abstraction at method boundaries. Space limitations prevents us from explaining the Nop-shadows Analysis any further. The author's dissertation [8, Section 5.2] explains the analysis in all detail.

4.2 Dependent Advice and previously published analysis stages

In previous work [9], we proposed "Dependent Advice", an AspectJ language extension that, similar to Dependency State Machines, expresses inter-dependencies between pieces of advice. Although both approaches share some ideas, Dependency State Machines improve over Dependent Advice in several ways. The most important improvement is that Dependency State Machines, unlike Dependent Advice, encode the order in which pieces of advice are meant to execute. The Nop-shadows Analysis from above makes heavy use of this information by propagating the state of Dependency State Machines through the program according to their transition tables, which expresses the execution order.

Our earlier approach, Dependent Advice, encoded no such information: a correct Dependent-Advice declaration for our ConnectionClosed example property would be the following.

dependency{ **strong** disconnect, write; **weak** reconnect; }

This declaration follows the syntax that we proposed in earlier work. The declaration states that `disconnect` and `write` share a "strong" dependency. This means that `disconnect` only needs to execute (on a connection `c`) if there is a chance of `write` executing (on `c`) as well, *and* the other way around. The additional "weak" reference to reconnect states that, if the strong dependency is fulfilled, i.e., if both `disconnect` and `write` may execute on the same connection `c` then `reconnect` has to be enabled on `c` as well, but *not* the other way around.

In our earlier work, we presented two flow-insensitive static program analyses that make use of this information. The first analysis, the Quick Check, uses syntactic information only, that we can obtain directly through the weaving process. In our example, if the analysis finds that a program disconnects and reconnects connections but never writes to any connection, i.e., there is no `write` shadow, then this program cannot fulfil the dependency, and hence the entire aspect can have no visible effect for this program.

The second analysis stage, the Orphan-shadows Analysis, performs the same check, but on a per-object basis. This stage uses a flow-insensitive, context-sensitive points-to analysis [26] to disambiguate pointer references. This allows the analysis to decide which joinpoint shadows could potentially refer to the same objects. The analysis then uses this information as follows. In our example, if the program disconnects a particular connection `c` but never writes to `c`, then for this `c` the dependency is not fulfilled and therefore one does not need to monitor any `disconnect`, `reconnect` or `write` events on this connection.

We specifically designed Dependency State Machines in such a way that they are backward compatible to Dependent Advice in the following way. In our previous work we described an algorithm "*genDeps*", which generates Dependent-Advice declarations from any given finite-state machine. We took care to define the semantics of Dependency State Machines in such a way that one can apply *genDeps* directly to any Dependency State Machine to obtain a set of Dependent-

Advice declarations. The flow-insensitive analyses that we proposed earlier can then directly operate on these declarations. This means, that for our ConnectionClosed example, one could obtain the Dependent-Advice declaration that we mentioned above simply by applying the *genDeps* algorithm to the Dependency State Machine from Figure 1.

In our view, Dependency State Machines are easier to understand than Dependent Advice because their semantics follow the semantics of finite-state machines, which are well understood. Especially, it is safe to assume that most programmers are familiar with the basic semantics of finite-state machines. For Dependent Advice, the semantics are less obvious. Therefore, Dependency State Machines combine two advantages: they encode richer information, and nevertheless they are potentially easier to use.

4.3 Implementation

The flow-insensitive Quick Check and the Orphan-shadows Analysis generally finish faster than the more involved flow-sensitive Nop-shadows Analysis. Therefore, it is a good idea to apply the Nop-shadows Analysis only after the Quick Check and the Orphan-shadows Analysis have been applied first.

We implemented Dependency State Machines as an extension to the AspectBench Compiler [4] (abc) that builds on exactly these ideas. Our abc extension first extracts Dependency State Machines from the aspect definitions. Then it uses the *genDeps* algorithm to generate Dependent-Advice declarations from these Dependency State Machines. Next, we instruct abc to weave all aspects (whether they contain dependency declarations or not) into the given program. Our extension then applies both the Quick Check and the Orphan-shadows Analysis from previous work. These analyses only access the generated flow-insensitive Dependent-Advice declarations, no Dependency State Machines. If potentially relevant shadows remain after applying these two stages, then our extension invokes the Nop-shadows Analysis. This analysis is flow-sensitive, and therefore it accesses the Dependency State Machines to extract the information about the advice-execution ordering that the state machine's transition structure expresses. Every analysis stage may identify "irrelevant" shadows. In the end, our extension instructs abc to un-weave and re-weave the program, this time with all "irrelevant shadows" disabled.

Our abc extension contains two front-ends that both create an internal representation of the Dependency State Machines that the given program contains. One front end is implemented as an extension to the abc-internal parser. We use this front end to parse AspectJ source files that contain declarations of Dependency State Machines. The second front end that we provide creates the internal representation of the Dependency State Machines directly from a given set of tracematches [3]. Tracematches is another AspectJ language extension that allows programmers to define an AspectJ-based runtime monitor in a declarative way, using a regular-expression syntax. The aspects that abc generates from tracematches are never written to disc. abc instead generates these aspects in the form of Jimple [28] three-address code, an internal representation of the compiler, and then weaves the aspects into the designated program directly on this representation. Our tracematch front end therefore extracts the state machine directly from abc's internal representation of the tracematch.

5. EXPERIMENTS

To validate our approach, we defined a set of twelve monitoring aspects as tracematches. All aspects monitor for violations of safety properties. Table 1 explains the properties that these aspects monitor. We then applied the twelve aspects to ten benchmark programs of the DaCapo benchmark suite [6]. This lead to 120 aspect/benchmark combinations.

We were interested in answering two research questions. The first question evaluates how much our approach improves over previous work: (1) How effective is the Nop-shadows Analysis in identifying irrelevant shadows when compared to our two previously published static analyses that consider flow-insensitive dependency information from Dependent Advice only? The second question evaluates our approach from the user's point-of-view: (2) How effective is the overall approach, i.e., the combination of all three analysis stages (Quick Check, Orphan-shadows Analysis and our novel Nop-shadows Analysis) in telling apart irrelevant shadows from potentially relevant shadows.

To answer both questions, we decided to compare the number of shadows that our approach fails to identify as irrelevant, i.e., the number of potentially relevant shadows, to two different baselines: (1) the number of shadows that remain potentially relevant after applying the first two analysis stages, and (2) the total number of shadows that a compiler that is unaware of our dependency annotation and conducts no static analysis would present to the user. Table 2 summarizes our analysis results with respect to both baselines.

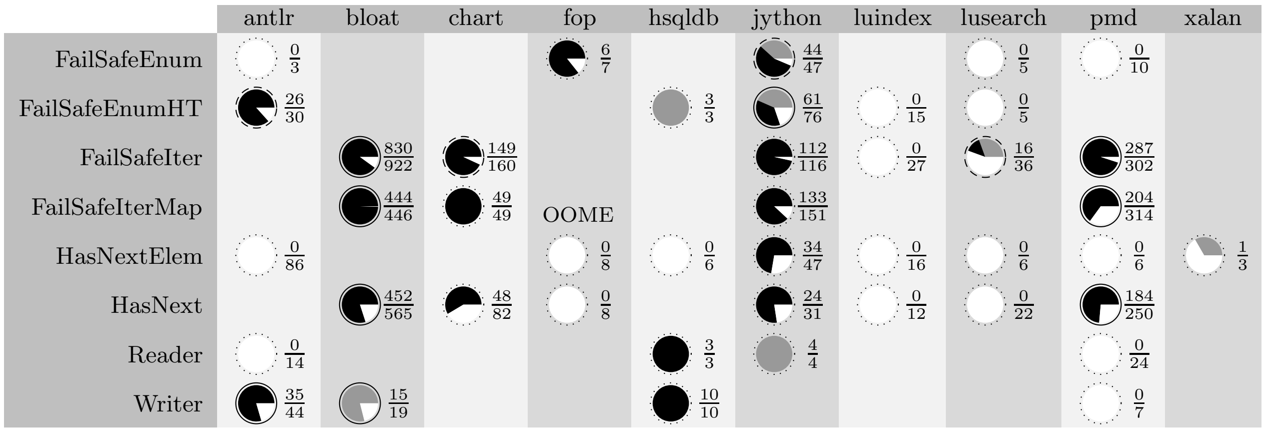
5.1 Analysis precision compared to previously published analyses

Table 2a shows the fraction of shadows that the Nop-shadows Analysis identified as irrelevant, where the baseline of this fraction is the number of potentially relevant shadows after applying the Quick Check and the Orphan-shadows Analysis. From this table we omitted those entries (and, where applicable, entire lines) for which the number of potentially relevant shadows was zero already after just applying these two analysis stages. The fraction of shadows that the Nop-shadows Analysis identified as nop shadows appears in white. For some combinations where only few shadows remained enabled after applying the analysis, we inspected these shadows manually. In gray we show the fraction of shadows that we manually determined to be relevant. The remaining black slices represent shadows that remain active even after analysis, either due to analysis imprecision or because they are actually relevant although they were not manually confirmed to be relevant. For the combination fop-FailSafeIterMap (1374 shadows to analyze), our Nop-shadows Analysis ran out of memory although we had provided abc with three gigabytes of heap space. As a research prototype, our analysis is currently not optimized towards low memory consumption.

For 18 out of these 43 combinations (41%), our novel Nop-shadows Analysis was able to identify all shadows as irrelevant. Because our aspects monitor safety properties, this means that, in these 18 cases, the analysis proved that the given program cannot possibly violate the given 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 appear as circles that only contain gray or white but no black slices. In other words, the analysis gave exactly the correct result, with no false posi-

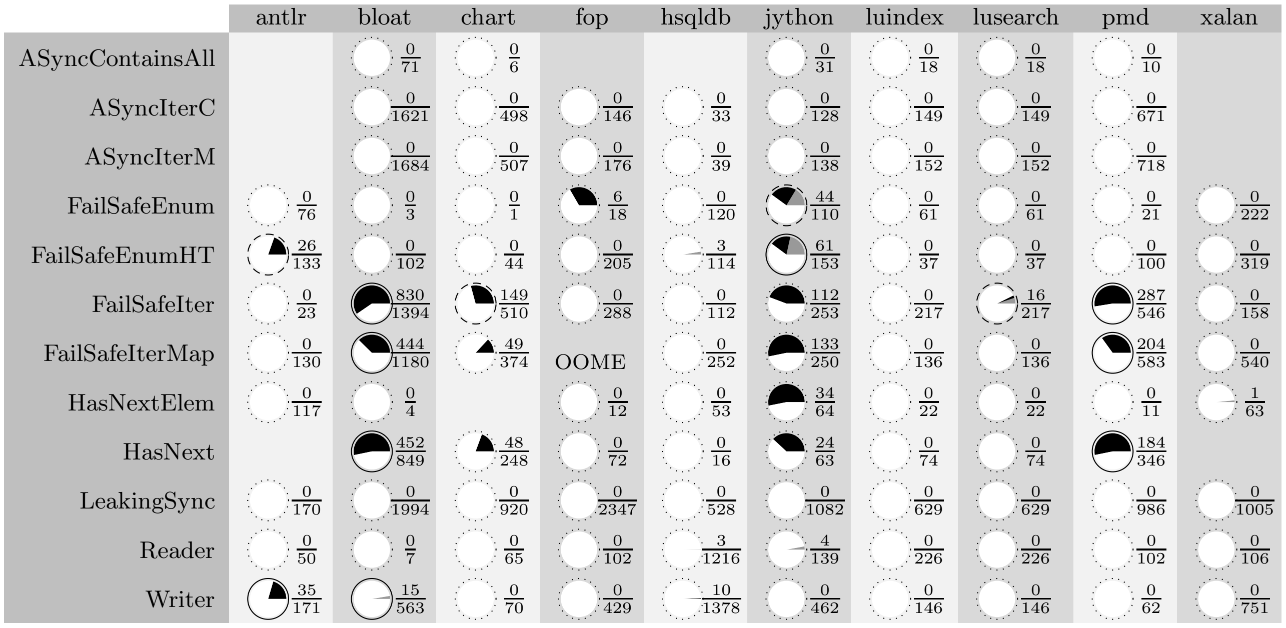
property name	description
ASyncContainsAll	synchronize on `d` when calling `c.containsAll(d))` for synchronized collections `c` and `d`
ASyncIterC	only iterate a synchronized collection `c` when owning a lock on `c`
ASyncIterM	only iterate a synchronized map `m` when owning a lock on `m`
FailSafeEnum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

Table 1: Relevant typestate properties and their names



	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	

(a) Potentially relevant shadows as fraction of shadows that remain after first two analysis stages



	antlr	bloat	chart	fop	hsqldb	jython	luindex	lusearch	pmd	xalan
ASyncContainsAll		0/71	0/6			0/31	0/18	0/18	0/10	
ASyncIterC		0/1621	0/498	0/146	0/33	0/128	0/149	0/149	0/671	
ASyncIterM		0/1684	0/507	0/176	0/39	0/138	0/152	0/152	0/718	
FailSafeEnum	0/76	0/3	0/1	6/18	0/120	44/110	0/61	0/61	0/21	0/222
FailSafeEnumHT	26/133	0/102	0/44	0/205	3/114	61/153	0/37	0/37	0/100	0/319
FailSafeIter	0/23	830/1394	149/510	0/288	0/112	112/253	0/217	16/217	287/546	0/158
FailSafeIterMap	0/130	444/1180	49/374	OOME	0/252	133/250	0/136	0/136	204/583	0/540
HasNextElem	0/117	0/4		0/12	0/53	34/64	0/22	0/22	0/11	1/63
HasNext		452/849	48/248	0/72	0/16	24/63	0/74	0/74	184/346	
LeakingSync	0/170	0/1994	0/920	0/2347	0/528	0/1082	0/629	0/629	0/986	0/1005
Reader	0/50	0/7	0/65	0/102	3/1216	4/139	0/226	0/226	0/102	0/106
Writer	35/171	15/563	0/70	0/429	10/1378	0/462	0/146	0/146	0/62	0/751

(b) Potentially relevant shadows as fraction of total shadows after weaving

Table 2: Irrelevant vs. potentially relevant shadows. White slices represent shadows that the Nop-shadows Analysis identified as irrelevant. Black slices represent shadows that we fail to identify as irrelevant, due to analysis imprecision or because the shadows are relevant. Gray slices represent shadows that we confirmed to be relevant, through manual inspection. The outer rings represent the aspect's runtime overhead after optimizing the advice dispatch. Solid: overhead $\geq 15\%$, dashed: overhead $< 15\%$, dotted: no overhead. OOME = OutOfMemoryException during static analysis

tives, in half of the cases. In three cases, the analysis failed to identify any nop shadow (black circles). In the remaining 18 cases, the analysis identified a sometimes significant amount of irrelevant shadows, but not all.

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 [2], 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 lose all context information. bloat, jython and pmd cause similar problems.

There appear to be only 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. Our current implementation of the Nop-shadows Analysis uses flow-sensitive information on an intra-procedural level only, and therefore cannot possibly produce precise analysis results in this context. In the future, we plan to extend the Nop-shadows Analysis into a fully inter-procedural version that will treat these cases more precisely.

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

For eclipse we were unable to determine where dynamic classes are loaded from. eclipse loads classes not from JAR files but from "resource URLs", which eclipse resolves internally, usually to JAR files within other JAR files. abc 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 unsound assumption that this code would not invoke any dependent advice.

5.2 Fraction of potentially relevant shadows over number of all shadows after weaving

Table 2b shows the fraction of shadows that the Nop-shadows Analysis identified as irrelevant, where the baseline of this fraction is the number of all shadows that a compiler without any of our static analysis would usually report to the user. This fraction shows to what extend users can benefit through the use of Dependency State Machines in general, when applying all three of our static analyses in combination, compared to not using Dependency State Machines at all. Opposed to Table 2a, this table shows the important piece of information that, in many cases, the Quick Check and the Orphan-shadows Analysis manage to identify many irrelevant shadows already. Often, these analyses are even sufficient to identify that all shadows are irrelevant, i.e., that the aspect will never have a visible effect. The high amount of white and gray in this table shows that our overall static-analysis approach is very effective in pinpointing to the user the relevant shadows that will cause the aspect to have a visual effect at runtime.

5.3 Reduction of runtime overhead

An added benefit of our analysis is that we can use the analysis result to optimize the advice dispatch, which may reduce the aspect's runtime overhead. Table 2 gives qualitative information about the optimized aspect's runtime overhead through the ring that surround each circle. (The author's dissertation [8] gives the full data.) Interestingly, the number of remaining shadows does not necessarily correspond directly to the resulting runtime overhead. For instance, only 15 out of 563 shadow remain in bloat-Writer, but these shadow executes so often that they cause a runtime overhead of more than 15%. chart-FailSafeIterMap, on the other hand, contains 49 residual shadows, but there is no observable overhead. Altogether, after applying the Nop-shadows Analysis, only nine combinations remain that have a significantly perceivable overhead of more than 15%. Most combinations show zero overhead, five combinations show an overhead of below 15%, which seems negligible in many cases.

6. RELATED WORK

We compare our work to the most related static program analyses and to aspect-oriented model-checking approaches. In addition, we discuss tools that generate AspectJ aspects from high-level specifications and whether these tools could generate Dependency State Machines as well.

6.1 Static program analysis

Whole-program analysis of tracematches. Tracematches [3] is an AspectJ language extension that allows programmers to express finite-state properties using a high-level language that is based on regular expressions. Like Dependency State Machines, tracematches are implemented on top of the AspectBench Compiler. During compilation, the compiler internally reduces tracematches to "normal" AspectJ aspects. Several people [10, 11, 24], including ourselves, have proposed static analyses that exploit the information that tracematches contain to optimize advice dispatch. The Quick Check and the Orphan-shadows Analysis, that we discussed in Section 4, are generalized versions of two similar analyses that we previously implemented specifically for tracematches [10]. The Nop-shadows Analysis that we presented in this paper is defined directly in terms of Dependency State Machines, and we never implemented a tracematch-specific version for it (although we could have). The advantage of Dependency State Machines is not that they improve the analyzability of tracematches or similar formalisms, but rather that Dependency State Machines make existing analyses applicable to aspects in general, no matter whether the aspects were generated from tracematches, from any other high-level specification or even written by hand.

Monitor optimizations. Avgustinov et al. [5] proposed optimizations to the monitoring aspect itself: *Leak elimination* discards monitoring state for objects that have been garbage collected. *Indexing* provides for fast access to partial matches. These optimizations are crucial to make runtime monitoring feasible at all and therefore we enabled them in our experiments when generating aspects from tracematches. JavaMOP [13] and PTQL [17] implement weaker variants of these optimizations.

One big advantage of Dependency State Machines is that they allow researchers to de-couple the optimizations of runtime monitors, i.e., the code that "goes into the advice bodies", from analyzing and optimizing the advice dispatch, based on the order in which these pieces of advice should execute. In theory it would be possible to determine an aspect's transition structure directly through static analysis, without requiring an explicit dependency annotation. However, general AspectJ code is Turing complete, which makes this analysis problem generally undecidable. In particular, the optimizations that aspect-generating monitoring tools conduct can lead to arbitrarily complex aspect code, much more complex than the code that we showed in Figure 1. This makes it very hard for static analyses to re-discover the transition structure directly from the code. Dependency State Machines elegantly solve this problem by simply specifying the transition structure directly in a machine readable format.

6.2 Model Checking

Goldman and Katz [16] propose a model-checking approach that can "once and for all" verify an aspect "relative to its specification", i.e., independently of any specific program that this aspect may be woven into. The authors developed the MAVEN tool that implements this modular aspect-verification mechanism. Like ourselves, the authors assume that the aspect's internal structure can be represented as a finite-state machine. However, unlike us, Goldman and Katz do not state how programmers would communicate this transition structure to the model checker. Our proposed syntax, Dependency State Machines, closed this gap. Another restriction of Goldman and Katz's approach is that the authors assume that one can represent the non-aspect parts of the program as a finite-state machine as well. This is necessary, because the authors model the weaving process through a series on transformations on state machines. In our approach, we make no such assumption. We leave the weaving semantics to standard AspectJ.

It would be an interesting piece of future work to determine whether the semantics that we gave to Dependency State Machines is compatible with the finite-state-machine semantics that Goldman and Katz's approach requires. If so, it should be easily possible to integrate MAVEN with our abc extension.

6.3 Aspect-generating tools

JavaMOP. JavaMOP [13] is an open research framework for generating AspectJ monitoring aspects from several kinds of formal specifications, including Extended Regular Expressions, Past-time and Future-Time Linear Temporal Logic. In previous work [9], Feng Chen has modified the JavaMOP implementation so that it internally generates a finite-state machine from all these formal specification, regardless of the formalism that is used. JavaMOP can directly benefit from Dependency State Machines by annotating the generated aspects with these state machines.

Association aspects and relational aspects. Sakurai et al. [25] proposed *association aspects*, an AspectJ language extension that allows programmers to restrict advice execution to joinpoints involving objects that the programmer explicitly associated with an aspect. A programmer associates an object `o` with an aspect `A` by calling `A.associate(o)`, and releases the association via `A.release(o)`. In earlier work [12] we showed that one can implement *relational aspects*, a variant of *association aspects*, via a syntactic transformation into tracematches. abc implements relational aspects that way, and the implementation automatically benefits from our extension: The optimizations proposed in this paper remove advice dispatch code for an advice contained in an aspect A from locations where the objects involved are known to be either not yet associated with A or to already have been released from A.

S2A, M2Aspects and J-LO. Maoz and Harel proposed S2A, a tool [21] to generate executable AspectJ code from Live Sequence Charts [15] (LSCs). An LSC and its generated aspects can either implement functional aspects of a system, or they can be used for runtime monitoring, reporting error messages when they match. Some of the aspects that S2A generates are history-based, and in fact even implement a finite-state machine. We confirmed with Maoz that S2A could, in principle, generate dependency annotations for these aspects and that they could lead to optimization potential similar to what we observed in our experiments, at least when LSCs are used to specify forbidden scenarios, implemented as runtime monitors. M2Aspects [20] generates AspectJ aspects from scenario-based software specifications, denoted as Message Sequence Charts (MSCs). MSCs are less expressive than LSCs. Hence we believe that one could also modify M2Aspects to generate dependent advice. J-LO, the Java Logical Observer [7, 27] generates AspectJ aspects from formulae written in a special future-time linear temporal logic with free variables. Internally, J-LO represents the formulae using alternating automata. There exists a standard algorithm to convert alternating automata into finite-state machines. J-LO could therefore easily benefit from Dependency State Machines by implementing this conversion and annotating the generated aspect with the appropriate Dependency-State-Machine declaration.

7. CONCLUSIONS AND FUTURE WORK

In this paper we have presented Dependency State Machines, a language extension to AspectJ that expressed the order in which pieces of advice have to occur, so that these pieces of advice, in combination have an effect that is visible outside the declaring aspect. We have shown how to use the information that Dependency State Machines provide to facilitate program understanding. We have outlined a set of static analyses that can identify "irrelevant" joinpoint shadows with high precision. When trying to determine the effects that their aspects may have, programmers do not need to consider such irrelevant shadows.

Nevertheless, we believe that also in fields different from program understanding, Dependency State Machines offer the potential for a lot of exciting research opportunities that researchers could address in the near future. One interesting field of research could be the inference of Dependency State Machines. Our current approach assumes that Dependency

State Machines are present in aspect code, i.e., that either the programmer or some aspect generating tool supplied the state-machine declaration. In many cases, it could be possible to infer these declarations automatically from aspect code or from dynamic executions.

Another interesting research question would be how Dependency State Machines can be used to verify or check an aspect's execution. According to the semantics that we gave in this paper, a Dependency State Machine expresses the order in which pieces of advice must execute to have a visible effect. One could give different semantics to Dependency State Machines, e.g. that a Dependency State Machine describes the order in which pieces of advice are allowed to be called by the surrounding context, i.e., the program which the pieces of advice are woven into. Static or runtime verification could then try to determine, for a particular execution context, whether this context fulfils the aspects execution requirements.

Acknowledgements. This work originated from a collaboration of the author with Feng Chen and his supervisor Grigore Roşu. Tragically, Feng passed away in summer 2009. May he rest in peace and may his legacy not be forgotten. The author conducted this work as a PhD student at McGill University, under supervision of Laurie Hendren. Thanks, Laurie, for all the support you gave me at McGill. This work was supported by NSERC and the Center for Advanced Security Research Darmstadt (www.cased.de).

8. REFERENCES

[1] The AspectJ home page. http://eclipse.org/aspectj/.

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

[3] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding Trace Matching with Free Variables to AspectJ. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 345–364. ACM Press, Oct. 2005.

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

[5] P. Avgustinov, J. Tibble, and O. de Moor. Making trace monitoring feasible. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 589–608. ACM Press, Oct. 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. ACM Press, Oct. 2006.

[7] E. Bodden. J-LO - A tool for runtime-checking temporal assertions. Diploma thesis, RWTH Aachen University, November 2005.

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

[9] E. Bodden, F. Chen, and G. Roşu. Dependent advice: a general approach to optimizing history-based aspects. In *AOSD '09: Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 3–14. ACM, 2009.

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

[11] E. Bodden, P. Lam, and L. Hendren. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In *Symposium on the Foundations of Software Engineering (FSE)*, pages 36–47, New York, NY, USA, 2008. ACM.

[12] E. Bodden, R. Shaikh, and L. Hendren. Relational aspects as tracematches. In *International Conference on Aspect-oriented Software Development (AOSD)*, pages 84–95. ACM Press, Mar. 2008.

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

[14] W. F. Clocksin and C. Mellish. *Programming in Prolog, 5th Edition.* Springer, 2003.

[15] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 1999.

[16] M. Goldman and S. Katz. MAVEN: Modular Aspect Verification. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 308–322, 2007.

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

[18] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java(TM) Language Specification, Third Edition.* Addison-Wesley Professional, 2005.

[19] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In *International Conference on Aspect-oriented Software Development (AOSD)*, pages 26–35. ACM Press, Mar. 2004.

[20] I. H. Krüger, G. Lee, and M. Meisinger. Automating software architecture exploration with M2Aspects. In *SCESM*, pages 51–58. ACM Press, 2006.

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

[22] H. Masuhara, Y. Endoh, and A. Yonezawa. A fine-grained join point model for more reusable aspects. *Programming Languages and Systems*, 4279:131–147, 2006.

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

[24] N. A. Naeem and O. Lhoták. Typestate-like analysis of multiple interacting objects. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 347–366, New York, NY, USA, 2008. ACM.

[25] K. Sakurai, H. Masuhara, N. Ubayashi, S. Matsuura, and S. Komiya. Association aspects. In *International Conference on Aspect-oriented Software Development (AOSD)*, pages 16–25, Mar. 2004.

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

[27] V. Stolz and E. Bodden. Temporal Assertions using AspectJ. In *5th Workshop on Runtime Verification*, volume 144 of *Electronic Notes in Theoretical Computer Science*, pages 109–124, July 2005.

[28] R. Vallée-Rai and L. Hendren. Jimple: Simplifying Java Bytecode for Analyses and Transformations. Technical Report 1998-4, Sable Research Group, July 1998.