AOSD'09, ACM, pp 3--14. 2009.

Dependent advice: A general approach to optimizing history-based aspects

Eric Bodden
McGill University
Montréal, Québec, Canada

Feng Chen
University of Illinois at
Urbana-Champaign, IL, USA

Grigore Roşu
University of Illinois at
Urbana-Champaign, IL, USA

ABSTRACT

Many aspects for runtime monitoring are *history-based*: they contain pieces of advice that execute conditionally, based on the observed execution history. History-based aspects are notorious for causing high runtime overhead. Compilers can apply powerful optimizations to history-based aspects using domain knowledge. Unfortunately, current aspect languages like AspectJ impede optimizations, as they provide no means to express this domain knowledge.

In this paper we present *dependent advice*, a novel AspectJ language extension. A dependent advice contains dependency annotations that preserve crucial domain knowledge: a dependent advice needs to execute only when its dependencies are fulfilled. Optimizations can exploit this knowledge: we present a whole-program analysis that removes advice-dispatch code from program locations at which an advice's dependencies cannot be fulfilled.

Programmers often opt to have history-based aspects generated automatically, from formal specifications from model-driven development or runtime monitoring. As we show using code-generation tools for two runtime-monitoring approaches, tracematches and JavaMOP, such tools can use knowledge contained in the specification to automatically generate dependency annotations as well.

Our extensive evaluation using the DaCapo benchmark suite shows that the use of dependent advice can significantly lower, sometimes even completely eliminate, the runtime overhead caused by history-based aspects, independently of the specification formalism.

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

Submitted to AOSD 2009.

1. INTRODUCTION

In this paper we present *dependent advice*, a novel language extension to aid efficient implementations of, and reasoning about, *history-based aspects*. A history-based aspect executes its pieces of advice conditionally, based on the observed execution history. There can be many uses of history-based aspects but programmers primarily use history-based aspects for runtime monitoring and verification.

Figure 1 shows a simplified example, the "ConnectionClosed" aspect. This aspect monitors the events of disconnecting and reconnecting a connection `c`, as well as writing data to `c`. Note that almost all the aspect code is concerned with bookkeeping internal state. This can induce a large runtime overhead [3, 7, 11, 14, 20]. The error message at line 17 implements the only functionality that is visible outside the aspect. Note that the aspect prints the error only if *both* the advice "disconn" and "write" execute on the same connection `c`. In addition, the advice "reconn" only has to execute on connections that are both disconnected and written to at some point in time. Compilers could use this important information to apply powerful optimizations: For example, one does not have to monitor "disconn(`c`)" if the connection `c` is never written to. Unfortunately a programmer cannot express this crucial domain knowledge in plain AspectJ syntax, and it would be very hard for an AspectJ compiler to re-construct this knowledge solely based on the aspect code. This impedes crucial optimizations.

Dependent advice solve this problem. A dependent advice

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

Figure 1: ConnectionClosed monitoring aspect

contains dependency annotations to encode crucial domain knowledge: a dependent advice needs to execute only when its dependencies are fulfilled. For the "connection" example from Figure 1, a programmer could add the annotation

dependency{ **strong** disconn, write; **weak** reconn; }.

This annotation conveys the information that the execution of the advice "disconn" and "write" both depend on one another, and in addition the execution of "reconn" depends on both "disconn" and "write" to execute at some point in time.

Programmers can use dependent advice to document design intent or to aid static verification. For instance dependencies could encode forbidden combinations of events and static whole-program analyses could prove that such combinations cannot occur. In this paper we focus however on using dependent advice to aid an efficient implementation of history-based aspects: we present a flow-insensitive whole-program analysis that removes dispatch code for dependent advice from program locations at which the advice's dependencies cannot be fulfilled. The analysis is a generalized version of a flow-insensitive static whole-program analysis that Bodden et. al originally designed [7] for tracematches [1], an AspectJ language extension for runtime monitoring. The results of our evaluation show that the use of dependent advice can yield significant speedups at runtime.

However, writing dependency annotations by hand can be error prone and very time consuming. Therefore it would be beneficial if tools could generate these annotations automatically. Fortunately, many people do not write history-based aspects by hand either: researchers have proposed several tools [1, 11, 17, 19] that generate history-based AspectJ aspects automatically, from formal specifications from runtime verification or model-driven development. As we show in this paper, these specifications convey enough domain knowledge to generate dependent advice automatically. We modified two runtime monitoring tools, tracematches [1] and *JavaMOP* [11], to generate dependent advice from specifications that express monitoring properties using past-time and future-time linear temporal logic and regular expressions.

To validate our approach we applied a large set of both generated and hand-written aspects with and without dependency annotations to the DaCapo [4] benchmark suite. Our results show that the use of dependent advice can significantly lower, and sometimes even completely eliminate, the runtime overhead caused by history-based aspects. Most interestingly however, while the result of this optimization depends on the monitored property and program, it is independent of the code generation tool and specification formalism. This is encouraging, as it indicates that dependent advice express dependencies within history-based aspects in a natural way.

To summarize, the main contributions of this paper are:

- an AspectJ language extension called *dependent advice*, encoding domain knowledge that helps compilers optimize advice execution, and an implementation of this extension in the AspectBench Compiler [2] (*abc*),
- an algorithm that generates dependent advice from finite-state models or specifications, along with an implementation of this algorithm for *JavaMOP* (regular expressions, past-time and future-time LTL) and tracematches, and
- a set of experiments proving that compilers can successfully optimize dependent advice (whereas normal advice could not be optimized any further) and that these optimizations are effective regardless of the specification tool and formalism that was used to generate the dependent advice.

We organized the remainder of the paper as follows. In the next section we explain dependent advice, their syntax, their static semantics and a parametrized matching semantics. The parameter to the semantics is a Boolean predicate "*activates*" that determines when exactly a dependent advice does or does not execute at runtime. A compiler can provide different implementations of *activates*, but the semantics demands that any correct implementation of dependent advice adheres to a special soundness condition. In Section 3 we present an efficient and effective implementation of dependent advice and prove this implementation sound with respect to the semantics. In Section 4 we explain an algorithm to generate dependent advice from any finite-state based monitor specification. In an accompanying Technical Report [5] we prove this algorithm correct: monitoring-aspects with dependent advice are behaviour-equivalent to monitoring-aspects that only use ordinary advice. In the Technical Report we also prove the algorithm "stable": it generates equivalent dependency annotations for equivalent finite-state specifications, even if these specifications are written in different formalisms. Section 5 explains our experiments, followed by a discussion of related work and conclusions.

2. DEPENDENT ADVICE

In this section we describe dependent advice. We start by explaining their syntax, first in a short form and then in a more verbose form. Then we give a matching semantics.

2.1 Syntax

Dependent advice are a backwards-compatible AspectJ language extension that comprise the following syntactic changes. (Our Report [5] formally defines the full syntax.)

- Pieces of advice can have a `dependent` *modifier*,
- every `dependent` advice is given a *name*, and
- an aspect can hold a set of *dependency declarations*.

A dependency declaration has the following form:

dependency{
  strong s1, . . . , sn;
  weak w1, . . . , wm;
}

Here `s1` through `sn`, and `w1` through `wm`, are names of dependent advice declared in the same aspect as the dependency declaration. Figure 2 shows how to use dependent advice for ConnectionClosed.

Informally, the meaning of "**strong** disconn, write;" is that the `disconn` advice only has to execute on a Connection `c` if at some point in time the advice `write` executes on `c` as well. In addition, `write` only has to execute on `c` if `disconn` executes on `c`. In other words, the dependency states that if `disconn` was to execute on a Connection `c` for which it is known that `write` *never* occurs on `c` then the execution of `disconn` can safely be omitted—and the other

AOSD'09, ACM, pp 3--14. 2009.

```
1  aspect ConnectionClosed {
2    dependency{ strong disconn, write; weak reconn; }
3
4    Set closed = new WeakIdentityHashSet();
5
6    dependent after disconn(Connection c) returning:
7      call(* Connection.disconnect()) && target(c) {
8      closed.add(c);
9    }
10
11   //... advice ''write'' and ''reconn'' ommited for brevity
12 }
```

Figure 2: ConnectionClosed with dependent advice

way around. Weak dependencies are slightly different: By adding "**weak** reconn;" the programmer states that "reconn" only has to execute on Connections `c` for which both "disconn" and "write" execute at some point, but *not* the other way around.

Note however that the dependency annotation in Figure 2 (line 2) omits the variable name `c` of the Connection. This is because, by default, a dependency annotation infers variable names from the formal parameters of the advice declarations that it references (e.g. line 6). The dependency annotation from Figure 2 is a short hand for the more verbose

```
dependency{ strong disconn(c), write(c); weak reconn(c); }
```

The semantics of variables in dependency declarations is similar to unification semantics in logic programming languages like Prolog [12]: 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 `k`:

```
dependency{ strong detectLoops(k,k); }
```

This concludes the discussion of the syntax of dependent advice. Out Technical Report contains further information about the type-checks that we apply to dependent advice and to dependency declarations. Here we just wish to state that we ensure that every dependent advice is referenced by at least one dependency declaration. Also, each dependency declaration references each dependent advice at most once.

2.2 Matching semantics

We define the matching semantics of dependent advice as a semantic extension to ordinary advice matching in AspectJ. A program can generally have multiple aspects with dependent advice. However, since the semantics of dependent advice in one aspect is defined independently from other aspects, in the following we assume one fixed aspect A, without loss of generality.

Let $\mathcal{A}$ be the set of A's pieces of advice, $\mathcal{D}$ the set of dependency declarations in A, $\mathcal{V}$ the set of all possible variable names, $\mathcal{O}$ the set of all heap objects allocated on a given program execution and $\mathcal{J}$ the set of all AspectJ joinpoints (i.e. events) on that execution.

Furthermore we declare functions *strong* and *weak* of type $\mathcal{D} \to \mathcal{P}(\mathcal{A})$, which return the set of advice that the dependency declaration $d \in \mathcal{D}$ references as strong, respectively weak advice. We define the set $\mathcal{A}^d \subseteq \mathcal{A}$ as $\mathcal{A}^d := \mathit{strong}(d) \cup \mathit{weak}(d)$.

In the following let us assume that variables in d have been fully inferred (see Section 2.1). The set $\mathcal{V}^d$ is the set of variables mentioned in d. Our type checks ensure that d references each advice $a \in \mathcal{A}^d$ only once. Therefore d induces for each advice a a mapping σ_a^d from a's parameters to variables in $\mathcal{V}^d$: If d references an advice declaration `adv(T1 p1,...,Tn pn)` using the advice reference `adv(v1,...,vn)` then we obtain the mapping

$$\sigma^d_{\mathtt{adv}} = \{\mathtt{p1} \mapsto \mathtt{v1}, \ldots, \mathtt{pn} \mapsto \mathtt{vn}\}.$$

Note that σ_a^d is the identity function in case that variable names were inferred for a in d.

2.2.1 Advice matching for normal advice

We model advice matching in AspectJ [15] as a function

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

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).

Compatible joinpoints.

In the remainder of this section we will refer to "compatible joinpoints". We say that two joinpoints j_a and j_b are *compatible* with respect to a dependency declaration d and two pieces of advice a and b if a executes at j_a with a variable binding β_a, b executes at j_b with a variable binding β_b respectively, and both β_a and β_b assign the same objects to equal variables, with variable names substituted as defined through d. Formally we define a predicate *compt* as follows:

$$\mathit{compt}: \quad \mathcal{J} \times \mathcal{A} \times \mathcal{J} \times \mathcal{A} \times \mathcal{D} \quad \to \quad \mathbb{B}$$

$$\begin{aligned}
&\mathit{compt}(j_a, a, j_b, b, d) = \\
&\quad \text{let } \beta_a := \mathit{match}(a, j_a), \beta_b := \mathit{match}(b, j_b) \text{ in} \\
&\qquad \beta_a \neq \bot \wedge \beta_b \neq \bot \wedge \\
&\qquad\quad \forall p_a \in \mathit{dom}(\sigma_a^d)\ \forall p_b \in \mathit{dom}(\sigma_b^d): \\
&\qquad\qquad \sigma_a^d(p_a) = \sigma_b^d(p_b) \to \beta_a(p_a) = \beta_b(p_b)
\end{aligned}$$

2.2.2 Advice matching for dependent advice

Dependent advice differ in their matching semantics from normal AspectJ advice and we therefore define a function *depMatch* that matches *dependent* advice against joinpoints, based on $\mathcal{D}$ and *match*. *depMatch* also has access to a function *activates*. This function is a parameter to *depMatch* (description follows).

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

$$\mathit{depMatch}(a, j) =$$

$$\begin{cases} \mathit{match}(a, j) & \text{if } \mathit{match}(a, j) \neq \bot \ \wedge \\ & \quad \exists d \in \mathcal{D} \ . \ \mathit{activates}(d, a, j) \\ \bot & \text{else} \end{cases}$$

The function *depMatch* refines the original *match* function provided by AspectJ: It only produces a match if the Boolean predicate *activates* holds for at least one advice dependency. When $\mathit{activates}(d, a, j)$ holds, we say that the dependency d *activates* the dependent advice a at j. The predicate *activates* is a parameter to our matching semantics. A compiler may choose between different implementations of *activates* but we define that any *sound* implementation of dependent advice *must* guarantee:

Condition 1 (Soundness condition)**.**

$$\forall d \in \mathcal{D} \;\; \forall a \in \mathcal{A} \;\; \forall j_a \in \mathcal{J} \;\; :$$

$$\Big(a \in \mathcal{A}^d \;\wedge\; \forall b \in \mathit{strong}(d) \; \exists j_b \in \mathcal{J} : \mathit{compt}(j_a, a, j_b, b, d)\Big)$$

$$\longrightarrow \mathit{activates}(d, a, j_a) = \textbf{true}$$

Informally, Condition 1 states that a dependency d *must* activate a at joinpoint j_a, if d references a (as strong or weak advice), and for each *strong* advice b in d there is some joinpoint j_b (at some time earlier or later in the program execution, or the current joinpoint itself) that is compatible with j_a (with respect to d, a and b).

The most conservative implementation would be the constant function **true**. This would effectively treat dependent advice just as ordinary AspectJ advice (*depMatch* degenerates to *match* as our type-checks ensure that $\mathcal{D} \neq \emptyset$).

An optimizing implementation would instead want to return **false** from *activates* whenever possible, but without jeopardizing soundness. A perfect implementation would determine *activates* such that it returns **false** whenever the premise of Condition 1 does not hold. That way, the implementation would disable dependent advice whenever possible but still guarantee soundness. Unfortunately determining *activates* that way is *undecidable*: At the time where *activates* needs to decide whether or not to activate a dependency at the current joinpoint, it may need to know whether a compatible joinpoint will occur in the future.

A sensible implementation of dependent advice must therefore approximate *activates*. It must *try* to return **false** on a best-effort basis, but only when the soundness condition permits, i.e. when the premise of the soundness condition does not hold. In the next section we explain an effective implementation based on this principle.

3. IMPLEMENTING DEPENDENT ADVICE

We next explain the static abstraction of Condition 1 that we use in our implementation. The abstraction considers *all* possible program executions. In Section 3.2 we prove this abstraction sound. We explain the details of our concrete implementation in the AspectBench Compiler in Section 3.3.

3.1 A static abstraction of Condition 1

Our soundness condition, Condition 1, defines the situations in which $\mathit{activates}(d, a, j)$ *must* return **true**. As noted earlier, an effective implementation of dependent advice should attempt to return **false** from this function whenever possible, i.e. whenever the premise of Condition 1 does not hold. This is exactly the case when its negation holds:

Condition 2 (Negation of the premise of Condition 1)**.**
$a \notin \mathcal{A}^d \;\vee\; \exists b \in \mathit{strong}(d) \; \forall j_b \in \mathcal{J} : \neg \mathit{compt}(j_a, a, j_b, b, d)$

According to Condition 2, a dependency d can fail to activate a dependent advice a for two reasons. In the first case d does not at all reference a, i.e. $a \notin \mathcal{A}^d$. This is the trivial case. (Note that our type checks demand that a be referenced by *some* dependency, so there must be another dependency d' which at least gives a a chance of being activated.) The second reason is that there is a strong advice b in d so that there exists no joinpoint j_b that is compatible with j_a. This is the condition that our static analysis exploits.

Note that we can fully determine the following parts of Condition 2 at compile time. For each dependency d we can determine the sets $\mathit{strong}(d)$ and $\mathcal{A}^d$. For any advice $a \in \mathcal{A}^d$ the variable substitution σ_a^d (used within *compt*) is also statically determined. Hence, the only parts of Condition 2 that our static analysis needs to approximate are:

1. the set $\mathcal{J}$ of all joinpoints, and
2. the variable binding $\mathit{match}(a, j)$ that occurs when advice a matches at joinpoint j (also used within *compt*).

Approximating joinpoints through joinpoint shadows.

A woven AspectJ program generates a joinpoint j by executing a piece of code generated by the AspectJ compiler at a specific program location, j's *joinpoint shadow* [22] $\mathit{shadow}(j)$. We define the set $\mathcal{S}$ of all shadows as:

$$\mathcal{S} = \bigcup_{j \in \mathcal{J}} \{s \mid s = \mathit{shadow}(j)\}$$

We can now define our static approximation of Condition 2 via joinpoint shadows. Given a dependent advice a, a shadow s_a and a dependency d, we define:

Condition 3 (Static approximation of Condition 2)**.**
$a \notin \mathcal{A}^d \;\vee\; \exists b \in \mathit{strong}(d) \; \forall s_b \in \mathcal{S} : \neg \mathit{sCompt}(s_a, a, s_b, b, d)$

The function *sCompt* is a static approximation of *compt* that accepts shadows instead of joinpoints. Both functions are very similar. The only difference is that *compt* uses *match* to compute mappings from variables to runtime objects. At compile time we have no access to runtime objects. *sCompt* therefore approximates this mapping through a compile-time function *sMatch*.

Approximating of objects through points-to sets.

Because we now deal with joinpoint shadows, we redefine *match* as a function *sMatch* over inputs from $\mathcal{S}$ instead of $\mathcal{J}$. A function call $\mathit{match}(a, j)$ returns $\bot$ when advice a does not execute at j. This is a runtime decision: Several AspectJ pointcuts have to be evaluated at runtime. For instance the pointcut `this(A)` only matches if the concrete runtime type of the currently executing object is a subtype of `A`. AspectJ compilers allow the AspectJ runtime to determine a match by weaving a *dynamic residue* [15] in place of the joinpoint shadow. In some cases a compiler can statically determine that an advice a can never apply at a given joinpoint shadow $s = \mathit{shadow}(j)$. For instance, in the above example it could be that the currently executing object must be of a final type (i.e. can have no subtypes) that is not a subtype of `A`. In this case `this(A)` cannot hold at s, and the compiler generates a "Never" residue that instructs the compiler not to weave any advice code for a at s. In the following we will say that $\mathit{never}(a, s)$ holds in this situation.

The other difference between *match* and *sMatch* is that, because *sMatch* is evaluated at compile time, it cannot return a mapping from advice parameters to runtime objects. Every joinpoint shadow does however give us access to a mapping ι which maps each advice parameter p to the local program variable l that the compiler inserts to bind p to its runtime value when the advice is executed at this shadow. For a local variable l we can determine its points-to set [18] $\mathit{pointsTo}(l)$. We denote the set of all points-to sets by $\mathbb{P}$. This allows us to define *sMatch* as follows.

$$\mathit{sMatch}: \quad \mathcal{A} \times \mathcal{S} \quad \rightarrow \quad \{\beta \mid \beta : \mathcal{V} \rightharpoonup \mathbb{P}\} \cup \{\bot\}$$

$$\mathit{sMatch}(a, s) = \begin{cases} \bot & \text{if } \mathit{never}(a, s) \\ \lambda v \; . \; \mathit{pointsTo}(\iota(v)) & \text{else} \end{cases}$$

This makes us almost ready for defining our static approximation of the function *compt*. The last insight that we exploit is that two run-time objects referenced by advice parameters p and q cannot point to the same objects if $\mathit{pointsTo}(\iota(p)) \cap \mathit{pointsTo}(\iota(q)) = \emptyset$: In this case p and q are only assigned values from local variables that themselves are definitely not assigned objects from the same allocation site. This yields the following definition of *sCompt*.

$$\mathit{sCompt}: \quad \mathcal{S} \times \mathcal{A} \times \mathcal{S} \times \mathcal{A} \times \mathcal{D} \quad \rightarrow \quad \mathbb{B}$$

$$\begin{aligned} &\mathit{sCompt}(s_a, a, s_b, b, d) = \\ &\quad \text{let } \beta_a := \mathit{sMatch}(a, s_a), \beta_b := \mathit{sMatch}(b, s_b) \text{ in} \\ &\quad\quad \beta_a \neq \bot \wedge \beta_b \neq \bot \wedge \\ &\quad\quad\quad \forall p_a \in \mathit{dom}(\sigma_a^d) \; \forall p_b \in \mathit{dom}(\sigma_b^d) : \\ &\quad\quad\quad\quad \sigma_a^d(p_a) = \sigma_b^d(p_b) \rightarrow \beta_a(p_a) \cap \beta_b(p_b) \neq \emptyset \end{aligned}$$

3.2 Soundness of the approximation

We next define what it means for this abstraction to be sound, and prove soundness based on this definition.

Theorem 1 (*sCompt* soundly approximates *compt*)**.**

$$\begin{aligned} &\forall j_a, j_b \in \mathcal{J} \;\; \forall d \in \mathcal{D} \;\; \forall a, b \in \mathcal{A}^d \;\; : \\ &\quad \mathit{compt}(j_a, a, j_b, b, d) \\ &\quad\quad \longrightarrow \mathit{sCompt}(\mathit{shadow}(j_a), a, \mathit{shadow}(j_b), b, d) \end{aligned}$$

Proof 1 (Proof of Theorem 1)**.** The proof of Theorem 1 is almost immediate if one assumes that points-to sets are computed in a sound way, i.e. if o is an object created at allocation site s and assigned to a program variable l then $s \in \mathit{pointsTo}(l)$—a general assumption that we make for this paper. We conduct the proof in inverse order, from the right to the left. If $\mathit{sCompt}(\mathit{shadow}(j_a), a, \mathit{shadow}(j_b), b, d)$ does not hold then this can have two reasons: (1) we have $\mathit{never}(a, s_a)$ or $\mathit{never}(b, s_b)$, or (2) the two shadows induce variable bindings that assign disjoint points-to sets to the same variable from d (used at different locations). In case (1) $\neg\mathit{compt}(j_a, a, j_b, b, d)$ holds trivially because $\mathit{never}(a, s_a)$ implies $\mathit{match}(a, j_a) = \bot$, and the same holds for b, s_b and j_b. Similarly, (2) disjoint points-to sets imply distinct runtime objects (assuming sound points-to sets). □

Theorem 1 directly implies the following corollary, therefore proving our approximation sound.

Corrolary 1 (Condition 3 soundly approximates Condition 2)**.** For every joinpoint $j \in \mathcal{J}$ with $s := \mathit{shadow}(j)$, every dependency d and every dependent advice $a \in \mathcal{A}^d$, it holds that Condition 3 implies Condition 2.

This concludes the discussion of our static abstraction. In the following we give some additional detail about the actual implementation within the AspectBench Compiler.

3.3 Implementation in abc

Figure 3 gives an overview of our implementation of dependent advice as an extension "`abc.da`" to the AspectBench Compiler (*abc*). The user provides a Java base program as input, plus a set of aspects augmented with dependency annotations. In a first step, our compiler extension parses and type-checks the aspects and annotations. It then splits apart the dependency information from the aspects. *abc* then matches the resulting plain-AspectJ aspects against the base program, producing a "weaving plan". This plan holds information about which advice applies where in the program. *abc* next weaves the appropriate pieces of advice into the program (based on the weaving plan) and produces a woven program—still un-optimized. At this stage, our extension intercepts the compilation to analyze the woven program based on the previously extracted dependency annotations. For each potential match recorded in the weaving plan, we statically analyze if the dependencies for the matched advice can potentially be fulfilled at the matched program location. If not, then we remove this potential match from the plan. After the analysis finishes, we re-weave the entire program, i.e. we instruct *abc* to un-do the previous weaving process and weave the base program again, this time with the updated weaving plan. After the program was re-woven, *abc* automatically emits Java bytecode for the woven (and now optimized) program. We next explain the internals of the analysis, highlighted in Figure 3.

As mentioned earlier, our analysis executes right after weaving, analyzing the woven program. It has access to the base program, all aspects, all dependent advice in these aspects, and *abc*'s weaving plan. The weaving plan $\mathcal{W}$ contains a list of tuples (s, a, r) where s is a joinpoint shadow, a is an advice applying at s, and r the dynamic residue that the runtime will evaluate to determine whether a must indeed execute at a concrete joinpoint induced by s.

Quick-check.

Our analysis iterates through the weaving plan, considering each entry separately, first using the "Quick-check" shown in Algorithm 1. The Quick-check modifies the residue of an entry $(s, a, r) \in \mathcal{W}$ to (s, a, Never) if no advice dependency d activates a at s for the trivial reason that at least one strong advice b in d matches *nowhere* in the entire program, as determined by the weaving plan, line 9. Note that the condition in line 9 depends on whether the algorithm already processed weaving-plan entries for b itself. We therefore iterate Algorithm 1 until a fixed point is reached. The

AOSD'09, ACM, pp 3--14. 2009.

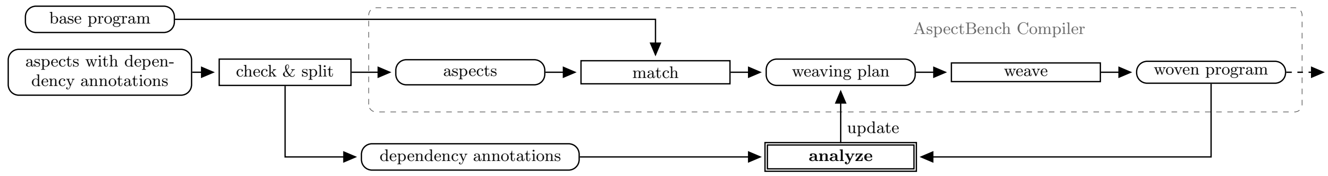


Figure 3: Overview of our implementation of dependent advice as an extension "abc.da" to *abc*

Algorithm 1 Quick-check

for $(s_a, a, r_a) \in \mathcal{W}$ **do**
  if $(r_a \neq \text{Never}) \wedge (a$ is dependent advice$)$ **then**
    $activated$:= **false**
    for $d \in \mathcal{D}$ with $a \in \mathcal{A}^d$ **do**
      $allStrongAdviceMatch$:= **true**
      for $b \in strong(d)$ **do**
        for $s_b \in \mathcal{S}$ **do**
          if $\neg\exists(s_b, b, r_b) \in \mathcal{W} \;:\; r_b \neq \text{Never}$ **then**
            $allStrongAdviceMatch$:= **false**
          end if
        end for
      end for
      if $allStrongAdviceMatch$ **then**
        $activated$:= **true**
      end if
    end for
    if $\neg activated$ **then**
      $\mathcal{W} := (\ \mathcal{W} \setminus \{(s_a, a, r_a)\}\ ) \ \cup\ \{(s_a, a, \text{Never})\}$
    end if
  end if
end for

Quick-check is "quick" because it does not require points-to information. In our benchmarks it therefore always finished in under 3.3 seconds.

If active advice applications remain after the Quick-check, then we next apply Sridharan and Bodík's demand-driven refinement-based context-sensitive points-to analysis [25] to the woven program. This analysis first produces context-insensitive points-to sets using Spark [18]. Then next, when queried for the points-to sets of a local variable l the analysis refines the points-to sets for l with context information. In previous work [7] we found that context information is necessary to effectively optimize pieces of advice that reference objects created inside factory methods, e.g. iterators, which are produced by a call to `iterator()`. Because we query the analysis only on variables that actually bind values at joinpoint-shadows of dependent advice, this demand-driven approach usually executes a lot faster than an analysis that determines context information for every program variable.

Flow-insensitive Orphan-shadows analysis.

We then apply a flow-insensitive "Orphan-shadows" analysis, shown in Algorithm 2. The algorithm essentially proceeds like the Quick-check (Algorithm 2 only shows the differences to Algorithm 1), however an advice a only activates a dependency d if every strong advice b of d has a shadow that is compatible with s_a, as determined by *sCompt*. Again we iterate Algorithm 2 until we reach a fixed point. This iteration is no bottle-neck: in all our experiments we reached the fixed point after two or three iterations. We named the

Algorithm 2 Flow-insensitive Orphan-shadows analysis (only showing differences to Algorithm 1)

...
          if $\neg\exists(s_b, b, r_b) \in \mathcal{W} \;:\; r_b \neq \text{Never} \vee$
           $sCompt(s_a, a, s_b, b, d) =$ **false** **then**
            $allStrongAdviceMatch$:= **false**
          end if
...

analysis Orphan-shadows analysis because it identifies shadows that are lacking other shadows to activate any dependency, and disables advice applications at these shadows.

4. GENERATING DEPENDENT ADVICE

The above optimizations assumed dependency annotations in the code. Programmers may write dependency annotations by hand but this can be time consuming and error prone. Fortunately, programmers often opt to have history-based aspects generated automatically, from finite-state monitor specifications. Runtime-monitoring tools generate state-machines from such specifications, along with aspects that trigger state transitions when monitored events occur. The state machine then executes a user-defined piece of code when those transitions drive it into a final state. If specifications bind free variables, there exists one state-machine instance per variable binding.

4.1 Generation from finite-state machines

We next present a general algorithm that exploits domain knowledge in given a finite-state specification to generate sound dependency annotations automatically.

Definition 1 (Finite-state machine)**.** A *finite-state machine* $\mathcal{M}$ is a tuple $(Q, \Sigma, q_0, \Delta, Q_F)$, where Q is a set of states, Σ is a set of input symbols, q_0 the initial state, $\Delta \subseteq Q \times \Sigma \times Q$ the transition relation and $Q_F \subseteq Q$ the set of accepting (or final) states. For this paper we assume that $q_0 \notin Q_F$. Further, one can easily transform $\mathcal{M}$ into an equivalent finite-state machine in which accepting states have no outgoing transitions and we assume that $\mathcal{M}$ has this form.

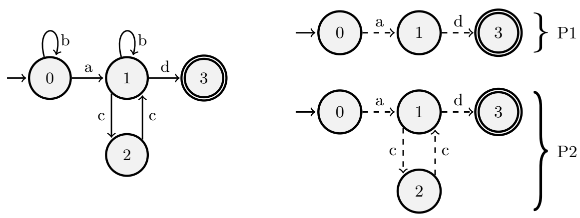
Definition 2 (Words and runs)**.** A *word* $w = (a_1, \ldots, a_n)$ is an element of Σ^*. We define a *run* ρ of $\mathcal{M}$ on w to be a sequence $(q_0, q_{i_1}, \ldots, q_{i_n})$ such that $\forall k \;:\; (0 \leq k < n) \rightarrow (q_{i_k}, a_{k+1}, q_{i_{k+1}}) \in \Delta$, with $i_0 := 0$. A run ρ is *accepting* if $q_{i_n} \in Q_F$. We say that $\mathcal{M}$ accepts w, $w \in \mathcal{L}(\mathcal{M})$, if there exists an accepting run of $\mathcal{M}$ on w. We assume that both words and runs are non-empty, i.e. that $n \geq 1$.

Algorithm 3 (page 7) defines the function *genDeps* which generates dependency declarations from $\mathcal{M}$. The programmer initializes the algorithm by calling $genDeps(q_0, \emptyset, \{\})$. Intuitively, *genDeps* recursively explores $\mathcal{M}$ in a depth-first manner to find all paths p through $\mathcal{M}$ that satisfy the following conditions: (1) the path ends in an accepting state (line

Algorithm 3 *genDeps*(q, p, c), with $q \in Q, p \in \mathcal{P}(Q \times \Sigma)$, $c : Q \rightharpoonup \mathbb{N}$

Global variables: $\mathcal{D} := \emptyset$

1: **if** $c(q) \leq 1$ **then**
2: $\quad c' :=$ copy of c; $c'(q) := c'(q) + 1$
3: $\quad$ **if** $q \in Q_F$ **then**
4: $\quad\quad strong := \{a \in \Sigma \mid \exists q \in Q \ : \ (q, a) \in p\}$
5: $\quad\quad Q_p := \{q \in Q \mid \exists a \in \Sigma \ : \ (q, a) \in p\}$
6: $\quad\quad weak := \{a \in \Sigma \mid a \notin strong \wedge \exists q \in Q_p \backslash Q_F \ : \ (q, a, q) \notin \Delta\}$
7: $\quad\quad \mathcal{D} := \mathcal{D} \cup \{(strong, weak)\}$
8: $\quad$ **end if**
9: $\quad$ **for** $a \in \Sigma$ **do**
10: $\quad\quad p' := p \cup \{(q, a)\}$
11: $\quad\quad$ **for** $q' \in Q$ such that $q' \neq q \wedge (q, a, q') \in \Delta$ **do**
12: $\quad\quad\quad$ genDeps(q', p', c')
13: $\quad\quad$ **end for**
14: $\quad$ **end for**
15: **end if**



(a) Example state machine (b) Paths determined by *genDeps*

for P1 : **dependency**{ **strong** a,d; **weak** c; } *//D1*
for P2 : **dependency**{ **strong** a,c,d; **weak** b; } *//D2*

(c) Dependency declaration generated for state machine

Figure 4: An example run of Algorithm 3

3), (2) it does not contain *self-loops* (line 11) and (3) it does not visit a state more than twice (line 1). ((3) assures that we visit each edge only once, assuring termination.) When *genDeps* finds such a path p, it adds a new dependency declaration to the global set $\mathcal{D}$. The dependency references the labels of all edges on p as strong. Further, it references all those symbols a as weak, which are not already strong on p and for which there is some non-final state on p that has no a-self-loop in $\mathcal{M}$.

Figure 4 shows an example run of Algorithm 3. 4(a) gives a state machine $\mathcal{M}$. 4(b) shows the two paths P1 and P2 that Algorithm 3 discovers; 4(c) shows the two resulting dependency declarations: D1 for P1 and D2 for P2. D1 does not reference b because b causes in $\mathcal{M}$ self-loops on all non-final states along P1. D2, however, includes b because state 2 has no b-self-loop: if $\mathcal{M}$ reads b while in state 2, $\mathcal{M}$ will discard the partial match.

Assume now a program in which the advice that normally triggers symbol c never matches at any joinpoint shadow. c is necessary to reach the accepting state via P2. Therefore c is strong in D2, and thus D2 is not activated for this program. Hence, there is no active dependency that references b, and it is safe to not monitor b.

4.1.1 Correctness and Complexity of Algorithm 3

One can prove that Algorithm 3 is correct, meaning that it generates dependency declarations that are both sound and complete, or in other words, the runtime monitor without dependency annotations accepts exactly the same words as the same monitor augmented with dependency annotations. Due to space limitations we present the proof in our accompanying Technical Report [5].

The theoretical worst-case complexity of Algorithm 3 is exponential in size of Δ and linear in the size of Σ. However, our experiments show that, for usual specifications, Δ will be very small: Algorithm 3 never generated more than nine dependencies for our example specifications. It always terminated within milliseconds.

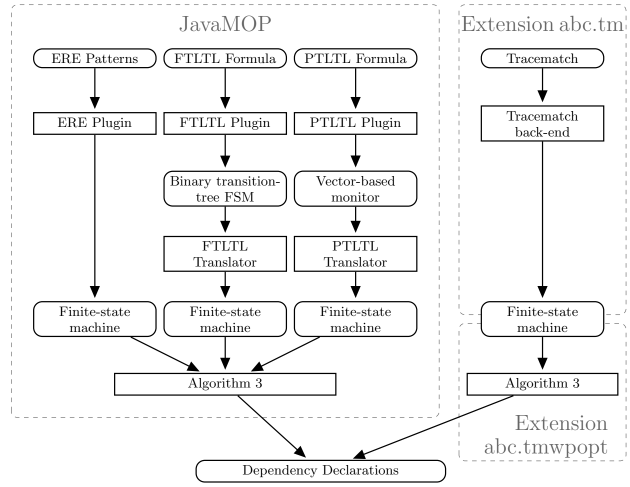


Figure 5: Generating dependent advice in *JavaMOP* and *abc*

4.1.2 Stability of Algorithm 3

In our Technical Report we further prove that *genDeps* is *stable*, i.e. that it computes equivalent sets of dependency declarations for equivalent finite-state machines:

Theorem 2 (Algorithm 3 is stable)**.** Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be finite-state machines with $\mathcal{L}(\mathcal{M}_1) = \mathcal{L}(\mathcal{M}_2)$. Let $\mathcal{D}_1$ and $\mathcal{D}_2$ be the set of dependency declarations that Algorithm 3 computes for $\mathcal{M}_1$ respectively $\mathcal{M}_2$. Then it holds that $\mathcal{D}_1 \equiv \mathcal{D}_2$, i.e. both dependency sets are logically equivalent.

4.2 Implementation in JavaMOP

The left-hand side of Figure 5 illustrates our implementation in *JavaMOP*. *JavaMOP* provides an extensible logic framework for specification formalisms [10, 11]. Via logic plug-ins, one can easily add new logics into *JavaMOP* and then use these logics within specifications. *JavaMOP* has several specification formalisms built-in, including extended regular expressions (ERE), past-time and future-time linear temporal logic (PTLTL/FTLTL), and context-free grammars. In this paper we focus on generating dependency information for ERE, PTLTL and FTLTL. Those three logics are finite-state, which allowed us to implement algorithms to translate the monitor generated from a ERE, FTLTL or PTLTL specification into a finite-state machine as defined in Definition 1.

ERE. The monitoring code generated by the ERE plug-in in *JavaMOP* is already a standard finite-state machine [10].

FTLTL. *JavaMOP*'s FTLTL plug-in outputs a binary transition tree finite-state machine (BTT-FSM) [10]. A BTT-

FSM is a state machine in which each state holds a Binary Transition Tree, i.e. a Boolean function. The BTT-FSM determines the target state of a transition by computing this Boolean function when an event is received. We translate a BTT-FSM into a standard finite-state machine by symbolically computing its BTTs exhaustively in each state.

PTLTL. Unlike the ERE plug-in and the FTLTL plug-in, the PTLTL plug-in in *JavaMOP* generates a monitor which has a vector of bits as its internal state [10]. We implemented an algorithm to exhaustively explore all possible states of the PTLTL monitor in order to construct an equivalent finite-state machine.

JavaMOP next applies the general Algorithm 3 to obtain the dependency information from the state machine. Every *JavaMOP* monitor supports both validation and violation handlers. *JavaMOP* executes a monitor's validation handler when the monitor accepts a trace, and its violation handler when it rejects a (partial) trace. We generate dependency declarations for validation handlers using Algorithm 3 directly. For a violation handler we instead fix $Q_F := \{q_r\}$, where q_r is the state from which no accepting state can be reached. *JavaMOP* uses minimized deterministic state machines and therefore q_r is unique. We then emit the appropriate set of dependencies, depending on whether the monitor uses only a validation handler, only a violation handler, or both.

JavaMOP writes AspectJ source code to disk. Our extension to *JavaMOP* adds dependency declarations to this output and also modifies the output so that each generated piece of advice is given a unique name. The dependency declarations reference those names. In a second step, the programmer can then use the dependent-advice extension of *abc* to read this generated code again from disk and weave monitoring code into a base program of her choice, making full use of the optimizations that we explained in Section 3.

4.3 Implementation for tracematches

Tracematches use yet another data structure to implement their runtime monitors: they use constraints [1]. A constraint $x = o \wedge y \neq p$ on a state q encodes that every binding that maps tracematch variable x to object o, and does not map tracematch variable y to object p, is in q. This allows tracematches to get around a current restriction of *JavaMOP*: In *JavaMOP*, programmers may only specify properties that bind *all* free variables to objects on the *first* observed event. As we show in Section 5, this makes it impossible to express some monitor specifications in *JavaMOP*.

The nature of these automata gives them a different structure from *JavaMOP*'s automata. *JavaMOP*'s automata are deterministic and minimized, and therefore have a unique reject state (the only state from which no final state can be reached). Tracematches however use non-deterministic automata. They reject traces using "skip-loops" [1]. Every state q holds a skip-loop with label a for every a for which q has no "normal" a-self-loop. In addition, the initial state q_0 of a tracematch state machine has no loops because the tracematch back-end assumes a Σ-loop on q_0 implicitly.

Despite these differences we can still use Algorithm 3 when transforming the state machine first: For each $a \in \Sigma$ we add an a-loop to q_0; and we remove all skip-loops. When defining Algorithm 3 we took extra care to formulate it in a way such that it is directly applicable to the resulting state machine.

Another notable difference of our tracematch-based implementation compared to *JavaMOP* is that for tracematches we never write AspectJ source code to disk. Tracematches, like dependent advice, are implemented as an extension to *abc*, and they generate history-based aspects directly in the form of Java bytecode. We therefore enhanced the *abc* extension "`abc.tm`" for tracematches with another extension "`abc.tmwpopt`" for whole-program optimization (see the right-hand side of Figure 5). This extension injects dependency annotations directly into the back-end of our *abc* extension "`abc.da`" for dependent advice (after the "split" in Figure 3). Every advice generated from a tracematch already carries a unique name, so we can re-use those names when we generate the dependency declarations.

5. EXPERIMENTS

To validate our approach we applied a set of twelve specifications for runtime monitoring to the ten benchmarks of the current version 2006-10-MR2 of the DaCapo benchmark suite [4]. We sought to determine whether or not dependent advice can indeed yield a significantly lower runtime overhead than normal advice in history-based aspects, and if so, whether this optimization effect depends on the code generation tool or specification formalism.

We first implemented all twelve specifications as tracematches, re-using some specifications from previous work [7]. Then we implemented plain AspectJ aspects for the same specifications by hand. Hand-writing such aspects proved time-consuming. It was particularly hard to assure proper memory-management: In many but not all cases [3], an aspect should discard monitoring state for objects that are garbage collected. In a second step, we augmented the hand-coded aspects with dependency annotations, which appeared comparatively simple. Next we wrote monitor specifications in the "extended regular expressions" (ERE) syntax for *JavaMOP*. *JavaMOP* currently assumes that the *first* monitored event in each specification binds *all* of the specification's variables. Four of the twelve specifications do not fulfil this requirement and *JavaMOP* therefore cannot express these specifications.

In the case of *JavaMOP* we were also interested to see whether the choice of specification formalism impacts the optimization results (as opposed to the choice of code generation tool). We therefore implemented the three specifications FailSafeIter, HasNext and LeakingSync not only in ERE but also in PTLTL and FTLTL. For each monitor specification we had *JavaMOP* generate history-based aspects with dependency annotations.

Altogether this gave us 26 history-based aspects (from *JavaMOP* and hand-coded) and 12 tracematches. (Remember that *abc* does not generate history-based aspects for tracematches in the form of source code, but rather both generates and consumes dependency annotations right in its back-end.) We compiled each of the ten DaCapo benchmarks with all 38 inputs, one at a time, for now with optimizations for dependent advice *disabled*. When optimizations are disabled, our compiler extension treats dependent advice just as normal advice. To establish a baseline, we further compiled each of the ten benchmarks without any aspects present. Altogether this gave us ten unwoven programs and 380 woven program versions.

Because we felt that it would be overwhelming to report results for so many programs, we first performed a simple triage: We ran each of the 380 woven programs and deter-

ASyncIter *	only iterate a synchronized collection c when holding a lock on c
ASyncIterM *	only iterate a synchronized map m when holding a lock on m
FailSafeEnum	do not update a vector while iterating over it at the same time
FailSafeEnumHT	do not update a hash table while iterating over its keys or elements
FailSafeIter	do not update a collection while iterating over it at the same time
FailSafeIterM *	do not update a map while iterating over its keys or values
HasNext	always call hasNext before calling next on an Iterator
LeakingSync	only access a synchronized collection or map 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: Monitor specifications that applied to our benchmarks (benchmarks with "*" cannot be expressed in *JavaMOP*)

mined their runtime overhead over the runtime of the respective unwoven program. This way we could determine 72 woven programs that showed a runtime overhead of more than 10%. The remainder of our discussion focuses on these 72 cases. For two of our twelve specifications, the runtime overhead was always below 10%. Table 1 gives informal descriptions of the remaining ten specifications.

5.1 Number of advice applications

We compiled all these 72 cases again, this time with optimizations for dependent advice *enabled*. We show the complete results of our experiments in Table 2 on page 10. The first column states the specification formalism; the names of the benchmark and specification follow. Columns four to seven show the number of advice applications that are enabled, i.e. the number of entries in the weaving plan for dependent advice that have a residue different from Never. We report (1) the initial number of advice applications, (2) the number after applying the Quick-check, (3) the number of advice applications at shadows reachable from the benchmark's main class, and (4) the number of advice applications enabled after applying the flow-insensitive Orphan-shadows analysis. The analysis ends early when the Quick-check already disables all advice applications, and therefore does not determine (3) and (4) in such cases.

Our results show that the Quick-check is very successful for the patterns that involve synchronized collections. This is not surprising: All benchmarks except hsqldb, lucene and xalan are single-threaded and therefore create no synchronized collections at all. The other specifications usually have *some* matches for all strong advice. This is not surprising either, because we here only consider benchmarks with 10% runtime overhead or more. When all strong advice match, the Quick-check is insufficient.

The flow-insensitive analysis stage is very successful in specifications that use multiple free variables, such as FailSafe*, Reader and Writer. It is less successful for specifications that only use a single variable, such as HasNext. The reason is simple: HasNext binds a single iterator and our optimization can only affect iterators on which a programmer invokes `hasNext()` but never `next()`. This is rarely the case. Flow-sensitivity is required [8,23] to handle such specifications more precisely. Other cases like bloat-FailSafeIter, are notoriously [7,23] hard to handle because they use very long-lived objects, dynamic class loading or reflection. All these cases lead to many overlapping points-to sets, impeding our analysis.

In Section 4.1.2 we mentioned that the way in which we generate dependency annotations is stable. As a result, the effectiveness of the analysis does not depend on the source of the dependency annotations. There are generally more advice applications for tracematches than for *JavaMOP*, simply because tracematches generate two to three additional advice applications shadow: tracematches use two additional advice for monitor synchronization and one to execute the tracematch body [1, S. 4.7]. Nevertheless, the fraction of removed advice applications is virtually equal for equal benchmarks and specifications, independent of the code generation tool and specification formalism. In case of *JavaMOP*, the number of disabled advice applications is not only similar but equal for equal benchmarks and specifications in all of ERE, PTLTL and FTLTL.

5.2 Reduction of runtime overhead

A reduction in the number of an aspect's advice applications does not necessarily reflect a 1:1 reduction of the runtime overhead caused by the aspect: If many optimized advice applications resided in dead code or code that is barely executed, then the overhead may remain unaffected. We therefore measured the actual runtime overhead of the optimized woven program over the un-woven program. The eighth column in Table 2 shows the runtime of the un-woven program (our baseline) in seconds, columns nine and ten show the runtime overhead for the un-optimized, respectively optimized version over this baseline in percent. A value of >10h means that the benchmark ran longer than ten hours and was aborted after this period of time. Overheads above 10% appear in boldface. We ran the benchmarks on Sun's HotSpot VM (build 1.4.2_12, mixed mode), with 2GB of maximal heap space on a machine with an AMD Athlon 64 X2 Dual Core Processor 3800+ running Ubuntu 7.10 (kernel version 2.6.22-14).

Our optimizations were able to bring the overhead below 10% in 44 out of all 72 cases. Of the remaining cases there were a few with significant reductions, e.g. FailSafeIterM. However, the benchmarks where our analysis failed to disable advice applications naturally show the same runtime overhead before and after optimizations. None of the optimized benchmarks runs significantly slower than the unoptimized versions, indicating that our implementation is sound. Again, the choice of code generation tool and formalism seems to have only a qualitative impact. Hand-coded aspects are usually the fastest. After all, a programmer can exploit domain knowledge which cannot be encoded in current monitoring specifications. For instance a programmer knows that every Java iterator is only ever associated with a single collection, and can therefore use an optimized data structure, e.g. a mapping from iterators to collections. Yet, the *relative* reduction in runtime caused by our optimizations is consistent over all specification languages and tools.

AOSD'09, ACM, pp 3--14. 2009.

formalism	benchm.	specification	initial	quick	reachable	flow-ins.	baseline (s)	un-opt. (%)	opt. (%)
tracematches	antlr	Reader	167	167	137	74	4.1	**398**	**21**
tracematches	antlr	Writer	475	270	57	48	4.1	**209**	1
tracematches	bloat	ASyncIter	5977	0			9.002	**1569**	2
tracematches	bloat	ASyncIterM	6166	0			9.002	**2367**	0
tracematches	bloat	FailSafeIter	5065	5065	4019	3991	9.002	**9150**	**9201**
tracematches	bloat	FailSafeIterM	3832	3832	3505	2927	9.002	**>10h**	**16533**
tracematches	bloat	HasNext	3328	3328	2254	2254	9.002	**1680**	**1692**
tracematches	bloat	LeakingSync	8595	0			9.002	**215**	8
tracematches	bloat	Writer	1774	1774	762	537	9.002	**449**	**452**
tracematches	chart	LeakingSync	3695	0			14.651	**88**	-1
tracematches	fop	FailSafeEnumHT	635	635	117	0	2.398	**12**	6
tracematches	fop	FailSafeIterM	2265	2265	193	193	2.398	**15**	**13**
tracematches	fop	LeakingSync	9403	0			2.398	**241**	5
tracematches	fop	Writer	1429	905	281	0	2.398	**16**	2
tracematches	jython	FailSafeEnumHT	539	539	539	480	11.054	**170**	**47**
tracematches	jython	FailSafeIterM	538	538	490	353	11.054	**17**	5
tracematches	lucene	FailSafeEnum	218	218	155	117	30.878	**20**	4
tracematches	lucene	FailSafeIter	732	732	432	386	30.878	**18**	7
tracematches	lucene	LeakingSync	2631	2631	1713	0	30.878	**206**	0
tracematches	lucene	Reader	557	557	154	0	30.878	**56**	1
tracematches	pmd	ASyncIter	2213	0			13.059	**36**	3
tracematches	pmd	ASyncIterM	2354	0			13.059	**53**	-1
tracematches	pmd	FailSafeIter	1823	1823	1700	1606	13.059	**139**	**25**
tracematches	pmd	FailSafeIterM	2078	2078	2014	1836	13.059	**>10h**	**>10h**
tracematches	pmd	HasNext	1223	1223	1009	1006	13.059	**70**	**71**
tracematches	pmd	LeakingSync	3959	0			13.059	**15**	2
hand-coded	bloat	ASyncIter	1387	0			9.002	**153**	4
hand-coded	bloat	ASyncIterM	1450	0			9.002	**159**	8
hand-coded	bloat	FailSafeIter	1535	1535	1047	1025	9.002	**390**	**388**
hand-coded	bloat	FailSafeIterM	1120	1120	745	220	9.002	**100621**	**2534**
hand-coded	bloat	HasNext	947	947	641	641	9.002	**1328**	**1322**
hand-coded	bloat	LeakingSync	2145	0			9.002	**39**	5
hand-coded	bloat	Writer	1153	1153	652	650	9.002	**36**	**34**
hand-coded	chart	LeakingSync	920	0			14.651	**29**	0
hand-coded	fop	FailSafeEnumHT	205	205	19	0	2.398	**13**	3
hand-coded	fop	LeakingSync	2347	0			2.398	**69**	**14**
hand-coded	lucene	FailSafeEnum	61	61	43	5	30.878	**15**	2
hand-coded	lucene	LeakingSync	652	652	368	0	30.878	**48**	0
hand-coded	lucene	Reader	136	136	33	0	30.878	**28**	1
hand-coded	pmd	ASyncIterM	556	0			13.059	**11**	0
hand-coded	pmd	FailSafeIterM	483	483	360	186	13.059	**551**	**274**
hand-coded	pmd	HasNext	346	346	246	243	13.059	**33**	**36**
MOP-ERE	antlr	Reader	64	64	53	20	4.1	**98**	4
MOP-ERE	antlr	Writer	273	217	33	18	4.1	**79**	4
MOP-ERE	bloat	FailSafeIter	1526	1526	1038	1010	9.002	**1563**	**1572**
MOP-ERE	bloat	HasNext	947	947	641	641	9.002	**1460**	**1452**
MOP-ERE	bloat	LeakingSync	2145	0			9.002	**58**	0
MOP-ERE	bloat	Writer	663	663	308	21	9.002	**119**	**112**
MOP-ERE	chart	LeakingSync	920	0			14.651	**58**	-1
MOP-ERE	fop	LeakingSync	2347	0			2.398	**124**	4
MOP-ERE	lucene	LeakingSync	652	559	330	0	30.878	**51**	1
MOP-ERE	pmd	FailSafeIter	546	546	367	273	13.059	**41**	-4
MOP-ERE	pmd	HasNext	346	346	246	243	13.059	**52**	**40**
MOP-PTLTL	antlr	LeakingSync	170	0			4.1	**12**	4
MOP-PTLTL	bloat	FailSafeIter	1526	1526	1038	1010	9.002	**1322**	**1318**
MOP-PTLTL	bloat	HasNext	947	947	641	641	9.002	**1058**	**1033**
MOP-PTLTL	bloat	LeakingSync	2145	0			9.002	**275**	4
MOP-PTLTL	chart	LeakingSync	920	0			14.651	**84**	0
MOP-PTLTL	fop	FailSafeIter	288	288	50	0	2.398	**16**	-1
MOP-PTLTL	fop	LeakingSync	2347	0			2.398	**217**	3
MOP-PTLTL	lucene	LeakingSync	652	559	330	0	30.878	**102**	-2
MOP-PTLTL	pmd	FailSafeIter	546	546	367	273	13.059	**26**	-4
MOP-PTLTL	pmd	HasNext	346	346	246	243	13.059	**18**	**19**
MOP-FTLTL	bloat	FailSafeIter	1526	1526	1038	1010	9.002	**1599**	**1561**
MOP-FTLTL	bloat	HasNext	947	947	641	641	9.002	**1633**	**1641**
MOP-FTLTL	bloat	LeakingSync	2145	0			9.002	**58**	0
MOP-FTLTL	chart	LeakingSync	920	0			14.651	**58**	0
MOP-FTLTL	fop	FailSafeIter	288	288	50	0	2.398	**13**	**12**
MOP-FTLTL	fop	LeakingSync	2347	0			2.398	**123**	1
MOP-FTLTL	lucene	LeakingSync	652	559	330	0	30.878	**53**	-1
MOP-FTLTL	pmd	FailSafeIter	546	546	367	273	13.059	**37**	-2
MOP-FTLTL	pmd	HasNext	346	346	246	243	13.059	**43**	**47**

Table 2: The results of our experiments: number of advice applications initially in the program, after Quick-check, reachable from the program's main class, and after the flow-insensitive Orphan-shadows analysis; runtime of the un-woven program in seconds, runtime overhead of the woven un-optimized and optimized program; overheads above 10% appear in boldface

5.3 Compilation and analysis time

We ran our static optimizations on IBM's J9 VM (build 2.3, J2RE 1.5.0 IBM J9 2.3 Linux amd64-64), with 3GB of maximal heap space. Space limitations prevent us from including detailed compilation times, nevertheless we wish to give a brief overview. The Quick-check took never longer than 3.3 seconds, on average it took 148 milliseconds. The Flow-insensitive analysis took never longer than 17 seconds, with an average of 1.4 seconds. A large factor is however the points-to analysis that the flow-insensitive stage requires. Computing points-to sets and context information can be costly, and largely depends on the benchmark. In the worst case, bloat-FailSafeIter, it took 58 minutes to compute. This benchmark has many more shadows than any other benchmark and we therefore need to query the demand-driven points-to analysis more often. On average, the points-to analysis took 11 minutes. This may appear long, yet many of our un-optimized benchmarks showed several minutes overhead too. Optimizations clearly pay off in these cases.

Our Technical Report gives additional information about the limitations of our approach.

5.4 Discussion

To conclude, dependent advice come at some compile-time cost, however their use can yield significant run-time improvements. The success of the optimization depends on the property that the history-based aspect monitors and on the monitored program, but not on the particular monitor implementation.

6. RELATED WORK

We next compare our work to earlier work on optimizations for runtime monitoring, discuss how our work can be applied to other aspect-generating tools and how it relates to dataflow pointcuts. In our Technical Report we also compare to maybeShared pointcuts [6] and LogicAJ [16].

Flow-insensitive tracematch optimizations. Our work was largely inspired by earlier work of Bodden et. al [7]. They were the first to propose a Quick-check and a flow-insensitive pointer-based analysis to remove unnecessary monitor instrumentation, however their approach was bound to tracematches only. The goal of this work was to distill the essence of their approach and make the same powerful optimizations available to history-based aspects generated from other sources (including hand-written aspects), while at the same time not compromising on the good results that the authors obtained for tracematches earlier. Our approach achieves exactly that: dependent advice allow optimizations to be successful independently of the chosen code-generation tool or specification formalism.

Flow-sensitive tracematch optimizations. Bodden et al. also proposed a second optimization [8] for tracematches that is intra-procedural and flow-sensitive. Naeem and Lhoták independently developed a fully inter-procedural flow-sensitive version [23]. Flow-sensitive approaches are potentially more precise than flow-insensitive ones, however they require significantly more domain knowledge. A minimal extension of dependent advice that encodes flow information would be an interesting area for future work.

Monitor optimizations. Avgustinov et al. [3] proposed optimizations to the runtime monitor 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 all our experiments. The authors' optimizations are however orthogonal to ours. With leak elimination and indexing disabled, our speedups would likely have been even more significant, as there would have been more overhead to remove. JavaMOP and PTQL [14] implement weaker variants of these optimizations.

Association aspects and relational aspects. Sakurai et al. [24] 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 [9] 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 from locations where the objects involved are known not to be associated with `A`. Further, for objects for which no advice in the relational aspect can ever execute, the optimization will remove the call to the code that associates the object with the aspect in the first place.

S2A and M2Aspects. Maoz and Harel proposed S2A, a tool [19] to generate executable AspectJ code from Live Sequence Charts [13] (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 [17] 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.

Dataflow pointcuts. Masuhara and Kawauchi proposed a pointcut `dflow` [21], to be used as p `&& dflow[`s,t`](`q`)`, which matches if data flows from s to t, where p is a pointcut binding s, and q is an inner pointcut binding t. `dflow` is evaluated at runtime, i.e. it only matches if dataflow does indeed exist. The authors suggest however, to devise a static analysis that would optimize data-flow pointcuts at compile time. Unfortunately, dependent advice are not expressive enough for this purpose: Dependent advice are defined using tests of pointer equality, and the may-alias analysis in our optimization therefore regards only pointer assignments. In general, data-flow can however also comprise the flow of primitive values and flow arising from String concatenation.

7. CONCLUSIONS AND FUTURE WORK

In this work we presented *dependent advice*, a novel AspectJ language extension to aid the optimization of history-based aspects. Dependent advice augment normal AspectJ advice with dependency annotations. A dependent advice only needs to execute when its dependencies are fulfilled.

We implemented a static flow-insensitive whole-program analysis to approximate dependencies in the AspectBench Compiler. Based on the analysis results, the compiler can remove dispatch code for a dependent advice from locations at which the advice's dependencies cannot be fulfilled. As our results show, this optimization can significantly lower the runtime overhead of history-based aspects.

We modified code generators for specifications written in four finite-state formalisms. We made them exploit domain knowledge contained in the specification to automatically augment their generated AspectJ code with dependency annotations. The code generation is "stable", i.e. it generates equivalent dependency annotations from equivalent specifications, independent of the particular specification formalism. In result, the observed optimization effects are stable as well. We believe that similar code generation should be possible for any modelling or specification language over which reachability can efficiently be decided. It would be interesting future work to determine if one can generate annotations in a stable way for classes of these other languages too.

It also seems that programmers could benefit from our approach outside the scope of AspectJ, or even of aspect-oriented programming in general. We believe that a mechanism similar to dependent advice can be useful in any system that uses implicit invocation and where the whole program or system can be analyzed.

All tools, benchmarks, scripts and instructions required to reproduce our experimental results are available at:

`http://www.aspectbench.org/benchmarks/`

8. REFERENCES

[1] 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 *OOPSLA*, pages 345–364. ACM Press, Oct. 2005.

[2] 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 *AOSD*, pages 87–98. ACM Press, Mar. 2005.

[3] P. Avgustinov, J. Tibble, and O. de Moor. Making trace monitoring feasible. In *OOPSLA*, pages 589–608. ACM Press, Oct. 2007.

[4] 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.

[5] E. Bodden, F. Chen, and G. Roşu. Dependent advice: A general approach to optimizing history-based aspects (Extended version). Technical Report abc-2008-2, `http://www.aspectbench.org/`, March 2008.

[6] E. Bodden and K. Havelund. Racer: Effective race detection using AspectJ. In *ISSTA*, pages 155–165. ACM Press, July 2008.

[7] E. Bodden, L. J. Hendren, and O. Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *ECOOP*, volume 4609 of *LNCS*, pages 525–549. Springer, July 2007.

[8] E. Bodden, P. Lam, and L. Hendren. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In *FSE*. ACM Press, Nov. 2008. To appear.

[9] E. Bodden, R. Shaikh, and L. Hendren. Relational aspects as tracematches. In *AOSD*, pages 84–95. ACM Press, Mar. 2008.

[10] F. Chen and G. Roşu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. In *RV*, volume 89(2) of *ENTCS*, pages 108–127, July 2003.

[11] F. Chen and G. Roşu. MOP: an efficient and generic runtime verification framework. In *OOPSLA*, pages 569–588. ACM Press, Oct. 2007.

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

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

[14] S. Goldsmith, R. O'Callahan, and A. Aiken. Relational queries over program traces. In *OOPSLA*, pages 385–402. ACM Press, Oct. 2005.

[15] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In *AOSD*, pages 26–35. ACM Press, Mar. 2004.

[16] G. Kniesel, T. Rho, and S. Hanenberg. Evolvable pattern implementations need generic aspects. In *ECOOP 2004 Workshop on Reflection, AOP and Meta-Data for Software Evolution*, June 2004.

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

[18] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In *CC*, volume 2622 of *LNCS*, pages 153–169, Apr. 2003.

[19] S. Maoz and D. Harel. From multi-modal scenarios to code: compiling LSCs into AspectJ. In *FSE*, pages 219–230. ACM Press, Nov. 2006.

[20] M. Martin, B. Livshits, and M. S. Lam. Finding application errors using PQL: a program query language. In *OOPSLA*, pages 365–383, Oct. 2005.

[21] H. Masuhara and K. Kawauchi. Dataflow pointcut in aspect-oriented programming. In *1st Asian Symposium on Programming Languages and Systems*, volume 2895 of *LNCS*, pages 105–121. Springer, Dec. 2003.

[22] H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In *CC*, volume 2622 of *LNCS*, pages 46–60. Springer, Apr. 2003.

[23] N. A. Naeem and O. Lhoták. Typestate-like analysis of multiple interacting objects. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 347–366, New York, NY, USA, 2008. ACM.

[24] K. Sakurai, H. Masuhara, N. Ubayashi, S. Matsuura, and S. Komiya. Association aspects. In *AOSD*, pages 16–25, Mar. 2004.

[25] M. Sridharan and R. Bodík. Refinement-based

AOSD'09, ACM, pp 3--14. 2009.

context-sensitive points-to analysis for Java. In *PLDI*, pages 387–400, June 2006.