A Flexible Architecture for Pointcut-Advice Language Implementations

Christoph Bockisch Mira Mezini
Darmstadt University of Technology
Hochschulstr. 10, 64289 Darmstadt, Germany
{bockisch, mezini}@informatik.tu-darmstadt.de

Abstract

Current implementations for aspect-oriented programming languages map the aspect-oriented concepts of source programs to object-oriented bytecode. This hinders execution environments with dedicated support for such concepts in applying their optimizations, as they have to recover the original aspect definition from bytecode. To address this representational gap we propose an architecture for implementations of pointcut-advice languages where aspect-oriented concepts are preserved as first-class entities. In this architecture, compilers generate a model of the crosscutting which is executed by virtual machines.

In this paper we discuss a meta-model for aspect-oriented concepts and a virtual machine-integrated weaver for the meta-model. As a proof of concept, we also provide an instantiation of the meta-model with the AspectJ language and an AspectJ compiler complying with the proposed architecture. We also discuss how preexisting high-performance optimizations of aspect-oriented concepts benefit from the proposed architecture.

1. Introduction

This paper is concerned with the implementation of the pointcut-advice sub-family of aspect-oriented languages, PA languages [1] for short. Most aspect-oriented languages belong to this family as they provide the pointcut and advice constructs to support the modularization of behavioral crosscutting. For simplicity, we will use the term "aspect-oriented languages" to refer to the pointcut-advice sub-family where no distinction is necessary.

The most prevalent implementation strategies of PA languages share the following conceptual workflow [2, 3]. First, the aspect-oriented program is parsed to retrieve pointcuts and advice from aspect modules. Next, join point shadows [2, 4] are searched for. Finally, bytecode instructions for dispatching advice functionality are generated and inserted at these shadows. The latter two steps are generally referred to as the weaving phase.

During this phase aspect-oriented (AO) concepts, which are expressed by special language constructs in the source code, drive code generation and transformations in the compiler. One implication of this approach is that code originally modularized in aspects is merged with code of the base language's modules. An obvious problem with this is the weakening of the continuity property of incremental compilation: Modifying an aspect potentially requires re-weaving in multiple other modules [5]. Moreover, the aspect-oriented concepts become implicit instead of staying first-class in the generated bytecode.

This is different from object-oriented programming languages, e.g., Java, where concepts like classes, methods, fields and even polymorphism are also reflected in the bytecode. In Java bytecode, polymorphic method calls can be immediately identified because they are represented by a special instruction rather than general-purpose instructions encoding the resolution logic. In contrast, current implementations of aspect-oriented languages generate sequences of general-purpose instructions to realize concepts like the cflow pointcut designator [6].

This representational gap hinders a range of optimizations by the execution environment [7, 8]. It is very difficult task for the just-in-time compiler of a virtual machine to recognize the intention of a sequence of instructions generated by the weaver, e.g., for checking whether a control flow is active. On the contrary, a first-class representation of quantifications over the control-flow, can be exploited by the virtual machine's just-in-time compiler to perform optimizations during the native code generation [9]. Other optimizations of this kind are also outlined in [9]. Furthermore, with a first-class representation of aspect-oriented concepts, object layouts that better suit the needs of aspects can be designed in the virtual machine [10].

Approaches like the AspectBench Compiler framework (`abc`) [11, 12] apply static analyzes based on a first-class model of the aspects to achieve some performance optimizations. Yet, this first-class model also only exists during compile- and weaving-time and is abandoned afterwards. Optimized implementations that can rely on virtual machine features to achieve efficient runtime execution are out of reach.

Besides facilitating optimizations in the execution environment, a first-class representation of AO concepts also supports development of new language extensions. The `abc` framework already supports re-using and extending the implementation of some core concepts in static compilers. It exposes interfaces for join point shadow search and weaving, which can be used to realize new kinds of pointcut designators [11, 12]. However, other concepts, e.g., concerning the aspect instantiation strategy or advice precedence, are not first-class. Hence, language extensions that target these parts of an aspect-oriented language can not re-use implementation efforts provided by the `abc` framework. Aspect-oriented languages that, e.g., support runtime deployment of aspects, like JAsCo [13, 14] or AspectWerkz [15, 16], can not benefit from `abc` at all, because the latter lacks an abstraction over the aspect deployment strategy.

The work presented in this paper targets the problems outlined so far. We propose an architecture for aspect-oriented language

Workshop VMIL '07 March 12-13, 2007, Vancouver, British Columbia, Canada.
Copyright © 2007 ACM 1-59593-661-5/07/03…$5.00

implementations where the aspect-oriented concepts stay first-class until execution. Centric to this architecture is a meta-model of core aspect-oriented concepts which acts as interface between compilers and runtime environments[1]. Compilers *instantiate* the meta-model with language constructs while runtime environments *implement* the execution of the meta-model. The core concepts made explicit in the current meta-model are: advice, join point shadows, dynamic properties of matching join points, context reification at join points, strategies for implicitly instantiating aspects, advice ordering at shared join points and aspect deployment.

A concrete instantiation of the architecture is also presented, consisting of two parts. First, to show that the meta-model is appropriate to express state-of-the-art aspect-oriented languages, we provide a compiler that expresses the AspectJ language features as instantiations of the meta-model. Second, we provide an implementation of the meta-model by means of a Java 5 agent and Class Redefinition [17]. This implementation serves as an unoptimized default solution to execute the meta-model on any standard Java 5 virtual machine.

The default implementation targets only one of the problems identified with current implementations: It better supports the continuity of incremental compilation. To show how the proposal supports sophisticated optimizations of the AO concepts, we discuss a second optimized implementation of the model by a dedicated virtual machine. Thereby, we show how the first-class AO concepts can drive advanced optimizations by the just-in-time compilers of virtual machines. Finally, to show that the architecture and its meta-model supports language extensions, a mapping for some advanced AO concepts and optimized implementations thereof is outlined.

The remainder of this paper is organized as follows. In section 2, the proposed architecture of PA language implementations and its meta-model is presented. Section 3 discusses a concrete instantiation and the default implementation of the proposed meta-model. We discuss alternative instantiations of the architecture in section 4 and present related work in section 5. Finally, section 6 concludes the paper and presents areas of future work.

2. Architecture and Meta-Model

In this section, an overview of the proposed architecture is given and the underlying meta-model of pointcut-advice languages is presented.

2.1 Overview of the Architecture

The proposed architecture is schematically shown in figure 1. The central block of the architecture is the *meta-model of pointcut-advice languages* which has been designed to be generic enough to accommodate the concepts of most current PA languages. It is defined as a set of interfaces and classes in the Java language. A concrete language implementation has different connections to the meta-model.

Concrete AO language features are mapped to the meta-model by implementing the interfaces according to the feature's semantics, we speak of *instantiating the meta-model*. Mapping a concrete language to the meta-model results in the model for the respective language.

Compilers adhering to the architecture (in the front-end block of the figure) generate bytecode as usual for the non-aspect parts of the program, i.e., object-oriented modules and object-oriented parts of aspect-oriented modules. Additionally, they create bytecode, called the *preamble*, that instantiates and configures model entities according to the behavioral crosscutting definitions in the source code under compilation. The preamble is executed before running

[1] One could also call the meta-model an intermediate representation for aspect-oriented programming languages.

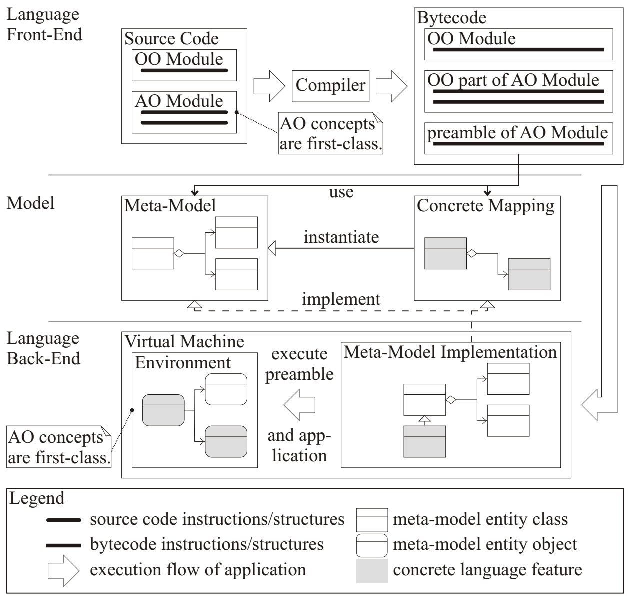

Figure 1. Schema of the proposed architecture for pointcut-advice language implementations.

the actual application. Its execution creates and possibly deploys the first-class representation of the crosscutting definitions.

Execution environments that adhere to the architecture (in the back-end block of the figure) *implement the meta-model*. Since all elements of the aspect-oriented programs are represented as objects at runtime, the execution environment has full access to a high-level description of the behavioral crosscutting defined by the AO program. This enables it to employ sophisticated optimizations. But also simple bytecode weaving as in existing implementations is possible. Beyond advanced optimizations, different implementations of the deployment interface can also provide advanced features such as synchronized or transactional aspect deployment.

In other words, the meta-model serves as the interface between AO compilers and execution environments, hence, decoupling the front-end from the back-end of an AO language. Besides providing the execution environment with first-class representations of AO concepts, this decoupling makes compilers and execution environments independently interchangeable: New implementations of the meta-model by new virtual machines can be used as the back-end for existing compilers, and the code produced by new compilers can execute on any execution environment that implements the meta-model.

2.2 A Meta-Model for Pointcut-Advice

The meta-model breaks down an aspect into small, differentiated units to improve re-usability and to avoid ambiguities. Figure 2 shows the elements of the model and their connections by means of a class diagram. The `Aspect` module provides a logical grouping of `AdviceUnits` – representing pointcut-advice pairs – to be deployed together. The other modules are concerned with expressing either pointcuts or advice functionality.

2.2.1 Expressing Pointcuts.

Pointcuts are represented as `JoinPointSet` data structures – the participating classes are marked by the box labeled *pointcut* in figure 2. A `JoinPointSet` corresponds to a simple clause in a pointcut expression of an aspect. A simple clause is one that consists of a single static pointcut designator, e.g., a call or a field access desig-

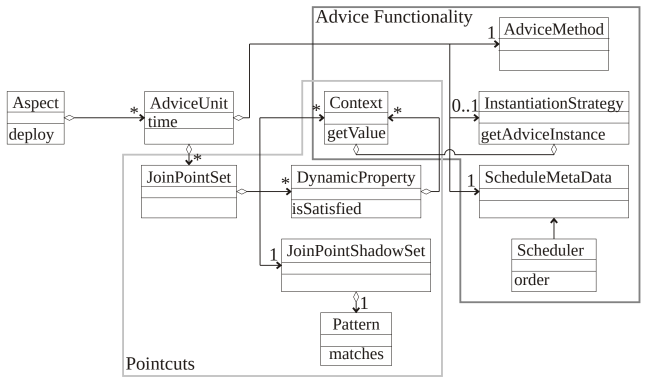

Figure 2. A meta-model for pointcut-advice languages.

nator, eventually combined with dynamic properties by an *and* (`&&`) operation. For instance, `call(pattern)&& dyn` is a simple pointcut clause where `call(pattern)` is a static pointcut designator selecting all calls to methods that match `pattern`, and `dyn` is some dynamic property that the method calls selected by the static designator must also fulfill.

Accordingly, each `JoinPointSet`, jps, has a static component, $jpss$ of type `JoinPointShadowSet`, representing its shadows [2], and a dynamic component, an set of `DynamicProperty` objects. A join point in jps occurs at runtime when an instruction in $jpss$ is executed and all dynamic properties are satisfied. In addition, a `JoinPointSet` may also refer to `Context` objects, representing values that are exposed to the advice at member join points.

`Context` objects represent some part of a join point's context – e.g., the `this` object or the source code location of a join point's shadow – and are used by several parts of the meta-model. Contexts can also be composed of other contexts, e.g., the `thisJoinPoint` value in AspectJ can be realized as a composed context.

A compound pointcut expression can be expressed in the meta-model by simple clauses combined with *or* (`||`) operators. An example of a pointcut in the *ored* form is `(call(pattern1)&& dyn1)|| (set(pattern2)&& dyn2)`. However, not all pointcuts in AspectJ-like pointcut languages are already in this form. For instance, the pointcut expression `call(pattern1)&& dyn && call( pattern2)` is not in the *ored* form. Yet such pointcut expressions can be transformed into the *ored* form. That is, the *ored* form does not constrain the valid pointcut expressions that can be defined.

The idea is that any pointcut expression that produces a non-empty join point set can be brought into an *ored* form. Given two pointcut designators, p_1 and p_2 with patterns pt_1 respectively pt_2, the pointcut p_1 && p_2 is equivalent to the pointcut p in *ored* form whose pattern is $pt_1 \wedge pt_2$.

For illustration, consider the following listing. It shows several pointcut definitions in a pseudo language, followed by an equivalent pointcut expression in *ored* form. The third example cannot be expressed in an *ored* form. However, this pointcut represents a family of pointcuts with empty join point sets. There can never be a join point which is a method call and a field set at the same time.

```
// 1.
// original pointcut definition :
call(pattern1) && dyn && call(pattern2)
// equivalent pointcut definition in ored form:
call(pattern1 && pattern2) && dyn

// 2.
// original pointcut definition :
(call(pattern1) || set(pattern2)) && dyn
// equivalent pointcut definition in ored form:
(call(pattern1) && dyn) || (set(pattern2) && dyn)

// 3.
// original pointcut definition :
(call(pattern1)) && (set(pattern2))
// this is an illegal pointcut
```

A pointcut in our meta-model is specified as an array of `JoinPointSets` which corresponds to a list of simple pointcut clauses combined with an *or* (`||`) operation.

An instance of `JoinPointShadowSet` refers to a `Pattern` object, which describes lexical properties of join point actions, e.g., the name or signature of the called method. A join point shadow set is, conceptually, evaluated by matching the lexical context of all join point shadows in the application against its pattern object.

Dynamic properties model the program's state at the time a join point is executed, e.g., the active control flow or the receiver object. A `DynamicProperty` can specify which values from the context of a join point are required for its evaluation by referring to respective `Context` objects.

2.2.2 Expressing Advice Functionality.

An `AdviceUnit` defines crosscutting functionality by specifying its *what*, *where*, and *when*. The *what* is defined in an `AdviceMethod` [2] and the *where* by a set of `JoinPointSets`. The `time` property of `AdviceUnit` (representing the *when*) determines whether the advice method is executed *before*, *after*[3] or *around* a join point. In the following, we will discuss the meta-model entities concerned with defining the *what*, marked in figure 2 by a box labled *advice functionality*.

Advice methods may need an object on which to execute – an *advice instance*. Generally, an advice may execute on different advice instances at different join points matched by its pointcut. For illustration, consider an AspectJ aspect declared with a `pertarget` clause and consisting of a call pointcut and an advice. The `pertarget` clause states that the advice of this aspect executes on different advice instances for different target objects at call join points matched by the pointcut.

The strategy for retrieving an advice instance is captured by `InstantiationStrategy`. To specify the values that it may need from a join point's context to retrieve an appropriate advice instance an instantiation strategy may refer to an arbitrary number of `Context` objects. When an advice method is executed at some join point, the required context values are determined and passed to the `InstantiationStrategy` which returns the advice instance on which to execute the advice method.

An advice unit also has `ScheduleMetaData` associated with it. `ScheduleMetaData` is used to determine the order of execution when multiple advice apply at the same join point. A simple form of schedule meta-data is a priority level associated with each advice unit. When two or more advice units are executed at the same join point, the one with the highest priority gets executed first. The `Scheduler` interface – which is responsible for generating a concrete order given some schedule meta-data – must be co-implemented with the schedule meta-data.

When multiple advice units must be executed at a join point shadow, the meta-model implementation will pass their schedule meta-data to the scheduler, which determines an ordering of the advice. The ordering is encoded as a chain of `AdviceOrderElement` objects. The structure of such a chain is defined by the class diagram in figure 3, whereby *action* is a reference to either an advice unit or the actual join point action.

[2] In the default implementation normal methods are used as advice methods.

[3] It is possible to specify that the advice executes after normal or exceptional execution of the join point.

Each `AdviceOrderElement` object stores a (possibly empty) list of *before* advice in the order of their execution, followed by an *around* advice, which, in turn, is followed by a (possibly empty) list of *after* advice to be executed after the *around* advice has finished. Any `AdviceOrderElement`, aoe_i, may be linked to a following element, aoe_{i+1}, thus, specifying the advice to execute, when the *around* advice of aoe_i **`proceed`**s. Once the execution of the last *after* advice of aoe_{i+1} is over, the *around* advice of aoe_i continues after its **`proceed`**, followed by the *after* advice of aoe_i, if any. The *around* of the last element of an `AdviceOrderElement` chain refers to the join point action rather than to an advice.

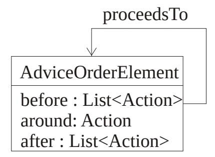

Figure 3. Data structure for representing the order of advice.

For illustration, consider a priority-based scheduler and three advice units - *A*, *B*, *C* - to be executed at the same join point shadow. *A* and *B* are *before* advice units with priorities 100, respectively 80; *C* is an *around* advice with priority 90. Further, assume that the scheduling strategy is such that a *before* advice with a lower priority than an *around* advice is executed only when the latter **`proceed`**s. Figure 4 shows the order structure returned by the scheduler for this example.

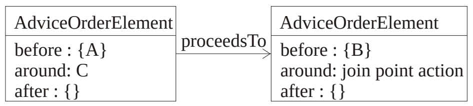

Figure 4. Example `AdviceOrderElement` data structure.

3. Default Instantiation of the Architecture

To show that the meta-model is appropriate for expressing aspect-oriented features of current languages and to illustrate the proposed architecture we will discuss an instantiation of the architecture, here. We have instantiated the meta-model with the AspectJ language features and implemented both, a compiler and an execution environment for the meta-model.

In the following subsection, we discuss how the AspectJ features are expressed as instantiations of the meta-model as well as the compiler for the AspectJ language that obeys the proposed architecture. In subsection 3.2 we present an implementation of an execution environment to execute the meta-model based on byte-code weaving. Dynamic aspect deployment is facilitated by means of a Java 5 agent and Class Redefinition [17].

3.1 Mapping AspectJ

In the following, we discuss how AspectJ features described in appendix B *Language Semantics* of the AspectJ Programming Guide [6] can be mapped to our meta-model. We also discuss the code generated by a compiler for building model entities that represent the crosscutting in the source code. The AspectJ compiler conforming to our architecture is built using the JastAdd compiler framework [18] which allows for flexibly extending the processed language features.

The AspectJ aspect in listing 1 will be used to illustrate the discussion in this section. Listing 2 shows the preamble code that expresses the same aspect in terms of our meta-model, using implementations of meta-model interfaces discussed in this subsection. Figure 5 shows an object diagram created by the preamble.

```
1  aspect BoundPoint issingleton() {
2
3    before(Point p):
4        call(void Point.set*(*)) &&
5        target(p)
6    {
7      // ...
8    }
9
10 }
```

Listing 1. Aspect definition in the AspectJ language.

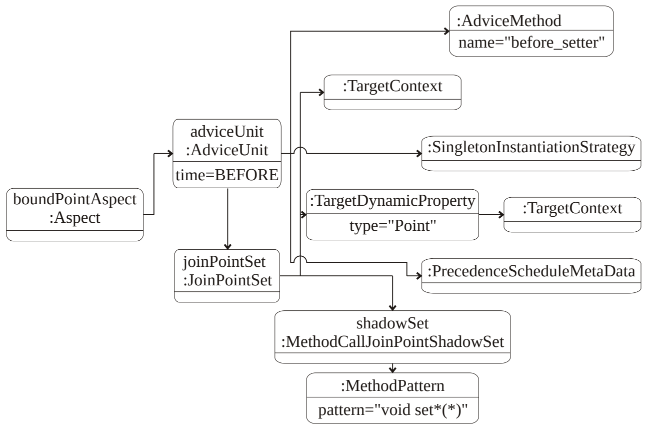

Figure 5. Meta-object structure modeling the AspectJ aspect.

To start with, the compiler creates a class for each aspect (lines 1 to 5 in listing 2) that contains a method (lines 2 to 4, listing 2) for each advice (lines 6 to 8, listing 1).

Next, the compiler generates the preamble code, shown in lines 8 to 30 in listing 2. Our AspectJ compiler generates this code into the static initializer of the main class. The preamble contains code for setting up the pointcut (lines 8 to 17) and the advice functionality (lines 19 to 26), finally, the preamble also contains the deployment of the defined aspects (lines 28 to 30).

3.1.1 Pointcut.

Join points described in [6], *method call*, *field set*, etc., are mapped to implementations of the classes `JoinPointShadowSet` and `Pattern` [4]. For illustration, consider the lines 8 to 17 in listing 2, where a `MethodPattern` and a `MethodCallJoinPointShadowSet` object are created to express the **`call`** pointcut from line 4 in listing 1.

`Pattern` classes are provided for matching the fully qualified signature of a method, field, and type against a pattern specified in the AspectJ wildcard notation. To evaluate type patterns containing the “+” operator, the `Pattern` implementation needs information about the type hierarchy. The required information can be gathered by intercepting class loading.

AspectJ also defines pointcut designators (PCD) that select join points based on dynamic properties. The pointcut designators **`target`**, **`this`**, and **`args`**, specify the dynamic type that the receiver object, the executing object, respectively the argument objects, must have in order for an execution point to classify as a join point.

[4] The current implementation does not support *Handler execution*, *Advice execution*, *Constructor execution*, *Object initialization* and *Object pre-initialization*; implementations of these constructs are subject to future work. As for object creation join points, we currently only provide an equivalent to *Constructor call*.

```
 1 class BoundPoint {
 2   before_setter(Point p) {
 3     // ...
 4   }
 5 }
 6
 7 ...
 8 MethodCallPattern setterPattern = Patterns.ajMethodPattern("call(void Point.set*(*))");
 9
10 JoinPointShadowSet shadowSet = AspectModelFactory.createMethodCallJoinPointShadowSet(setterPattern);
11
12 Set<DynamicProperty> dynamicProperties =
13   Collections.singleton(AspectModelFactory.createTargetDynamicProperty(Point.class));
14
15 List<Context> contexts = Collections.singleton(AspectModelFactory.createTargetContext());
16
17 JoinPointSet joinPointSet = AspectModelFactory.createJoinPointSet(shadowSet, dynamicProperties, contexts);
18
19 Class boundPointClass = BoundPoint.class;
20 Method adviceMethod = boundPointClass.getDeclaredMethod("before_setter", new Class[]{Point.class});
21 AdviceUnit adviceUnit = new AdviceUnit(
22   BEFORE,
23   Collections.singleton(joinPointSet),
24   AspectModelFactory.createSingletonInstantiationStrategy(boundPointClass),
25   adviceMethod,
26   new PrecedenceScheduleMetaData());
27
28 Aspect boundPointAspect = AspectModelFactory.createAspect(Collections.singleton(adviceUnit));
29
30 AspectModelFactory.deploy(boundPointAspect);
```

Listing 2. Aspect from listing 1 in our model.

We provide implementations of the interface `DynamicProperty` for these designators. These implementations are configured with the type to which the respective context value must conform. For illustration, consider the code in line 13 in listing 2, which expresses the **`target`** pointcut designator in line 5 of listing 1.

In our model, the pointcut designators **`within`**, **`withincode`**, **`cflow`** and **`cflowbelow`**, are also mapped to implementations of `DynamicProperty`. In AspectJ, **`within`** and **`withincode`** select join points based on their lexical scope and are statically resolved by the `ajc` and `abc` weaver [2, 19]. At the conceptual level, we consider them more generally as dynamic properties that select join points based on the topmost frame in the call stack. This allows to keep the mapping independent of a certain weaving strategy. For instance, in the efficient weaving technique presented in [20, 21], it is not possible to evaluate **`within`** statically.

When a mapping of concrete language features to the meta-model is provided, the feature must be realized in terms of the interface of the meta-model. This interface is very general – for dynamic properties it is simply the method `isSatisfied` – which allows (a) a uniform treatment by meta-model implementations (e.g., a weaver) and (b) an implementation of the feature using Java's full computational power. A weaver that optimizes certain features will not use this general interface, but instead directly generate code driven by the special knowledge about the feature.

Listing 3 sketches the default implementation of the **`cflow`** pointcut designator of AspectJ, which basically follows the implementation scheme of the AspectJ compilers [19].

The constructor of `CflowDynamicProperty`'s default implementation receives an array of `JoinPointSets`, corresponding to the pointcut of a **`cflow`** in AspectJ. From these join point sets a *before* `AdviceUnit` is created, whose advice method is the `increase` method defined in the class `CflowDynamicProperty`. Similarly, an *after* advice unit is created that decreases the counter. The `ExplicitInstantiationStrategy` used in line 19 always returns the specified object as the advice instance. The `isSatisfied` method of a `CflowDynamicProperty` returns **`true`** if the counter is greater than zero.

Because we use the abstract factory design pattern [22] to create the model entities, it is easily possible to replace their concrete implementations. A virtual machine with dedicated optimizations for cflow, e.g., can overwrite the factory and construct an appropriate object for representing the cflow dynamic property in a way transparent to the user. The factory method receives all information that describes the cflow dynamic property on an abstract level, i.e., a description of the join points constituting the control flow in question. An alternative implementation would not use this description as in the example above, but store it and make it available to the virtual machine. Execution environments that do not overrise the factory will override the default implementation. This topic will be discussed further in section 4.2.

AspectJ defines the pointcut designators **`target`**, **`this`** and **`args`** to bind values from the dynamic context of a join point and to make them accessible to the advice. An example is given in line 5, listing 1. These pointcut designators are modeled as subclasses of `Context`[5]. In the default implementation, the `getValue` method of these `Context` subclasses exploit the Java Virtual Machine Tools Interface (JVMTI) [23] to access local values in the join point's context. An optimized implementation is discussed in section 3.2. Lines 15 to 15 in listing 2 show an example that uses these context implementations.

The special forms **`thisJoinPoint`**, **`thisJoinPointStaticPart`** and **`thisEnclosingJoinPointStaticPart`** available in AspectJ are also mapped to `Context` implementations. These implementa-

[5] It is also possible to bind values at entry points of a control flow in AspectJ. For these designators a default implementation can be provided similar to the cflow dynamic property. As for all parts of the meta-model, an optimized implementation is also possible.

```
public class CflowDynamicProperty
    extends DynamicProperty {

  private int counter;

  public void increase() {
    counter++;
  }

  // ...

  public CflowDynamicProperty
      (Set<JoinPointSet> joinPointSets) {
    AdviceUnit increaseAdviceUnit =
      AspectModelFactory.createAdviceUnit(
        BEFORE,
        joinPointSets,
        AspcetModelFactory.
          createExplicitInstantiationStrategy(this),
        CflowDynamicProperty.class.getDeclaredMethod(
          "increase", new Class[0]),
        new PrecedenceScheduleMetaData());

    // ...
  }

  public boolean isSatisfied(Object[] contextValues)
       {
    return counter > 0;
  }
}
```

Listing 3. Implementation of *cflow* in our model.

tions require context values such as the target object or the signature of the join point, which are used to create an instance of `JoinPoint`.

3.1.2 Advice Functionality.

The *per clause* in AspectJ controlls the retrieval of advice instances. The keyword **`issingleton`** in listing 1, line 1, specifies that all advice are executed on the same advice instance. Line 24 in listing 2 illustrates the use of the `SingletonInstantiationStrategy`, whose implementation is sketched in listing 4. The first time an advice instance is needed, a new instance of the specified class is created and used for all subsequent advice method executions. Respective `InstantiationStrategy` implementations are also provided for **`perthis`**, **`pertarget`**, **`percflow`** and **`percflowbelow`**.

```
public class SingletonInstantiationStrategy
extends InstantiationStrategy {

  private Class type;
  private Object singleton;

  public SingletonInstantiationStrategy (Class type) {
    this.type = type;
  }

  public Object getAdviceInstance
      (Object[] contextValues) {
    if(singleton == null) {
      singleton = type.newInstance();
    }
    return singleton;
  }
}
```

Listing 4. Singleton instantiation strategy.

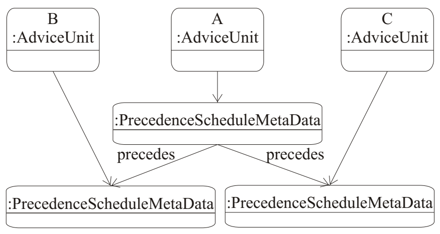
B
:AdviceUnit
A
:AdviceUnit
C
:AdviceUnit
:PrecedenceScheduleMetaData
precedes
precedes
:PrecedenceScheduleMetaData
:PrecedenceScheduleMetaData

Figure 6. An example for the `PrecedenceScheduleMetaData`.

Advice precedence is specified in AspectJ either implicitly by the order in which advice are defined in an aspect, or explicitly by `declare precedence`. To map this feature, we provide the classes `PrecedenceScheduler` and `PrecedenceScheduleMetaData`, which implement the interfaces `Scheduler`, respectively `ScheduleMetaData` . In listing 2, line 26, the advice unit is initialized with an empty `PrecedenceScheduleMetaData`, because it is the only advice unit and does not precede another one.

For a more sophisticated example consider three *before* advice units *A*, *B* and *C*, where *A* precedes both *B* and *C*. The precedence schedule meta-data of *A* stores a reference to the precedence schedule meta-data of *B* and *C* (figure 6 shows a corresponding object diagram). When advice unit *A* and *B* have to be executed at the same join point shadow, the `PrecedenceScheduler` implementation is passed the schedule meta data of *A* and *B*; it returns that *A* must be executed before *B*.

It might seem naive to model aspects at this level of granularity risking poor performance. However, the meta-model was designed to preserve all the concepts that have been present in the source code in a way independent from the weaver implementation. In our architecture, the implementation of the meta-model by components of the virtual machine is responsible for performing optimizations to the model, as discussed in the following subsection and in section 4.

The AspectJ compiler presented here as well as the corresponding execution environment implementation discussed in the following subsection have been integrated in a modified version of the AJDT, called the AJDT-EM (EM stands for Execution Model). Documentation about the AJDT-EM can be found in [24], installation and usage instructions are available at [25].

3.2 Default Weaver for the Meta-Model

The factory for our meta-model provides the `deploy` operation for aspects. When an aspect is deployed by invoking this operation, the execution environment must take care that the aspect is active during the subsequent execution. As the default implementation of such an execution environment, we provide a Java agent as an extension to a standard Java 5 virtual machine (JVM). The default weaver implementation is also discussed in [26, 27].

The agent uses the bytecode instrumentation package to intercept class loading: When a class is loaded by the virtual machine, the agent processes the class data and stores meta-information about the class to be used for join point shadow search and weaving. Upon aspect deployment, the agent performs bytecode weaving similar to existing aspect weavers and uses the Class Redefinition facility (also part of the bytecode instrumentation package) of JVMs to replace the old bytecode of classes with the bytecode where the aspect is woven in. When a class is loaded the contained join point shadows are matched against the join point shadow sets of the aspects currently deployed. When a shadow is matched all corrsponding advice units are deployed.

Upon aspect deployment, join point shadow search is performed. The `JoinPointShadowSets` of all `JoinPointSets` associated with the advice units in the aspect are evaluated. The result of the evaluation is a set of join point shadow meta-objects. The latter store information about advice units applying to the corresponding shadow: the `AdviceMethod`, the `ScheduleMetaData` and the `InstantiationStrategy` of the `AdviceUnit` as well as the `Contexts` and the `DynamicPropertys` of the matched `JoinPointSet`. After all advice units are processed in this way, bytecode is generated for all affected join point shadows and the Class Redefinition facility is used to replace the bytecode of affected methods in the virtual machine.

The code that is generated by the weaver for checking the dynamic properties and executing the advice method is called *advice dispatch block*. The execution order of the advice dispatch blocks is determined by the scheduler based on the `ScheduleMetaData` objects of the corresponding advice units. Figure 7 shows (a) an advice order structure and (b) how it is mapped onto a sequence of advice dispatch blocks.

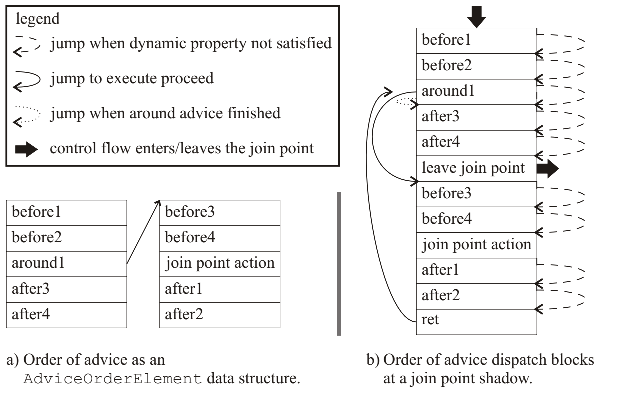
legend
jump when dynamic property not satisfied
jump to execute proceed
jump when around advice finished
control flow enters/leaves the join point
before1
before2
around1
after3
after4
before3
before4
join point action
after1
after2
a) Order of advice as an AdviceOrderElement data structure.
before1
before2
around1
after3
after4
leave join point
before3
before4
join point action
after1
after2
ret
b) Order of advice dispatch blocks at a join point shadow.

Figure 7. Instruction sequence generated for ordered advice units.

A dispatch block for **before** and **after** advice consists of code to:

1. invoke the method `isSatisfied` on each of the dynamic properties,
2. (for non-static advice methods) invoke `getAdviceInstance` on the instantiation strategy,
3. invoke `getValue` on all context values,
4. invoke the advice method.

Each call to `isSatisfied` on a dynamic property is followed by a conditional jump (if the call returns **false**) behind the last instruction of the advice dispatch block. In this case, the execution continues with the next advice dispatch block.

In order to support **proceed**, the dispatch block for **around** advice is slightly different. Instead of generating code for invoking the advice method (item 4 in the structure of dispatch blocks for **before** and **after** advice), the default weaver copies the **around** advice method's implementation into the advice dispatch block.

Inlining of **around** advice can not be performed when the **proceed** occurs in an anonymous nested type in the **around** advice as observed in [19, 2]. There, an implementation based on closures is described that can be used in all situations. A similar implementation in the default weaver is subject to future work.

The inlined code is modified in two ways. First, local variables are re-numbered to avoid interference with local variables of the method containing the join point shadow or other inlined around advice. Second, return instructions are replaced by instructions that jump behind the inlined advice method's code to ensure that the execution continues with the next advice dispatch block after the inlined advice method.

Our AspectJ compiler produces bytecode for around advice methods in a way to facilitate the necessary rewrites. A **proceed** is represented as an invocation to a static method with the same signature as the around advice's method. This method will never be called, though. Whenever such a call is discovered, the weaver replaces it with a "jump to subroutine" (`jsr`) to the first advice dispatch block of the next `AdviceOrderElement`. Before this, the current program counter is stored to enable resuming execution immediately behind the `jsr` instruction. Correspondingly, a `ret` instruction is inserted right behind the last advice dispatch block of an `AdviceOrderElement` for returning to the `jsr`.

The default weaver presented here conforms to the meta-model. But in our implementation, we already treat some of the AspectJ-instantiations of the meta-model specially, thereby showing how a meta-model implementation benefits from the AO concepts being first-class. This implementation provides optimized implementations for two kinds of context values: those that are locally available at the join point, and those that can be evaluated at weaving time.

An example of the first kind is the `Target` context: The weaver simply generates instructions that load it from the call frame instead of generating a call to the `Target` objects `getValue` method. An example of the second kind is the join point signature (e.g., exposed as part of **`thisJoinPointStaticPart`** in AspectJ): A string constant and an instruction for loading it are generated by the weaver.

With these optimizations, the default implementation of the meta-model is comparable to conventional aspect weaving in terms of the runtime performance. Advanced optimizations are also possible as will be discussed in the following section.

4. Alternative Instantiations of the Architecture

In this section, we evaluate the proposed architecture in terms of its flexibility in supporting variability with respect to both, different instantiations of the meta-model (i.e., compilers and language features) and different implementations of the meta-model (i.e., execution environments).

4.1 Different Instantiations of the Meta-Model

Currently, there is a variety of aspect-oriented programming languages offering additional features to AspectJ [28]. We will discuss how some advanced features can be realized as instantiations of our meta-model; more discussion of mapping languages to the meta-model can be found in [27, 26].

To start with, several languages support dynamic scoping of aspects, e.g., an aspect can be active only within certain threads or only for certain base objects. Examples are the languages CaesarJ [29, 30] and JAsCo [13, 14]. In our meta-model, the scope of an aspect can be mapped to a dynamic property: When an aspect is deployed with a specific scope, the aspect is copied and all join point sets of its copied advice units get an additional dynamic property implementing checks for the scope. Afterwards, the copied aspect is deployed.

Aspect-oriented languages also differ with respect to their pointcut languages. We have observed that, for the most part, the join point shadows, i.e., the statically resolvable part, is identical; differences exist in the dynamic properties that can be specified. Several languages allow to define pointcuts in terms of the execution history beyond the abilities of the **`cflow`** pointcut designator of AspectJ. The respective pointcut languages provide expressions to describe "interesting" sequences of join points, e.g., by means of regular expressions [11, 31, 32] or linear temporal logic [12]. When such a sequence is detected at runtime, the pointcut matches.

Join point sequences can be realized as dynamic properties in our meta-model, in a similar way as the **`cflow`** realization discussed in subsection 3.1. `AdviceUnits` are generated and deployed that get notified at the join points participating in the sequence. Internally, the advice units keep track of the already encountered join points, e.g., by updating an automaton. When the dynamic property's `isSatisfied` method is called, it checks the automaton's state.

Another dimension of variability concerns advice ordering. In section 3.1, we discussed how the proposed approach enables to determine the order of advice execution at a given join point using aspect precedence. More advanced strategies can be realized as well. For example, `ScheduleMetaData` objects can also specify conditional inter-advice dependencies, such as, *if advice a and b but not c apply then execute a before b; if, however, a, b and c apply then the order is b before c before a*. Given such specifications, constraint solving techniques, e.g., discussed in [33], could be used to determine the advice' order.

4.2 Different Implementations of the Meta-Model

We already discussed optimizations that can be performed by the weaver, e.g., for the `Target` context. Besides generating more efficient code for accessing context values, alternative weaver implementations may also be able to evaluate dynamic properties statically. For instance, in contrast to our default implementation, the **`within`** and **`withincode`** dynamic properties could be statically evaluated.

There are other kinds of optimizations which are only possible within a virtual machine. In previous work [20], we have implemented virtual machine techniques that speed up dynamic aspect deployment. The idea is to treat join point shadows similar to method calls and use well established speculative optimization techniques for virtual methods [34, 35]. The idea of these techniques is to perform no virtual method dispatch when the just-in-time compiler (JIT) can determine the concrete type of the receiver object. This determination is based on the assumption that certain properties of the application, e.g., the type hierarchy, will not change. The virtual machine can detect changes to these properties, class loading for example, and efficiently replace the non-virtual method dispatch with a full virtual method dispatch [36].

Similarly, these techniques are used in [20] for deploying aspects. The assumption when compiling a join point shadow is that the set of deployed aspects will not change. At the event of aspect deployment, the code of join point shadows is replaced. By using these techniques an efficient implementation of the deployment for the meta-model can be provided.

The evaluation in [20] measured the time for deploying an aspect affecting all calls to public application methods. Deploying this aspect on the SPEC JVM98 benchmark applications [37] with the VM integrated deployment only took 3 ms on average. Other implementations of dynamic deployment averagely took between 229 ms and 3360 ms in the same scenario.

In the Steamloom virtual machine [7], we have experimented with optimized implementations of control flow checks [9] and of advice instance retrieval [10].

For the optimized control flow check, an identifier is assigned to each **`cflow`** pointcut designator defined in the application. Instead of weaving in instructions that execute the control flow check, the weaver flags the place where the check has to be executed and inserts the **`cflow`**'s identifier as a reference to the first-class entity. When the JIT compiler encounters a place where a control flow check is to be executed, it can access the **`cflow`**'s full definition because it is a first-class entity. This allows the JIT compiler to check whether the control flow is always **`true`** or always **`false`** in the currently compiled context. In [9], this optimization as well as others are described in more detail.

In [9] we present a worst case evaluation of the effect of **`cflow`** pointcut designators on single operations. We measured how much the performance of method calls degrades for methods that (a) constitute the control flow referred to by **`cflow`** and (b) only contains a check of the current control flow. The integrated **`cflow`** implementation imposed an overhead of 166% for case (a) and 67% for case (b). The performance loss of other investigated implementations ranged from 498%[6] to 7370% for case (a) and from 687% to 4240% for case (b) in single-threaded applications. For multi-threaded applications our implementation exposed the same performance as in the single-threaded case while the other implementations expose a performance loss of at least 1856 % (case (a)) and 2108% (case (b)).

When using this optimization in a meta-model implementation, the optimized control flow check implementation can use the control flow based `DynamicProperty` implementations as first-class representations of the check. When the JIT compiles an advice dispatch block that contains such a check, it can use the associated join point sets to determine if the check will, e.g., always succeed and omit instructions calling the dynamic property's `isSatisfied` method. If the check can not be omitted, the JIT can generate code for a more efficient check that is executed instead of the call to the default implementation of `isSatisfied`, as has been shown in [9].

In other work, the object model of the virtual machine has been modified to store a table of advice instances to realize optimized access to them [10]. This way, the lookup costs are reduced. The extended virtual machine provides a special bytecode instruction for loading the instance from its storage location in the extended object layout.

The enhanced object model for virtual machines to support per instance aspects can play its strength for **`pertarget`** aspects. In the approach with advice instance tables, the performance of executing an advice from a **`pertarget`** aspect is at least circa one order of magnitude faster than in other investigated approaches [10].

While all these optimizations are promising, they currently lack a common interface to make them available to a wide range of language implementations. The meta-model presented in this paper can act as such an interface.

5. Related Work

The Nu project [5] also aims at providing an interface between compilers and execution environments. For this purpose, two new instructions are provided in the intermediate language (e.g., the bytecode) for associating and disassociating a *pattern of join points* with a *delegate*. The model of this approach is less differentiated and less complete than our meta-model. For example, dynamic properties of join points such as **`cflow`** can not be expressed [38].

In [39], a meta-model to capture the semantics of features in aspect-oriented languages is defined. The meta-model is implemented as an interpreter in the Smalltalk language, called *metaspin*, whereby each computational step is represented as a closure and can be a join point. The aspect sand-box project [40] follows similar goals. The semantics of aspect-oriented languages are expressed as interpreters in the Scheme language. Similar to our approach, both aforementioned approaches represent aspect-oriented concepts as first-class entities. However, these approaches only target language design and the connection to optimized implementations are not considered.

The Reflex project [41, 42] aims at providing an extensible kernel for aspect-oriented programming based on behavioral reflection

[6] We leave the results for the stack-walking implementation out of this discussion, as the performance at **`cflow`** checks is probibitively bad.

by means of a Java embedded language. This kernel has some similarities with the meta-model proposed in this paper. What we call an advice unit, is a *link* in their terminology; join point shadow sets are called *hooksets* and Dynamic properties are called *activation condition* in Reflex and are also first-class objects. However, in the proposed way of using Reflex, i.e., by means of the Reflex kernel language [41], activation conditions have to be specified as blocks of code which hinders re-use. What is more important, Reflex covers only the meta-model part of our proposed architecture. Questions related to exploiting the reflective aspect definitions in the execution environment, including different implementations of aspect deployment, e.g., transactional aspect deployment, are not considered.

The AspectBench Compiler (`abc`) [43, 19] offers a workbench for implementing compilers for aspect-oriented languages. It provides an extensible parser based on the Polyglot framework. Furthermore, interfaces for *shadow types* and *shadow matcher* are provided, which are used for join point shadow search and weaving. Bytecode analyzes and optimizations of the Soot framework are also part of the workbench. Language extensions implement the `ShadowType` and `ShadowMatch` interfaces for new kinds of pointcut designators and extend the parser such that it creates instances of these classes when it encounters a pointcut definition. These objects are passed as first-class entities to the weaver which generates an intermediate code. The analyzes and optimizations provided by Soot work on this intermediate code and can, thus, be reused by language extensions.

The `abc` model is less flexible than our meta-model, e.g., with regard to instantiation strategies, as the methods for loading the receiver object for advice calls are hard-coded for the per clauses defined in the AspectJ language. Also, as the compiler weaves the aspects into the bytecode, the concepts are not first-class in the execution environment, preventing it from making additional optimizations that are not possible to perform at compile time.

The `ajc` compiler from the AspectJ distribution [44] is split into a front-end which parses the AspectJ code and generates pure Java bytecode from it and a back-end which weaves in the aspects. In the first step, aspects and advice are transformed into classes and methods. Non-java constructs such as pointcut declarations are stored as Java bytecode attributes. In a second step, the back-end of the compiler reads these attributes and performs the weaving [2]. Thus, similarly to `abc`, pointcut declarations are passed as first-class entities to the weaver but lose this state after weaving, i.e., before the execution. Also, the interface between the front-end and the back-end is not officially documented and also not extensible.

6. Summary and Future Work

In this paper, we presented an architecture for implementations of aspect-oriented programming languages. Central to this architecture is a meta-model of aspect-oriented language features that decouples the definition of language features from their implementation in a virtual machine.

The meta-model is generic in a sense that a wide variety of current aspect-oriented language concepts can be mapped onto it. This mapping can be expressed in a way that abstracts from optimized implementation issues. This enables language designers to concentrate exclusively on the semantics of language features and yet profit from optimization techniques implemented in virtual machines that adhere to the architecture.

The proposed architecture requires that the front-end of a language implementation, i.e., the compiler, produces code that creates runtime objects which define the aspects according to the meta-model. The back-end, i.e., the execution environment, recognizes these runtime objects and weaves the program accordingly. Since aspect-oriented concepts are expressed as first-class entities via the runtime objects, execution environments are enabled to make sophisticated optimizations.

The architecture is flexible in the sense that compilers and execution environments adhering to it can be flexibly exchanged. Optimizations that are made in special execution environments adhering to the architecture can be be exploited by programming languages also adhering to the architecture. More information and downloads can be found at the project's home page [25].

In future work, we will target three different areas. One area is to improve the default implementation of the model. Currently not all join point shadows provided by AspectJ are supported, namely, exception handlers as well as different variants of object initialization (e.g., **`preinitialization`**). Support for these join point shadows and an implementation of **`around`** and **`proceed`** based on closures will be provided in future versions of the default weaver.

A second area of future work will target other concrete instantiations of the architecture as discussed in section 4. On the one side, we will provide virtual machine optimizations presented in earlier work, as a special implementation of the meta-model. This will include a comprehensive performance evaluation and comparison. On the other side, other aspect-oriented languages will be mapped to the meta-model and respective compilers will be implemented. In this process, the meta-model might need to be refined.

Finally, we we will increase the expressiveness of our model. In concrete we will research possibilities to also capture structural crosscutting in the model. Further, we will investigate support for more advanced join point models and more expressive pointcut languages, e.g., similar to Prolog queries.

Acknowledgments

We would like to thank Michael Eichberg who contributed to the discussion of advice ordering, Michael Hausl for implementing the AspectJ compiler discussed in the paper, and Markus Maus for contributing to the AspectJ instantiation of the meta-model.

This work was supported by the AOSD-Europe Network of Excellence, European Union grant no. FP6-2003-IST-2-004349.

References

[1] Masuhara, H., Kiczales, G.: Modeling crosscutting in aspect-oriented mechanisms. In: 17th European Conference on Object-Oriented Programming (ECOOP). (2003) 2–28

[2] Hilsdale, E., Hugunin, J.: Advice weaving in AspectJ. In: AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development, New York, NY, USA, ACM Press (2004) 26–35

[3] Masuhara, H., Kiczales, G., Dutchyn, C.: Compilation semantics of aspect-oriented programs. In: Foundations Of Aspect-Oriented Languages (Workshop at AOSD 2002). (April 2002)

[4] Hidehiko Masuhara, G.K., Dutchyn, C.: Compilation semantics of aspect-oriented programs. In: Foundations Of Aspect-Oriented Languages (Workshop at AOSD 2002). (April 2002)

[5] Rajan, H., Dyer, R., Hanna, Y., Narayanappa, H.: Preserving Separation of Concerns through Compilation. Technical Report 405, Iowa State University (21 March 2006 2006)

[6] : The AspectJ Programming Guide. `http://www.eclipse.org/aspectj/doc/released/progguide/index.html` (2006)

[7] : The Steamloom Homepage. `http://www.st.informatik.tu-darmstadt.de/static/pages/projects/AORTA/Steamloom.jsp`

[8] : Homepage of the Envelope-Aware Jikes RVM. `http://www.st.informatik.tu-darmstadt.de/EBW-aware`

[9] Bockisch, C., Kanthak, S., Haupt, M., Arnold, M., Mezini, M.: Efficient Control Flow Quantification. In: ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2006). (2006)

[10] Haupt, M., Mezini, M.: Virtual Machine Support for Aspects with Advice Instance Tables. In: First French Workshop on Aspect-Oriented Programming, Paris, France (September 2004)

[11] Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Adding trace matching with free variables to AspectJ. In: OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications, New York, NY, USA, ACM Press (2005) 345–364

[12] Stolz, V., Bodden, E.: Temporal Assertions using AspectJ. In: Fifth Workshop on Runtime Verification (RV'05), Electronic Notes in Theoretical Computer Science., Elsevier Science Publishers (2005)

[13] : Jasco homepage. `http://ssel.vub.ac.be/jasco/`

[14] Suvée, D., Vanderperren, W., Jonckers, V.: JAsCo: an Aspect-Oriented Approach Tailored for Component Based Software Development. In: Proc. AOSD 2003. (2003) 21–29

[15] : AspectWerkz homepage. `http://aspectwerkz.codehaus.org/`

[16] Bonér, J.: AspectWerkz - Dynamic AOP for Java. `http://codehaus.org/~jboner/papers/aosd2004_aspectwerkz.pdf` (2003)

[17] : Api specification for package java.lang.instrument. `http://java.sun.com/j2se/1.5.0/docs/api/java/lang/instrument/package-summary.html`

[18] Ekman, T., Hedin, G.: Rewritable Reference Attributed Grammars. Lecture Notes in Computer Science **3086** (January 2004) 147–171

[19] Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Optimising AspectJ. In Hall, M., ed.: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), ACM Press (2005)

[20] Bockisch, C., Arnold, M., Dinkelaker, T., Mezini, M.: Adapting Virtual Machine Techniques for Seamless Aspect Support. In: ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2006). (2006)

[21] Bockisch, C., Haupt, M., Mezini, M., Mitschke, R.: Evenelope-based Weaving for Faster Aspect Compilers. In: In Net.ObjectDays. (2005)

[22] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, Reading, Massachusetts (1994)

[23] : Jvmti (java virtual machine tool interface) homepage. `http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/`

[24] Jackson, A., Clarke, S., Bockisch, M.C.C.: Deliver preliminary support for next-priority use cases. Technical Report AOSD-Europe deliverable D80, AOSD-Europe-IBM-80, IBM UK (26 February 2007 2007)

[25] : Homepage of the aspect language implementation architecture (alia). `http://www.st.informatik.tu-darmstadt.de/static/pages/projects/ALIA/alia.html`

[26] Bockisch, C., Mezini, M., Gybels, K., Fabry, J.: Initial definition of the aspect language reference model and prototype implementation adhering to the language implementation toolkit architecture. Technical Report AOSD-Europe Deliverable D72, AOSD-Europe-TUD-7, Technische Universität Darmstadt (27 February 2007 2007)

[27] Jackson, A., Clarke, S., Chapman, M., Dean, A., Bockisch, C.: Deliver Preliminary Support For Top Priority Use Cases. Technical Report AOSD-Europe deliverable D64, AOSD-Europe-IBM-64, IBM UK (30 October 2006 2006)

[28] Dinkelaker, T., Haupt, M., Pawlak, R., Navarro, L.D.B., Gasiunas, V.: Inventory of Aspect-Oriented Execution Models. Technical Report AOSD-Europe Deliverable D40, AOSD-Europe-TUD-4, Technische Universität Darmstadt (28 February 2006 2006)

[29] Aracic, I., Gasiunas, V., Mezini, M., Ostermann, K.: Overview of caesarj. Transactions on AOSD I, LNCS **3880** (2006) 135 – 173

[30] : CaesarJ homepage. `http://caesarj.org/`

[31] Douence, R., Fritz, T., Loriant, N., Menaud, J.M., Ségura-Devillechaise, M., Südholt, M.: An expressive aspect language for system applications with Arachne. In: AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development, New York, NY, USA, ACM Press (2005) 27–38

[32] Vanderperren, W., Suvée, D., Cibrán, M.A., Fraine, B.D.: Stateful Aspects in JAsCo. `http://ssel.vub.ac.be/jasco/media/sc2005.pdf`

[33] Eichberg, M., Mezini, M., Kloppenburg, S., Ostermann, K., Rank, B.: Integrating and Scheduling an Open Set of Static Analyses. In: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE). (2006)

[34] Arnold, M., Fink, S.J., Grove, D., Hind, M., Sweeney, P.F.: A survey of adaptive optimization in virtual machines

[35] Detlefs, D., Agesen, O.: Inlining of virtual methods. In: 13th European Conference on Object-Oriented Programming (ECOOP). Volume 1628 of LNCS. (June 1999) 258–278

[36] Fink, S.J., Qian, F.: Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In: International Symposium on Code Generation and Optimization (CGO). (2003) 241–252

[37] : SPEC JVM98 homepage. `http://www.spec.org/osg/jvm98/`

[38] Rajan, H., Dyer, R., Narayanappa, H., Hanna, Y.: Nu: Towards an AspectOriented Invocation Mechanism. Technical report, Iowa State University (26 March 2006 2006)

[39] Brichau, J., Mezini, M., Noyé, J., Havinga, W., Bergmans, L., Gasiunas, V., Bockisch, C., Fabry, J., DHondt, T.: An Initial Metamodel for Aspect-Oriented Programming Languages. Technical Report AOSD-Europe Deliverable D39, AOSD-Europe-VUB-12, Vrije Universiteit Brussel (27 February 2006 2006)

[40] Dutchyn, C., Kiczales, G., Masuhara, H.: Aop language exploration using the aspect sand box. In: Aspect-Oriented Software Development (AOSD 2002). (April 2002)

[41] Tanter, É.: An Extensible Kernel Language for AOP. In: Proceedings of AOSD Workshop on Open and Dynamic Aspect Languages, Bonn, Germany (2006)

[42] Tanter, É., Noyé, J.: Motivation and Requirements for a Versatile AOP Kernel. In: 1st European Interactive Workshop on Aspects in Software (EIWAS 2004), Berlin, Germany (September 2004)

[43] Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: abc: an extensible AspectJ compiler. In: AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development, New York, NY, USA, ACM Press (2005) 87–98

[44] : AspectJ homepage. `http://www.eclipse.org/aspectj/`