The Art of the Meta-Aspect Protocol

Tom Dinkelaker Mira Mezini Christoph Bockisch

Technische Universität Darmstadt
Hochschulstrasse 10
Darmstadt, Germany
{dinkelaker,mezini,bockisch}@informatik.tu-darmstadt.de

ABSTRACT

Alternative semantics for aspect-oriented abstractions can be defined by language designers using extensible aspect compiler frameworks. However, application developers are prevented from tailoring the language semantics in an application-specific manner. To address this problem, we propose an architecture for aspect-oriented languages with an explicit meta-interface to language semantics. We demonstrate the benefits of such an architecture by presenting several scenarios in which aspect-oriented programs use the meta-interface of the language to tailor its semantics to a particular application execution context.

Categories and Subject Descriptors

D.3.3 [**Software Engineering**]: Language Constructs and Features—*Classes and Objects, Frameworks*; D.2.11 [**Software Architectures**]: Languages

General Terms

Design, Languages

Keywords

Aspect-Oriented Programming, Meta-Object Protocols, Open Implementation, Debugging, Aspect Interactions

1. INTRODUCTION

Within a particular programming paradigm a broad range of semantical variations may be available for certain mechanisms; e.g., different semantics for *method dispatch* have been proposed for object-oriented languages. However, most languages do not allow programs written in the language to change the semantics built into the implementations of their compilers, respectively their runtime environments.

Kiczales et al. have proposed *meta-object protocols* [27] (MOP for short) to open up the implementations of object-oriented programming languages to systematically support alternative semantics. *"Metaobject protocols are interfaces to the language that give users the ability to incrementally modify the language's behavior and implementation"* [27]. MOPs have been, e.g., used to realize flexible implementation strategies for *inheritance* and *class instantiation* that can be adapted by users of the language.

This flexibility is possible because (part of) the semantics of objects is reified in *meta-objects* associated with them. By exchanging these meta-objects, alternative semantics can be plugged-in resulting in a new *variant OO language* [27]. As a consequence, the same object-oriented program can be interpreted under different semantics.

Similar to object-oriented languages, different languages for *aspect-oriented programming* (AOP) [26, 25] expose a large variety in their semantics. Different realizations are possible for *aspect instantiation*, *scoping*, *advice ordering*, etc., corresponding to different user requirements on the aspect language in different application domains. Yet, most aspect-oriented languages only provide one rigid form of semantics.

There are open platforms (compilers, interpreters, runtimes) for aspect-oriented (AO for short) languages that allow to provide alternative semantics for subsets of language semantics. Examples for supported adaptations are: new pointcut designators [36, 2], new join points [47], and advice composition orders [45, 46, 31]. However, there are two drawbacks in the extensibility supported by such extensible AO compilers and runtimes.

First, run-time adaptations of language semantics by applications are not supported. To motivate dynamic adaptation of AO language semantics, consider the mechanism for handling aspect interactions. Detecting aspect interactions and resolving potential conflicts depends on the strategy that is built into the infrastructure to compose multiple aspects at a shared join point. For example, with existing technology the strategy composes interacting aspects in a fixed order specified by the user. Nonetheless the strategy for coordinating aspect interactions may depend on the run-time state of application objects and aspects; such interactions are also called semantical [12, 28] and more precisely context-dependent [41]. Conflicts arising from context-dependent interactions cannot be resolved by any static ordering but only by dynamically changing the aspect composition strategy according to the application context.

Second, language technology based on open compilers/runtimes lacks regularity in the programming model: different technologies and programming models are needed for ex-

AOSD'09, March 2–6, 2009, Charlottesville, Virginia, USA.

tending the language and for using the language to program application semantics. For instance, when extending the language semantics supported by the *abc* compiler [2], advanced compiler technology – with a steep learning curve – such as *attribute grammars* and extensions thereof [1] need to be known and applied. In contrast, *AO technology* is used for programing the application logic using the extended semantics. Non-regular technologies do not scale well; what is needed is an adaptation mechanism for semantics that is available to application developers.

In this paper, we propose the concept of an AO language with an integrated *meta-aspect protocol*. The proposed language follows the *open implementation* design principle [24] and is inspired by meta-object protocols; it provides a meta-interface to its implementation, which opens up the aspect language semantics for dynamic user adaptations. When using the meta-aspect protocol, the same set of aspects can be interpreted under different semantics. Analogous to MOPs, this flexibility is enabled by reifying important parts of the language semantics as first-class entities available to AO programs. We have implemented the concept as an open runtime called *Pluggable and OPen Aspect RunTime* (POPART for short), on top of the meta-object protocol of *Groovy* [18]. Users can tailor the default AO semantics for special needs and experiment with the semantics by providing a possibly application-specific extension of the meta-aspect protocol. The POPART programs and user extensions are compiled to Java bytecode that then can be executed in the Java VM. To demonstrate the usefulness of POPART's meta-aspect protocol, we have used it to implement diverse variations of AO semantics which are flexibly selected by AO programs[1].

The contributions of this paper are twofold:

- To the best of our knowledge, this is the first paper to propose AO language technology with a meta-aspect protocol. While related work has expressed the need for meta-aspect protocols [5, 40], no concrete concept has been proposed so far.

- We demonstrate the benefits of meta-aspect protocols by implementing new variant languages that solve non-trivial problems in AOP. The applications of the proposed meta-aspect protocol range from providing support for debugging residual pointcuts over resolving aspect interactions that depend on the dynamic program context up to enabling dynamic aspect deployment.

The remainder of the paper is structured as follows. Sec. 2 provides some background knowledge on open language implementations and meta-object protocols as well as their implementation in Groovy. Sec. 3 discusses the concept of the meta-aspect protocol and presents details of the aspect runtime that we have built as a proof-of-concept. Sec. 4 evaluates the concept by applying the proposed meta-aspect protocol to implement several language variations. Sec. 5 discusses related work and Sec. 6 concludes the paper.

2. BACKGROUND

In this section we summarize concepts upon which our meta-aspect protocol is based and their realization in Groovy.

[1]The source code of POPART and all examples in the paper can be downloaded from: `http://www.stg.tu-darmstadt.de/popart`.

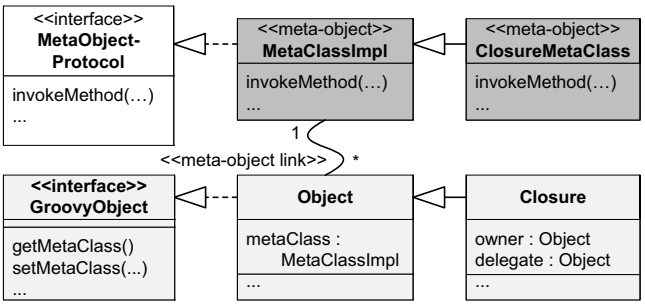

Figure 1: The Meta-Object Protocol of Groovy

2.1 Open Implementation and Meta-Object Protocols

The *open implementation* [24] design principle proposes to expose a part of the implementation strategy of systems (operating systems, databases, etc.) to the application-level. A system built according to the open implementation principle provides two orthogonal interfaces: the so-called *primary interface*, exposes to applications the primary functionality of the system. The other interface, called the *meta-interface*, provides application-level access to the implementation strategy of the primary-interface. Using the meta-interface allows users to change the implementation strategy behind the primary interface that is hidden for applications. This principle allows system designers to design their system open for later (unforeseen) adaptations. In particular, a meta-object protocol is an open implementation of an object-oriented language.

Meta-objects provide high-level and reflective operations, called *meta-methods*, that interpret object semantics at the *meta level* (e.g., method dispatch). Meta-object protocols (MOP) [27] define workflows in which these meta-methods participate to build an extensible semantics of the language. Thereby, a MOP-based language implementation covers a range in the design space of language semantics rather than only a single point with one specific behavior. Application programmers can use the MOP to create new *variant languages* that meet application-specific requirements by using standard object-oriented techniques. Thus, the distinction between language designers and users is blurred.

2.2 Groovy

Groovy [18, 32] is a pure object-oriented *scripting language* that nicely integrates with Java [17]. Besides a *meta-object protocol*, Groovy provides attractive language features, such as *class reloading* and *closures* as first-class entities.

As shown in the lower half of Fig. 1 every value is an instance of the class `Object` which implements the interface `GroovyObject`. The meta-level is shown in the upper half of Fig. 1. Method calls and field accesses on an object are handled by a corresponding meta-object, which is an instance of `MetaClassImpl` that implements the interface `MetaObject-Protocol`. The latter declares *meta-methods* used for interpreting an object's behavior, e.g., `invokeMethod`, `getProperty`, `setProperty`, etc.

When a base-level method is called on an object, the *meta-method* `invokeMethod` is called on the object's meta-object. The default implementation of `invokeMethod` in `MetaClass-`

Impl dispatches the base-level method call to the most specific implementation defined in the object's class or in one of its super classes. In a similar way, the meta-methods getProperty and setProperty dispatch field accesses to the most specific instance field[2].

The user can specialize MetaClassImpl and override specific meta-methods. Every class has a link to its default meta-object. When a class is instantiated, the default meta-object is used for the created instance. One can change the default meta-object defined for a class in the registry, thus changing the semantics of all objects of that class. Alternatively, one can change the meta-object of a single object by calling the method setMetaClass on it.

Special values may be associated with specific meta-objects. For instance, a special meta-class is provided for closures, which are objects of the class Closure. Closures encapsulate their own evaluation context, which consists of bindings from the lexical context, i.e., instance variables of their creating object (owner), and local variables. Hence, closures cannot use the default MetaClassImpl, which would dispatch method calls and field/variable accesses to this – the closure itself that does not define them. The special ClosureMetaClass dispatches method calls as well as field/variable accesses inside the closure to its context. Moreover, every closure can have a delegate object. If the latter is not null, method calls and field/variable accesses are dispatched to it. This allows to manipulate the evaluation context of a closure after the closure is created.

3. POPART META-ASPECT PROTOCOL

POPART programs consisting of classes and aspects are written in Groovy closures. The POPART runtime is embedded as an extensible library in Groovy. Further, the Groovy MOP is extended to realize the meta-protocol of the aspect language implemented by the POPART runtime; this extension is called *meta-aspect protocol* (MAP) in the following.

3.1 High-Level View of POPART

Our motivation for designing a meta-aspect protocol is to achieve the same extensibility and flexibility in customizing aspect semantics that meta-object protocols provide for objects. Consequently, we lean the definition of the MAP against the definition of the MOP and define a meta-aspect protocol as *an interface to the aspect language that gives users the ability to incrementally modify behavior and implementation of aspect-oriented abstractions.*

Fig. 2 shows the overall architecture of an aspect runtime with a meta-aspect protocol built on top of the MOP. Such a runtime provides two kinds of interfaces to programs, a *primary interface* and a *meta-interface*.

The *primary interface* defines aspect language abstractions. The primary interface of the default POPART implementation is similar to that of AspectJ [25]. POPART supports before, around, and after advice with common semantics. Pointcut designators such as method_call(regExp[3]), method_execution(regExp), advice_execution(), not(pc[4]),

[2]If an accessed field is not present, the MOP tries to find a corresponding getter or setter method by convention.

[3]A regular expression that should match the method name.

[4] pc is another pointcut expression.

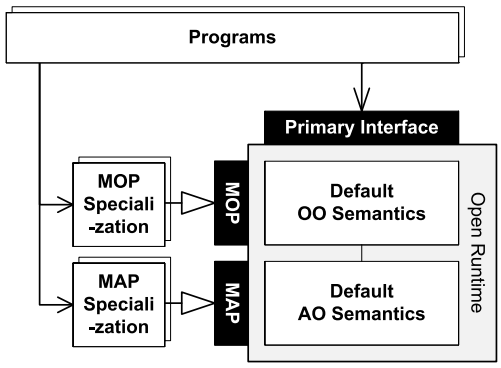

Figure 2: Architecture of POPART

cflow(pc), cflowbelow(pc), and if(boolClosure[5]) are supported. Advice bodies can access context information available for a certain join point type: thisJoinPoint, thisAspect, thisObject, and targetObject.

The *meta-interface* opens parts of the language implementation; it can be used by programs to create and use specialized semantics of the abstraction exposed by the primary interface. POPART's meta-interface consists of two building blocks: While the MOP provides a meta-interface to the the object-oriented semantics, the MAP provides a meta-interface to the aspect-oriented semantics. Application programmers can refine MAP classes to implement specialized aspect semantics and instances of the refined MAP classes can replace the POPART's (default) meta-level entities. Thereby, a POPART program can dynamically change the language semantics, e.g., in order to add run-time debugging support, or to activate an application specific ordering strategy for aspects that co-advise certain join points.

For illustration consider the example in Fig. 3. The aspect definition (lines 6–12) uses abstractions from the primary interface (keywords in bold). In line 21, the (default) meta-aspect associated with aspect is replaced with a specialized meta-aspect that debugs pointcuts and their subexpressions. During a pointcut's evaluation, the specialized meta-aspect prints the result of the evaluation in a tree structure on the console as shown in Fig. 4.

As demonstrated by the code in Fig. 3, the border between implementing application and language semantics is blurred in POPART. Also, there is no gap between the technology used to implement applications and language semantics. In the example presented here and those following in the rest of the paper, only object technology is used for implementing and customizing the language semantics. However, in principle, aspects can also be employed to adapt the meta-level classes. The choice is rather a matter of design decisions about the kind of modularity best suited for implementing particular language implementation concerns.

3.2 Aspects, Pointcuts, Advice, Join Points

At run-time, POPART represents AO program elements as data at the meta-level. The behavior of meta-entities such as *aspects*, *pointcuts*, *join points*, etc., is modeled in designated classes, shown as shadowed boxes in the right lower corner of Fig. 5. Each Aspect has one or several Pointcut-

[5]boolClosure is a closure that must evaluate to boolean.

```
1  class X {
2    void foo() { ... }
3    void baz() { foo(); }
4  }
5
6  aspect(name:"ToyAspect") {
7    Pointcut pc =
8                  method_call("foo.*") &
9                  not(cflow(method_call("bar.*")));
10
11   before ( pc ) { println "foo() called outside of bar().";  }
12 }
13 ...
14 // Code runs with default semantics
15 X x = new X();
16 x.baz(); // prints "foo() called outside of bar()."
17 ...
18
19 AspectManager am = AspectManager.getInstance();
20 Aspect aspect = am.getAspect("ToyAspect");
21 aspect.metaAspect = new DebugMetaAspect(aspect.class);
22 ...
23 // Code runs with specialized semantics.
24 // Also prints information about pointcut evaluation process.
25 x.baz();
```

Figure 3: A POPART Program

```
1  Fired join point at shadow 'Main.foo()' Evaluation:
2  match 'and (method_call(foo.*),not(cflow(method_call(bar.*))))'
3   \−left  = match 'method_call ('foo.*')'
4   \−right = match 'not (cflow (method_call('bar.*')))'
5      \−no match 'cflow (method_call('bar.*'))' Stack ...
6         \−Stack[1] no match at exec shadow 'Main.main(...)'
7         \−Stack[2] no match at call shadow 'Main.baz()'
8         \−Stack[3] ...
```

Figure 4: Debugging View for Pointcut Evaluation

AndAdvice objects each associating a Pointcut object with an advice (the latter are modeled as Groovy closures).

3.2.1 Aspects

As any Groovy object, an Aspect instance has a meta-object. However, meta-objects associated with aspects are of type MetaAspect, a specialization of Groovy's MOP with aspect-specific functionality; in the following, these specialized meta-objects are called *meta-aspects*. MetaAspect exposes well-defined points in the interpretation of aspects, reflected in the interface of meta-aspects defined in MetaAspectProtocol, shown in Fig. 6. MetaAspectProtocol declares several meta-methods, one for each point in the interpretation of aspects that is open to specialization. The class MetaAspect implements the meta-methods, defining a default semantic for aspect interpretation.

There are several meta-methods modeling the *join point reception semantics*, one per each advice type (Fig. 6, lines 4–9). The default implementation of these meta-methods in MetaAspect finds advice whose pointcuts match the received join point and adds the corresponding PointcutAndAdvice to the list applicablePAs. Another group of meta-methods (lines 12–17) is concerned with the evaluation of pointcuts in the workflow triggered by the reception of join points. Meta-methods for join point reception internally call matchPointcut that defines the semantics of evaluating a single pointcut. Methods matchedPointcut and notMatchedPointcut

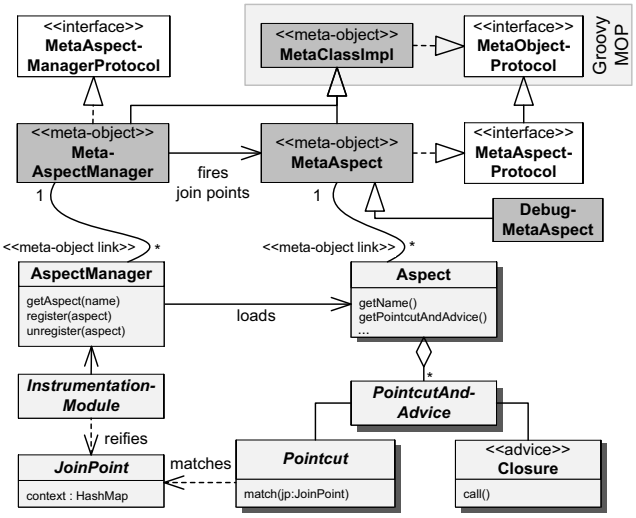

Figure 5: Overview of POPART's MAP

```
1  interface MetaAspectProtocol {
2
3  // join point reception semantics
4    void receiveBefore(Aspect aspect, JoinPoint jp,
5      List applicablePAs);
6    boolean receiveAround(Aspect aspect, JoinPoint jp,
7      List applicablePAs);
8    void receiveAfter(Aspect aspect, JoinPoint jp,
9      List applicablePAs);
10
11 // pointcut matching semantics
12   boolean matchPointcut(Aspect aspect, JoinPoint jp,
13     Pointcut pc);
14   void matchedPointcut(Aspect aspect, JoinPoint jp,
15     PointcutAndAdvice pa);
16   void notMatchedPointcut(Aspect aspect, JoinPoint jp,
17     PointcutAndAdvice pa);
18
19 // aspect interaction semantics
20   void interactionAtJoinPoint(Aspect aspect, JoinPoint jp,
21     Set aspects, List applicablePAs);
22 }
```

Figure 6: The MetaAspect Interface

are called whenever a pointcut matches, respectively does not match. Finally, interactionAtJoinPoint exposes aspect interaction semantics; whenever several aspects apply at a join point, this method is called on each of them.

The meta-link from an aspect to its meta-aspect plays an important role for the flexibility of our architecture. When adjusting an aspect's meta-link at run-time, i.e., exchanging the meta-aspect, other semantics will be used for that aspect instance (e.g., debugging support or customized advice ordering). By default, aspects share the same default meta-aspect instance, but each aspect instance may have its own special meta-aspect instance.

3.2.2 Pointcuts and Join Points

Pointcuts are represented by the subclasses of the abstract class Pointcut. For each designator in the primary interface, there is a corresponding subclass of Pointcut, e.g., for the method_call designator, there is the subclass class Method-

```
1  public aspect MethodCallInstrumentation {
2    Object around () : call(* *.*(..)) && !inExcludedShadows() {
3      AspectManager am = AspectManager.getInstance()
4      MetaAspectManager mam = am.getMetaAspectManager();
5      HashMap context = new HashMap();
6      context.put("method", thisJoinPoint.getMethodName());
7      context.put("args", thisJoinPoint.getArgs());
8      context.put("targetObject", thisJoinPoint.getTarget());
9      ...
10     Proceed _proceed = new Proceed() {
11       Object call(Object[] args) { return proceed(args);} };
12     context.put("proceed", _proceed);
13     JoinPoint jp = new MethodCallJoinPoint(...,context);
14     context.put("thisJoinPoint", jp);
15
16     mam.fireJoinPointBeforeToAspects(jp);
17     mam.fireJoinPointAroundToAspects(jp);
18     mam.fireJoinPointAfterToAspects(jp);
19     return context.get("result");
20 } }
```

Figure 7: Instrumentation for Method Call JPs

```
1  public interface MetaAspectManagerProtocol {
2    void loadAspects() {...}
3    void loadAspect(String name) {...}
4    void startup() {...}
5    void finalize () {...}
6
7    void fireJoinPointBeforeToAspects(JoinPoint jp);
8    void fireJoinPointAroundToAspects(JoinPoint jp);
9    void fireJoinPointAfterToAspects(JoinPoint jp);
10
11   Set calculateAspectInteractionSet(List applicPAs);
12   void interactionAtJoinPoint(JoinPoint jp, Set aspects,
13     List applicPAs);
14
15   void invokeAllApplicPAs(JoinPoint jp, List applicPAs);
16   Object invokeAdvice(JoinPoint jp, PointcutAndAdvice pa);
17 }
```

Figure 8: The MetaAspectManager Interface

`CallPCD`. Any pointcut implements the method `match` that takes a `JoinPoint` object as a parameter; e.g., the implementation of this method in `MethodCallPCD` matches against instances of `MethodCallJoinPoint`, testing whether the pattern of the receiver pointcut object is satisfied.

Join points are represented by subclasses of `JoinPoint`. For each join point type, a dedicated instrumentation module – an AspectJ program in the current implementation – instruments programs, such that objects of a corresponding subclass of `JoinPoint` are created at the execution of corresponding shadows. For illustration, the snippet in Fig. 7 shows the hook code to be inserted at all shadows of method call join points. The current implementation supports an AspectJ-like join point model. Other join point models, including domain-specific ones, can be accommodated by adding new subclases of `JoinPoint` and by exchanging the instrumentation modules.

3.3 Managing and Composing Aspects

Controlling aspect management and composition workflows is the responsibility of the aspect manager, the single instance of the `AspectManager` class, and its meta-object, an instance of `MetaAspectManager`. While the class `AspectManager` implements the fixed parts of the meta-aspect protocol responsible for aspect management and composition, the `MetaAspectManager` can be dynamically exchanged and allows run-time adaptation of management and composition semantics.

Unlike the meta-methods declared in `MetaAspectProtocol` that are responsible for the interpretation of individual aspect instances, the meta-methods of `MetaAspectManager` interpret aspects at a more coarse-grained level, i.e., adapting such a meta-method will change the interpretation of all aspects, and not only of one aspect. The role of the different meta-methods of `MetaAspectManager`, shown in Fig. 8, will be discussed in detail in the following sub-sub-sections.

3.3.1 From Aspect Programs to Meta-Level Entities

The aspect language supported by POPART is embedded into Groovy by exploiting its flexible syntax, closures, and its meta-object protocol. For an in depth-description of embedding domain-specific languages in Groovy we refer to previous work [11]. Aspect modules are defined in Groovy scripts that are compiled to Java bytecode [18]. The `MetaAspectManager` as part of the embedded POPART library is responsible for loading such scripts into POPART (methods `loadAspects()` and `loadAspect(String)` in Fig. 8, lines 2 and 3), thereby creating a graph of instances of the meta-level classes, which can than be executed. Fig. 9 shows the workflow for loading aspect scripts.

First, the method `loadAspect()` encloses each aspect into a Groovy `Closure` whose delegate field is set to be the `MetaAspectManager` instance (cf. Sec. 2.2). Besides the meta-methods of in Fig. 8, `MetaAspectManager` also implements the *primary interface*, in that it defines a method for each keyword used in aspect scripts, e.g., `aspect`, `method_call`), `before`, etc. Second, the created closure is called which starts the evaluation of the enclosed aspect script. When encountering an aspect keyword in the closure, the Groovy MOP maps it to a corresponding method call dispatched to the `MetaAspectManager` delegate. For example, the keyword `aspect` is mapped to a call to method `aspect` on the `MetaAspectManager` delegate, which creates a new instance of `Aspect`. Out of the pointcut designator keywords, a structure of meta-entities of type `Pointcut` is created (interactions inside the dashed box in Fig. 9). The `before` keyword will call the corresponding method that adds a `BeforePointcutAndAdvice` object to the created `Aspect` object, which refers to the pointcut hierarchy and the advice closure. Finally, the loaded `Aspect` is registered with the `AspectManager`.

To summarize, the result of loading an aspect is a graph of POPART meta-level instances. For illustration, the meta-level representation of the `ToyAspect` from the example program in Fig. 3 is shown in Fig. 10. Initially, the `Aspect` instance for `ToyAspect` is associated with a default `MetaAspect` instance (index 1). During the run of the program (from Fig. 3), the meta-object-link of the `ToyAspect` instance is changed from the default `MetaAspect` to the `DebugMetaAspect` (index 2).

After loading all aspects, a call to the `startup` method (Fig. 8, line 4) on the `MetaAspectManager` instance completes the initialization of POPART and the execution of the program is started. Once the program's execution is completed, `finalize` (line 5) will be called.

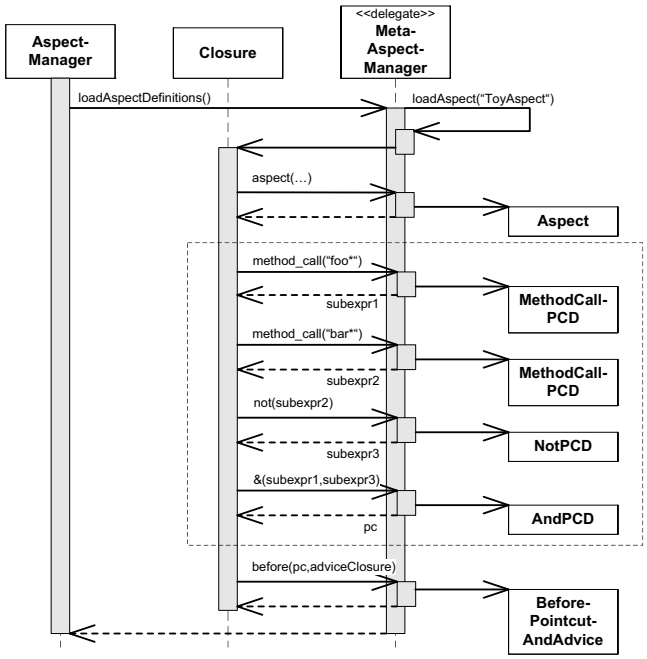

Figure 9: Sequence of Loading an Aspect

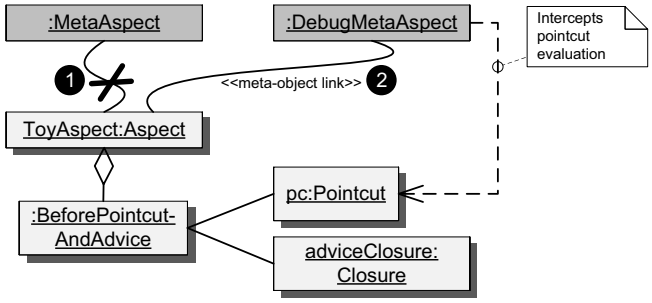

Figure 10: The Meta-Level of `ToyAspect`

3.3.2 Composing Aspects at Run-Time

The overall semantics of aspect composition is determined by the interaction of the meta-aspects and the `MetaAspectManager`. Aspect composition happens at run-time (after programs are instrumented to fire join point events and aspects are loaded) and follows an *open abstract aspect composition process* [30] that consists of four steps: *reify*, *match*, *order*, and *mix*.

In step *reify* program execution is intercepted at at each join point shadow. The hook code inserted by the instrumentation extracts the join point context and fires a new `JoinPoint` instance to the `MetaAspectManager`, which, in turn, passes it to the `MetaAspect` instance of each loaded aspect, thus entering the *match* step.

In step *match*, each `Aspect`'s `MetaAspect` determines what pointcuts match the current join point by passing it to the `match` method of each `Pointcut` instance referred to by the aspect. The result of this step is a list of advice to execute.

In step *order*, the lists of applicable advice retrieved from all aspects in the previous step are merged into one list by the `MetaAspectManager`, thereby determining the advice execution order. The default semantic implemented in `MetaAspectManager` is to order advice according to the sequence in which the advice and their declaring aspects have been loaded.

Finally, in step *mix*, the execution of program actions and advice is *mixed* by executing each advice in the order of the list resulting the *order* step. For *before* and *after*, all advice `Closure`s are called in sequence. For *around*, a special `Proceed` closure is used for wrapping multiple advice around a join point. Whenever `proceed` is called, the `Proceed` closure executes the next advice in the list of applicable advice. After all around advice have been called, the next call to `proceed` will execute the join point action.

3.4 Variation Points in POPART

POPART provides three main *variation points* (denoted VP1, VP2, and VP3 below) that have been selected to allow semantic variations found in the literature. One can extend POPART by defining subclasses of meta-entities (VP1), by defining subclasses of `MetaAspect` (VP2), and of `MetaAspectManager` (VP3), and setting up *abstract factories* [15]. In the following, we briefly mention possible extensions that could be provided with the current variation points. More elaborated scenarios will be given in the following section.

Users may specialize and extend all meta-level entities (VP1). One can e.g., create a subclass of `Aspect` to support dynamic activation, or to enable fine-grained scoping of aspects. Domain-specific join point models (JPM) [47] can be provided by subclassing `JoinPoint` and by implementing new instrumentation modules. For new join point types, new *primitive pointcut designators* can be provided by subclassing `Pointcut`. New abstract and higher-order designators can also be added.

The `MetaAspect` can be specialized by overriding its meta-methods (VP2). For example, by overriding `matchPointcut`, we can intercept pointcut evaluation for debugging or optimize pointcut matching using *partial evaluation*. To record online execution analysis, one may override `matchedPointcut` and `notMatchedPointcut` to count matching pointcuts. The method `interactionAtJoinPoint` can be overridden in order to refine the resolution strategy.

Meta-methods of `MetaAspectManager` (VP3) can be overridden to change the management and the composition semantics. One can adapt the *reify* step, e.g., by adding new meta-methods to receive join points of an alternative JPM, such as, the point-in-time JPM [35]. In step *match*, the evaluation of pointcuts can be adapted, e.g., to provide debugging support. The detection of aspect interactions and their resolution can be adapted in step *order*; we will show example scenarios for this in the next section. The step *mix* can be specialized to change the application of advice, e.g., to profile advice execution times or to support *concurrent advice*.

The *inversion of control* [14] enabled by callbacks to meta-methods in the control flow of the default implementation of the aspect semantics allows plug-ins comprising semantic specializations to be provided as user extensions. POPART has a fine-grained aspect meta-model in which the core aspect-oriented programming concepts are well isolated from each other. This allows to substitute them separately without affecting other parts of the implementation. For example, when changing the semantics of one pointcut designator, another pointcut designator's implementation should not be affected and the default implementation of aspect composition should not be invalidated.

Last but not least, semantic customizations can be applied dynamically by changing meta-object-links to meta-aspects or to the meta-aspect-manager. For example, as indicated in Fig. 10, during the run of the example program from Fig. 3, the meta-object-link of the `ToyAspect` instance is changed from the default `MetaAspect` (index 1) to the `DebugMetaAspect` (index 2). As a result, in step *match*, the `MetaAspectManager` will use the `DebugMetaAspect` for pointcut evaluation, which enables debugging for its pointcuts.

4. CREATING VARIANT LANGUAGES

In this section, we present scenarios of using the meta-level of POPART to specialize or extend the language semantics. First, a specialization of language semantics is realized by exchanging the meta-aspect of individual aspects to support debugging. Next, the part of the MAP concerned with aspect composition semantics is specialized to implement specific ordering strategies for co-advising aspects. Finally, the MAP is used to extend the primary interface.

4.1 Adding Pointcut Debugging Support

Although adequate support for debugging is crucial for the adoption of AOP, many AO languages lack such support. Especially, no adequate support for debugging pointcut evaluation down to the level of subexpressions and for debugging residual pointcuts is available. For performance reasons, many AO approaches perform *static weaving*, they partially evaluate pointcuts before run-time. But, since AO concepts lose their first-class status after static weaving, debugging is hindered [13].

In this section, we present a specialization of the default AO semantics realized by POPART to support debugging of pointcuts – in particular, pointcuts that cannot be fully evaluated statically – whereby all parts of the default semantics that are not affected by debugging are reused. MOPs have been used to support debugging of object-oriented programs; they allow visual debuggers to be implemented [27] and to be combined with program analysis. Similarly, our MAP can be used to extend aspect composition semantics with debugging support for pointcuts.

```
1  public class DebugMetaAspect extends MetaAspect {
2    boolean matchPointcut(aspect, jp, pc) {
3      boolean result = super.matchPointcut(aspect, jp, pc);
4      println "#Debug breakpoint at '$jp'";
5      println "#Pointcut under inspection = '${pc}'";
6      println "#It matches " + (result? "yes" : "no");
7      print   "#Please select:(1)step over,(2)step into, ...>";
8      String input = System.in.readLine();
9      ... // handle input option
10     return result;
11 } }
```

Figure 11: A Meta-Aspect for Debugging

A specialized meta-aspect class, called `DebugMetaAspect`, is shown in Fig. 11. It reuses aspect execution semantics by inheriting the default implementation of `MetaAspect` and overrides the `matchPointcut` meta-method to add interactive access to the pointcut evaluation for debugging via the console (lines 4–9). A *visual object inspector* similar to inspectors in Smalltalk [16] has also been developed on top of the MAP. This inspector allows to introspect the runtime state of meta-level entities, such as the current join point with the program context, the current pointcut, and the current aspect and meta-aspect. For brevity, only the console-based debugger is elaborated here.

When a pointcut is evaluated by a `DebugMetaAspect`, i.e., the method `matchPointcut` is called, the program execution is paused and the user is presented context information accessed by introspecting the reified meta-entities' state. For example, the current join point, the pointcut expression matched against, and the result of the match are displayed. Next, the user can interactively select (in Fig. 11, line 8) from a list of options. These options allow the user to specify the granularity at which the evaluation is traced, e.g., *step over* or *step into* functionality. Further, the user may also change the result of the evaluation, e.g., by *forcing* the subexpression *to match* or *not to match*.

The above debugging support can be used for an aspect by replacing the default meta-aspect with an instance of `DebugMetaAspect`. Recall that we have already seen such a scenario in the example in Fig. 3, line 21. When reaching line 24 in Fig. 3 with our interactive debugging support in place, the user can select an option to *step into* the pointcut. As effect, the tree structure of the pointcut evaluation is displayed as shown in Fig. 4 from Sec. 3.

Note that the example program in Fig. 3 defines a broken pointcut (lines 7–9). The pointcut still refers to the old method name `bar()` that was renamed to `baz()` (line 3). Such errors in pointcuts whose partial evaluation produces residuals, e.g., errors in `cflow` subexpressions, (also called residual pointcuts in the following) are hard to find because most existing debugging support only helps to find errors in parts of pointcuts that can be statically analyzed. The debugging support implemented with the MAP can be used to trace errors in residual pointcuts. Consider again the tree structure in Fig. 4 that shows details of the broken pointcut's evaluation at run-time; a join point is fired that matches, but that is not expected to match against the pointcut (e.g., `foo()` is called in `baz()`). The user can trace the bug in the tree structure by following the evaluation of subexpressions

as highlighted in bold. The user notices that the evaluation of subexpression `not(cflow(method_call("bar.*")))` (in line 7) matches unexpectedly because the renamed method `baz()` is on stack.

Adding debugging support is only one example of a language specialization that can be realized by attaching specialized meta-aspects to individual aspect instances. Other specializations can be realized in a similar way. For example, we have implemented specializations for online execution analyses that profile the execution times of advice, or automatically detect *co-advising aspect interactions* [34].

4.2 Customizing Aspect Composition

The specialization presented so far has adapted the behavior of single aspects. Another form of specialization is to adapt the interplay of aspects realized by the steps in the composition process, e.g., to customize advice ordering semantics. In Sec. 4.2.1, the MAP is used to implement an extensible advice ordering mechanism, which can be adapted at run-time to resolve context-dependent aspect interactions [41]. In Sec. 4.2.2, we discuss an example scenario from the telecommunication domain, in which dynamic aspect interaction occurs, that can be resolved using the presented extension.

4.2.1 Customizing Advice Ordering

Many aspect-oriented languages and systems do not adequately manage *aspect interactions* [12]. An important type of aspect interaction occurs where multiple pointcuts match a given join point [12, 7, 44, 34], thus several pieces of advice will *co-advise* [34] the shared join point[6]. If advice are not executed in the right order, co-advising may lead to conflicts. Different conflict resolution approaches [12, 7, 37, 46] and language extensions for specifying advice execution order have been proposed. Also aspect interactions have been found to be domain-specific [31, 19]. This suggests that there possibly is no general ideal solution for aspect interaction, therefore extensible advice ordering mechanisms and conflict resolution strategies [46, 31] are needed.

In Fig. 12, a specialized meta-aspect-manager class, called `OrderedMetaAspectManager`, is shown; it specializes the *order* step in the abstract aspect composition process to order advice in case of aspect interactions right before executing them, hence, resolving co-advising interactions. `OrderedMetaAspectManager` inherits the aspect composition semantics from `MetaAspectManager` and overrides the `interactionAtJoinPoint` meta-method. Recall that in the default implementation of the MAP, this meta-method is called whenever more than one pointcut-and-advice is found to be applicable at a join point during the *match* step.

The overridden meta-method uses the standard sort operation provided by the *Java Collection Framework* passing to it the set of advice to order and a comparator, which is delivered by a factory object (line 6). The implementation of `interactionAtJoinPoint` in `OrderedMetaAspectManager` is a template for ordering strategies. By using the factory pattern to retrieve the comparator, this method introduces a new variation point that allows further adaptations to provide customized (application-specific) advice ordering. New ordering strategies can be implemented by providing a new

[6]Interactions other than *co-advising* [31] are out of scope of this paper and will be addressed in the future.

```
1 class OrderedMetaAspectManager
2         extends MetaAspectManager {
3   ...
4   void interactionAtJoinPoint(jp, aspects, applicablePAs) {
5     super.interactionAtJoinPoint(jp, aspects, applicablePAs);
6     Comparator comparator = AspectFactory.getComparator();
7     Collections.sort(applicablePAs, comparator);
8   }
9 }
```

Figure 12: Ordered Advice Executions

implementation of `Comparator`. Because we acquire a `Comparator` via the factory the ordering strategy can be replaced at run-time.

Based on the template for advice ordering strategies, we have implemented several reusable advice ordering strategies as variations of the *ordered meta-aspect-manager*. An intuitive semantics of advice ordering is based on *priorities*. In this implementation the class `Aspect` is extended with a field `priority` and a `Comparator` is provided that orders pointcut-and-advice according to the priority value of their aspects. Another strategy is *rule-based* and allows to define precedence rules at the level of pointcut-and-advice. An *advice-type*-specific strategy uses different ordering strategies for *before*, *around*, and *after* advice. A generic *combinator* strategy allows to combine different strategies, e.g., to automatically apply the priority-based strategy, whenever the rule-based strategy does not define a specific order. In the following, we discuss how we can use the ordering mechanism to implement a strategy, where the order to choose depends on the application state.

4.2.2 Resolving Context-Dependent Aspect Interactions

An challenging kind of aspect interactions are those whose emergence depends on the dynamic state of a program [41, 12, 37, 38, 34, 28]. For illustration, consider the two aspects in Fig. 13 that implement two features of a phone management system [23, 37]: (a) the call forwarding supplementary service feature initiates call transfers to other phones, if the calls are not answered, and (b) the answering machine forwarding feature.

Both *Alice* and *Bob* have a `Phone` (line 16 and 17) and each phone has an `AnswerMachine` (line 3) which can be activated (line 9). In lines 20–29, the aspect `Alice_To_AM4Alice` is defined that forwards *Alice*'s received calls to her answer machine, if the call is not answered and the answer machine is active. Note that the aspect's parameter `perInstance:alice` defines the aspect to be instance-local[7], so that only calls on the phone `alice` will be advised. Another aspect, `Alice_To_Bob` (lines 31–38), forwards *Alice*'s un-answered calls to her `forwardPhone`, which is set to the Phone `bob` (line 18). Finally, `Bob_To_AM4Bob` (lines 40–43), forwards *Bob*'s un-answered calls to his answer machine.

Alice prefers that calls are answered by her colleague *Bob*, if the latter is available. Hence, in first sight the desired ordering strategy seems to be that `Alice_To_Bob` is given higher priority than `Alice_To_AM4Alice`. However, with the

[7]POPART's support for dynamic AOP is discussed in Sec. 4.3. This includes *dynamic deployment* and *instance-local deployment*, similar to existing approaches [10, 39, 42].

```
1  class Phone { //Phone.java
2    boolean receiveCall(String phoneNumber) {...}
3    AnswerMachine getAnswerMachine() {...}
4    Phone getForwardPhone() {...}
5    void setForwardPhone(Phone forwardTo) {...}
6  }
7
8  class AnswerMachine extends Phone { //AnswerMachine.java
9    boolean active = false;
10   boolean receiveCall(String phoneNumber) {
11     ...
12     return active;
13 } }
14
15 //Remaining lines in POPART
16 Phone alice = new Phone("Alice");
17 Phone bob = new Phone("Bob");
18 alice.setForwardPhone(bob);
19
20 aspect(name:"Alice_To_AM4Alice",
21        perInstance: alice ) {
22   around(method_execution("receiveCall.*")) {
23     boolean answered = proceed();
24     AnswerMachine am = targetObject.getAnswerMachine();
25     if (!answered && am.active) {
26       answered = am.receiveCall(args[0]);}
27     }
28     return answered;
29 } } };
30
31 aspect(name:"Alice_To_Bob",
32        perInstance: alice ) {
33   around(method_execution("receiveCall.*")) {
34     boolean answered = proceed();
35     Phone bob = targetObject.getForwardPhone();
36     if (!answered) {answered = bob.receiveCall(args[0]);}
37     return answered;
38 } } };
39
40 aspect(name:"Bob_To_AM4Bob",
41        perInstance:bob) {
42   // similar to the aspect Alice_To_AM4Alice
43 };
44
45 alice.receiveCall("+1-555-444-3333");
46 ...
```

Figure 13: A Phone Management System

`Alice_To_Bob`-before-`Alice_To_AM4Alice` strategy in place, the following undesired scenario may happen, if *Bob* has activated his answer machine. *Alice* does not answer a call, and `Alice_To_Bob` forwards it to *Bob*. Next, *Bob* also does not answer, and `Bob_To_AM4Bob` will forward the call to *Bob*'s answer machine. Obviously *Alice* prefers that calls are recorded on her own answer machine instead of the answer machine of someone else.

So, what is the right order for executing the co-advising aspects `Alice_To_Bob` and `Alice_To_AM4Alice`? As a matter of fact, any static ordering, e.g., based on priorities, will possibly be wrong, as the right order of advice depends on a dynamic condition, namely whether *Bob*'s answer machine is active or not. An advice order strategy is required that takes into account the dynamic context of the application, in this case, the activation status of *Bob*'s answer machine.

Our meta-aspect protocol can be used to implement and activate an advice execution order strategy that takes into account dynamic context information by using the reflective access provided by the MAP. The `PhoneForwardComparator` in Fig. 14 is tailored for solving the conflict in the above example, defining an advice ordering mechanism that takes

```
1  class PhoneForwardComparator<T extends PointcutAndAdvice>
2          implements Comparator<T> {
3    int compare(T pa1, T pa2) {
4      if (pa1.aspect.name.equals("Alice_To_Bob") &&
5          pa2.aspect.name.equals("Alice_To_AM4Alice")) {
6        Phone fromPhone = thisJoinPoint["targetObject"];
7        Phone toPhone = fromPhone.getForwardPhone();
8        if (toPhone.getAnswerMachine().active) {
9          return LOWER_PRECEDENCE;
10       else
11         return HIGHER_PRECEDENCE;
12     } else {
13       return AspectFactory.getDefaultCmp().compare(pa1,pa2);
14 } } }
```

Figure 14: Dynamic Advice Ordering

into account the state of the interacting aspects at a join point. If the two interacting pointcut-and-advice stem from aspects `Alice_To_AM4Alice` and `Alice_To_Bob` (lines 4–5), the advice defined in `Alice_To_Bob` gets a higher precedence if and only if `Bob`'s answer machine is turned off (cf. line 8). Otherwise, the advice of `Alice_To_AM4Alice` gets a higher precedence. If the two pointcut-and-advice do not stem from `Alice_To_AM4Alice` and `Alice_To_Bob`, the default priority-based advice ordering is used.

The accessible application context on which a `Comparator` may depend includes (a) the program state (e.g., objects, stack), (b) the aspect context (e.g., aspects and all their subcomponents), (c) the state of the composition mechanism (e.g., aspect interaction set and all applicable pointcut-and-advice at a join point), and (d) other external sources. Due to this broad reflective access, the MAP of POPART could be used for resolving other kinds of *semantical interferences* [12, 46, 28, 41]. For instance, one can intercede the list of applicable advice in order to enforce *mutual exclusion* or *implicit cuts* [46], by removing or adding pointcut-and-advice from the list.

4.3 Extending the Primary Interface

One may want to extend the primary interface to add a new feature to the aspect language, e.g., dynamic aspect deployment. For instance, in Fig. 15, a logging aspect is defined. Unlike other POPART aspects seen so far, this aspect has a new parameter `deployed:false` in line 1. As the result, the logging aspect is not immediately active after it is loaded by the aspect manager. Next, the program is executed three times. After the first run, the aspect is *deployed* by calling the method `deploy()` on it. This method is not provided by instances of `Aspect`, the default implementation of aspects in POPART. Subsequently, when executing the program the second time the logging aspect will advise the program. Next, the aspect is *undeployed*, again leaving the program unadvised in the third run.

In this scenario, the application uses the additional aspect abstractions: the `deployed` parameter in the aspect definition and the `deploy`/`undeploy` methods. To support these additional abstractions, and hence to realize dynamic deployment, we have implemented a class `DynamicAspect` that extends `Aspect` and holds an instance field `deployed` indicating the deployment status of its instances. Calling `deploy` on such an aspect instance will set the `deployed` field, hence activating the aspect. In contrast, `undeploy` resets the field and deactivates the aspect instance.

```
1 def a = aspect(name:"DynLogAspect",deployed:false) {
2   before( method_call(".*") ) {
3     println "$thisObject calls $targetObject";
4 } }
5 Program.main();
6 a.deploy();
7 Program.main();
8 a.undeploy();
9 Program.main();
```

Figure 15: A Dynamic Logging Aspect

```
1 class DynamicMetaAspect extends MetaAspect {
2   void receiveBefore(aspect,jp,applicablePAs) {
3     if (!((DynamicAspect)aspect).deployed) return; ...
4     super.receiveBefore(aspect,jp,applicablePAs);
5   }
6   ...
7 }
```

Figure 16: An Extension for Dynamic AOP

A *dynamic aspect* is composed by using the specialized meta-aspect class `DynamicMetaAspect`. When join points are fired to such a meta-aspect, it only matches pointcuts and executes advice if the aspect instance is deployed. The methods `receiveBefore`, `receiveAround`, and `receiveAfter` are all overridden in the same way. E.g., in Fig. 16 line 3, a dynamic condition checks whether `deployed` is `true`. In this case, `DynamicMetaAspect` forwards to the corresponding `super` method in `MetaAspect`, which composes the aspect. In contrast, the aspect is not composed if the condition evaluates to `false`. It is worth mentioning that, in addition to this *global-scoped* deployment mechanism, `DynamicAspect` also provides means for *instance-scoped* and *class-scoped* deployment similar to Steamloom [42] and CaesarJ [10].

To summarize, the MAP allowed us to implement dynamic activation strategies for aspects in a modular way. The extension changed the primary aspect interface, since new parameters are used in the aspect definitions and additional methods for deployment are provided. Examples of other kinds of extensions to the primary interface that could be realized by means of the MAP are the definition of new (domain-specific) join points and new (domain-specific) pointcut designators by adding new keywords to the *primary interface* under which aspects are evaluated (cf. Sec. 3.3.1). For example, we have implemented a domain-specific join point model for a workflow-oriented language; extending POPART with join points and pointcut designators for field access is another example extension.

5. RELATED WORK

The variation points in the design-space of POPART have been inspired from semantic variations of AO concepts found in related work.

The specialization for supporting dynamic AOP presented in Sec. 4.3 bears similarity to corresponding concepts in other systems with built-in support for dynamic AOP [10, 39, 42]. Unlike POPART, the language constructs for dynamic AOP and their respective semantics is fixed into the implementations of these systems. Further, none of these approaches provides a general solution to application-level customization of several parts of language semantics.

The advice order mechanism presented in Sec. 4.2.1 considers results of [44, 46, 28, 34, 19]. An instance for application-level semantic adaptations in AO solutions are customizable resolutions of *aspect interactions*, that change the composition semantics of aspects. Several approaches have been proposed, such as *logical meta-programming* [7], special *composition modules* that allow to specify logic to order advice [44], as well as providing *rules* to explicitly order, include, and exclude advice [45, 46].

These approaches support to a certain degree the customization of aspect interaction semantics, however, conflict resolution is not the only semantical variation in aspect languages. Moreover there are two major problems with the above approaches. First, the advice ordering strategies cannot dynamically be changed based on dynamic context. Second, there is a technology break between the way aspects are specified and how conflicts are resolved. On the contrary, our approach allows to specify aspects and their resolution logic – as well as other semantic variations – in the same language by either using the primary interface or the meta-interface. Moreover, the MAP allows the user to define a resolution strategy for context-dependent aspect interactions.

Extensible aspect-oriented language infrastructures [44, 45, 2, 46, 31] allow semantics to be adapted when building a special compiler or a special runtime. To the best of our knowledge, none of these systems support adaptations at run-time by users at the application-level. The *abc* compiler is often used for defining new kinds of pointcuts [2] as extensions of AspectJ. The *Aspect SandBox* (ASB) supports prototyping alternative AOP semantics and implementation techniques [36], e.g., new kinds of pointcut designators and alternative weaving techniques. However, providing new concepts and semantics results in excessive changes throughout the interpreter, e.g., when a new type of join point is added [47]. This is because ASB does not provide clear interfaces for controlling the underlying implementation strategy. There is no meta-interface available to programmers for tailoring AO semantics at the application level. In contrast to the extensible AO solutions above, POPART allows the adaptation of semantics at run-time by providing a meta-interface that can be extended using well-known object-oriented techniques.

Reflex [45, 46], *JAMI* [19], *MetaSpin* [8], and FIAL [6] are aspect-oriented runtimes that employ aspect-oriented meta-models that are open for alternative AO semantics when the run-time is build. In their meta-models, interfaces for AO abstractions are provided that can be extended, e.g., with new pointcut designators or composition strategies. The four approaches differ in their objectives and implementation techniques which are, however, not relevant for a comparison to our approach. In general, they do not support changing the AO semantics at run-time.

Reflection and MOPs have been used to implement AOP technology by adapting the semantics of objects. This approach has been followed by Sullivan [43], *AspectS* [20, 21], Lorenz and Kojarski [33, 29], *Composition Filters* [3, 4], Tanter [45], *AspectLua* [9], *Aquarium* [48], and *GroovyAOP* [22]. However none of the MOP-based solutions comes with a well-designed meta-aspect protocol to complement MOPs with a meta-interface that allows programmers to adapt the

semantics of aspects. Moreover, adapting the semantics of aspect composition both at the application-level and at run-time has not been in the focus.

Kojarski and Lorenz [33, 29] analyze the relation of reflection and AOP and argue that AOP is a first-class computational reflection mechanism and therefore is at the same level as reflection. In this work, we have presented the MAP at the same level as the MOP.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we proposed an architecture for a meta-aspect protocol that enables application developers to adapt the semantics of the aspect language at run-time. The benefits of the proposed architecture were demonstrated by deriving several aspect-oriented variant languages.

We conclude the paper by discussing in what extent the proposed architecture meets the following requirements for the design and implementation of a MOP defined by Kiczales et al. [27], which also apply to a meta-aspect protocol.

1. *Robustness.* Altering one part of the protocol should not affect other parts of the MOP implementation.
2. *Abstraction.* The user does not need to know the details of the language implementation.
3. *Ease of use.* The change of the default implementation should be natural and straight forward.
4. *Efficiency.* The flexibility of the MOP should not undermine the performance of the default language.

The first requirement is met to a large extent due to the fine-granularity of the meta-model underlying our MAP, where each AO concept is represented by a designated meta-level entity. Yet, further empirical assessment of the expressiveness of the architecture in general and of the meta-interface in particular needs to be conducted. Future work needs to explore the ability of our MAP to support an application-level implementation of further AO language features. For instance, we plan to explore in how far and with which benefits a security infrastructure designed for AOP could be provided on top of our meta-aspect protocol.

The second requirementis met because the meta-interface is an abstraction of the language implementation. Due to the integration of our MAP into Groovy's MOP and due to the implementation of POPART as an embedded language, changing the language semantics is as easy as changing designated fields of objects; hence, the third requirement is also met.

While this paper focuses on the flexibility provided by MAP, *efficiency* and *performance* issues related to the fourth requirement have not been in the focus and will be addressed in future work. Although the POPART code is subject to *adaptive optimization* provided by the just-in-time compiler of the Java VM due to the tight integration of Groovy into Java, a large run-time overhead has been measured for the Groovy-specific features especially for the Groovy MOP. It would be interesting to investigate if the *Groovy JIT* [22] is also suited to reduce the overhead imposed by our MAP. Also, *optimistic optimizations* similar to those proposed for MOPs [27, 43] could be applied. *Partial reflection* [45] could also be investigated as a means to optimize the reification overhead in our instrumentation modules. We would like to use Steamloom [42] for a dynamic adjustable instrumentation, which could virtually remove our run-time overhead in case no aspects are defined by withdrawing the instrumentation if it is not needed.

7. ACKNOWLEDGMENTS

This work was partly supported by the *feasiPLe* project, Federal Ministry of Education and Research (BMBF), Germany. We would like to thank Michael Eichberg, Vaidas Gasiunas, and Martin Monperrus for valuable discussions and the anonymous reviewers for their comments.

8. REFERENCES

[1] P. Avgustinov, T. Ekman, and J. Tibble. Modularity first: A Case for mixing AOP and Attribute Grammars. In *AOSD*, pages 25–35, 2008.

[2] P. Avgustinov, J. Tibble, A. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, and G. Sittampalam. abc: An extensible AspectJ Compiler. In *AOSD*, pages 87–98, 2005.

[3] L. Bergmans and M. Aksit. Composing Crosscutting Concerns using Composition Filters. *Communications of the ACM*, 44(10):51–57, 2001.

[4] L. Bergmans and M. Aksit. Principles and Design Rationale of Composition Filters. *Aspect-Oriented Software Development. Addison-Wesley*, pages 0–32, 2004.

[5] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual Machine Support for Dynamic Join Points. In *AOSD*, pages 83–92, 2004.

[6] C. Bockisch, M. Mezini, W. Havinga, L. Bergmans, and K. Gybels. Reference Model Implementation. Technical Report AOSD-Europe-TUD-8, Technische Universität Darmstadt, 2007.

[7] J. Brichau, K. Mens, and K. De Volder. Building Composable Aspect-Specific Languages with Logic Metaprogramming. In *GPCE*, pages 110–127, 2002.

[8] J. Brichau, M. Mezini, J. Noyé, W. Havinga, L. Bergmans, V. Gasiunas, C. Bockisch, J. Fabry, and T. D'Hondt. An Initial Metamodel for Aspect-Oriented Programming Languages. `http://www.aosd-europe.net/deliverables/d39.pdf`, 2006.

[9] N. Cacho, T. Batista, and F. Fernandes. AspectLua-A Dynamic AOP Approach. *Journal of Universal Computer Society*, 11(7):1177–1197, 2005.

[10] CaesarJ Homepage. `http://caesarj.org/`.

[11] T. Dinkelaker and M. Mezini. Dynamically Linked Domain-Specific Extensions for Advice Languages. In *Workshop on Domain-Specific Aspect Languages*, 2008.

[12] R. Douence, P. Fradet, and M. Südholt. A Framework for the Detection and Resolution of Aspect Interactions. In *GPCE*, pages 173–188, 2002.

[13] M. Eaddy, A. Aho, W. Hu, P. McDonald, and J. Burger. Debugging Aspect-Enabled Programs. *Symposium on Software Composition*, pages 200–215, 2007.

[14] M. Fayad, D. Schmidt, and R. Johnson. *Building Application Frameworks: Object-Oriented Foundations of Framework Design.* John Wiley, NY, USA, 1999.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[16] A. Goldberg and D. Robson. *Smalltalk-80: the Language and its Implementation*. Addison-Wesley, Boston, MA, USA, 1983.

[17] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Mass, 2000.

[18] The Groovy Home Page. `http://groovy.codehaus.org/`.

[19] W. Havinga, L. Bergmans, and M. Aksit. Prototyping and Composing Aspect Languages using an Aspect Interpreter Framework . In *ECOOP*, pages 180–206, 2008.

[20] R. Hirschfeld. AspectS: AOP with Squeak. In *OOPSLA Workshop on Advanced Separation of Concerns in OO Systems*, 2001.

[21] R. Hirschfeld. AspectS: Aspect-Oriented Programming with Squeak. In *Netobjectdays (NODe)*, pages 216–232, 2003.

[22] C. Kaewkasi and J. Gurd. Groovy AOP: A Dynamic AOP System for a JVM-based Language. In *Workshop on Software Engineering Properties of Languages and Aspect Technologies*, 2008.

[23] D. O. Keck and P. J. Kuehn. The Feature and Service Interaction Problem in Telecommunications Systems: A Survey. *IEEE Trans. Softw. Eng.*, 24(10):779–796, 1998.

[24] G. Kiczales. Beyond the Black Box: Open Implementation. *IEEE Software*, 13(1):8–11, 1996.

[25] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *ECOOP*, pages 327–353, 2001.

[26] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *ECOOP*, pages 220–242, 1997.

[27] G. Kiczales, J. d. Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, 1991.

[28] G. Kniesel. Detection and Resolution of Weaving Interactions. *TAOSD: Dependencies and Interactions with Aspects*, LNCS, 2007. Special Issue on Aspect Dependencies and Interactions, edited by R. Chitchyan.

[29] S. Kojarski and D. Lorenz. AOP as a First Class Reflective Mechanism. In *OOPSLA*, pages 216–217, 2004.

[30] S. Kojarski and D. Lorenz. Modeling Aspect Mechanisms: A top-down Approach. In *ICSE*, pages 212–221, 2006.

[31] S. Kojarski and D. Lorenz. Awesome: an Aspect Co-Weaving System for Composing Multiple Aspect-Oriented Extensions. In *OOPSLA*, pages 515–534, 2007.

[32] D. König and A. Glover. *Groovy in Action*. Manning, 2007.

[33] D. Lorenz and S. Kojarski. Reflective Mechanisms in AOP Languages. Technical report, Northeastern, 2003.

[34] D. Lorenz and S. Kojarski. Understanding Aspect Interactions, Co-Advising and Foreign Advising. In *ECOOP Workshop Aspects, Dependencies and Interactions*, Berlin, Germany, 2007.

[35] H. Masuhara, Y. Endoh, and A. Yonezawa. A Fine-Grained Join Point Model for More Reusable Aspects. *LNCS Programming Languages and Systems*, 4279:131, 2006.

[36] H. Masuhara, G. Kiczales, and C. Dutchyn. A Compilation and Optimization Model for Aspect-Oriented Programs. In *CC 2003*, volume 2622 of *LNCS*, pages 46–60, 2003.

[37] J. Pang and L. Blair. An Adaptive Run Time Manager for the Dynamic Integration and Interaction Resolution of Features. In *Distributed Computing Systems*, pages 445–450, 2002.

[38] J. Pang and L. Blair. Separating Interaction Concerns from Distributed Feature Components. *Electronic Notes in Theoretical Computer Science*, 82(5):70–84, 2003.

[39] The PROSE Homepage. `http://prose.ethz.ch/Wiki.jsp`.

[40] A. Rashid. Aspects and Evolution: The Case for Versioned Types and Meta-Aspect Protocols. In *Workshop on Reflection, AOP, and Meta-Data for Software Evolution*, 2006.

[41] F. Sanen, E. Truyen, and W. Joosen. Modeling Context-Dependent Aspect Interference using Default Logics. In *ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution*, 2008.

[42] The Steamloom Homepage. `http://www.st.informatik.tu-darmstadt.de/Steamloom`.

[43] G. Sullivan. Aspect-Oriented Programming using Reflection and Metaobject Protocols. *Communications of the ACM*, 44(10):95–97, 2001.

[44] D. Suvée, W. Vanderperren, and V. Jonckers. JAsCo: An Aspect-Oriented Approach tailored for Component based Software Development. In *AOSD*, pages 21–29, 2003.

[45] É. Tanter. *From Metaobject Protocols to Versatile Kernels for Aspect-Oriented Programming*. PhD thesis, Université de Nantes, France, 2004.

[46] E. Tanter. Aspects of Composition in the Reflex AOP Kernel. *LNCS*, 4089:98, 2006.

[47] N. Ubayashi, H. Masuhara, and T. Tamai. An AOP Implementation Framework for Extending Join Point Models. In *ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution*, 2004.

[48] D. Wampler. Aquarium: AOP in Ruby. In *AOSD*, 2008.