Composing Aspects with Aspects

Antoine Marot*
Université Libre de Bruxelles
Brussels, Belgium
amarot@ulb.ac.be

Roel Wuyts
IMEC and KULeuven
Leuven, Belgium
roel.wuyts@imec.be

ABSTRACT

Aspect-oriented programming languages modularize crosscutting concerns by separating the concerns from a base program in aspects. What they do not modularize well is the code needed to manage interactions between the aspects themselves. Therefore aspects cannot always be composed with other aspects without requiring invasive modifications. This paper proposes a join point model that makes it possible to express aspect composition itself as an aspect, liberating the composed aspects from composition-specific code. We have implemented this new join point model in our OARTA language, an extension of ASPECTJ, and we show how to use it to write aspects that manage aspect interactions.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Feature—*Classes and Objects*

General Terms

Languages

Keywords

Aspect composition, Aspect interaction, Crosscutting concern, Semantic interference

1. INTRODUCTION

Aspect-oriented programming (AOP) is a programming paradigm to modularize crosscutting concerns. The concerns are specified in modules called *aspects* and are composed with a base system during a process called *weaving*.

AOP has proven to be good way to modularize concerns that crosscut the base code. Unfortunately aspects that were not designed from the ground up to collaborate often conflict when used together. For example it is known that semantic interferences can occur when using an aspect for encrypting information in combination with an aspect that finds and removes inappropriate words in a text [1]. In order to avoid semantic interferences, the conflictual aspect interactions need to be managed.

Aspect interference is a practical problem and therefore several techniques were developed to help with *detecting* aspect interactions. However, few techniques exist to then *manage* the interactions to remove the conflicts and obtain the desired behaviour. The state-of-the-art solutions boil down to the following two non-optimal solutions:

- use precedence rules between the aspects that are being composed. The advantage is that the precedence is given by the composer of the system without needing to change the composed aspects. The disadvantage is that the composition is coarse-grained and limited because it means that one aspect will *always* take precedence over another aspect, which does not fit all usage scenarios. Moreover, several kinds of interactions are not related to the ordering of the aspects.

- destructively change the aspects to make them aware of each other. This solution makes it possible to have a fine-grained composition but implies that all aspects that participate in the composition are modified. While the aspects are therefore separated from the base code, they are no longer separated from each other and cannot be used in other usage scenarios.

This paper explores a novel way for managing aspect interactions by empowering aspects to express aspect compositions. This makes it possible to manage aspect interactions outside of the aspects themselves, liberating the composed aspects from any composition-specific code.

The contribution of this paper is composed of three parts:

- We identify the lack of support to modularize composition concerns that crosscut aspects and motivate the need for aspects to compose aspects.

- We propose the following AO language features to express aspect composition aspects: *named advices* to precisely identify advices, *advice patterns* to quantify over advices, *foreign pointcut modifications* to extend existing pointcuts, *weaving orderings* specified at the advice level using the advice patterns, and *user-defined instantiation policies* for foreign aspects. Our proposition encompasses different AOP language models, features and capabilities.

*Research Fellow of the *Fond National de la Recherche Scientifique* (FNRS-FRS)

AOSD'10 March 15–19, Rennes and St. Malo, France

- We extend AspectJ with these features. The language extension (called Oarta[1]) has been implemented using the extensible *AspectBench Compiler* (abc) [4].

To validate the usefulness of aspects to compose aspects we show how to use Oarta to implement several scenarios of composition concerns.

The remainder of the paper is structured as follows: Sec. 2 shows the kinds of problematic interactions that can occur when aspects are composed and lists the invasive modifications needed to make the compositions possible. Sec. 3 describes the features we propose that empowers aspects to express aspect compositions. Sec. 4 presents Oarta and evaluates it on several examples. Sec. 5 provides a discussion while Sec. 6 discusses related work. Finally, Sec. 7 concludes the paper.

2. INVASIVE ASPECT COMPOSITION

It is well-known that aspects that coexist in an application may badly interact with each other and cause unwanted interference that needs to be taken care of. This can currently be done in two unsatisfactory ways. The first is by giving explicit precedence relationships between the aspect at composition time. Unfortunately not all interactions are related to the execution order, and a precedence relation alone is therefore not sufficient to handle all possible interaction problems. The second option currently is to invasively modify the aspects that interfere to remove the unwanted interaction of a particular composition.

This section illustrates these problems with a concrete example of an online editing application. Based on this example we then discuss in more detail what elements of an aspect may require to be modified to eliminate unwanted interference with another aspect.

Note that in Sec. 4.3 we revisit the example given in this section and show how to manage the interactions in a non-intrusive way using our Oarta language.

2.1 Motivating Example

A company has developed an online document editing application. Users are allowed to read or edit documents after they have successfully logged in. Their actions are monitored for which a logging aspect has been implemented (see Fig. 1). The first advice (lines 5–9) logs login attempts for security reasons while the other advice (lines 11–15) keeps track of any action performed on a document.

We will now make various extensions of this program to highlight various practical aspect interaction problems.

2.1.1 *Adding a* Turbo *feature*

Runtime analysis determined that execution performance decreases heavily when many users are connected. After analysis the company came up with the following strategy to speed up the execution when necessary:

- when the number of connected users exceeds a particular threshold the secondary program features are deactivated and a buffering cache is activated to delay saves of the most used documents.
- as soon as the number of connected users dips below the threshold the secondary features are (re)activated and the buffering cache is deactivated.

[1]stands for *O*ne *A*spect to *R*ule *T*hem *A*ll.

```
1  public aspect Logging {
2
3    LogFile logF=...;
4    /* primary feature */
5    before(String login, String pswd):
6          call(* User.login(String,String))
7          && args(login,pswd) {
8      logF.logLoginAttempt(login,pswd);
9    }
10   /* secondary feature */
11   before(Document doc): call(* Document.*(..))
12                && this(doc) {
13     logF.logDocAction(doc,
14          thisJoinPoint.getSignature());
15   }
16
17 }
```

Figure 1: A logging aspect

```
1  public aspect CountActions {
2    private int Document.count=0;
3
4    public int Document.getCount() { ... }
5    public void Document.setCount(int i) { ... }
6
7    before(Document doc): call(* Document.*(..))
8          && !call(* Document.*Count(..))
9          && target(doc) {
10     doc.setCount(doc.getCount()+1);
11   }
12 }
```

Figure 2: A counting aspect

The part of the logging aspect that logs the login attempts is considered to be a primary feature for security reasons and should therefore always be active. The part that keeps track of what actions are performed on documents is considered to be a secondary feature and should be deactivated when needed.

To be able to select what documents are used the most, which is needed to implement the performance strategy, an aspect is implemented to count actions performed on documents (see Fig. 2). This aspect adds a counter variable to each document (line 2) and increments it for each action performed (lines 7–11).

The buffering cache responsible to delay document saves is implemented as an aspect as well (see Fig. 3). This aspect is instantiated and bound to each document on which a save action is performed (lines 1 and 4). Every instance of this aspect has its own counter of already delayed saves (line 2). Only when it reaches a certain number will the saves be done (by using proceed) (lines 6–11).

On top of these two aspects, an aspect called Turbo (see Fig. 4) is responsible to manage the (de)activation of the buffering cache and the secondary features with regards to the number of connected users. This functionality is achieved by toggling a flag when the amount of users exceeds a certain threshold (lines 5–15) and by referencing this flag when needed in the other aspects. The rest of the aspect manages the list of most used documents: when the action counter of a document is modified (lines 21–23) the list of the ten most used documents is updated (lines 24–27).

Impact on aspects.

While the solution given above works and manages the interactions between aspects there are many problems with

```
1  public aspect DelaySave pertarget(save()) {
2    int delayedsaves=0;
3
4    pointcut save(): call(* Document.saveOnDisk());
5
6    Object around(): save() {
7      if(++delayedsaves>10) {
8        proceed();
9        delayedsaves=0;
10     }
11   }
12 }
```

Figure 3: A buffer caching aspect

```
1  public aspect Turbo {
2    int threshold=100;
3    public static boolean activated=false;
4
5    after(): call(* User.login(..)) {
6      if(User.nbOnlineUsers()>threshold) {
7        activated=true;
8      }
9    }
10
11   after(): call(* User.logout(..)) {
12     if(User.nbOnlineUsers()<threshold) {
13       activated=false;
14     }
15   }
16
17   static List cached=new LinkedList();
18   public static boolean isCached(Document d)
19     { return cached.contains(d); }
20
21   after(Document d):
22     call(* Document.setCount(..))
23       && target(d) {
24     Turbo.cached.add(d);
25     sort(Turbo.cached);
26     if(Turbo.cached.size()>10)
27       Turbo.cached.removeLast();
28   }
29 }
```

Figure 4: Turbo aspect

```
1  before(Document doc): call(* Document.*(..))
2          && target(doc)
3          && !call(* Document.*Count(..))
4          && if(!Turbo.activated) {
5    doc=doc.getRootVersion();
6    logF.logDocAction(doc,
7        thisJoinPoint.getSignature());
8  }
```

Figure 5: Modified logging advice

```
1  void around(): save() &&
2    if(Turbo.isCached(thisJoinPoint.getTarget()))
3    && if(Turbo.activated) {
4    int dsaves=++delayedsaves;
5    Document d=
6      (Document) thisJoinPoint.getTarget();
7    Iterator it=d.getAllVersions().iterator();
8    while(it.hasNext()) {
9      Document v=(Document) it.next();
10     if(DelaySave.hasAspect(v))
11       dsaves+=
12         DelaySave.aspectOf(v).delayedsaves;
13   }
14   if(dsaves>10) proceed();
15   return;
16 }
```

Figure 6: Modified buffer caching advice

the way that interactions between the aspects are handled.

First of all the (de)activation of the secondary logging advice is realized by modifying its pointcut in order to disable its execution when the turbo is not activated (line 4 in Fig. 5). This is the only way to realize the strategy. Not even runtime (un)weaving capability (in AO languages that support it [26, 33]) can express this composition cleanly because it is not the complete aspect that needs to be (un)woven, but only a subpart (one advice). Splitting the aspect such that the advices can be (un)woven individually would introduce other difficulties since that advice needs the state of the aspect to log information and that state is also used by the other advice.

Secondly the same modification has to be performed on the buffering cache aspect to enable it when needed (line 3 in Fig. 6).

Thirdly the logging pointcut has to be modified again in order to not match calls to setCount() or getCount() introduced by the action counter aspect because such calls do not represent actual user actions on documents (line 3 in Fig. 5).

Finally, the presence of the aspect Turbo has another impact on the buffering cache aspect. Indeed, Turbo implies that only saves on most used documents are delayed. The pointcut of the advice delaying saves is therefore adapted to fit this requirement (line 2 in Fig. 6).

2.1.2 *Adding a versioning feature*

The company decides to allow users to concurrently edit the same document. Inspired by the aspect-oriented transaction framework AspectOptima [21] an aspect is implemented to make users work on different copies of a document (see Fig. 7). When a user loads a document (line 10) a copy of the original document (called root version) is created (lines 11–12) and actually loaded by calling proceed (line 13). To keep the example simple, the source code to

```
1  public aspect Versioning {
2
3    Document Document.rootVersion;
4
5    public Document.getRootVersion() {...}
6
7    ...
8
9    boolean around(Document doc):
10     call(boolean User.load(..)) && args(doc) {
11     Document newDoc=newVersion(doc);
12     newDoc.rootVersion=doc;
13     return proceed(newDoc);
14   }
15
16 }
```

Figure 7: A versioning aspect

commit several versions of a single document is intentionally ignored.

Impact on aspects.

The versioning feature has a big impact on the meaning of a Document instance. Indeed, a document is no more represented at runtime as one instance of the class Document but as several. As a consequence, other aspects have to be invasively modified to handle this semantic alteration. For instance, Logging now has to work on the root version of the document (line 5 in Fig. 5) because it makes no sense to have different log entries for several Document instances when they actually represent a single entity. The aspect CountActions has to be modified in a similar way as well.

Versioning also interacts with the aspect DelaySave. Indeed, this aspect is instantiated and bound to every Document instance on which a save action is performed. Because Versioning implies that several instances of Document represent a single document, we can have several instances of DelaySave when only one would actually be necessary. As a consequence, the counter of already delayed saves of each aspect instance no longer represents the number of saves performed on a document. To decide whether the save action has to be delayed or not the aspect DelaySave is modified to compute the sum of each counter of each aspect instance bound to a different version of a single document (lines 5–13 in Fig. 6).

2.2 Issues

Section 2.1 gives a concrete example of how aspects sometimes need to be modified to avoid semantic interferences with other aspects when used together. This section delves a bit deeper and discusses what modifications can be necessary to remove unwanted interactions between aspects. It looks at modifications of pointcuts, advices, and instantiation policies.

2.2.1 Pointcut Modification

The example showed that it is sometimes necessary to change pointcuts in order to compose aspects successfully. In general we identified three scenarios which may require this kind of invasive change.

1. **Accidental Join Point Matching/Miss.** When an aspect is added to the set of aspects of an application, it may happen that this aspect adds, removes or modifies join points [22]. As a consequence, pointcuts of other aspects may not match the same set of join points as they were matching before the aspect was added. To manage such interactions and avoid possible semantic interferences, some pointcuts may have to be adapted to handle the change (see for example line 3 in Fig. 5).

2. **Mutual Exclusiveness.** Pointcuts may have to be changed even when no join points are added, removed or affected. Indeed, several aspects may for instance implement a similar feature but with different strategies (e.g. different transaction strategies applied on different objects [21]). As a result, it is often required by the application constraints that these aspects should not coexist at the same join points. Pointcuts then have to be modified to ensure they do not intersect.

3. **Runtime Advice (De)Activation.** The aspect composition may sometimes require an advice or an aspect to be (de)activated under runtime conditions. This can occur for instance to ensure mutual exclusiveness in control-flow segments (e.g. features competing for resources like our logging advice which must be deactivated to speed up the execution) or to provide a composition feature (e.g. our DelaySave which is only activated for most used documents). In AO languages that support the if(..) pointcut predicate, runtime advice (de)activation can be realized by adding a runtime condition to the pointcut of the advice (see line 4 in Fig. 5).

```
1 Object around(): pointcut() {
2   if(condition) {
3     /* original advice code */
4   }
5   else
6     return proceed();
7 }
```

Figure 8: if(..) pointcut simulation.

2.2.2 Advice Modification

Modifying advice is a second way to manage aspect interactions, and we list a number of possibilities.

1. **Runtime Advice (De)Activation.** In AO languages which do not support the if(..) pointcut predicate, runtime (de)activation of advice is realized by modifying the advice code in order to simulate that predicate. The code given in Fig. 8 illustrates the simulation for an advice of kind *around*.

2. **Advice Execution Context.** Advice and methods can be considered as similar in the sense that they both accept arguments. In the case of advice, arguments are contextual values from the join points on which advice are being applied. Certain aspect compositions may require modifications of the advice to adapt their execution context.

   We illustrate this by revisiting the Versioning aspect from Sec. 2.1. This aspect alters the semantics of the environment: several instances of the class Document may represent a single actual document. This change in the environment interacts with the Logging and the CountActions aspects since they manipulate

documents. In such situations, the interaction cannot be handled by ensuring mutual exclusiveness or runtime (de)activation because these aspects are somehow complementary and need to coexist. To manage the interactions the source code of the involved advice is modified to make them work on the correct contextual input (line 5 in Fig. 5).

2.2.3 Instantiation Policy Modification

An aspect instance represents the state of an aspect. It defines the values of the aspect variables which are used in the aspect's advice. Aspect instances can have different scopes, determined by instantiation policies. Scopes are either global (a singleton instance for the whole system) or local (one instance per object, thread, control-flow segment, etc.). Different languages have different instantiation policies. AspectJ, for example, does not support the *per thread* policy.

When an aspect alters the environment it may have an impact on the instantiation policy of other composed aspects. For instance, our versioning aspect has an impact on the instantiation policy of the aspect DelaySave. Indeed, since a document may be represented as several instances of the class Document, the scope of an instance of DelaySave should not be per Document object anymore but per set of Document objects which represent a single document.

As a second example suppose that an aspect is implemented to monitor a certain sequence of events and check if the sequence is correct with respect to some rules [3]. The state of that aspect contains the history of already met events. This monitoring aspect is then composed with an aspect intercepting certain loop join points to parallelize their iterations in different threads [15]. In order to ensure the correct behavior of the application, the monitoring aspect has now to be instantiated per thread. Otherwise, it could identify erroneous patterns of events because of the parallelization.

Note that this interaction management is invasive because the instantiation policy of an aspect is specified in that aspect.

2.3 Conclusion

When an aspect is developed independently it makes assumptions about its environment. These assumptions can be made on various things such as the available resources, where, when and how it will be applied on the base system or even the environment semantics. Moreover aspects can impact their environment when used. Composing aspects therefore means that assumptions made by one aspect may be broken by another one. Such interactions can lead to semantic interferences if they are not correctly treated.

In our examples, source code of aspects need to be invasively modified in order to manage the interactions implied by the aspect composition. As a consequence, the extra code needed to handle the aspect composition is tangled and scattered over several aspects, making the code hard to understand and less reusable.

3. ASPECTS TO COMPOSE ASPECTS

Aspect-oriented programming is a programming paradigm to modularize concerns which are tangled and scattered across an application. Since composing aspects may imply the specification of additional behavior which crosscuts aspects, it seems quite reasonable and relevant to expect aspect-oriented languages to have mechanisms to modularize it.

However, current aspect-oriented languages do not support the specification of aspects over aspects very well. This section discusses this absence of support for aspect-orientation and then proposes some requirements an AO language should meet to support the composition of aspects with aspects.

3.1 Base Language-Related Join Point Model

To separate crosscutting concerns from the base system, an aspect-oriented language relies on a *join point model* (JPM). A JPM specifies three elements: (1) the points in the program exposed to the aspects (the *join points*), (2) a *means of identifying join points* and (3) a *means of specifying semantics at join points* [24]. The last two elements are generally referred to as *pointcut* and *advice* respectively.

For most aspect-oriented languages, the base system (or root concern) is a program specified in an object-oriented language [6]. JPMs of these languages are therefore mostly focussed on object-orientation. For example, the typical dynamic join points for object-oriented languages are events such as method or constructor call, class or object initialization, or exception handling.

Separating the base program from the aspects by using base-related join points has proven to be a good way to modularize concerns that crosscut the base code. But the aspect composition is a concern that does not crosscut the base system but the aspects themselves. Therefore, base-related join points cannot be used to specify such aspects over aspects since aspects use concepts and elements that do not appear in the base application (e.g. pointcuts or advice). For instance, it is not possible for an aspect to modify pointcuts of another aspect using base-related pointcuts, advice or inter-type declarations.

Note that some AO languages have features for reasoning about aspects. But these features are quite limited since their purpose is rather to gain expressivity while specifying base-related aspects than to specify aspects over aspects. AspectJ, for example, has a pointcut to intercept the execution of advice (the adviceexecution() predicate) that is often used to avoid advising loops. We will explain more precisely why this predicate is limited to compose aspects in Section 4.

3.2 Aspect-Related JPM Requirements

In this section we enumerate the minimal features we feel a join point model should support to empower an aspect to express aspect composition. We discuss the ability to modify foreign aspects, to specify advice precedence relationships and to express aspect dependency relationships.

3.2.1 Foreign Aspect Modification

We saw previously in this paper that composing aspects may require invasive modification of the composed aspects.

In particular, Sec. 2.2.1 and 2.2.2 highlighted that some composition cases require pointcuts and advice to be adapted. To encapsulate such modifications in an aspect, the join point model of the language should offer a means to modify

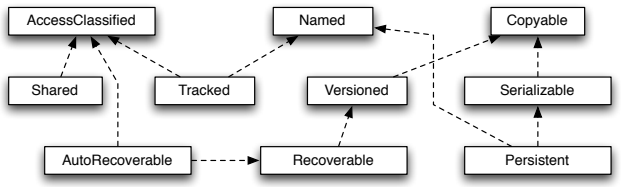

Figure 9: Complex aspect dependencies in AspectOptima [21]

pointcuts and advice of foreign aspects. Moreover, because an aspect composition may need to adapt a specific subpart of an aspect, the join point model should provide a means to precisely identify pointcuts and advice. For instance, when composing the counting and the logging aspects there was a need to adapt only one pointcut of the logging aspect shown in Fig. 1 (more precisely, the pointcut at lines 11-12).

The join point model should also allow for access and adaptation of the contextual input of an advice. Indeed, we saw previously that invasive interaction management may depend on the advice's input (line 2 in Fig. 6) and may even require its modification (line 5 in Fig. 5).

The instantiation policy of an aspect also needs to be changeable to solve particular aspect interactions, like we showed in Sec. 2.2.3. We stress that a finite set of instantiation policies would not handle all usage scenarios and that *user-defined* policies are needed. For example, the interaction between the DelaySave and Versioning aspects of Section 2 can then be solved elegantly by changing the instantiation policy for DelaySave from *per instance of class Document* to *per set of instances of class Document that represent a single document*. Note that for example AspectC++ [30] supports user-defined instantiation policies.

3.2.2 Precedence Management

It may happen that the order in which the weaver applies aspects to the base program determines the behavior of the resulting application. This phenomenon occurs with interacting aspects and is even used in [1, 16] to detect interactions.

Consider for example two different aspects, *caching* and *authorization*. Weaving the caching aspect before the authorization aspect gives rise to semantic interferences. Indeed, when the result of a method has already been cached, that cached value is directly returned before authorization is performed, potentially resulting in security breaches. This is not the case when the authorization aspect is woven first.

To manage such interactions, the programmer has to specify which ordering produces the desired application semantics. [23, 7] give examples showing that a weaving ordering at the aspect level cannot resolve all precedence-related interactions correctly. Instead the order has to be specified at the more fine-grained advice level.

Composing aspects with aspects therefore requires the aspects to have the ability to specify advice precedence relationships.

3.2.3 Dependency Management

An aspect may assume that another aspect is woven in order to behave correctly. This aspect interaction is known as *dependency* [29]. If an aspect on which another aspect is dependent is not woven in the base system, the resulting application does not behave correctly.

The most used example to illustrate aspect dependency in the literature is probably the couple *Authentication - Authorization*. *Authentication* implements the identification of a user while *Authorization* manage access rights with respect to the identified user. These aspects depend on each other to ensure access policy.

Dependency relations can however be much more complex. Figure 9 shows the inter-aspect dependency graph from AspectOptima [21]. It shows that aspects may depend on several aspects and that an aspect may be required by several different aspects.

As a consequence, the management of dependency interactions becomes tricky when runtime weaving/unweaving are possible (like for example in Prose [26] or JAsCo [33]). Indeed, on the one hand the weaving of an aspect implies to weave all its dependencies if they are not woven yet and on the other hand, the unweaving of an aspect implies to unweave all its dependencies if they are not required by another aspect which is still woven.

We are not aware of any aspect-oriented language that supports to explicitly declare aspect dependency relations. Instead, when needed, the relationships exist implicitly in the aspects that are involved.

4. OARTA

This section describes Oarta, an AOP language we made that implements all features we introduced in Sec. 3. We decided to make Oarta an extension of AspectJ because we could take advantage of the extensible *AspectBench Compiler* (abc) [4]. We believe however that our model can be applied to other aspect-oriented languages without much conceptual problems, although a complete realization like we show here for AspectJ might be technically challenging.

The rest of this section is structured as follows: Section 4.1 discusses features for managing aspect interactions that exist in AspectJ. Section 4.2 explains the Oarta extensions, followed by Section 4.3 that shows how composition can be implemented as an aspect in Oarta. Then, Section 4.4 gives some implementation details.

4.1 AspectJ's JPM

In this section we discuss the basic AspectJ features that can be used to implement aspect compositions in dedicated aspects.

4.1.1 The adviceexecution Pointcut Predicate

It is possible in AspectJ to intercept executions of advice using the predicate adviceexecution(). However, this predicate does not allow to identify executions of one particular advice because it identifies *any* advice execution.

When used in conjunction with the predicate within(..) we can restrain the selection to a certain aspect but we cannot distinguish two advice of that aspect. When used in conjunction with the predicate args(..) the selection can be restrained to advice with signatures with particular specified argument types. However, using these additional predicates does not guarantee that we can identify one particular advice if several advice of an aspect have the same signature.

Note that an advice has arguments if its pointcut uses a predicate (target(..), args(..) or this(..)) to bind values

```
1 public aspect Example {
2   Object around(Object obj):
3     adviceexecution() && args(obj)
4     && !within(Example) {
5       print(obj+" is used in an advice");
6       return proceed(new Object());
7   }
8 }
```

Figure 10: Retrieving and modifying the contextual input of an advice execution.

```
1  public interface Container { }
2
3  public aspect Loader {
4    public Object Container.aField;
5    public void Container.aMethod() { ... }
6    before(): call(* Container+.*(..)) { ... }
7  }
8
9  public aspect Connector {
10   declare parents: aClass implements Container;
11 }
```

Figure 11: Indirection strategies.

from the join point to variables. Interestingly, we can therefore use adviceexecution in conjunction with args to retrieve contextual values from the intercepted advice execution (see Fig. 10).

Using advice of kind around, contextual values can also be modified by giving other values as arguments when calling proceed (line 4 in Fig. 10).

4.1.2 *The* if *Pointcut Predicate*

AspectJ supports the if(..) pointcut predicate. In the absence of true runtime weaving, this predicate can be used to activate or deactivate advice and aspects at runtime.

4.1.3 *Indirection Strategies*

Indirection strategies (or patterns) are described in [14]. These strategies are intended to separate the specification of an aspect's effects from the specification of the places where the effects should be applied.

Several participants are involved in order to realize the strategies: an empty interface (the *container*), an aspect which specifies the desired effects by referencing the container (the *loader*) and an aspect which connects actual classes to the container (the *connector*). This is illustrated in Fig. 11. As a result, the loader aspect does not have to be invasively configured with application-specific information and is more reusable.

When expressing aspect composition as an aspect these strategies are really helpful. They allow the modification of foreign pointcuts thanks to the indirection. However, these strategies can only modify the parts of a pointcut related to types. They cannot handle behavior-specific modifications. For instance, they do not allow to add an if(..) pointcut predicate.

4.2 Oarta's Extensions

We now enumerate the extensions Oarta makes to AspectJ in order to support our conceptual join point model.

4.2.1 *Named Advice*

Advice now have to be named. The following examples illustrate the syntax changes:

- void around **myName**(int i): call(* *.*(..)) { ... }
- before **anotherName**(): set(* *.*) { ... }

This language extension allows us to precisely identify both advice and pointcuts, because an advice has exactly one pointcut in AspectJ.

4.2.2 *Advice Patterns*

In order to quantify over advice, our extension makes it possible to use *advice patterns*. An advice pattern is similar to a method pattern except that it requires additional information about the kind of the advice (and has no modifier patterns nor exception throwing). The advice kind information allowed in an advice pattern is either before, after, around, afterthrowing, afterreturning or the wildcard * (which identifies any of them).

Examples:

- String around *.foo(int)
  identifies all around advice named foo taking one argument of type int and returning a String.
- * * Aspect.*(..)
  identifies all advice of the aspect Aspect.

Note that only around advice have a return type. All other advice kinds return void.

4.2.3 *Foreign Pointcut Modification*

In AspectJ, pointcuts can use conjunctions, disjunctions and negations of pointcut predicates. As a result, pointcuts can be extended by adding another pointcut using a disjunction, or specialized using a conjunction.

The modification of foreign pointcuts is realized by using the orpointcut and andpointcut constructs:

- orpointcut: an_advice_pattern : a_pointcut;
- andpointcut: an_advice_pattern: a_pointcut;

orpointcut affects the pointcuts of advice matched by the given advice pattern by replacing their pointcut with the disjunction of the original pointcut and the pointcut given as argument. andpointcut has the same effect but using conjunction.

When used together with the *indirection strategies* described in Sec. 4.1.3 this language extension allows us to support the modification of foreign pointcuts.

Note that this language feature is similar to the *global pointcut* extension proposed in [4], that differs from our proposal because it only supports conjunctions and affects all pointcuts of an aspect.

4.2.4 *Modified* declare precedence *construct*

The original declare precedence construct specifies weaving orderings at the aspect level. Our extension allows it to be used at the more fine-grained advice level using advice patterns. The modified construct is given here:

- declare precedence: advpattern1, advpattern2;

4.2.5 *Modified* adviceexecution *predicate*

We saw in Sec. 4.1.1 that the pointcut predicate adviceexecution() cannot always identify executions of a specific advice. We therefore extend it to take an advice pattern as argument: adviceexecution(an_advice_pattern) so that it can match executions of advice matched by the advice pattern.

Moreover, in order to be able to expose the entire join point context to aspects (and not only the arguments of the advice (see Sec. 4.1.1)), the join point matched by the advice can be retrieved by using the pointcut predicate target(..). Here is an example to illustrate this:

```
1 before catchAdvEx(JoinPoint jp):
2   adviceexecution(* around *.*(..))
3   && target(jp) {
4       print(jp+": matched by an around adv.");
5 }
```

This advice intercepts every execution of an around advice and prints the join point that has been matched by that advice. The target(..) predicate is used to bind the join point runtime representation to the variable jp. Conceptually, we believe it makes sense to consider a join point as the *target* of an advice's execution.

4.2.6 *User-Defined Instantiation Policies*

In AspectJ, an aspect instance is implicitly retrieved at runtime by calling the static method aspectOf(). An aspect's advice is then 'called' on the retrieved instance in order to perform the advice's behavior. This method is automatically generated for each aspect during the compilation. Another static method is generated as well: hasAspect(). That method is used at runtime to check if an aspect instance exists.

These methods take no arguments for aspects with an instantiation policy that is either *singleton* or *per control-flow segment*. They take one object argument if the policy is *per object*.

AspectJ compilers do not allow these methods to be redefined (contrary to for example AspectC++). Moreover, they cannot be intercepted by pointcuts. Oarta changes this to support user-defined instantiation policies: a composition aspect can specify instantiation policies for other aspects by intercepting these methods and 'overriding' them using advice of kind around. We illustrate this in the next section.

4.3 Examples

In this section we give three examples to show how our extended join point model makes it possible to separate the interaction management from the aspects. We first revisit the motivating example of this paper, then we give an example involving dependency management to conclude with an advice-level ordering example.

4.3.1 *Revisiting the motivating example*

The example from Sec. 2 illustrated several practical aspect interaction problems that can occur. This section shows how to handle them in Oarta featuring our JPM. We assume for simplicity that the source code given earlier respects our extended language's syntax (in other words, we assume that advice are named).

Turbo. The Turbo aspect impacts the pointcuts of the Logging and DelaySave aspects (see Sec. 2). Instead of modifying these pointcuts by destructively changing these two

```
1  public aspect Turbo {
2    ...
3    /* code from Fig. 4 */
4    ...
5
6    andpointcut: (* * Logging.*(Document)):
7      if(!Turbo.activated);
8    andpointcut: (* * DelaySave.*()):
9      if(Turbo.isCached(thisJoinPoint.getTarget()))
10     && if(Turbo.activated);
11
12 }
```

Figure 12: Extended Turbo aspect.

```
1 public aspect CAInteractions {
2   andpointcut: (* * Logging.*(Document)):
3     !call(* Document.*Count(..));
4 }
```

Figure 13: Aspect that manages the CountActions interactions.

aspects, we use the andpointcut construct to alter them from the outside (see Fig. 12). The first declaration modifies the pointcut of the logging advice considered as a secondary feature (lines 6–7) while the second one modifies the pointcut in DelaySave (lines 8–10).

CountActions. The aspect CountActions introduces new call join points which must be avoided in Logging. Again, we can use andpointcut to manage the interaction in a dedicated CAInteractions aspect (see Fig. 13).

Versioning. The Versioning aspect had very complex interactions which required us to invasively modify advice to support it in Sec. 2. We are now able to modularize all of these interactions in a dedicated VInteractions aspect (see Fig. 14). The first advice of that aspect handles the join point context modification required in Logging and CountActions (lines 2–7). To do so, the advice has to intercept executions of these aspects' interacting advice (lines 3–4). It also binds the Document instance of the advice retrieved from the join point context to the variable doc (line 5). Finally, it calls proceed with the root version of the document as argument in order to make the intercepted advice work on the correct instance (line 6).

The rest of the VInteractions aspect manages the interaction with DelaySave. The origin of this interaction is the instantiation policy of the aspect DelaySave (see Sec. 2). To manage it, VInteractions replaces the instantiation policy with a user-defined instantiation policy. This is achieved by intercepting executions of the static hasAspect() and aspectOf() methods of DelaySave and replacing their original behavior (lines 11–29).

The first advice handles calls to hasAspect() quite simply: if the argument is an instance of class Document, the advice answers that an aspect instance exists. Otherwise, it answers the opposite.

The other advice manages the calls to aspectOf(). If the argument is an instance of class Document, it retrieves the root version of that instance (line 23). It then checks if an instance of the aspect has already been created for that root version (lines 24–25). If no instance has been associated, an instance is generated and stored in the object (in a field created by the aspect (line 9)) before being returned. Note

```
1  public aspect VInteractions {
2    void around logNcount(Document doc):
3      (adviceexecution(* * Logging.*(Document)) ||
4      adviceexecution(* * CountActions.*(Document)))
5      && args(doc) {
6        proceed(doc.getRootVersion());
7    }
8
9    public DelaySave Document.d_aspect;
10
11   boolean around hasAspect(Object o):
12     execution(static * DelaySave.hasAspect(..))
13     && args(o) {
14       if(o instanceof Document) return true;
15       else return false;
16   }
17
18   DelaySave around aspectOf(Object o):
19     execution(static * DelaySave.aspectOf(..))
20     && args(o) {
21       if(o instanceof Document) {
22         Document doc=(Document) o;
23         doc=doc.getRootVersion();
24         if(doc.d_aspect==null)
25           { doc.d_aspect=new DelaySave(); }
26         return doc.d_aspect;
27       }
28       else throw new RuntimeException();
29   }
30 }
```

Figure 14: Aspect that manages the Versioning interactions.

that AspectJ forbids explicit instantiation of aspects but Oarta allows it.

All interactions of our motivating example have now been managed and no interacting aspect had to be modified. Everything has been realized in dedicated aspects.

4.3.2 Dependency Management

In the previous example, we created an aspect VInteractions responsible for the management of interactions provoked by the aspect Versioning. As a consequence, Versioning depends on VInteractions. In this example, we show how dependencies can be managed in dedicated aspects.

Dependencies have to be managed when runtime weaving is possible (for compile-time weaving it suffices to ensure their presence at the compilation). Figure 15 shows how to add an aspect that enables runtime weaving in Oarta. It implements static methods to weave and unweave aspects (lines 4–8). Unfortunately, we could not use the indirection strategies presented in Sec. 4.1.3 because they do not support introduction of static members. An alternative solution is to have a static set which contains aspects that are currently woven (line 2).

We use indirection to add a runtime condition in all pointcuts of aspects implementing RWeavable (line 13–14). As a result, advice of these aspects are only executed if their respective aspect is woven. Figure 16 shows the aspect that connects Versioning and VInteractions to the interface RWeavable.

Now that our aspects are weavable at runtime, we can manage their dependencies. A dependency has to be managed at two moments: at weaving-time and at unweaving-time. The aspect responsible for managing dependencies is given in Fig. 17. The first advice of the aspect handles weaving-time situations: when Versioning is woven (lines 4–6), the aspect weaves VInteractions (line 7). The handling at

```
1  public aspect RuntimeWeaving {
2    static Set woven=new HashSet();
3
4    public static void weave(Class aspect)
5      { woven.add(aspect); }
6
7    public static void unweave(Class aspect)
8      { woven.remove(aspect); }
9
10   public static boolean isWoven(Class aspect)
11     { return woven.contains(aspect); }
12
13   andpointcut: (* * RWeavable+.*(..)):
14     if(RuntimeWeaving.isWoven(this.getClass()));
15 }
```

Figure 15: Aspect enabling aspects to be woven at runtime.

```
1  public aspect Connector {
2    declare parents: Versioning
3      implements RWeavable;
4    declare parents: VInteractions
5      implements RWeavable;
6  }
```

Figure 16: Aspect making two aspects weavable at runtime.

unweaving-time is simply the opposite.

4.3.3 Advice-Level Ordering

Assume an application that handles lots of files, that sometimes need to be encrypted or compressed to ensure privacy or to save space. Depending on the use case one encryption/-compression technique may be preferred over another (e.g., a symmetric encryption to save files on disk and asymmetric to transfer files over a network). Each encryption/compression technique is implemented as an aspect. Figure 18 illustrates the *zip* compression and symmetric encryption techniques. We assume for convenience that other encryption/compression aspects adopt the same advice nomenclature.

The problem is that, when these advice are applied on the same join points, we might try to decrypt a compressed file (or decompress an encrypted one), which will provoke an error. Therefore, precedence relationships have to be specified:

- declare precedence: * * *.encrypt(..), * * *.compress(..);
- declare precedence: * * *.decompress(..), * * *.decrypt(..);

```
1  public aspect VersioningDependencyManagement {
2
3    before weave(Class c):
4      call(static void RuntimeWeaving.weave(..))
5      && args(c)
6      && if(c==Versioning.class) {
7        RuntimeWeaving.weave(VInteractions.class);
8    }
9
10   after unweave(Class c):
11     ...
12
13 }
```

Figure 17: Aspect managing the dependency of the Versioning aspect.

```
1  public aspect ZipCompression {
2    before compress(File f):
3      call(* FileSystem.save(File)) && args(f) {
4      zip(f);
5    }
6    after decompress(File f):
7      call(* FileSystem.open(File)) && args(f) {
8      unzip(f);
9    }
10 }
11
12 public aspect SymEncryption {
13   String key= ... ;
14
15   before encrypt(File f):
16     call(* FileSystem.save(File)) && args(f) {
17     symenc(f,key);
18   }
19   after decrypt(File f):
20     call(* FileSystem.open(File)) && args(f) {
21     symdec(f,key);
22   }
23 }
```

Figure 18: ZipCompression and SymEncryption

Note that it would have been impossible to solve this composition issue non-invasively with an ordering at the aspect level. Indeed, it would have required us to modify the aspects in order to get only one advice per aspect and therefore being able to order them correctly.

4.4 Implementation

ABC is an implementation of ASPECTJ with the intention of easing language extensions or optimisations. We chose it as platform for our experiments for exactly that reason.

ABC provides a frontend built on the POLYGLOT framework [27]. Among other things, this framework allows to specify a grammar as an incremental set of modifications to the existing JAVA grammar. Hence, we used it to add new elements to the ASPECTJ's syntax specified in ABC (i.e., advice patterns and the andpointcut/orpointcut constructs), and to modify existing elements (i.e., advice which now have a name, and declare precedence and adviceexecution which now takes advice patterns as arguments).

POLYGLOT is also structured as a list of passes that rewrite the abstract syntax tree. Inspired by the implementation of the *global pointcuts* (see Sec. 4.2.3), we have implemented the modification of foreign pointcuts as a pass that rewrites pointcut nodes of advice matched by the patterns specified in andpointcut and orpointcut constructs. Another pass has been implemented to collect precedence declarations and compute the concrete advice ordering used at weaving-time.

User-defined instantiation policies for foreign aspects is based on two principles: (1) allowing pointcuts to match executions of the aspectOf and hasAspect methods, and (2) allowing users to explicitly instantiate aspects using new. This more permissive behavior of the compiler has been achieved by simply toggling a flag in ABC for the former and by removing a test in the type checker for the latter.

5. DISCUSSION

The goal of this paper is to highlight the fact that aspect composition is itself a concern that crosscuts aspects and that using aspects to implement it is as relevant and useful as using aspects for concerns that crosscut the base system.

The features we describe can be used to solve many aspect composition problems, as indicated through the various examples. However we are aware that our solution is not yet complete and that some composition issues remain that we do not handle. For instance, we did not propose a *context-dependent* ordering feature, which may be required in certain cases [7], nor join point model elements that would handle the new andpointcut/orpointcut constructs from the outside (if composition aspects need to be composed as well). We are currently in the process of designing a complete model of aspectual composition.

6. RELATED WORK

Aspect interactions and aspect interference occur frequently in practice. Therefore there is already quite some research that investigates these problems.

Detection of interactions or interferences caused by aspects is tackled in many works [25, 22, 19, 18, 1, 16, 17, 13, 9, 8, 10, 11, 32, 31, 28, 5, 20]. However, in this paper we do not address the detection of interactions but instead focus on the step that follows detection: how to resolve conflicts in a non-intrusive manner. Note that the research of Douence *et al.* [9, 8], not only detects interactions but also introduces some explicit aspect composition operators. However these operators are too coarse-grained to handle the various complex scenarios highlighted in the paper (they basically consist of precedence and mutual exclusiveness relationships).

Related papers have proposed some new language mechanisms to deal with a specific composition concern [21, 4, 2], but none have clearly expressed the need for modularization of a large variety of composition concerns.

In [21], an aspect-oriented transaction framework is implemented consisting of ten reusable aspects. The authors report some limitations they encountered in separating the aspects and with inter-aspect configurability using ASPECTJ. To solve some of these limitations they propose amongst others to have (1) explicit constructs to declare dependencies and (2) methods to enable/disable pointcuts at runtime. These propositions clearly share ideas with our approach.

Ernst *et al.* [12] also proposes named advice for ASPECTJ in order to support advice polymorphism and to allow advice to be 'overridden'. We used named advice to precisely identify advice and pointcuts in order to adapt them from the outside.

Dinkelaker *et al.* [7] address the situation where developers are prevented from tailoring aspect-oriented semantics in an application-specific manner. The solution proposed is a meta-aspect protocol where parts of the semantics can be redefined. Their approach makes it possible to define interaction management as semantic redefinitions. As a consequence, aspects may not need to be changed to compose them. Note that when many interactions have to be managed this all needs to be done in a single redefined element of the protocol. In our approach we can manage single interactions in a single aspect, which is much easier to maintain and understand.

In JASCO [33], an *aspect bean* defines the behavior of an aspect while a *connector* can define pointcuts and runtime composition strategies. While this enables JASCO to have independency between the base system and the aspect beans, it does not solve invasive composition-specific modifications of the aspects: depending on the composed aspects, pointcuts might still have to be adapted as well as advice.

Moreover, similarly to the last mentioned work, all composition code or strategies required by the interacting aspects have to be located in a single connector handling these aspects.

7. CONCLUSION

This paper focusses on the problem of managing interactions between aspects that are being composed. It answers a practical need that developers often face when using several aspects together that crosscut each-other. Our work complements various existing research that detects aspect interactions or aspect interferences by handing constructs to manage the interactions without needing to change the aspects that are being composed. We propose an aspect join point model that focusses on the relations between aspects. It extends existing join point models that focus on the relations of aspects with the base program. We extended AspectJ to incorporate this extended join point model. The resulting language, Oarta, can manage interactions between aspects and as a result can treat aspect composition itself as an aspect.

Acknowledgments

The authors would like to thank the anonymous reviewers for their comments that helped to improve the presentation of the paper.

This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy.

8. REFERENCES

[1] M. Aksit, A. Rensink, and T. Staijen. A graph-transformation-based simulation approach for analysing aspect interference on shared join points. In *AOSD '09: Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 39–50. ACM, 2009.

[2] A. Assaf and J. Noyé. Dynamic AspectJ. In *DLS '08: Proceedings of the 2008 symposium on Dynamic languages*, pages 1–12. ACM, 2008.

[3] P. Avgustinov, E. Bodden, E. Hajiyev, L. Hendren, O. Lhoták, O. de Moor, N. Ongkingco, D. Sereni, G. Sittampalam, J. Tibble, and M. Verbaere. Aspects for trace monitoring. In *Formal Approaches to Testing Systems and Runtime Verification (FATES/RV)*. Springer, 2006.

[4] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, , and J. Tibble. abc: An extensible AspectJ compiler. In *Transactions on Aspect-Oriented Software Development*, 2005.

[5] L. Bergmans. Towards detection of semantic conflicts between crosscutting concerns. In *AAOS Workshop at ECOOP 2003*, 2003.

[6] J. Brichau, M. Mezini, J. Noyé, W. Havinga, L. Bergmans, V. Gasiunas, C. Bockisch, T. D'Hondt, and J. Fabry. An initial metamodel for aspect-oriented programming languages. *Deliverable D39, AOSD-Europe*, Feb. 2006.

[7] T. Dinkelaker, M. Mezini, and C. Bockisch. The art of the meta-aspect protocol. In *AOSD '09: Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 51–62. ACM, 2009.

[8] R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In *GPCE*, pages 173–188, 2002.

[9] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 141–150. ACM, 2004.

[10] P. E. A. Durr, L. M. J. Bergmans, and M. Aksit. Reasoning about behavioral conflicts between aspects. Technical Report TR-CTIT-07-15, Enschede, Feb 2007.

[11] P. E. A. Durr, T. Staijen, L. M. J. Bergmans, and M. Aksit. Reasoning about semantic conflicts between aspects. In *EIWAS 2005: 2nd European Interactive Workshop on Aspects in Software*, 2005.

[12] E. Ernst and D. H. Lorenz. Aspects and polymorphism in AspectJ. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 150–157. ACM, 2003.

[13] B. D. Fraine, P. D. Quiroga, and V. Jonckers. Management of aspect interactions using statically-verified control-flow relations. In *Proceedings of the 3rd International Workshop on Aspects, Dependencies and Interactions*, 2008.

[14] S. Hanenberg and P. Costanza. Connecting aspects in AspectJ: strategies vs. patterns. In *First AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, 2002.

[15] B. Harbulot and J. R. Gurd. A join point for loops in AspectJ. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 63–74. ACM, 2006.

[16] W. Havinga, I. Nagy, L. Bergmans, and M. Aksit. Detecting and resolving ambiguities caused by inter-dependent introductions. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 214–225. ACM, 2006.

[17] W. Havinga, I. Nagy, L. Bergmans, and M. Aksit. A graph-based approach to modeling and detecting composition conflicts related to introductions. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 85–95. ACM, 2007.

[18] E. Katz and S. Katz. Incremental analysis of interference among aspects. In *FOAL '08: Proceedings of the 7th workshop on Foundations of aspect-oriented languages*, pages 29–38. ACM, 2008.

[19] E. Katz and S. Katz. Modular verification of strongly invasive aspects. In *Languages: From Formal to Natural: Essays Dedicated to Nissim Francez on the Occasion of His 65th Birthday*, pages 128–147. Springer-Verlag, 2009.

[20] B. Kessler and E. Tanter. Analyzing interactions of structural aspects. In *Workshop on Aspects, Dependencies and Interactions*, 2006.

[21] J. Kienzle, E. Duala-Ekoko, and S. Gélineau. Aspectoptima: A case study on aspect dependencies

and interactions. In *Transactions on Aspect-Oriented Software Development V*, pages 187–234. Springer-Verlag, 2009.

[22] G. Kniesel. Detection and resolution of weaving interactions. *Transactions on Aspect-Oriented Software Development, special issue on Aspect Dependencies and Interactions*, pages 1–53, Apr 2009.

[23] A. Marot and R. Wuyts. Composability of aspects. In *SPLAT '08: Proceedings of the 2008 AOSD workshop on Software engineering properties of languages and aspect technologies*. ACM, 2008.

[24] K. Masuhara, G. Kiczales, and C. Dutchyn. Compilation semantics of aspect-oriented programs. In *FOAL 2002 Proceedings: Foundations of Aspect-Oriented Languages Workshop at AOSD 2002*, pages 17–26, 2002.

[25] F. Munoz, B. Baudry, and O. Barais. Improving maintenance in aop through an interaction specification framework. In *ICSM08, 24th International conference on Software Maintentance*. IEEE Computer Society Press, 2008.

[26] A. Nicoara and G. Alonso. Dynamic aop with prose. In *Proceedings of the International Workshop on Adaptive and Self-Managing Enterprise Applications (ASMEA 2005)*, 2005.

[27] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In *Compiler Construction, 12th International Conference, CC 2003*, volume 2622 of LNCS, pages 138–152. Springer, 2003.

[28] R. Pawlak, L. Duchien, and L. Seinturier. Compar: Ensuring safe around advice composition. In M. Steffen and G. Zavattaro, editors, *FMOODS*, volume 3535, pages 163–178. Springer, 2005.

[29] F. Sanen, E. Truyen, B. D. Win, W. Joosen, N. Loughran, G. Coulson, A. Rashid, A. Nedos, A. Jackson, and S. Clarke. Study on interaction issues. In *AOSD-Europe Deliverable D44, AOSD-Europe-KUL-7*, 2006.

[30] O. Spinczyk and D. Lohmann. The design and implementation of aspectc++. In *Know.-Based Syst.*, volume 20, pages 636–651. Elsevier Science Publishers B. V., 2007.

[31] M. Stoerzer, J. Krinke, and U. Passau. Interference analysis for AspectJ. In *In Workshop on Foundations of Aspect-Oriented Languages*, 2003.

[32] M. Storzer and F. Forster. Detecting precedence-related advice interference. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 317–322. IEEE Computer Society, 2006.

[33] D. Suvée, W. Vanderperren, and V. Jonckers. JAsCo: an aspect-oriented approach tailored for component based software development. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 21–29. ACM, 2003.