

Transparent and Dynamic Aspect Composition

Anis Charfi¹, Michel Riveill², Mireille Blay-Fornarino², Anne-Marie Pinna-Dery²
charfi@informatik.tu-darmstadt.de, {riveill,blay,pinna}@essi.fr

ABSTRACT

In this paper we present an aspect-oriented approach with an advanced aspect merging mechanism. This approach is geared towards modularizing crosscutting concerns in component based applications and it originated from our previous work on the *interaction model*, which focuses on component interactions as programmatic and runtime entities for dynamic component composition and adaptation. In the interaction model, aspects are specified as a set of interaction rules using the Interaction Specification Language ISL. The ISL language provides a commutative and associative aspect composition mechanism, which improves the comprehensibility, the analyzability, and the predictability of the system behavior at shared join points. Unlike in most AOP approaches, the programmer does not have to care about the composition of aspects, their ordering, and their interactions. Instead, the rule merging mechanism automatically merges the advices according to a finite set of composition rules and equivalence axioms and generates a semantically equivalent advice. Aspect inter-dependencies are passed to the merging mechanism as additional constraints.

1. INTRODUCTION

Aspect-orientation [1] is a paradigm for the modularization of crosscutting concerns in software. In recent years this research area made major advances. However the problems of aspect composition, aspect interaction and conflict resolution received only minor attention. These problems arise when several advices need to execute at the same join point. Such join point is called *shared join point* [2]. The composition of multiple aspects at a shared join point raises several issues like the order of aspect execution and the dependencies between aspects. Most AOP approaches provide means to specify precedence between aspects, which implies an execution order. Each time a new aspect is added to the application, the programmer has to figure out the effect of this new aspect on the system (base application and aspects) and manually resolve potential conflicts if detected by the programmer himself.

The main requirement to the *interaction model* is the ability to dynamically introduce aspects into a distributed component-based application. The target aspects can be used to compose components with the application on-the-fly. Note also that components can be heterogeneous (different programming language, platform, component model).

Another important requirement is composition support i.e., several parties can adapt the application by defining aspects according to different view points. A genius composition mechanism is required to keep the consistency and coherence of all adaptations (i.e., the different aspects). The availability of such a mechanism would improve the comprehensibility, the analyzability, and the predictability of the system behavior at shared join points. A closer look at current AOP languages revealed that none of them fully fulfills this requirement.

In the interaction model [3], aspects are expressed as a set of interaction rules. An interaction rule consists of a notifying part (the pointcut) and a reaction (the advice). Aspects are expressed in the Interaction Specification language ISL, which defines a set of operators with well-defined semantics e.g., for message receiving, sequential execution, parallel execution, etc.

The interaction model solves the problem of aspect composition in a novel way. Instead of having an unpredictable random aspect behavior at shared join points, the composition mechanism generates an advice, which is the result of merging all advices that match that join point. Whenever a shared join point is reached one single advice is executed, which is semantically equivalent to the composed advices. The merging mechanism is based on a finite set of merging rules and equivalence axioms.

Automatic aspect composition yields several benefits such as releasing the programmer from manual composition and allowing an easier understanding for system behavior at shared join points. This also improves the quality of the software (comprehensibility, ease of evolution, and predictability), enables better testing and verification and gives the programmer more control in an AOP system. At each shared join point, the programmer can see the resulting advice that will be executed.

This remainder of this paper is organized as follows: Section 2 presents the dynamic AOP approach provided by the interaction model. In Section 3, we outline aspect composition with the rule merging mechanism. In section 4, we report on related work. In Section 5 we conclude the paper.

2. AOP WITH THE INTERACTION MODEL

The interaction model original focus was the expression of component interactions as first-class entities in a distributed component-based setting. Interactions are specified in a dedicated language, the Interaction Specification Language (ISL), which allows the definition of interactions among heterogeneous components like .NET; Java Components, EJB, etc. Interactions

¹ Darmstadt University of Technology, Germany

² Laboratoire I3S (CNRS/UNSA), Sophia Antipolis, France

can be activated or deactivated at run time, thus adapting the component behavior. The basic idea of interactions is to rewrite a method body (the pointcut) using the reaction (advice). Unlike in traditional AOP, the pointcut also appears in the reaction. This is especially required by the composition mechanism: If we have several advices to compose at the same pointcut, we use the pointcut as fix point.

The *interaction service* is an implementation of the interaction model. It can be thought of as a dynamic aspect repository with a weaver that uses a commutative and associative aspect composition mechanism. Since the interaction model is for a component-oriented setting, we are only interested in join points that appear in the public interface of the component (we do not break the component/object encapsulation).

To illustrate aspect definition in the interaction model we go through an example showing the encapsulation of the authorization concern in an agenda application. Our base application consists of the classes *Agenda* and *UserProfile*. The class *Agenda* stores the meetings of a given user and provides methods to add/remove a meeting. The class *UserProfile* holds user information.

```
class agenda {
    public void addMeeting(String identifier);
    public void removeMeeting(String
                               identifier);
    public Owner agenda.getOwner();
}
```

Listing 1. Class Agenda

Assuming that the authorization concern was not anticipated in the first version of the application, we now want to dynamically add the authorization functionality to the running application. Caller authorization is required for the methods *addMeeting()* and *removeMeeting()* of the class *Agenda* and for some methods of the class *UserProfile*. The authorization concern cuts across the application classes *Agenda* and *UserProfile*. To modularize this concern we define the following interaction pattern:

```
interaction authorization(Agenda agenda,
                          SecurityService security)
{
    agenda.addMeeting(String identifier),
    agenda.removeMeeting(String identifier)
    ->
    if security.authorize(_call)
    then agenda._call
    else throw "unauthorized user"
    endif
}
```

Listing 2. Authorization aspect

We also have to define a similar pattern for the class *UserProfile*. The interaction pattern is the modularization unit for crosscuts in the interaction model i.e., this is the counter part to the *aspect* construct in AspectJ [8]. An interaction pattern is specified in the

Interaction Specification language ISL and contains at least one interaction rule. The left side of the interaction rule is the *notifying* part i.e., the *pointcut* and the right side is the *rule's reaction* i.e., the *advice*. An interaction rule corresponds to a pair of pointcut and advice. In the interaction model the advice code is specified in ISL, which is platform, component model and language independent. One could e.g., define interactions between a .NET component, an Enterprise Java Bean, and a Web Service.

The semantics of an interaction rule is to execute the rule's reaction instead of the rule notifying part. This also means that the interaction model only provides the *around* advice. Nevertheless, the before and after advices can be easily expressed using the around advice. The counter part to *proceed* in AspectJ is the ISL operator *_call*, which represents the notifying message or notifying field set/get operation. There is however a finer difference to other AO languages: In order to have the before and after advice we write the following rules whereby *_call* refers to the join point at hand:

before: *pointcut* -> *advice*; *_call*

after: *pointcut* -> *_call*; *advice*

An interesting feature of the interaction model as an aspect-oriented approach is that interaction patterns take component type parameters, which represent the context of the advice execution. This eliminates the need for passing the context from the pointcut to the advice (keywords *this* and *args* in AspectJ). Several other papers [11][12] have also recognized this benefit.

The join point model of the interaction model encompasses primarily message receiving in a component setting (corresponds to the method execution pointcut in AspectJ). It also supports setting/reading of public fields. This choice may seem limited but it is reasonable for the component context as it does not break the encapsulation. The pointcut language of the interaction model allows the use of several wild card operators. For example, if we use the expression *agenda.** as a notifying expression in the authorization interaction pattern, this will capture all executions of any business method of the *Agenda* component. Business methods are those exposed in the component interface.

Interactions in the interaction model apply to some instance of a class and not all instances of that class. That is our approach is instance-based unlike AspectJ which is rather class-based. A sample implementation of the interaction model as the one found in [6] allows the programmer to write the interaction patterns and register them with an interaction server. Later, interaction patterns can be instantiated on specific component instances. When an interaction pattern is instantiated, specific instances of the types in the interaction pattern parameters should be selected. After that, interaction objects representing the interaction rules are created and passed them to the respective component instances. Then, the behavior of those instances is adapted correspondingly.

3. ASPECT COMPOSITION IN THE INTERACTION MODEL

First we give a short overview of ISL. More details on this language and its semantics can be found in [4]. Then, we illustrate the aspect composition mechanism through examples.

3.1 Interaction Specification Language

ISL is used to specify interaction patterns independently of any programming language and component model. It introduces several operators such as the conditional operator (if ... then ... else ... endif), the sequential operator (;), the concurrency operator (//), the waiting operator, and the exception operators throw and try-catch. These operators can also be used to compose aspects. The keyword *this* within an interaction pattern refers to the interaction object itself. In the following, we shortly discuss some of the ISL operators:

- The method invocation operator "." denotes a method call on the receiving component. Using the keyword `_call` in place of a method call refers to the notifying method call. Another form of a "method call" is the access of a public attribute (setter or getter)
- The assignment operator "!=" assigns the return value of a method call or of an assignment to a variable.
- The sequence operator ";" states that two behaviors should be executed one after the other.
- The concurrency operator "/" states that two behaviors should be executed in any order.
- The waiting operator ("_X" where X is a label) states that the execution of a message, variable assignment or another waiting behavior is blocked, until the end of the execution of a behavior labeled by "[X]".
- The conditional operator (*if then else endif*) states a conditional execution of a behavior depending on the boolean result of the execution of another behavior.
- The exception handling operators (*try ... catch and throw*). The thrown exception is then returned to the user (if it is not caught in the interaction rule).
- The delegation operator states that an action that does not contain the triggering message of the current rule should be considered as the triggering message. There is no keyword to denote this operator. It is implicitly added during the phase of semantic analysis.
- The ISL keyword `_call` represents the notifying message call. It also represents the reified notifying message when used alone as a method parameter.

There are shortcuts in ISL such as the wild card operator "*" that matches all messages on the receiving component. The comma operator "," is a shortcut that is used to select many methods on the left side of the interaction rule.

3.2 Interaction Composition

In the previous section we presented the interaction pattern that implements the authorization concern. In this section, we will define two other interactions patterns to capture the concerns *notification* and *persistence*. The notification interaction pattern shown in listing 3 consists of a single interaction rule capturing the reception of the message `addMeeting()`. The advice states that the original call should proceed and in parallel to that the method `notify()` of the class `Display` must be invoked with the same parameters as the notifying call. In this way, when the notification aspect is active, the display component outputs a notification

message each time a new meeting is added to the agenda. The pointcut of this interaction rule could be easily extended to notify the display whenever a meeting is removed from the agenda.

```
interaction notification(Agenda agenda,
                        Display display)
{
    agenda.addMeeting(String identifier)
    ->
    agenda._call // display.notify(_call)
}
```

Listing 3. The notification aspect

The persistence concern is captured by the interaction pattern shown in listing 4. The pointcut captures the reception of the messages `addMeeting()` and `removeMeeting()` by the Agenda component. The advice code proceeds with the original call and then stores the changed Agenda instance into the database.

```
interaction persistence(Agenda agenda,
                       DBManager dbMgr)
{
    agenda.addMeeting(String identifier),
    agenda.removeMeeting(String identifier)
    ->
    agenda._call ;
    db.store(agenda.getOwner(), agenda)
}
```

Listing 4. The persistence aspect

The three aspects shown so far have a shared join point namely the reception of the message `addMeeting()` by the Agenda component. When all three aspects are activated on certain instances of Agenda it is not clear what the resulting code would be like and what advice will be executed in what order.

The ISL language provides a mechanism for merging interaction rules. It is a *commutative* and *associative* aspect composition mechanism that merges the reactions of interaction rules which have common notifying parts (shared join points).

In our example, the rule merging mechanism automatically generates the interaction rule shown in Listing 5 for the notifying message `addMeeting()`. The generated interaction rule is semantically equivalent to the three interaction rules defined in the interaction patterns notification, persistence and authorization.

By looking at that interaction rule, the programmer can easily understand and predict the behavior at the shared join point because he sees one aspect with the resulting behaviour instead of three aspects.

For the message reception join point of `removeMeeting()` there are two interaction rules that need to be merged: the notification rule and the persistence rule. The resulting interaction rule is shown in Listing 6.

```

interaction notification&persistence&
authorization (Agenda agenda, Display display
, DBManager dbMgr, SecurityService security)
{
    agenda.addMeeting(String identifier)
    ->
    if security.authorize(_call)
    then
        display.notify(_call) //
        (agenda._call ;
        db.store(agenda.getOwner(), agenda))
    else throw "unauthorized user"
    endif
}

```

Listing 5. Rule merging for addMeeting()³

```

interaction persistence&authorization
(Agenda agenda, DBManager dbMgr,
SecurityService security)
{
    agenda.removeMeeting(String identifier),
    ->
    if security.authorize(_call)
    then
        agenda._call ;
        db.store(agenda.getOwner(), agenda)
    else throw "unauthorized user"
    endif
}

```

Listing 6. Rule merging for removeMeeting()

When the ISL *sequence* and *concurrency* operators are merged no chronological order is introduced if it was not already specified in the original input rules. This means that there would be several merging results that are semantically equivalent. For illustration we merge the rules *ra* and *rb*:

ra: *pointcut*-> *pointcut* ; *adviceA*

rb: *pointcut*-> *pointcut* ; *adviceB*

Since we do not know any thing about the order of execution of *adviceA* and *adviceB* the merging of *ra* and *rb* should not induce any new order. This means that the merging mechanism should not use the sequence operator in the generated rule. In this case the correct merging result *rab* uses the parallel operator:

rab: *pointcut* -> *pointcut* ; (*adviceA* // *adviceB*)

³ See appendix for rules description. The composition of "ithenelse" with a term different from exception is first applied. Then for each branch, the composition is operated recursively: for example sequence and message call, sequence and concurrency... We can notice that exception throwing is absorbent for the merging mechanism.

Thus no new chronological or priority order is introduced. According to [9], conflicts must then be detected as actions that are not ordered. Additional information is then needed such that the actions commute or that a specific execution order must be respected. We are working on insuring the commutativity and associativity taking into account dependencies between advices by considering them as functions [15]. According to [5] there are three forms of conflicts between aspects: **conditional execution** i.e., the execution of an aspect is necessary to the execution of another, **mutual exclusion** i.e., the execution of an aspect excludes another, **ordering** of aspect executions. .

Such inter-aspect dependencies can be passed as additional constraints to the rule merging mechanism. We are currently extending the mechanism in that direction. Our objective is to support various kinds of constraints. These constraints in turn express interactions between interactions (aspects) and are therefore called *meta interactions*. As an example for using meta interactions we give the following scenario. We have an interaction pattern *A* which binds the object *o1* and *o2* and there is an inverse interaction pattern *A'* that binds *o2* and *o1*. We use a meta interaction to state that whenever *A* (respectively *A'*) is instantiated the other pattern must also be instantiated. This can be useful to model father-son relationships for example. Another usage for meta interactions is to express aspect dependencies e.g., given the interaction patterns *A*, *B*, and *C* we use a meta interaction to specify that if *A* and *B* are activated then *C* must also be activated [14].

In [4], the author gives a precise description of *merging rules* for each ISL operator. These are rules on how to merge interaction rules for the various ISL operators: A merging rule explains how to get the result of merging a rule *ra* which contains an ISL operator *A* with a rule *rb*, which contains an ISL operator *B*. They are listed in the appendix at the end of this paper. There are also several *equivalence axioms* presented in [4]. The completeness, the commutativity, and associativity of the merging mechanism were proved in [4].

The Merging mechanism detects some conflicts such as the redefinition of one call in two different ways. This case corresponds to composition of *around* which do not use *proceed*. Based on the result of the merging, potential conflicts are detected when non-determinism calls to actions are identified. Fewer conflicts are detected than in aspect-oriented approaches based on *around* operators because we distinguish delegation, exception and conditions. As meta interactions do not occur at the merging level but at the activation level, they have no effect on the merging rules.

3.3 Implementation

We had developed two prototypes. In Noah [6], the byte code of the Java or .NET component class is modified at compile time or at load time using platform-specific bytecode engineering tools. GenInt [6] is a tool to make a Java class support interactions. MSILGenInt [6] is a similar tool for .NET assemblies.

Only the classes that appear at the left side of the interaction rule (*pointcut*) require byte code modification. This includes the introduction of a *meta class* and a *meta object* fields as well as wrappers for methods and field read/write operations. The other prototype works with the Shared Source Common Language Infrastructure SCCLI [7]. The in-memory class representation is

changed at runtime making the method table point to interception stubs that execute the advice when the pointcut matches..

The basic idea in both alternatives is that each *interacting component* (i.e., a component that can manage interactions) has a table of join points and advices. For each join point there is an array of advices. For example, for message reception join points we have an array holding the advices. Whenever the first aspect that matches the join point is deployed the respective advice is put in the first position in the array. When a new advice is added and its pointcut matches the join point, both advices are composed by rule merging and the resulting advice is put at the slot zero in the array. The same procedure applies when an advice is undeployed i.e., the result of rule merging is always put at position zero.

When the join point is reached in the course of the execution, the *meta object* of the corresponding class executes the advice at position zero i.e., the merging result. Advice execution is done by the *meta object* which interprets the ISL advice by visiting the ISL tree specified in the rule's reaction.

We developed supporting tools which enable the definition of interactions that connect Java components with .NET components. We also support Enterprise Java Beans with the extended JOnAs [13] application server.

We successfully implemented several crosscutting concerns with those prototypes e.g., *object replication* with interactions and java groups. Object replication has been implemented in two versions (replication for *fault tolerance*, replication for *management of the disconnected mode* together with several reconciliation protocols according to the application needs).

4. RELATED WORK

Although the number of AOP frameworks has flourished over the last few years, the problems related to aspect composition, shared join points, aspect interactions and inter-aspect dependencies have not been addressed adequately. To our best knowledge, the interaction model is the only proposal that supports advice merging. The originality of our proposal lies in automatically merging of the active advices at a shared join point into a single advice, which is semantically equivalent.

The problem of aspect interactions, conflicts and dependencies have been also addressed by other works. However, most approaches provide primitive support by letting the programmer specify the order of aspect execution. AspectJ [8] defines the *declare precedence* construct to specify the order of aspects. In our approach, even if the programmer does not specify the order of aspect execution, he still gets a good understanding of what happens at a shared join point.

In [1], the issues of composing multiple aspects at a shared join point are outlined. The paper presents a generic approach based on declarative specification of both ordering and control constraints among aspects. The work presented in [9] presents a generic framework for the formal definition and analysis of aspect interactions. The authors also introduce several aspect composition operators enabling the definition of more precise aspects at shared join points. This provides expressive support for the resolution of conflicts among interacting aspects. Our work is complementary to this one as we address the advice composition with a set of operators and do not investigate the expression of

interactions between aspects as they did. In this way, we are closer to the work of Batory [15] which considers aspects as a function, and composition of aspects as a composition of functions. In [5], the authors present a formal approach based on model analysis to capture aspect interference and detect potential conflicts early in the software development process.

The advantage of our proposal is that aspect merging is done automatically and is not the task of the programmer. Moreover, by presenting one single advice at the shared join point, the programmer can better understand the resulting behavior after aspect activation. This improves the quality of the software and eases testing and verification.

Our approach can be considered as an AOP for components, which is platform, component model and language independent. The work presented in [10] only provides language independence but it is bound to .NET as a platform. In our approach, it is possible to define an interaction rule between a .NET and a Java component. We also support EJB and CORBA CCM. Our work has been extended to take into account other pointcuts (call sending and event emitting). We added other operators such as return and "nop" to express "no operation to do". To take into account these new operators we just add new rules.

5. CONCLUSION

The main advantage of the interaction model over other AOP approaches is the merging mechanism, which is a novel aspect composition technique that composes the advices of various aspects with a shared join point into one semantically equivalent advice. The merging mechanism has been defined inductively. It is complete, associative and commutative. Automatic aspect composition improves the comprehensibility of the AOP application by telling the programmer which code will be executed at a shared join point. It also boosts the analyzability and the predictability of the system and makes testing and verification much simpler. In our experiments with students, as merging rules are intuitive, users learn them quickly. The tool, we developed to visualize the interactions between components, is used at the beginning when learning rules merging, or when a lot of aspects have been weaved on the same joint point. Even if the interaction patterns are dynamically merged, their visualization is useful to deal with the complexity of applications.

In the future, we will work on better integration of the *meta interactions* that express inter-aspect dependencies into the rule merging mechanism. We will extend the merging rules by adding rules on how precedences between actions affect ISL operators.

Acknowledgments: We would like to thank Laurent Berger, David Emsellem for building the Noah prototype. We also thank Mira Mezini and Klaus Ostermann for their fruitful comments on earlier drafts of this paper.

6. REFERENCES

- [1] Aspect-Oriented Software Development, <http://www.aosd.net>
- [2] I. Nagy, L. Bergmans, M. Aksit. *Declarative Aspect Composition*. SPLAT Workshop in conjunction with AOSD 04, Lancaster, U.K, March, 2003

