JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2007**

Vol. 6, No. 7,Special Issue: Aspect-Oriented Modeling, August 2007

Aspect Composition in the Motorola Aspect-Oriented Modeling Weaver

Jing Zhang, **Thomas Cottenier, Aswin van den Berg, and Jeff Gray**

Abstract

One of the fundamental issues in Aspect-Oriented approaches is aspect-to-aspect interference, which occurs when multiple aspects are deployed jointly such that different composition orders may give rise to various inconsistency problems. This paper describes how aspect precedence can be specified explicitly at the modeling level in order to derive a correct composition order and therefore reduce the aspect interference problem in Aspect-Oriented Modeling (AOM). The paper presents a modeling approach to achieve aspect reuse by introducing three distinct categories of aspect composition mechanisms. These composition concepts have been implemented in the Motorola ***WEAVR***, which is an AOM weaver developed at Motorola as a plug-in component for Telelogic TAU G2.

1 INTRODUCTION

Aspect-Oriented Modeling (AOM) [AOM] is an Aspect-Oriented Software Development (AOSD) [AOSD] extension applied to earlier stages of the software lifecycle. AOM aims at supporting separation of crosscutting concerns at the modeling level, with the purpose of enhancing productivity, quality and reusability through the encapsulation of requirements that cut across software components.

One of the fundamental issues in AOSD is the potential conflicts that may occur in the presence of interactions among aspects (i.e., when multiple aspectual behaviors are superimposed at the same joinpoint, different composition orders may reveal various inconsistency problems). In such circumstances, the aspects interfere with each other in a potentially undesired manner, either due to the side-effects caused by the aspects (e.g., several aspects change the state of the base program simultaneously) or due to the requirements enforced by the system (e.g., the logging aspect may be applied only in the presence of the encryption aspect because some particular systems require all logged data to be encrypted). A number of aspect interference examples have been described in [Durr, Kienzle06, Lagaisse, Nagy, Pawlak, Sihman].

Cite this article as follows: Jing Zhang, Thomas Cottenier, Aswin van den Berg, and Jeff Gray: "Aspect Composition in the Motorola Aspect-Oriented Modeling Weaver", in *Journal of Object Technology*, vol. 6, no. 7, Special Issue: Aspect-Oriented Modeling, August 2007, pp. 89-108
http://www.jot.fm/issues/issue_2007_08/article4

Several techniques have been proposed and developed to resolve or reduce aspect interference. For the most light-weight approach, the execution orders between aspects are governed by declaring precedence relationships, such as in AspectJ [AspectJ] and some aspect modeling approaches [Reddy]. Some other approaches extend the simple precedence declaration and introduce more complex dependencies and ordering relationships between aspects, such as [Kienzle03, Nagy]. Aspect interactions can be identified through static analysis on the crosscutting concerns and the base module [Douence02, Stoerzer06]. Advanced approaches require extra behavior specifications from the user for each advice [Durr, Pawlak] or each aspect [Lagaisse, Sihman]. The conflict between aspect semantics can then be detected automatically based on the specified contracts.

The problem of aspect interference is intrinsic to every AOSD technique (i.e., interference is at the essence of aspects due to the focus of multiple concerns that may crosscut at common locations). As an initial step towards resolving the interference issue in AOM, we adopt a light-weight approach following and extending the AspectJ [AspectJ] notation. This paper is not intended to analyze and detect the interference between aspects, nor does it concentrate on reasoning about the correctness of the system after composing multiple aspects simultaneously. Instead, we describe how aspect precedence can be specified explicitly at the modeling level in order to reduce the occurrences of aspect interference in AOM. Based on the precedence declarations, the underlying composition mechanism will derive an appropriate weaving and execution order automatically.

Furthermore, the paper presents a unique and powerful mechanism for pointcut specification based on state machines. The paper also shows how to facilitate aspect reuse to a larger extent by introducing three distinct categories of aspect composition mechanisms (i.e., pointcut composition, advice composition, and aspect composition). The approach has been implemented in the Motorola ***WEAVR***, which is an AOM plug-in for weaving aspects into executable UML state machine models in Telelogic TAU G2 [TAU]. The main benefit of this work is to improve the expressiveness and reusability of crosscutting concerns by handling aspect interference and composition at a higher level of abstraction.

The remainder of the paper is structured as follows. Section 2 gives a brief overview of the Motorola ***WEAVR***, including its basic language constructs and weaving procedure. Section 3 further illustrates the aspect definition and pointcut designators by examples. Section 4 presents the three categories of the composition mechanisms that have been implemented in the current version of the ***WEAVR***. Section 5 discusses the related work about aspect interference and composition. The paper concludes in Section 6 by summarizing contributions and ongoing work.

2 MOTOROLA *WEAVR*

The most essential feature of the Motorola ***WEAVR*** is to enable aspect-oriented weaving for UML statecharts that include action semantics [OMG]. By weaving aspects into executable UML models, the platform-specific models and the source code can be generated in an automated manner. This section provides a background introduction to the ***WEAVR*** in order to set the context for our contribution in aspect composition in models.

Two fundamental language constructs are introduced in the ***WEAVR***. First, we need to specify the "where" (i.e., the locations, or **joinpoints**, in the models where the crosscutting behavior emerges). Based on the UML concepts that actions are executed during a transition from one state to another state, two distinct types of joinpoints are supported in the ***WEAVR***: *action* and *transition* joinpoints, referring to the actions and transitions declared in the state machines, respectively. Examples of action joinpoints are variable definition, assignment, new operation, signal output, timer set, and expression method/constructor invocation. Transition joinpoints capture sequences of execution paths (e.g., from one state to another state, or from the starting point to the return point of a method execution) within a state machine.

A set of particular joinpoints are encapsulated in a **pointcut**, specially denoted by one of two dinstinct UML stereotypes: `<<ActionPointcut>>` or `<<TransitionPointcut>>`. The notation used for both types of pointcuts is identical: a pointcut is always represented as a transition from a set of source states to a set of target states. A pointcut has an interface that can specify the particular parameters exposed at the identified joinpoints. Further explanation of pointcut designators will be provided in the next section.

Second, we need to specify the "what" (i.e., the behavior of the crosscutting concern). In the ***WEAVR***, this behavior is implemented using state machines and encapsulated into a special kind of construct stereotyped by the name `<<advice>>`. An **advice** is named, containing the proceed operation, reflective API calls, as well as several parameters that are bound to the pointcut parameters.

Pointcuts and advice are encapsulated in an **aspect**, stereotyped by the name `<<aspect>>`. Aspects can own multiple pointcuts and advice. An aspect contains a binding diagram that defines what advice are bound to which pointcuts. Those bindings are stereotyped by the name `<<binds>>`. Aspects are applied to the base models through a deployment diagram.

Two phases are involved in the weaving process: advice instantiation and advice instance binding. During the first phase, advice are instantiated based on the pointcuts they are bound to, with most of the calls to the reflective API resolved. The `proceed` operation is replaced by the original joinpoint action. Matched joinpoints are annotated and linked to the corresponding advice. At this point, the base model has not been modified, except for the joinpoint annotations.

During the second phase, the aspects are woven into the base models in one of the following two ways: wrapping or inlining. In the wrapping mode, the original joinpoint actions are replaced by an operation call to the corresponding advice instance. For the inlining version, all advice instances are actually inlined in the base model. By allowing specific behavioral aspects to be woven into the abstract models, the ***WEAVR*** makes the use of executable UML more practical. For more details about the features of the Motorola ***WEAVR***, please refer to [Cottenier06, Cottenier07]. The ***WEAVR*** resources and academic free license can be obtained at [WEAVR].

3 POINTCUT DESIGNATORS

The ***WEAVR*** offers a unique mechanism for pointcut designators, which enables joinpoints to be strategically selected from state machine specifications. This section will illustrate the aspect definition and two distinct types of pointcut designators through an example based on a simplified authentication process model shown in Figure 1.

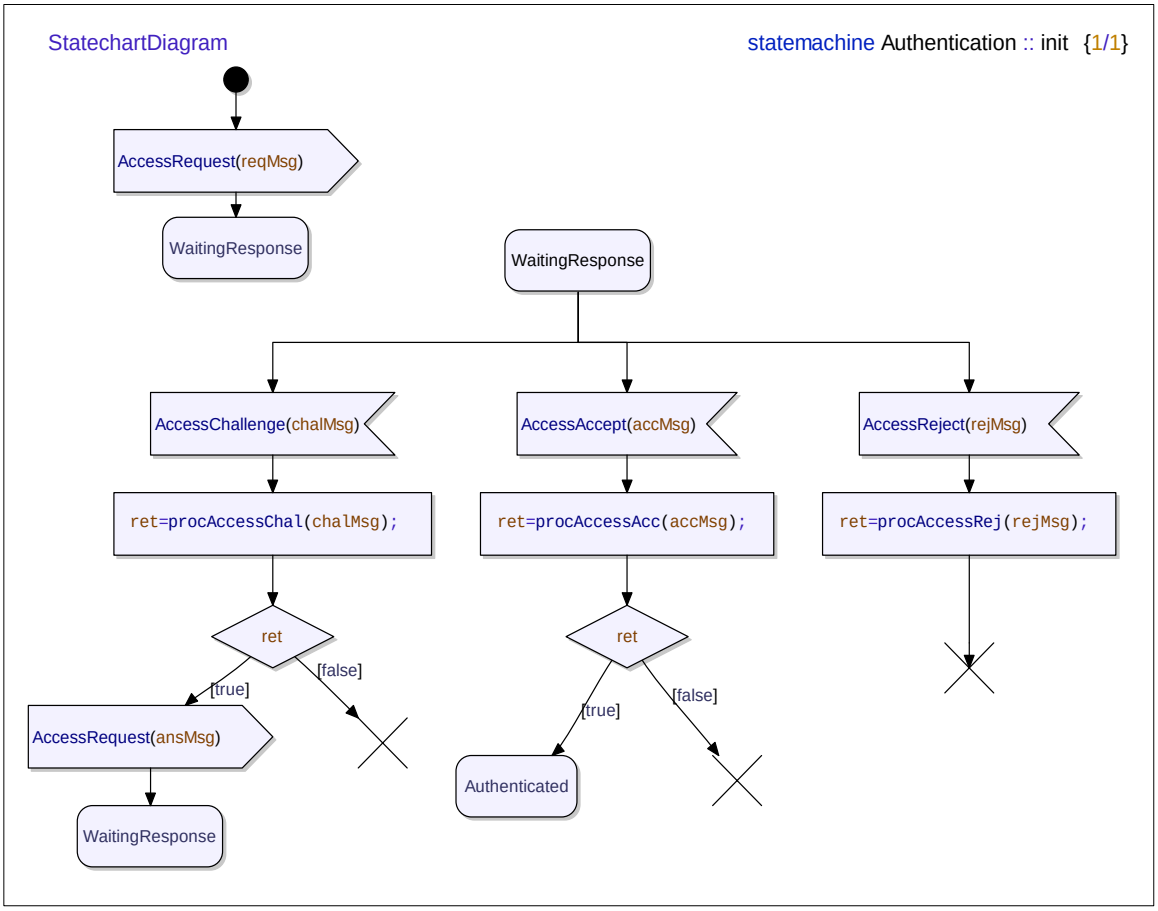


Figure 1. A simple authentication process model

This figure illustrates the process of an authentication request from a client to a server. After the client initiates a request for access, it enters the state of `WaitingResponse`, waiting for the reply from the server. Three kinds of responses (i.e., challenge, accept, and reject) could be received, each of which carries a distinct type of message. Depending on these different signals, the authentication process will trigger specific actions that lead to different states. If the challenge signal is received, the client will have to present an answer for the challenge question and request authentication again. If the accept signal is received and the actions for handling the signal return properly, the client will be authenticated. Otherwise, the whole process will terminate.

Figure 2 defines a tracing aspect applied to the authentication model. The aspect contains one advice that is bound to two pointcuts. The advice `tracing` wraps the original joinpoint action (denoted by `proceed`) by inserting print statements using reflective API calls (e.g., `thisJoinPoint`). The actions before `proceed` are called

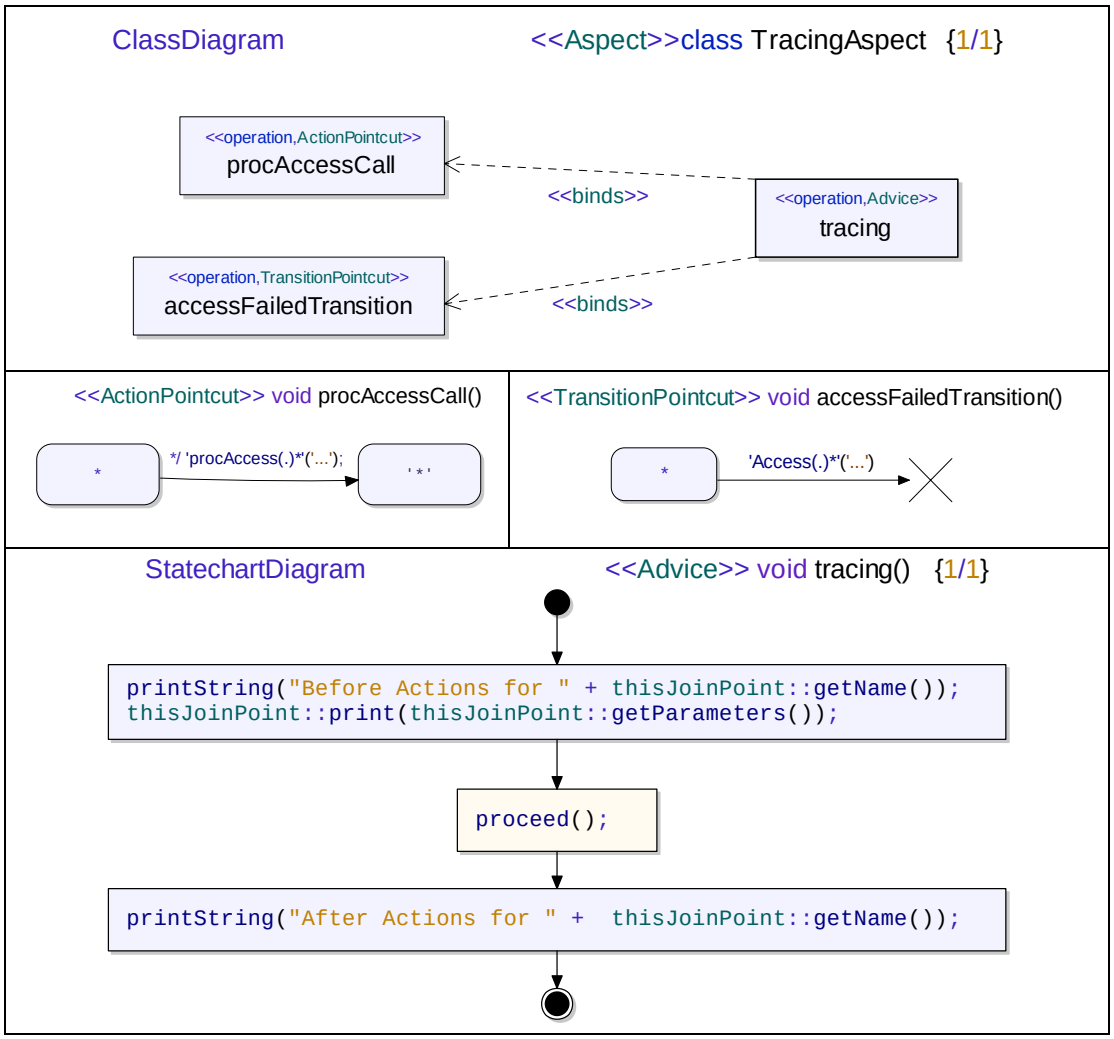


Figure 2. A tracing aspect definition with two pointcuts and an advice

"before actions" and the actions after `proceed` are named "after actions." The action pointcut `procAccessCall` captures all the call actions to a method whose name starts with `procAccess` (based on regular expression matching) and it may contain any number of parameters of any type. The transition pointcut `accessFailed-Transition` matches a selection of execution paths that lead to the state of `Stop` (represented by the symbol `X`) after receiving a signal whose name starts with `Access`. The joinpoints selected from these two pointcuts are shown in Figure 3. The red solid bars denotate the "before" and "after" advice actions to be inserted by the matching of the action pointcut `procAccessCall`. The green dashed bars delimit portions of execution paths that match the transition pointcut `accessFailedTransition`. The marks that occur first in the execution path correspond to the "before" actions whereas the second marks refer to the "after" actions. The joinpoint selection mechanism performs static control flow analysis to determine the earliest points that match the pointcut definition. The "before" marks are placed at the first location in the execution paths for which the only reachable next states match the target state of the pointcut designators (i.e., `Stop` state in this case), and the "after" marks are positioned right before the next state actions.

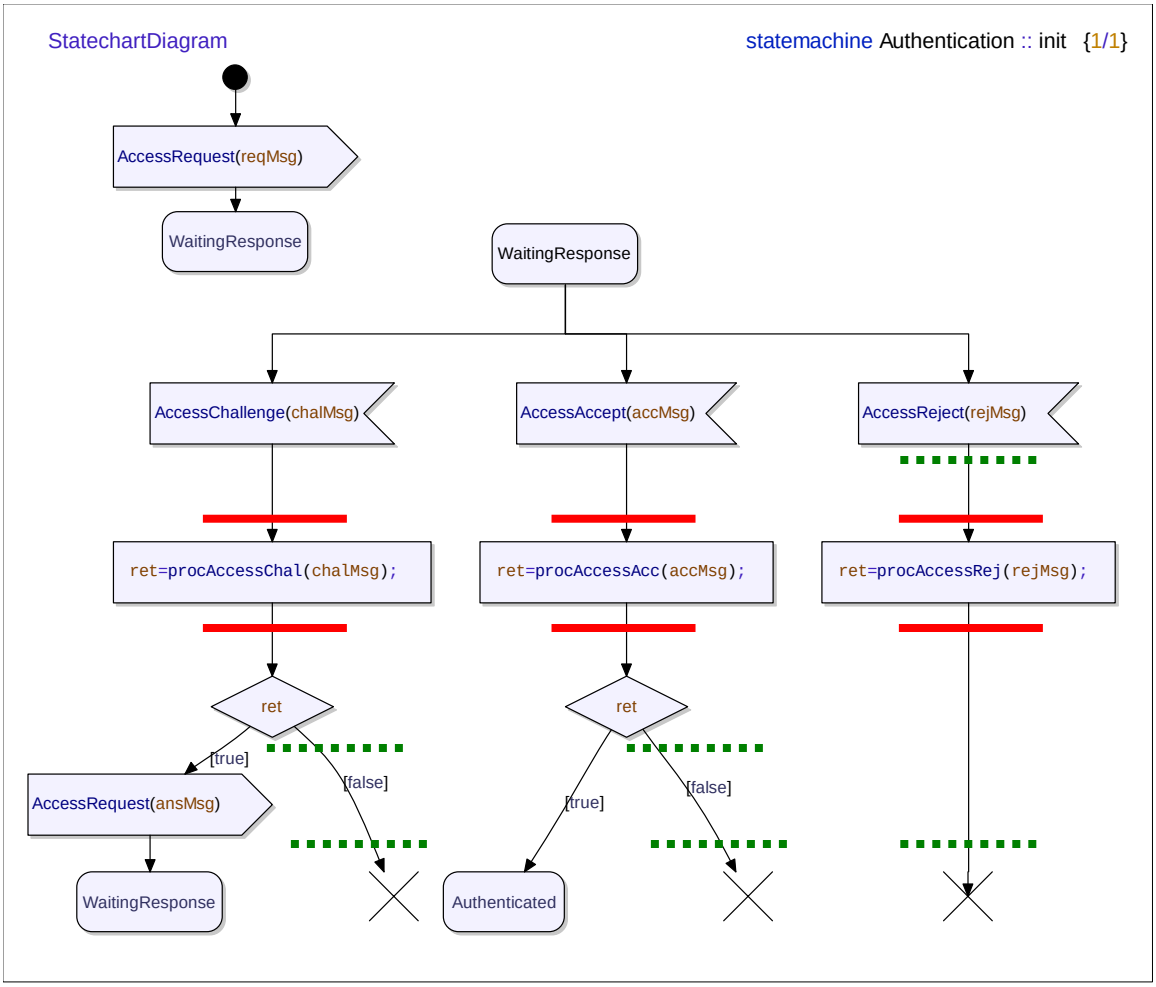


Figure 3. Selections of joinpoints that match the pointcut definitions in Figure 2

Due to the intrinsic difference of the definitions and the joinpoint selection mechanisms, action pointcuts and transition pointcuts will never match to the same joinpoint. (In Figure 3, the two "before" marks after the receiving signal `AccessReject` are essentially located at two different joinpoints.) Action pointcuts and transition pointcuts are comparable to the call pointcuts and execution pointcuts in AspectJ but with more powerful expressiveness.

Figure 4 illustrates a pointcut designator that can be interpreted as either an action pointcut or a transition pointcut. It captures all the action joinpoints executed in the context of a transition, as well as all the transition joinpoints that execute any actions in their context. Therefore, this pointcut matches every single joinpoint in the whole state machine model.

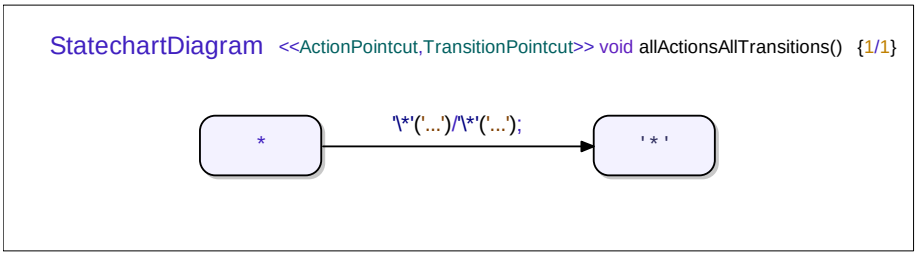


Figure 4. A pointcut designator to match every joinpoint in the state machine model

The context relationship between action and transition pointcuts also implies a **cflow** relationship – actions are executed within the control flow of transitions. Suppose in the authentication example, the actions `procAccessChal`, `procAccessAcc` and `procAccessRej` will all invoke a common routine `parse` in their control flow, which is used to analyze the authentication response information. One of the debugging requirements is, "whenever the `parse` operation returns an error, print out the values of the signal parameters that are associated by the signal received in the current context of transition." In this example, the authentication process prints `chalMsg` for `AccessChallenge`, `accMsg` for `AccessAccept`, and `rejMsg` for `AccessReject`.

This crosscutting concern can be captured easily by the `AccessAspect` shown in Figure 5. The action pointcut `ParseMsg` matches all the joinpoints where the operation `parse` (with any number parameters and a Boolean type return value) is invoked in the **cflow** of the transitions from state `WaitingResponse` to any target states by a trigger signal whose name starts with `Access`. The advice `PrintMsg` will call the `parse` routine and check whether the return value is "OK." If not, the advice will print out the signal name as well as its parameters for debugging purpose, through the use of `thisJoinPoint` reflective APIs. **Cflow** can also be used to compose two poincuts as will be seen in the next section.

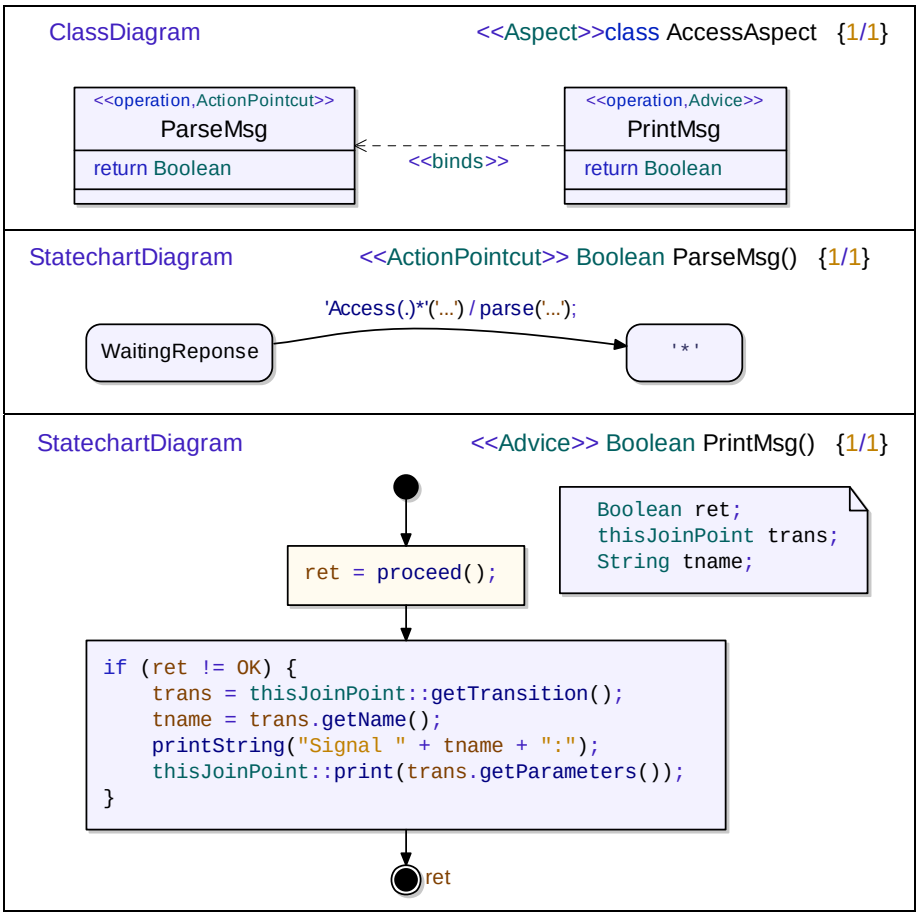


Figure 5. Access aspect definition with an action pointcut and a print advice

4 COMPOSITIONS IN THE *WEAVR*

Based on the distinct definition of the aspect constructs introduced in the previous sections, two kinds of interference problems may occur during the weaving process: advice-to-advice and aspect-to-aspect. (Note: We do not consider pointcut-to-pointcut interference in this paper because pointcuts do not own any behavior actions. Analysis on the pointcut-to-pointcut interference is considered as part of future work.) This section introduces precedence declarations on advice and aspects. The explicitly specified precedence constraints reduce undesired interference at the shared joinpoint and will be passed to the underlying model composition mechanism to compute a proper weaving order.

Reusability in AOM can be defined as the ability to reuse pointcuts, advice and aspects. To achieve reusability to the largest extent, we have implemented three distinct categories of composition mechanisms in the ***WEAVR*** (i.e., pointcut composition, advice composition and aspect composition). In the following sub-sections, each mechanism will be illustrated in detail and compared to the corresponding AspectJ notation.

Pointcut Composition

In the ***WEAVR***, the pointcut composition semantics strictly follow the AspectJ semantics. Pointcuts can be composed with Boolean operators to build other pointcuts. The Boolean expression is specified in a separate text box within the composite pointcut diagram. (Note: Currently we adopt a primitive and intuitive way to represent the pointcut compositions. However, the composition syntax can also be specified in a graphical notation through the extensions of the pointcut metamodel definition.) The supported Boolean operators are: AND (`&&`), OR (`||`) and NOT (`!`), indicating the intersection, union and difference of the set of the joinpoint selections, respectively. Furthermore, the ***WEAVR*** also supports `cflow`, `cflowbelow` and `within` pointcut designators. As illustrated in Figure 6, `CompositePointcut` is constructed by two sub-pointcuts – `procAccessCall` and `accessFailedTransition` (see pointcut definitions from Figure 2), which means that `CompositePointcut` will pick out joinpoints matched by `procAccessCall` that are not in the control flow of any joinpoint picked out by `accessFailedTransition`. After applying pointcut matching to the authentication model in Figure 1, the resulting joinpoints will be two method call actions: `procAccessChal` and `procAccessAcc`.

One advantage of our approach over AspectJ is that a pointcut can be directly referenced (e.g., through dragging and dropping in the model view) and reused in any other aspect. AspectJ, however, only allows the abstract aspect to be reused by inheritance. Concrete aspects extending an abstract aspect must provide concrete definitions of abstract pointcuts. Reusing pointcuts among multiple aspects is not possible in AspectJ.

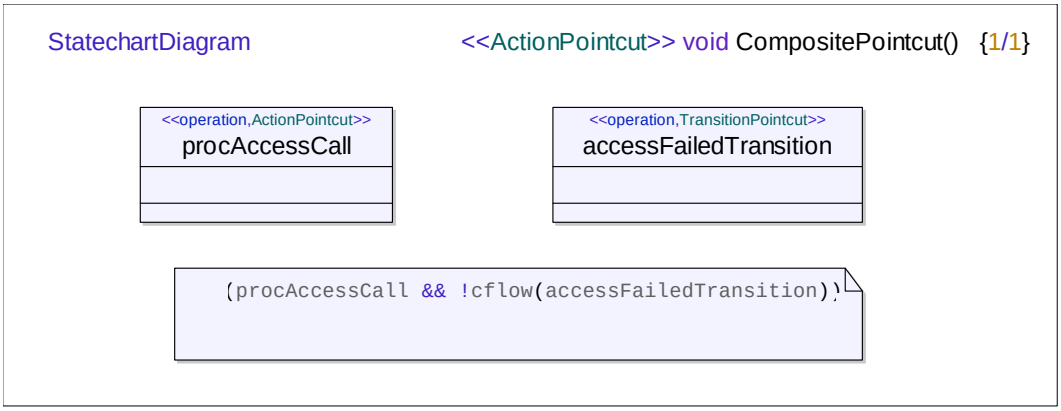


Figure 6. Pointcut composition

Advice Composition

Advice composition intends to bind and execute the advice instances that perform at the same joinpoint in a certain appropriate order. In the ***WEAVR***, advice are ordered based on the precedence relationships that are specified by the aspect developers. Currently, we have implemented the `<<follows>>` relationship between advice. As shown in Figure 7, `Advice2` follows `Advice1`, which means that at a particular joinpoint (either an action or a transition joinpoint), `Advice1` has precedence over `Advice2`, and the instances of `Advice2` will be executed closer to the joinpoint than the instances of `Advice1` (i.e., the "before" actions in `Advice1` instances will always be executed prior to the "before" actions in `Advice2` instances while the "after" actions will be carried out in the opposite order). In the absence of an ordering constraint, the execution order of the corresponding advice instances is undefined and controlled by the underlying ***WEAVR*** compiler.

Ordering relationships specify a partial order upon the execution of a set of advice instances. In order to obtain a composition and execution order, a topological sort is performed on the advice. Circular dependencies among the advice are detected when their corresponding pointcuts match to the same joinpoint. Under such circumstance, the ***WEAVR*** will abort with an error message, indicating the problematic advice involved in the circularity.

When executing an advice instance, the call to proceed will be redirected to the invocation of the advice instance with the next precedence, or the computation under the joinpoint if there is no further advice instance. In the case of Figure 7, suppose `Pointcut1` and `Pointcut2` both match a single joinpoint (in the following, `PCT` replaces `Pointcut` and `ADV` replaces `Advice`). The advice instantiation order at this joinpoint could be:

```
PCT1-ADV1, PCT2-ADV1, PCT1-ADV2, PCT2-ADV2
```

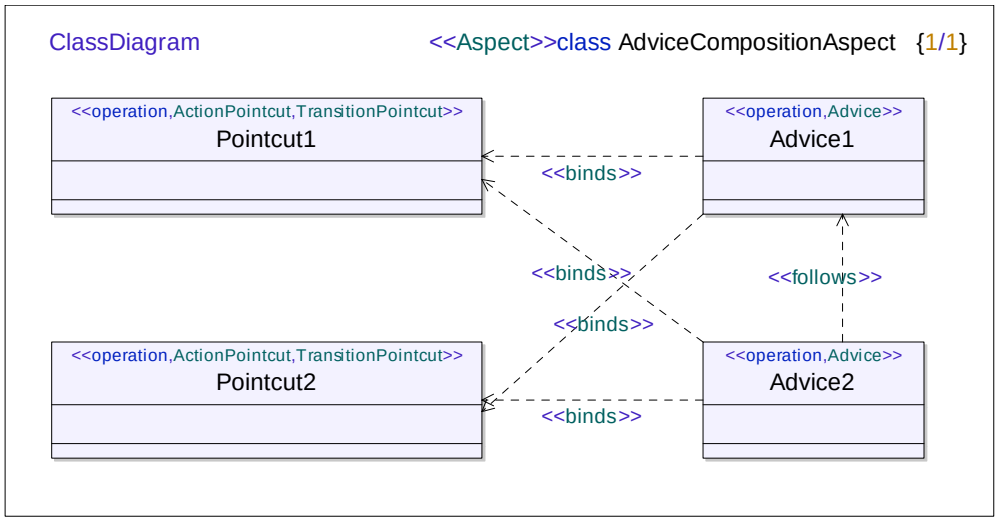
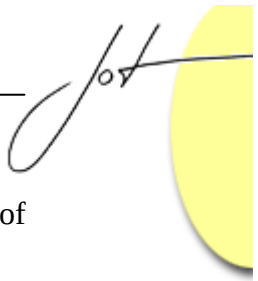


Figure 7. Advice composition



If both `Advice1` and `Advice2` contain a `proceed` action, the execution order of the woven model at this joinpoint would be as follows:

- Before actions in the advice instance: `PCT1-ADV1`
- Before actions in the advice instance: `PCT2-ADV1`
- Before actions in the advice instance: `PCT1-ADV2`
- Before actions in the advice instance: `PCT2-ADV2`
- Original actions at the joinpoint
- After actions in the advice instance: `PCT2-ADV2`
- After actions in the advice instance: `PCT1-ADV2`
- After actions in the advice instance: `PCT2-ADV1`
- After actions in the advice instance: `PCT1-ADV1`

By comparing our advice composition mechanism with AspectJ, we believe that our approach offers two advantages:

1. In the ***WEAVR***, the concepts of pointcuts and advice are loosely decoupled. An advice is named, which allows it to be associated with not just one, but multiple pointcuts as long as they share compatible interfaces. Therefore, an advice can be directly referenced (e.g., through dragging and dropping in the model view) and reused in different aspects in a compositional way. In AspectJ, advice is unnamed and can only be bound to one particular pointcut. The tight coupling between pointcuts and advice makes aspects difficult to reuse. The only way to reuse advice in AspectJ is by means of inheritance, which is known to be more brittle and less flexible than the composition-based solution [Gamma].
2. In AspectJ, the precedence of advice relies completely on their textual locations in an aspect file. The underlying interpretation rules, as stated in the AspectJ Programmers Guide [AspectJ], declare that, "for two advice within a single aspect, if either is after advice, then the one that appears later in the aspect has precedence over the one that appears earlier; otherwise, the one that appears earlier in the aspect has precedence over the one that appears later." These rules have limitations and cannot express all composition orders, as pointed out in [Herrejon]. Our approach resolves the above problems because there is only one advice type (i.e., around advice) in the ***WEAVR***, which decreases the complexity of handling three different types (i.e., before, after and around) of advice as in AspectJ. Furthermore, by declaring the advice precedence explicitly, the interference between the advice is reduced.

Aspect Composition

Aspect composition is achieved through a deployment diagram (Figure 8), which is used to bind aspects to the base models, with the precedence relationships declared. Aspects can be bound to multiple base models through the stereotype `<<crosscuts>>` (e.g., `ExceptionAspect` and `TracingAspect` are both applied to the base package `Authentication`). Aspects can also be deployed to other aspects or advice. In the

absence of the <<crosscuts>> relationship, aspects will be applied to all the base models in the current active project (e.g., LoggingAspect and EncryptionAspect). The precedence relationships between aspects can be <<follows>>, <<hidden_by>> and <<dependent_on>>. The remainder of this section explains these three concepts in detail based on the example provided in Figure 8.

1. **TracingAspect follows ExceptionAspect:** TracingAspect (as defined previously in Section 3, Figure 2) is used to print out signatures and parameter values of some particular action and transition joinpoints. ExceptionAspect captures all of the transitions that lead to the stop state and sends out an exception notice right before the system terminates. There is no "before" advice in ExceptionAspect. The <<follows>> relationship between these two aspects means that at a single joinpoint, ExceptionAspect has higher precedence than TracingAspect. Therefore, all of the advice in the ExceptionAspect have higher precedence than the ones in the TracingAspect. In other words, the advice instances instantiated from TracingAspect will be executed closer to the joinpoint than the ones instantiated from ExceptionAspect. The execution order of the woven model at this shared joinpoint would be as follows:
    - Before advice instance in TracingAspect
    - Joinpoint action(s)
    - After advice instance in TracingAspect
    - After advice instance in ExceptionAspect
    - Stop state entered

2. **LoggingAspect is hidden by TracingAspect:** LoggingAspect stores the attributes and data of a particular interest into a database whenever they are used or modified. However, the system may not always want to log everything, such as those data that are being traced by TracingAspect. The <<hidden_by>> relationship inactivates LoggingAspect whenever it matches the same joinpoint as TracingAspect. The correlation between TracingAspect and LoggingAspect can be described using the following expression:

    $$\text{TracingAspect} \Rightarrow \neg\text{LoggingAspect}$$

    This notation means that the presence of TracingAspect implies the absence of LoggingAspect. For each pointcut denoted as $\text{Pointcut}_{\text{LoggingAspect}}$ in LoggingAspect, the actual corresponding pointcut exposed by this particular deployment strategy is

    $$\text{Pointcut}_{\text{LoggingAspect}}' = \text{Pointcut}_{\text{LoggingAspect}} \;\&\&\; \neg\text{Pointcut}_{\text{TracingAspect}}$$

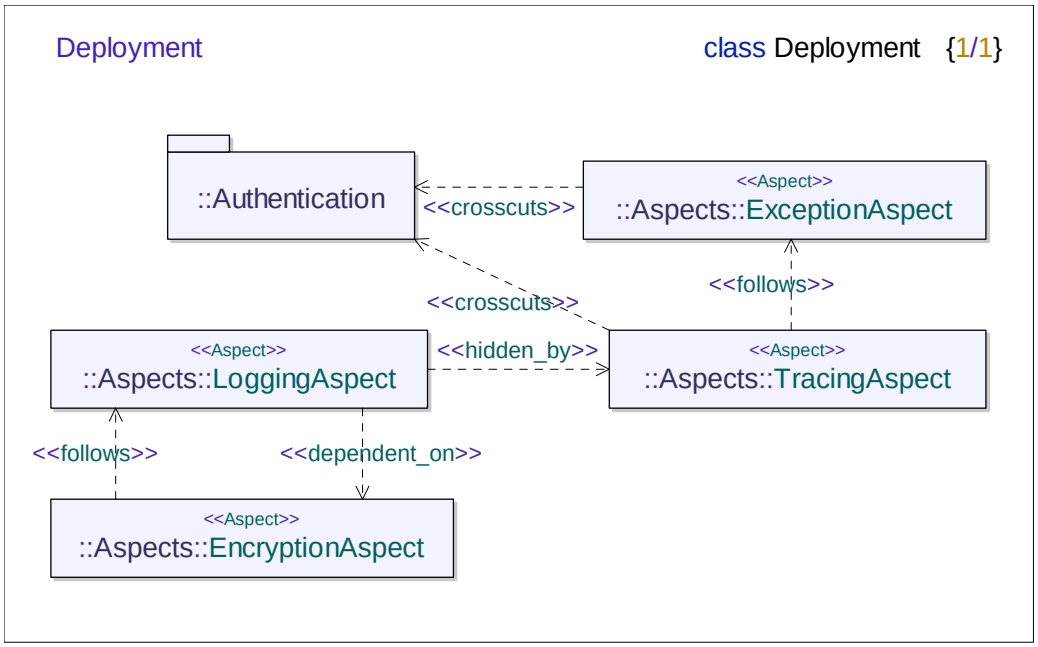


Figure 8. Aspect Composition

3. **`LoggingAspect` is dependent on `EncryptionAspect`:** The `<<dependent_on>>` relationship enforces the `LoggingAspect` to be applied only in the presence of the `EncryptionAspect`. This is enforced because some systems may require all logged data to be encrypted (i.e., `LoggingAspect` will only be applied at the joinpoints when `EncryptionAspect` and `LoggingAspect` both match. `LoggingAspect` will be disabled at the other joinpoints where it matches apart from `EncryptionAspect`). The relationship between `LoggingAspect` and `EncryptionAspect` is denoted as

   `LoggingAspect => EncryptionAspect`

   This means that the presence of `LoggingAspect` implies that `EncryptionAspect` has to be present at the same joinpoint as well. Therefore, under this particular condition, the actual pointcut exposed by `LoggingAspect` is

   $$Pointcut_{LoggingAspect}' = Pointcut_{LoggingAspect} \&\& Pointcut_{EncryptionAspect}$$

   The resulting joinpoint selection set for `LoggingAspect` is indicated by the striped area in Figure 9. In addition, as illustrated in Figure 8, `EncryptionAspect` also `<<follows>>` `LoggingAspect`, which forces encryption actions to be executed closer to the joinpoint than logging procedures.

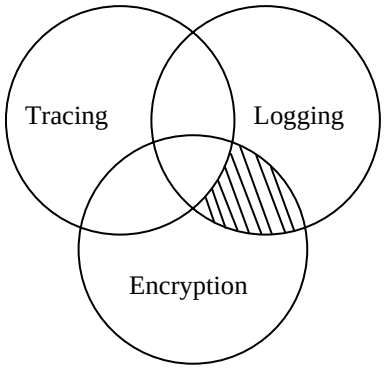


Figure 9. Joinpoint selection for LoggingAspect
by the deployment strategy in Figure 8

In order to detect and collect all of the joinpoints by traversing the whole model in linear time, the base model is divided into several exclusive sets from the deployment diagram. The derived aspect composition order for Figure 8 is:

```
Authentication      <-   ExceptionAspect,
                         TracingAspect,
                         LoggingAspect',
                         EncryptionAspect

ALL-Authentication  <-   LoggingAspect',
                         EncryptionAspect
```

Within the scope of package `Authentication`, `ExceptionAspect` will be applied first, followed by `TracingAspect`, `LoggingAspect` (with the new composite pointcut) and `EncryptionAspect`. For all of the other models that are not within the scope of `Authentication` (denoted by subtracting `Authentication` from `ALL` with a minus sign "`-`"), only `LoggingAspect` and `EncryptionAspect` will be applied. Circular and conflict relationships among the aspects will be detected and reported when they are superimposed at the same joinpoint.

The advantages of our approach over AspectJ are:

1. In the ***WEAVR***, aspects are explicitly deployed by means of a deployment diagram. Aspects can be bound to different fragments of the base models; in AspectJ, aspects are applied everywhere. The only way to apply an aspect to a certain scope is to restrict *every* pointcut specification by using the "within" keyword, which are bound lexically to method or class names; this makes pointcuts and aspects less reusable.

2. The semantics of the `<<follows>>` relationship in the ***WEAVR*** correspond to the "declare precedence" form in AspectJ. In addition, the ***WEAVR*** is able to handle two more dependency relationships between aspects (i.e., `<<hidden_by>>` and `<<dependent_on>>`), which further restrict application of an aspect at the same joinpoint. In AspectJ, however, when a pointcut matches a certain joinpoint, the corresponding aspect is always applied.

In this section, we introduced composition mechanisms implemented in the ***WEAVR*** in three distinct categories: pointcut, advice and aspect. From our experience, we have found that by integrating compositions at different granularity levels, the aspect expressiveness and reusability can be extended to a larger extent. By declaring precedence relationships between crosscutting concerns, the aspect interference can be reduced and controlled by the aspect developers.

5 RELATED WORK

Aspect interference is a well-known problem to every AOSD approach and has received considerable attention in the research literature. The Aspects, Dependencies, and Interactions (ADI) Workshop [ADI] is particularly dedicated to discussing this issue. An exhaustive classification and documentation of aspect interactions has been investigated by Sanen et al. [Sanen]. This section will give a brief overview on some of the existing techniques that provide support for handling aspect interference.

As a light-weight approach, AspectJ [AspectJ] controls the aspect ordering by the "declare precedence" statement. As noted by Reddy et al. [Reddy], weaving order is defined by two composition directives, i.e., "follows" and "precedes," at the class design modeling level. Theme/UML [Clarke] resolves aspect conflicts by indicating precedence order using a "prec" tag. We extend these approaches by introducing two additional precedence relationships between aspects. Furthermore, our approach allows precedence to be specified explicitly between advice.

A number of advanced approaches have been proposed to manipulate aspect interference at the programming level. Kienzle et al. [Kienzle03] defined an aspect based on the services it provides, requires, and removes. They also established a set of composition rules to solve inter-aspect dependencies. Similarly, aspect integration contracts were introduced by Lagaisse et al. [Lagaisse] to manage semantic interference between the aspects. Sihman and Katz [Sihman] united the theory of superimposition with AOP, allowing interactions and relations to be expressed among generic aspects, which can be used to define proof obligations for the correctness of superimpositions and to check feasibility of combining superimpositions. Nagy et al. [Nagy] proposed ordering, control and structural constraints to address the aspect interference issue. Pawlak et al. [Pawlak] developed a language called "CompAr" to specify composition-relevant information on the advice, such as Boolean choices, action executions/invocations and post-execution constraints, with the purpose of detecting and solving aspect composition issues. As a similar approach, Durr et al. [Durr] defined the

semantics of advice in terms of operations on an abstract resource model, in order to reason about semantic conflicts between aspects. Douence et al. [Douence02] presented static conflict analysis for aspect interactions and proposed some linguistic support for conflict resolution. Later on, they extended their work by defining applicability conditions for aspects, which made interaction analysis more precise [Douence04]. Stoerzer et al. [Stoerzer06] proposed an interference criterion by analyzing data flow and control flow of the advice in order to detect aspect conflicts.

The primary benefit our approach compared to these other techniques is that aspect interference is reduced even before proceeding to the implementation level, which enhances the comprehensibility and reusability of the crosscutting concerns. Furthermore, our unique pointcut designator provides more powerful expressiveness because it allows joinpoints to be selected from the state machine specifications.

6 CONCLUSIONS

The paper introduces the aspect composition mechanisms in the Motorola ***WEAVR***, an industrial AOM tool that is currently being deployed in production in the Motorola network infrastructure business unit [WEAVR]. A key point when dealing with aspects is the notion of aspect interference. One of the primary contributions of this paper is an approach that allows precedence relationships to be specified at the modeling level to prevent undesirable interference. Model engineers make design decisions explicitly based on the dependencies between aspectual behaviors. The underlying composition mechanism in the Motorola ***WEAVR*** derives a composition order automatically. Furthermore, the three distinct categories of aspect composition mechanisms implemented in the ***WEAVR*** are introduced with the purpose of facilitating aspect reusability and comprehensibility to a larger extent than the other existing AOSD approaches.

We are currently investigating the interference problem in a more systematic way in order to explore and validate the precise needs for various aspect dependences and constraints that can be introduced in the ***WEAVR***. We are also working on integrating the composition mechanism with the debugging and simulation feature, so that the model engineers can simulate the model in different perspectives and verify the impacts that are caused by each applied aspect.

As this research is still in the preliminary phases, there are some limitations. The current version has not taken into account pointcut-to-pointcut interference. However, in recent AOSD literature, the so-called “fragile pointcut” problem [Stoerzer04] has been studied as an important aspect interference issue. The future work will include investigation on the topic of pointcut interference. Also, the current research does not take into account the correctness of the system after composing multiple aspects simultaneously. The future work will exploit the reasoning mechanisms for weaving correctness.

7 ACKNOWLEDGEMENTS

This work is partially supported by the National Science Foundation under CAREER-CCF-0643725.

REFERENCES

[ADI] ADI, *Aspects, Dependencies, and Interactions Workshop at European Conference on Object-Oriented Programming (ECOOP)*, Nantes, France, July 2006. http://www.aosd-europe.net/adi06/

[AOM] AOM Website: http://www.aspect-modeling.org/

[AOSD] AOSD Website: http://www.aosd.net/

[AspectJ] AspectJ Website: http://www.eclipse.org/aspectj/

[Clarke] Siobhàn Clarke and Elisa Baniassad, *Aspect-Oriented Analysis and Design: The Theme Approach*, Addison Wesley, 2005.

[Cottenier06] Thomas Cottenier, Aswin Van Den Berg and Tzilla Elrad, "An Add-in for Aspect-Oriented Modeling in Telelogic TAU G2," *Telelogic User Group Conference (UGC)*, Denver, CO, October 2006.

[Cottenier07] Thomas Cottenier, Aswin van den Berg, and Tzilla Elrad, "Joinpoint Inference from Behavioral Specification to Implementation," In *Proceedings of the 21st European Conerence on Object-Oriented Programming (ECOOP)*, Berlin, Germany, July 2007.

[Douence02] Rémi Douence, Pascal Fradet and Mario Südholt, "A Framework for the Detection and Resolution of Aspect Interactions," In *Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE)*, Pittsburgh, PA, October 2002, pp. 173-188.

[Douence04] Rémi Douence, Pascal Fradet and Mario Südholt, "Composition, Reuse and Interaction Analysis of Stateful Aspects," In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD)*, Lancaster, UK, March 2004, pp. 141-150.

[Durr] Pascal Durr, Tom Staijen, Lodewijk Bergmans, and Mehmet Aksit, "Reasoning About Semantic Conflicts Between Aspects," *2nd European Interactive Workshop on Aspects in Software (EIWAS)*, Brussels, Belgium, September 2005.

[Gamma] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[Kienzle03] Jörg Kienzle, Yang Yu, Jie Xiong, "On Composition and Reuse of Aspects," In *Proceedings of the 2nd Workshop on Foundations of Aspect-Oriented Languages (FOAL)*, Boston, MA, March 2003.

[Kienzle06] Jörg Kienzle and Samuel Gélineau, "AO Challenge - Implementing the ACID Properties for Transactional Objects," In *Proceedings of the 5th International Conference on Aspect-oriented Software Development (AOSD)*, New York, NY, March 2006, pp. 202-213.

[Lagaisse] Bert Lagaisse, Wouter Joosen, and Bart De Win, "Managing Semantic Interference with Aspect Integration Contracts," *International Workshop on Software-Engineering Properties of Languages for Aspect Technologies (SPLAT)*, Lancaster, UK, March 2004.

[Herrejon] Roberto Lopez-Herrejon, Don Batory, and Christian Lengauer, "A Disciplined Approach to Aspect Composition," In *Proceedings of Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM)*, Charleston, SC, January 2006, pp. 68-77.

[Nagy] Istvan Nagy, Lodewijk Bergmans and Mehmet Aksit, "Composing Aspects at Shared Joinpoints," In *Proceedings of International Conference NetObjectDays (NODe)*, Erfurt, Germany, September 2005, pp. 19-38.

[OMG] OMG. Semantics for a foundational subset for executable UML models - request for proposal. *Request for Proposal ad/2005-04-02*, Object Management Group, 2005.

[Pawlak] Renaud Pawlak, Laurence Duchien, and Lionel Seinturier, "CompAr: Ensuring Safe Around Advice Composition," In *Proceedings of 7th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, Athens, Greece, June 2005, pp. 163-178.

[Reddy] Raghu Reddy, Sudipto Ghosh, Robert France, Greg Straw, James M. Bieman, N. McEachen, Eunjee Song, and Geri Georg, "Directives for Composing Aspect-Oriented Design Class Models," *Transactions on Aspect-Oriented Software Development I*, LNCS 3880, Springer-Verlag, 2006, pp. 75-105.

[Sanen] Frans Sanen, Eddy Truyen, Wouter Joosen, Andrew Jackson, Andronikos Nedos, Siobhan Clarke, Neil Loughran, and Awais Rashid, "Classifying and documenting aspect interactions," *The 5th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, Bonn, Germany, March 20, 2006.

[Sihman] Marcelo Sihman and Shmuel Katz, "Superimpositions and Aspect-Oriented Programming," *The Computer Journal*, 46(5), September 2003, pp. 529-541.

[Stoerzer04] Maximilian Stoerzer and Christian Koppen, "PCDiff: Attacking the Fragile Pointcut Problem," In *Proceedings of the 1st European Interactive Workshop on Aspects in Software (EIWAS)*, Berlin, Germany, July 2004.

[Stoerzer06] Maximilian Stoerzer, Robin Sterr and Florian Forster, "Detecting Precedence-Related Advice Interference," In *Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE)*, Tokyo, Japan, September 2006, pp. 317-322.

[TAU] Telelogic TAU G2 Website: http://www.telelogic.com/corp/products/tau/g2/index.cfm

[WEAVR] Motorola ***WEAVR*** Website: http://www.iit.edu/~concur/weavr/

About the author(s)

Jing Zhang is a researcher at Motorola Labs, where she is responsible for conducting research on Autonomic Network Management. Jing is also a part-time PhD student in the Department of Computer and Information Sciences at the University of Alabama at Birmingham. Her PhD research is focused on techniques that combine model transformation and program transformation in order to assist in evolving large software systems. She can be reached at j.zhang@motorola.com.

Thomas Cottenier works as a researcher for the Motorola Software Group and is a PhD student at the Computer Science department of the Illinois Institute of Technology, USA. Thomas holds a Communication Engineering degree from the Université Libre de Bruxelles, Belgium and a MS in Computer Engineering from the Illinois Institute of Technology. Thomas' interests include Aspect-Oriented Software Development and Model-Driven Software Development applied to telecommunication and distributed computing systems. He can be reached at thomas.cottenier@motorola.com.

Aswin van den Berg has a PhD in Computer Science from Cornell University and has been working with the Software and System Engineering Research Lab at Motorola Labs from 2000 until 2007 and for the Motorola Software Group in Schaumburg, IL since 2007. His graduate work focused on Program Transformation Systems and Higher Order Attribute Grammars. At Motorola he has been working on automatic code generation from SDL/UML software models and is now the project leader of the Motorola WEAVR project. He can be reached at aswin.vandenberg@motorola.com.

Jeff Gray is an Assistant Professor in the Computer and Information Sciences Department at the University of Alabama at Birmingham (UAB), where he co-directs research in the Software Composition and Modeling (SoftCom) Laboratory. His research interests are in model-driven engineering, aspect-oriented software development, and generative programming. He can be reached at gray@cis.uab.edu.