Aspect Interference and Composition in the Motorola Aspect-Oriented Modeling Weaver

Jing Zhang
Motorola Labs
Department of Computer and Information Sciences
University of Alabama at Birmingham
zhangj@cis.uab.edu

Thomas Cottenier
Aswin van den Berg
Motorola Labs
{thomas.cottenier, aswin.vandenberg}@motorola.com

Jeff Gray
Department of Computer and Information Sciences
University of Alabama at Birmingham
gray@cis.uab.edu

ABSTRACT
Aspect-Oriented Modeling (AOM) aims at supporting separation of concerns at the modeling level, with the purpose of improving productivity, quality and reusability through the encapsulation of requirements that cut across software components. One of the fundamental issues in Aspect-Oriented approaches is aspect-to-aspect interference – when multiple aspects are deployed jointly, different composition orders may give rise to various inconsistency problems. This position 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 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.

Categories and Subject Descriptors
D.2.2 [**Software Engineering**]: Design Tools and Techniques.

General Terms
Algorithms, Design, Experimentation, Languages

Keywords
Aspect-Oriented Software Development, Aspect-Oriented Modeling, Aspect Interference, Aspect Composition

1. INTRODUCTION
Aspect-Oriented Modeling (AOM) [2] is an Aspect-Oriented Software Development (AOSD) [3] extension applied to earlier stages of the software lifecycle. AOM aims at supporting separation of concerns at the modeling level, with the purpose of enhancing the productivity, quality and reusability through the encapsulation of requirements that cut across software components.

One of the fundamental issues in AOSD concerns the potential conflicts that may occur during interaction among aspects (i.e., when multiple aspectual behaviors are superimposed at the same join point, 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 [7, 10, 12, 14, 17].

Several techniques have been proposed and developed to resolve or reduce the aspect interference problem. For the most light-weight approach, the execution orders between aspects are governed by declaring precedence relationships, such as in AspectJ [4] and some aspect modeling approaches [15]. Some other approaches extend simple precedence declaration and introduce more complex dependencies and ordering relationships between aspects, such as [9, 12]. Advanced approaches require extra behavior specifications for each advice [7, 14] or each aspect [10, 17] from the user. The conflict between aspect semantics can 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 [4] 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. Additionally, the paper shows how to facilitate aspect reuse to a larger extent by introducing three distinct categories of aspect composition mechanisms: pointcut composition, connector composition (similar to advice in AspectJ [4]), and aspect composition. The approach has been implemented in the Motorola ***WEAVR***, which is an AOM plug-in for manipulating executable UML models in Telelogic TAU G2 [19]. The main benefit of this work is to improve the expressiveness and reusability of crosscutting concerns by handling the aspect interference and composition at a high level of abstraction.

The remainder of the paper is structured as follows. Section 2 gives a brief overview of the Motorola ***WEAVR***, including the basic concepts and weaving procedure. Section 3 introduces three different categories of the composition mechanisms that have been implemented in the current version of the ***WEAVR***. Section 4 discusses the related work. The paper concludes in Section 5 by summarizing contributions and ongoing work.

2. MOTOROLA *WEAVR* OVERVIEW

The essential feature of the Motorola ***WEAVR*** is to enable aspect-oriented weaving for UML statecharts that include action semantics [13]. By weaving aspects into the executable UML models, the platform-specific models and the source code can be fully automatically generated. Two fundamental language constructs are introduced in the ***WEAVR***. First, we need to specify the "where" (i.e., the locations, or join points, 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 types of join points are supported in the ***WEAVR***: *action* and *transition* join points. A set of particular join points are encapsulated in a special kind of construct, denoted by the **pointcut** stereotype. A pointcut has an interface that specifies the particular parameters or return value exposed at the identified join points.

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 **connector**. A connector corresponds to the advice construct in Aspect-Oriented Programming (AOP) languages such as AspectJ [4]. A connector is named, containing the `proceed` operation, reflective API calls, as well as several parameters that are bound to the pointcut parameters.

Pointcuts and connectors are encapsulated in a special construct, stereotyped by the name **aspect**. Aspects can own multiple pointcuts and connectors. An aspect contains a binding diagram that defines which connectors are bound to which pointcuts. Those bindings are stereotyped by the name `<<binds>>`. Figure 1 illustrates an example for an aspect definition (top of the figure) with one pointcut and one connector. This aspect imposes an error handling concern for each setup operation in the system. `Pointcut1` (bottom-left of the figure) captures all the join points whose expression is an invocation of a method with a signature matched by "`int *::*Setup*(*, *, *)`." This pointcut exposes the first and second arguments (denoted by `p1a` and `p1b`) of the call expression, as well as its return value (denoted by `return int`). `Connector1` is bound to `pointcut1` through the `<<binds>>` relationship with a compatible list of parameters (indicated by `c1a`, `c1b` and `return int`). The connector implementation (bottom-right of the figure) first executes the original join point action (denoted by `proceed`), then checks the return value of the join point. If the value is negative, the connector will print a logging message using a reflective API call (e.g., `thisJoinPoint`) and then move into an `idle` state. The actions before `proceed` are called "before actions" and the actions executed after `proceed` are named "after actions."

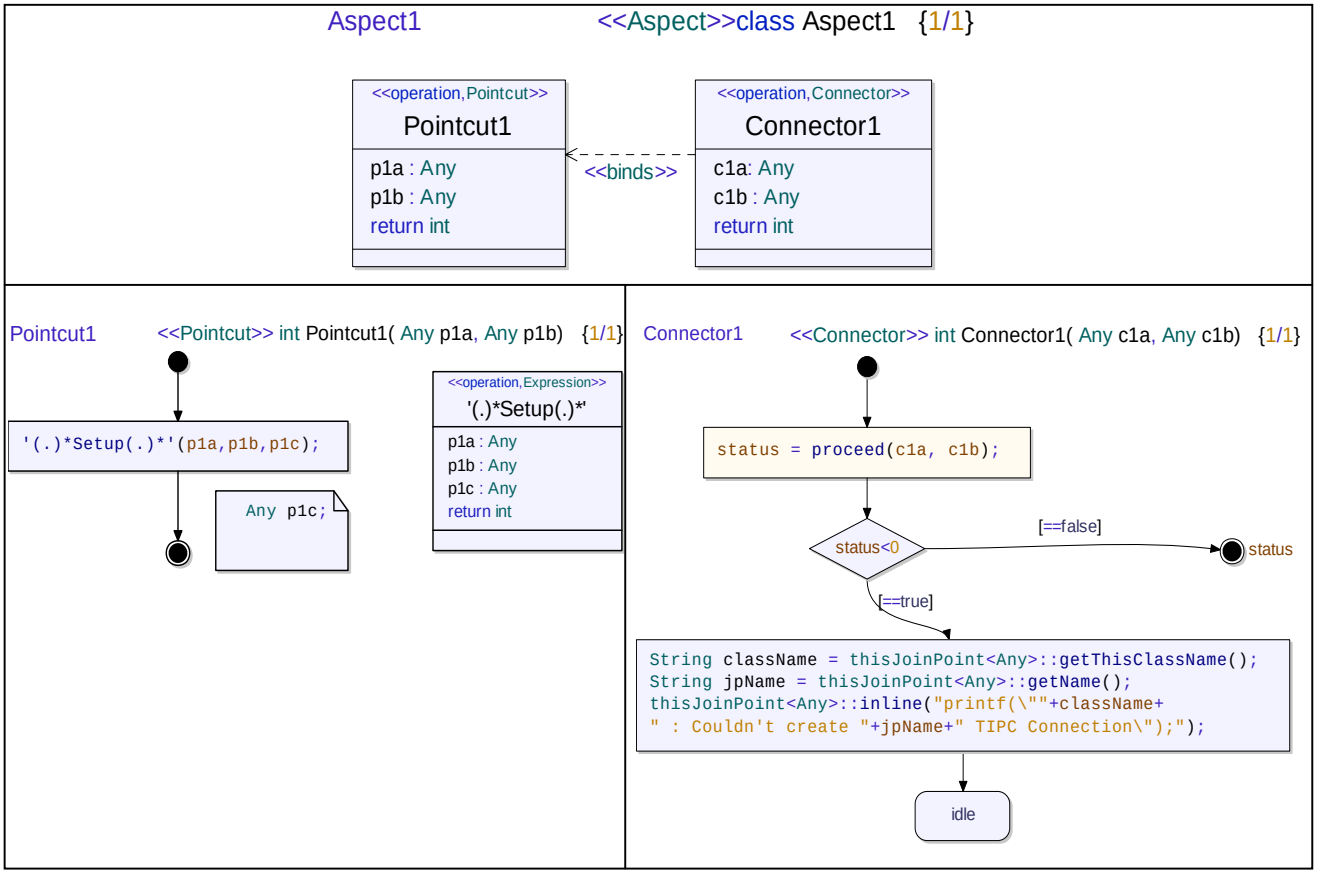

Figure 1. An aspect definition with a pointcut and a connector

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

During the second phase, the aspects are woven into the models in one of two ways: wrapping or inlining. For the wrapping version, the original join point calls are replaced by a call to the corresponding connector instance. For the inlining version, all connector instances are actually inlined in the base model. The ***WEAVR*** also allows simulation on the woven model with the perspective of 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 Motorola ***WEAVR***, please refer to [6].

3. COMPOSITIONS IN THE *WEAVR*

Based on the different constructs introduced in Section 2, two kinds of interference problems may occur during the weaving process: connector-to-connector and aspect-to-aspect. (Note: we do not consider pointcut-to-pointcut interference at the time being because pointcuts do not own any behavior actions. Analysis on the pointcut-to-pointcut interference is considered as part of the future work.) This section will introduce precedence declarations on connectors and aspects. The explicitly specified precedence constraints can help reduce undesired interference at the same join point and will be passed to the underlying composition mechanism in order to compute a proper weaving order.

Reusability in AOM can be defined as the ability to reuse pointcuts, connectors 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, connector composition and aspect composition. In the following sub-sections, each mechanism will be illustrated in detail and compared with the corresponding AspectJ notation.[1]

3.1 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. The supported Boolean operators are: AND (`&&`), OR (`||`) and NOT (`!`). Furthermore, the ***WEAVR*** also supports `cflow`, `cflowbelow` and `within` pointcut designators. As shown in Figure 2, `Pointcut1` is constructed by the three sub-pointcuts (`Pointcut1a`, `Pointcut1b` and `Pointcut1c`), which means that `Pointcut1` will pick out join points matched by `Pointcut1a` that are not in the control flow of any join point picked out by `Pointcut1b`, or it will pick out all join points matched by `Pointcut1c`.

[1] We compare our approach with AspectJ because it is one of the most mature and complete AO systems described in the literature. Although AspectJ is focused at the implementation level and our AOM approach is at the modeling level, the interference and composition concepts are universal in every AOSD context.

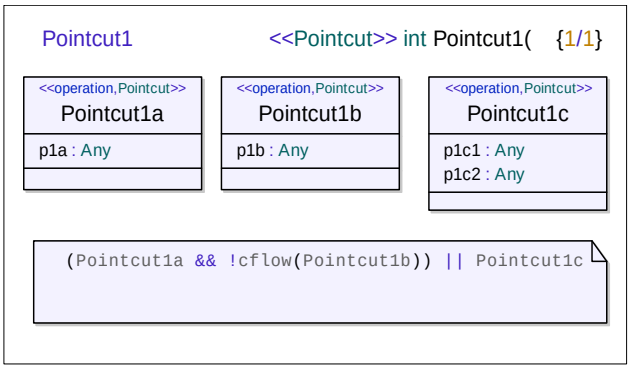

Figure 2. Pointcut composition.

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.

3.2 Connector Composition

Connector composition intends to bind and execute the connector instances that perform at the same join point in a certain appropriate order. In the ***WEAVR***, connectors are ordered based on the precedence relationships that are specified by the users. Currently, we have implemented the `<<follows>>` relationship between connectors. As shown in Figure 3, `Connector2 follows Connector1`, which means that at a particular join point, `Connector1` has precedence over `connector2`, and the instances of `Connector2` will be executed *closer* to the join point than the instances of `Connector1` (i.e., the before actions in `Connector1` instances will always be executed prior to the before actions in `Connector2` 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 connector instances is undefined and controlled by the underlying ***WEAVR***.

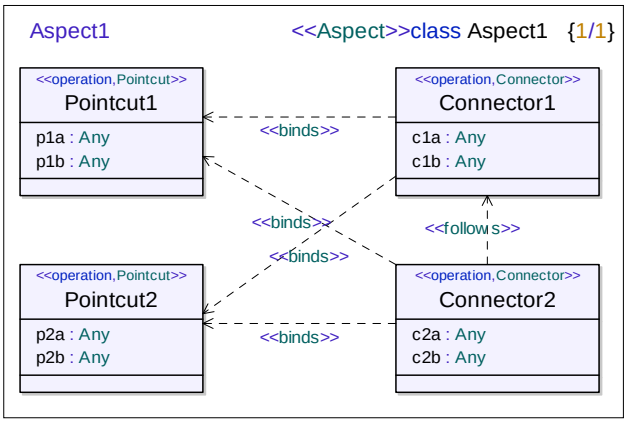

Figure 3. Connector composition.

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

When executing a connector instance, the call to `proceed` will be redirected to invocation of the connector instance with the next precedence or the computation under the join point if there is no further connector instance. In the case of Figure 3, suppose `Pointcut1` and `Pointcut2` both match to a single join point (in the following `PC` replaces `Pointcut` and `CONN` replaces `Connector`). The connector instantiation order at this join point could be: [2]

```
PC1-CONN1, PC2-CONN1, PC1-CONN2, PC2-CONN2
```

If both `Connector1` and `Connector2` contain a `proceed` action, the execution order of the woven model at this join point would be as follows:

- Before actions in the connector instance: `PC1-CONN1`
- Before actions in the connector instance: `PC2-CONN1`
- Before actions in the connector instance: `PC1-CONN2`
- Before actions in the connector instance: `PC2-CONN2`
- Original actions at the join point
- After actions in the connector instance: `PC2-CONN2`
- After actions in the connector instance: `PC1-CONN2`
- After actions in the connector instance: `PC2-CONN1`
- After actions in the connector instance: `PC1-CONN1`

Comparing our connector composition mechanism with the advice semantics in AspectJ, we believe that our mechanism offers two advantages:

1. In the ***WEAVR***, the concepts of pointcuts and connectors are loosely decoupled. A connector is named, which allows it to be associated with not just one, but multiple pointcuts as long as they share compatible interfaces. Therefore, a connector 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 be reused. 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 [8].
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 [4], say 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 [11]. Our approach resolves the above problems because there is only one connector type (i.e., around connector) 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 connector precedence explicitly, the interference between the connectors is reduced.

3.3 Aspect Composition

Aspect composition is achieved through a deployment diagram. A deployment diagram (Figure 4) 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>>`. Aspects can also be deployed to other aspects or connectors. In the absence of the `<<crosscuts>>` relationship, aspects will be applied to all the models in the current active project (e.g., `Aspect3` and `Aspect4` in Figure 4). The precedence relationships between aspects can be `<<follows>>`, `<<hidden_by>>` and `<<dependent_on>>`. The remaining of this section will explain these three concepts in detail based on the example provided by Figure 4.

Aspect2 follows Aspect1:

At a single join point, `Aspect1` has higher precedence than `Aspect2`, which means that all of the connectors in `Aspect1` have higher precedence than the ones in `Aspect2`. All of the connector instances instantiated from `Aspect2` will be executed following the ones instantiated from `Aspect1`.

Aspect3 is hidden by Aspect2:

`Aspect3` will be inactivated when both `Aspect2` and `Aspect3` match at the same join point. The relationship between `Aspect2` and `Aspect3` can be described using the following expression:

```
Aspect2 => ¬Aspect3
```

This notation means that the presence of `Aspect2` implies the absence of `Aspect3`. For each pointcut denoted as $\texttt{Pointcut}_{\texttt{Aspect3}}$ in `Aspect3`, the actual corresponding pointcut exposed by this particular deployment strategy is:

$$\texttt{Pointcut}_{\texttt{Aspect3}}\texttt{'=Pointcut}_{\texttt{Aspect3}}\texttt{\&\& ¬Pointcuts}_{\texttt{Aspect2}}$$

Aspect4 is dependent on Aspect3:

`Aspect4` will only be applied at the join points when both `Aspect3` and `Aspect4` match. `Aspect4` will be disabled at the other join points that it matches apart from A3. The relationship between `Aspect3` and `Aspect4` is denoted as:

```
Aspect4 => Aspect3
```

This means that the presence of `Aspect4` implies that `Aspect3` has to be present at the same join point as well, so that

$$\texttt{Pointcut}_{\texttt{Aspect4}}\texttt{'=Pointcut}_{\texttt{Aspect4}}\texttt{ \&\& Pointcut}_{\texttt{Aspect3}}$$

---

[2] One `<<binds>>` relationship corresponds to one connector instantiation. In this example, three other orders are possible because the precedence between `Pointcut1` and `Pointcut2` is undefined and determined by the ***WEAVR***:

```
{PC2-CONN1, PC1-CONN1, PC1-CONN2, PC2-CONN2},
{PC2-CONN1, PC1-CONN1, PC2-CONN2, PC1-CONN2},
{PC1-CONN1, PC2-CONN1, PC2-CONN2, PC1-CONN2}.
```

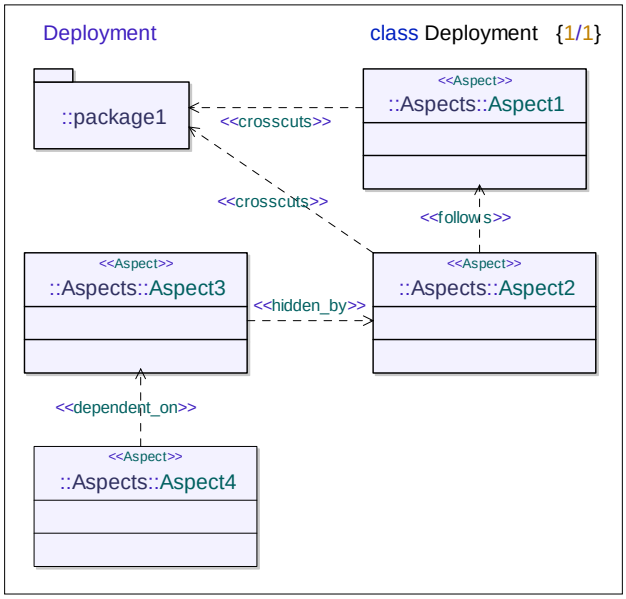

Figure 4. Aspect composition.

In order to detect and collect all of the join points 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 4 is:

```
package1 <- Aspect1,Aspect2,Aspect3,Aspect4
ALL – package1 <- Aspect3,Aspect4
```

Within the scope of `package1`, `Aspect1` will be applied first, followed by `Aspect2`, `Aspect3` and `Aspect4`. Meanwhile, `Aspect3` will be disabled at the presence of `Aspect2` and `Aspect4` will only be applied at the presence of `Aspect3`. For all of the other models that are not within the scope of `package1` (denoted by subtracting `package1` from `ALL` with a minus sign "–"), only `Aspect3` and `Aspect4` will be applied with `Aspect4` depending on `Aspect3`. Circular and conflict relationships among the aspects will be detected and reported when they are superimposed at the same join point.

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 any where. The only way to apply an aspect to a certain scope is to restrict *every* pointcut specification by using the "within" keyword, thus making 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 applying an aspect at the same join point. In AspectJ, however, when a pointcut matches a certain join point, the corresponding aspect is always applied.

4. 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 [1] is particularly dedicated to discussing this issue. An exhaustive classification and documentation of aspect interactions is under investigation by Sanen et al. [16]. 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 [4] controls the aspect ordering by the "declare precedence" statement. As noted by Reddy et al. [15], weaving order is defined by two composition directives, i.e., "follows" and "precedes," at the class design modeling level. Theme/UML [5] resolves the aspect conflicts by indicating precedence order using a "prec" tag. We extend these approaches by introducing two more precedence relationships between aspects, and furthermore allowing precedence to be specified explicitly between connectors.

A number of advanced approaches have been proposed to manipulate aspect interference at the programming level. Kienzle et al. [9] 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. [10] to manage semantics interference between the aspects. Sihman and Katz [17] 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 the combining superimpositions. Nagy et al. [12] proposed ordering, control and structural constraints to address the aspect interference issue. Pawlak et al. [14] 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. [7] defined the semantics of advice in terms of operations on an abstract resource model, in order to reason about semantics conflicts between aspects. The primary benefit our approach offers over these techniques is that aspect interference issue is reduced even before proceeding to the implementation level, which enhances the expressiveness and reusability of the crosscutting concerns.

5. CONCLUSIONS

A key point when dealing with aspects is the notion of aspect interference. The primary contribution 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 reasonable composition order automatically. Furthermore, the three distinct categories of aspect composition mechanisms implemented in the ***WEAVR*** are introduced with the purpose of facilitating aspect reuse to a larger extent than AspectJ.

As this research is still in the preliminary phases, we do have 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 [18] has been studied as an important aspect interference issue. The future work will include investigation on the topic of pointcut interference. We are currently studying 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 model engineers can simulate the model in different perspectives and verify the impacts that are caused by each applied aspect.

6. ACKNOWLEDGEMENTS

This work is partially supported by the National Science Foundation under CSR-SMA-0509342.

7. REFERENCES

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

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

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

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

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

[6] 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.

[7] 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.

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

[9] Jorg 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.

[10] 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.

[11] 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.

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

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

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

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