

# Model Composition Directives

Greg Straw, Geri Georg, Eunjee Song, Sudipto Ghosh, Robert France, and  
James M. Bieman

Department of Computer Science  
Colorado State University, Fort Collins, CO, 80523  
{straw, georg, song, ghosh, france,  
bieman}@cs.colostate.edu

**Abstract.** An aspect-oriented design model consists of a set of aspect models and a primary model. Each of these models consists of a number of different kinds of UML diagrams. The models must be composed to identify conflicts and analyze the system as a whole. We have developed a systematic approach for composing class diagrams in which a default composition procedure based on name matching can be customized by user-defined composition directives. This paper describes a set of composition directives that constrain how class diagrams are composed.

## 1 Introduction

Solutions to design concerns (e.g., security and fault tolerance concerns) may cross-cut many modules of a design model. The cross-cutting nature of these solutions can make understanding, analyzing and changing the solutions difficult. This complexity can be addressed through the use of aspect-oriented modeling (AOM) techniques, where the design of a cross-cutting solution is undertaken in an independent fashion, and the resulting *aspect* models are composed with *primary* models of core functionality to create a complete system design. Composition is necessary to identify conflicts across aspect and primary models, and to identify undesirable emergent properties in composed models.

We have developed an AOM technique in which aspect and primary models are expressed using the UML [12]. Each model consists of a variety of UML diagrams. Composition of aspect and primary models involves composing diagrams of the same types. For example, the class diagram in an aspect model is composed with the class diagram in a primary model. The AOM technique uses a default, name-based composition procedure in which model elements with the same syntactic type and name are merged to form a single element in the composed model. The default procedure assumes that elements of the same syntactic type with the same name represent different and consistent views of the same concept. This may not be the case if the aspect and primary models are developed independently. Often, a more sophisticated form of composition is needed to produce composed models with desired features. Composition directives can be used to modify the default composition procedure [6]. In this paper we rigorously define and significantly extend a set of composition directives informally described in our previous work [6], and show how the composition direc-

tives can be used to alter the basic default composition procedure. We also give examples of how composition directives can be used to resolve conflicts in composed models produced by the default composition procedure.

## 2 Composition in AOM

An aspect-oriented design model consists of a primary model and aspect models [6]. A primary model consists of one or more UML diagrams that each describes a view of the core functionality. The core functionality determines the primary structure of a design. In our AOM approach an aspect model describes a family of solutions for a design concern that each cross-cuts the primary model. Aspect models consist of parameterized UML artifacts that describe generic solutions to design concerns [7,11]. An aspect model cannot be directly composed with a primary model. A *context-specific aspect model* must first be created by binding the aspect model's template parameters to application-specific values. The context-specific aspect can then be composed with the primary model.

Conflicts across aspect and primary model views and undesirable emergent properties can be identified during composition or during analysis of the composed model. Composition directives can be used to resolve conflicts or remove undesirable emergent properties during composition. For example, a composition directive can (1) indicate that properties in aspect models override conflicting properties in primary models (or vice versa), (2) specify that particular primary model elements must be removed or added during composition, and (3) determine the order in which two or more aspects are composed with a primary model.

Figure 1 illustrates a simple composition example. Figure 1(a) shows an aspect model consisting of a single class diagram template. The aspect model describes a family of solutions (from a structural perspective) in which entities that produce outputs (buffer writers) are decoupled from output devices through the use of buffers. Template parameters are preceded by the symbol “[”. Figure 1(b) shows a context specific aspect model created from the aspect model in Figure 1(a). The context specific aspect model is obtained using the following name bindings:

```
( |Buffer<-Buffer), ( |Output<-FileStream),  
  ( |BufferWriter<-Writer), ( |write()<-writeLine())
```

The result of composing the context-specific aspect class diagram shown in Figure 1(b) with the primary model class diagram shown in Figure 1(c) is shown in Figure 1(d). In the primary model the output producer sends outputs directly to the output device. In the composed model a buffer is introduced between the output producer and the output device. Composition of the context-specific aspect model and the primary model is carried out using a default name-based composition procedure. The procedure merges model elements that have the same name and syntactic type to produce a single model element in the composed model. If the matching model elements are associated with invariants (e.g., expressed in the OCL) the invariant associated with the merged element in the composed model is formed by taking the logical ‘AND’ of the invariants. Operation specifications, expressed as OCL pre and post-

conditions, can also be merged for matching operations. The precondition of the merged operation in the composed model is formed by taking the logical “OR” of the preconditions associated with the matching operations, and the postcondition is formed by taking the logical “AND” of their postconditions. Composition using the simple default procedure can produce undesirable results. In Figure 1(d) an association between *Writer* and *FileStream* exists in the composed model, but the intent is that the writer be completely decoupled from the output device, and thus the association should be removed. A composition directive can be used to alter the composition so that it includes removal of this association.

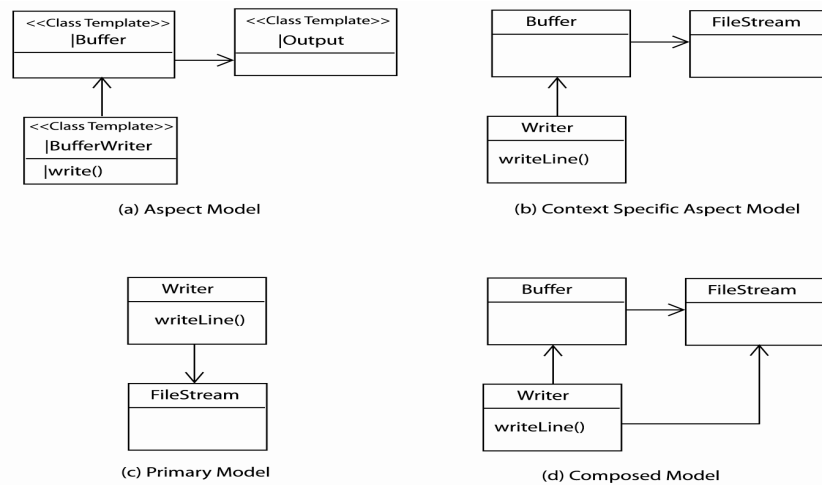


Fig. 1. Default Composition Example

### 3 Requirements for Composition Directives

In this section we motivate the need for composition directives and identify some of the directives that can be used to resolve conflicts. We restrict our attention to models that consist only of class diagrams.

Consider operations  $addUser(u: User, mId: MgrID)$ ,  $doAddUser(u: User)$  in a class *Repository* that is part of a context specific aspect model, and an operation  $addUser(u: User)$  defined in a primary model class named *Repository*. The  $addUser$  operation in the primary model adds a user (instance of *User*) to a collection of users (instance of a class *Users*). The  $addUser$  operation in the context specific aspect model calls the  $doAddUser$  operation if and only if the client calling the operation is authorized to add a user. The  $doAddUser$  operation adds a user to the collection. Composition of the two matching  $addUser$  operations produces a conflict because the two operations have different specifications. This is an example of a *property conflict* – a property conflict occurs when two matching elements (elements with the same name and syntactic type) are associated with conflicting properties (in this case pre and postconditions). In this example, the intention is to merge the  $doAddUser(u:$

*User*) operation in the context specific aspect model with the *addUser(u: User)* operation in the primary model. To resolve this conflict, composition directives should rename the *addUser* operation in the (context specific) aspect model to *checkAndAddUser*, and rename the *doAddUser* operation in the aspect model to *addUser*.

In some cases, renaming elements may not be the appropriate way to resolve a property conflict. Consider a context specific aspect model that includes a class *FileStream* with an attribute *maxWriters: int* that is associated with the constraint  $\{maxWriters = 1\}$ . Now consider a primary model with a class named *FileStream* that contains an attribute *maxWriters: int*, with the constraint  $\{maxWriters = 2\}$ . If the matching attributes are merged, a property conflict will arise because the merged constraint ( $\{maxWriters = 1 \text{ and } maxWriters = 2\}$ ) is inconsistent. This conflict can be resolved by specifying, through a composition directive, that one operation overrides the other, such that properties from the overriding element take precedence over those in the element being overridden. However, override relations can produce a *cyclic-override* conflict when a cycle exists between two elements such that there are override relations specifying both as dominant elements.

In some cases, elements may need to be added or deleted during composition in order to produce a composed model that has desired properties. For example, associations may be added to provide correct access to other elements, or may be removed if they pose a security risk. Composition directives can be used to add or delete model elements during composition.

With the ability of renaming, adding, and removing elements comes the risk of yet another type of conflict: the *nonexistent-reference conflict*. A nonexistent-reference conflict arises when a reference in one of the models refers to an element that no longer exists, or exists under a different name. To resolve this conflict, the reference elements in a model must be updated.

In a system with multiple aspects, the order in which aspect models are composed with a primary model may be important in the cases where different orderings produce different composed models [6]. Composition directives can be used to specify the order in which multiple aspects are composed with a primary model.

With the addition of ordering relationships, an additional type of conflict becomes possible. A *cyclic-ordering conflict* occurs when there is a cycle among ordering relationships defined over multiple aspects. These conflicts can be resolved by analyzing the aspects to correct the order relationships.

The above examples give rise to the following set of actions that can be specified by composition directives:

- Creating new elements.
- Adding elements to a Namespace.
- Deleting elements from a Namespace.
- Renaming elements.
- Changing references to an element.
- Specifying override relationships between matching elements.
- Specifying ordering relationships among multiple aspects.

## 4 Composition Directives

In this section, short descriptions of the composition directives are followed by illustrated examples. For more detailed specifications of the directives, refer to our technical report [13]. Many of these descriptions refer to *ModelElements*, *Names*, and *Namespaces* from the UML meta-model [12]. References to *Aspect* in these descriptions refer to a context specific aspect model. Two types of composition directives are used in our AOM approach: Low-level composition directives are used to customize the composition of a single context specific aspect model and a primary model, and high-level composition directives are applied to two or more aspect models and are used primarily to specify the order in which aspect models are composed with a primary model. The set of composition directives defined in this paper is not intended to be a complete set, but serves as a starting point for the eventual definition of a complete set of composition directives.

### 4.1 Low-level Composition Directives

**Creating New ModelElements.** The directives for creating new ModelElements are collectively referred to as *constructors*. Each constructor will have a different set of operands, but each will consist of the necessary properties to define each (see Example 2 in section 4.2). The use of a constructor results in the creation of a reference to a new ModelElement. The constructors are used as follows:

```
newHandle = create<ModelElement.name> { parameters ...}
```

**Adding ModelElements to a Namespace.** Once a new ModelElement is created, it is not yet a member of a Namespace. The directive for adding ModelElements to a Namespace is `add`. The `add` directive has two operands: (1) The ModelElement to be added, and (2) the Namespace the ModelElement is being added to. The `add` directive is used as follows:

```
add addition :ModelElement to owner :Namespace
```

**Removing ModelElements.** The directive for removing ModelElements from a Namespace is `remove`. It has two operands: (1) The ModelElement to be removed, and (2) the Namespace to remove the ModelElement from. The `remove` directive is used as follows:

```
remove member :ModelElement from owner :Namespace
```

**Renaming ModelElements.** The directive for renaming ModelElements is `rename`. The `rename` directive has two operands: (1) The ModelElement to rename, and (2) the new Name. The `rename` directive is used as follows:

```
rename target :ModelElement to newName :Name
```

**Replacing References to a ModelElement.** Removing a ModelElement may lead to invalid references that refer to a non-existent ModelElement. The `replaceReferences` directive can change these references to a different ModelElement. This directive has three operands: (1) The original Name of the ModelElement the references refer to, (2) the replacement Name for the references, and (3) the Namespace containing the references. The third operand defines a scope for the replacement of references. The `replaceReferences` directive is used as follows:

```
replaceReferences originalName :Name  
  with replacementName :Name in owner :Namespace
```

**Overriding ModelElements.** An override relationship specifies that one ModelElement's properties take precedence over properties of another ModelElement. The `overrides` directive has two operands: (1) The ModelElement that will take precedence, and (2) the ModelElement that will be overridden. When an override relation is defined for two ModelElements, the relationship is honored for all contained ModelElements. This directive is used as follows:

```
superior :ModelElement overrides inferior :ModelElement
```

## 4.2 Composition Examples

Illustrated examples show the use of the composition directives for composing a single primary and context specific aspect. Each aspect model may be woven into multiple areas of a primary model. For simplicity, these examples only show only one portion of the primary model, which represents one portion of the design for which the aspect is to be woven.

**Example 1.** Consider the example in Figure 2. In the context specific aspect model, the `UserMgmt` class contains a operation called `getRepositorySize()` that retrieves the size of `SystemMgmtAuthRepository`. Note that this operation has been created from the aspect model with a name that will cause a property conflict. The conflict is with the operation of the same name in `UserMgmt` in the primary model. The operation `primary::UserMgmt::getRepositorySize()` returns the size of `UserRepository`, which is a different operation. To resolve this conflict, the `rename` directive can rename one or both operations, and the `replaceReferences` directive can update any references to the old Name. The following composition directives are applied:

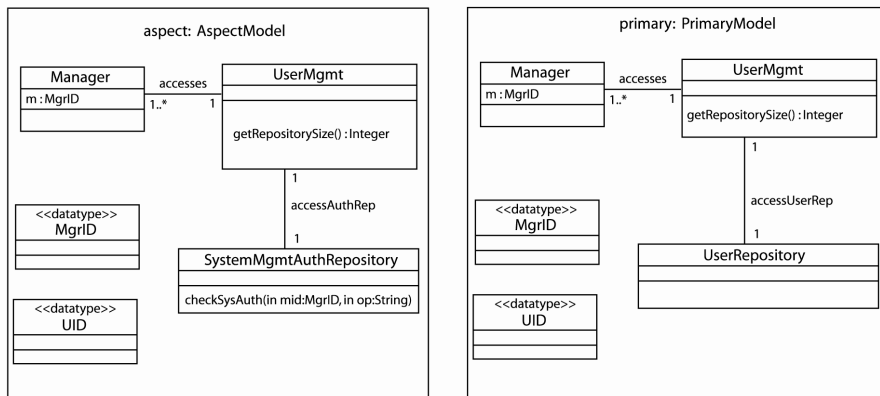
```
(1) rename aspect::UserMgmt::getRepositorySize()  
    to aspect::UserMgmt::getAuthRepositorySize()  
  
(2) replaceReferences  
    aspect::UserMgmt::getRepositorySize()  
    with aspect::UserMgmt::getAuthRepositorySize()  
    in aspect
```

```

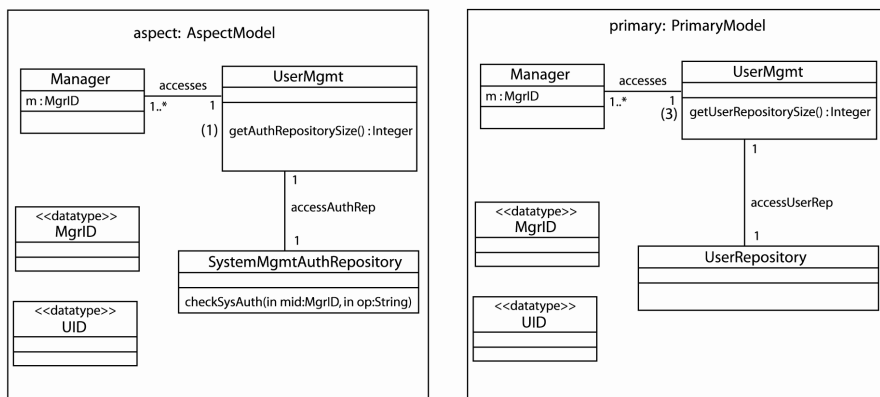
(3) rename primary::UserMgmt::getRepositorySize()
    to primary::UserMgmt::getUserRepositorySize()

(4) replaceReferences
    primary::UserMgmt::getRepositorySize()
    with primary::UserMgmt::getUserRepositorySize()
    in primary

```



**Fig. 2.** Example 1: Before Application



**Fig. 3.** Example 1: After Application. (1) and (3) note the name changes.

The result of applying the directives is shown in Figure 3. Where applicable, the effects of the composition directives are denoted in the composed model using the corresponding numbers. The names of *getRepositorySize()* in *aspect* and *primary* are changed to *getAuthRepositorySize()* and *getUserRepositorySize()*, respec-

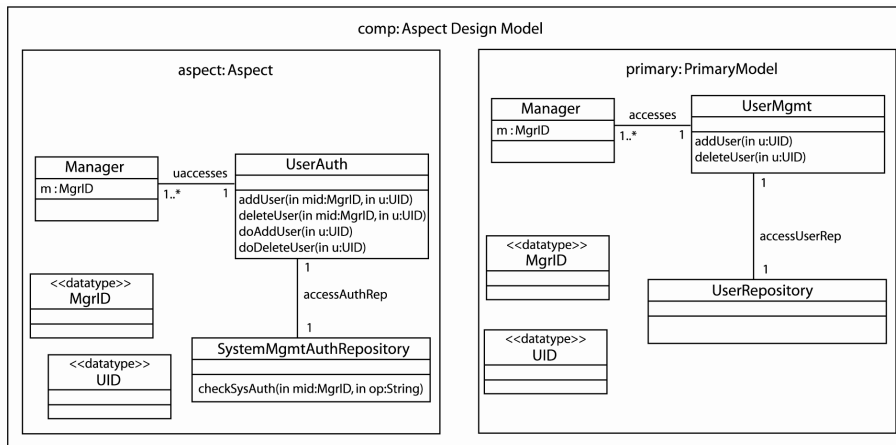
tively. The references to the operation names are changed throughout each model to reflect the name change, and to avoid reference conflicts.

**Example 2.** The following example, from France *et al.*[6], illustrates the use of the `create`, `add`, `remove` and `replaceReferences` directives. In Figure 4, the `UserAuth` class performs authorization checks for `Managers` requesting the addition or deletion of users from the system. In the primary model, `Manager` has a direct association with `UserMgmt`, which provides the `addUser` and `deleteUser` services. In the composed model, `Manager` needs to make these requests to `UserAuth` and should have no direct access to the `UserMgmt` class. The first step to specifying this composition is to recognize the `accesses` association as a prohibited element to be removed:

```
(1) remove primary::Manager::accesses
    from primary::Manager
```

This does not result in a well-formed primary model however, since there may be references to the `accesses` association in `Manager`. References to `accesses` in the primary model must be changed to `uaccesses` in the context specific aspect model, since it is the association intended for making the `addUser` and `deleteUser` requests, and replaces `accesses`:

```
(2) replaceReferences primary::Manager::accesses
    with aspect::Manager::uaccesses
    in primary::Manager
```



**Fig. 4.** Example 2: Before Application

The definitions of the `addUser` and `deleteUser` operations in `UserAuth` include an authorization check for a given `MgrID`, and if the `Manager` is authorized its request, a call is made to the appropriate operation `doAddUser` or `doDeleteUser`. The `doAddUser` and `doDeleteUser` operations are intended to request the add and delete services, however there is no connection to the `UserMgmt` class that provides these ser-

vices. The first step to solving this problem is to create an association between *UserAuth* and *UserMgmt*:

```
(3) userAuthEnd = createAssociationEnd {
    isNavigable = true,
    aggregation = aggregate,
    participant = aspect::UserAuth,
    multiplicity = 1 },

userMgmtEnd = createAssociationEnd {
    isNavigable = true,
    aggregation = none,
    participant = primary::UserMgmt,
    multiplicity = 1 },

userAuth-userMgmt = createAssociation {
    name = "UserAuth-UserMgmt",
    connection = [userAuthEnd,
    userMgmtEnd] }
```

Once the new Association is created, we need to add it to the appropriate Namespace, which in this case is the composable aspect design model (i.e., a single primary model and a context specific aspect) since the association spans both the primary and context specific aspect models. The new AssociationEnds must be added to their respective participants as well:

```
(4) add userAuth-userMgmt to comp,
    add userAuthEnd to aspect::UserAuth,
    add userMgmtEnd to primary::UserMgmt
```

There are two options for specifying the correct operation calls: The first option is to define *doAddUser* and *doDeleteUser* to delegate to *UserMgmt* via the new association. The second option is more concise, and simply replaces the call of *doAddUser* and *doDeleteUser* to the appropriate operations in *UserMgmt*, and deletes *doAddUser* and *doDeleteUser*. This is the option we will use, and results in the following composition directive:

```
(5) replaceReferences aspect::UserAuth::doAddUser
    with primary::UserMgmt::addUser()
    in aspect,

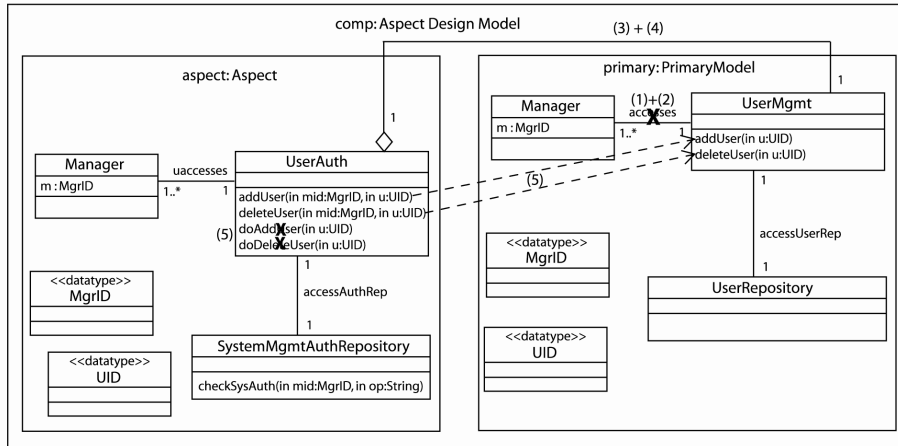
remove aspect::UserAuth::doAddUser
from aspect::UserAuth,

replaceReferences aspect::UserAuth::doDeleteUser
with primary::UserMgmt::deleteUser()
in aspect,

remove aspect::UserAuth::doDeleteUser
from aspect::UserAuth
```

The result of the composition is shown in Figure 5. The **X**'s mark the ModelElements removed by the composition directives. The dependencies from the *addUser*

and *deleteUser* operations in *UserAuth* illustrate the calls to the respective operations in *UserMgmt*.



**Fig. 5.** Example 2: After Application

**Example 3.** Figure 6 illustrates the need for defining an override relationship. The primary model shows a simple system for writing to a *FileStream*. This system only supports one *Writer*, as there is no concurrency control.

The following invariants are defined for *maxWriters* in the context specific aspect model and primary models:

```
context primary::FileStream::maxWriters
  inv: maxWriters = 1

context aspect::FileStream::maxWriters
  inv: maxWriters = 2
```

The *FileStream* in the context specific aspect model supports up to two *Writers*, while in the primary model, it only supports one. The default composition behavior is to take the logical 'AND' of constraints over matching properties, but in this case that behavior would not be appropriate. In the composed model, the intended result is the support of up to two *Writers*, so the following override relationship is defined:

```
(1) aspect::FileStream::maxWriters
  overrides primary::FileStream::maxWriters
```

Another override relationship is needed. The definition of *write()* in *primary::FileStream* simply writes the *Line* to the *FileStream* without any checks. The definition for *write()* in *aspect::FileStream* supports multiple *Writers*, and thus obtains the semaphore through calls *wait()* and *signal()* in the *Semaphore* class. This is the desired behavior in the composed model, so the following override relationship is defined:

```
(2) aspect::FileStream::write()
    overrides primary::FileStream::write()
```

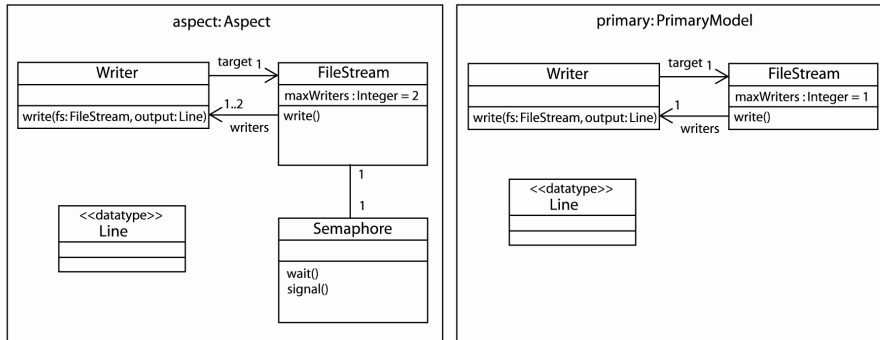


Fig. 6. Example 3: Before Application

All properties of *primary::FileStream* are overridden by their respective properties in *aspect::FileStream*. This same behavior can be achieved using the following composition directive, since any declared override relationship is honored for contained, matching ModelElements:

```
(3) aspect::FileStream overrides primary::FileStream
```

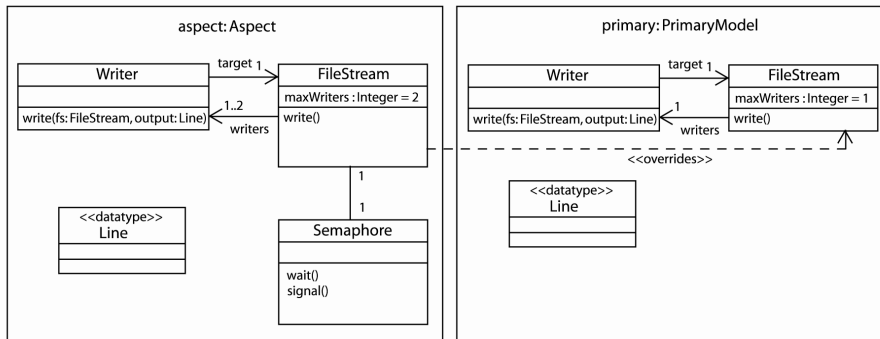


Fig. 7. Example 3: After Application

Figure 7 shows the result of applying the composition directive (3). The dependency from *aspect::FileStream* to *primary::FileStream* illustrates the created override relationship between the two classes. When composition is performed, the definitions and constraints for *write()* and *maxWriters* in the context specific aspect model are used rather than those for the respective properties in the primary model.

### 4.3 High-level Composition Directives

In a system with multiple aspects, the order in which aspect models are composed with a primary model is important: Different ordering can result in different composed models [6]. The weave order for an aspect design model containing multiple aspects can be defined using weave-order relationships that specify that one aspect is to be woven before another. Any ordering of multiple aspects can be achieved using binary relations [13], which allows a developer to specify the important relationships in the weave order. A weave ordering relationship can be created using either the `follows` directive or the `precedes` directive. The `precedes` directive has two operands: (1) the aspect to be woven first, and (2) the aspect to be woven second.

```
former :Aspect precedes latter :Aspect
```

Conversely, the same relationship can be created using the `follows` directive, both directives are only provided for convenience.

```
later :Aspect follows earlier :Aspect
```

### 4.4 Weave-Ordering Example

**Example 4.** Consider the following aspect design model in Figure 8(a). There are three different aspects and the primary model. Without any direction, the aspects will be woven in an arbitrary order. In this example, the aspect *authentication* needs to be woven before the aspect *authorization*, since authorization without authentication is meaningless. Therefore, we declare the following composition directive to make the order explicit.

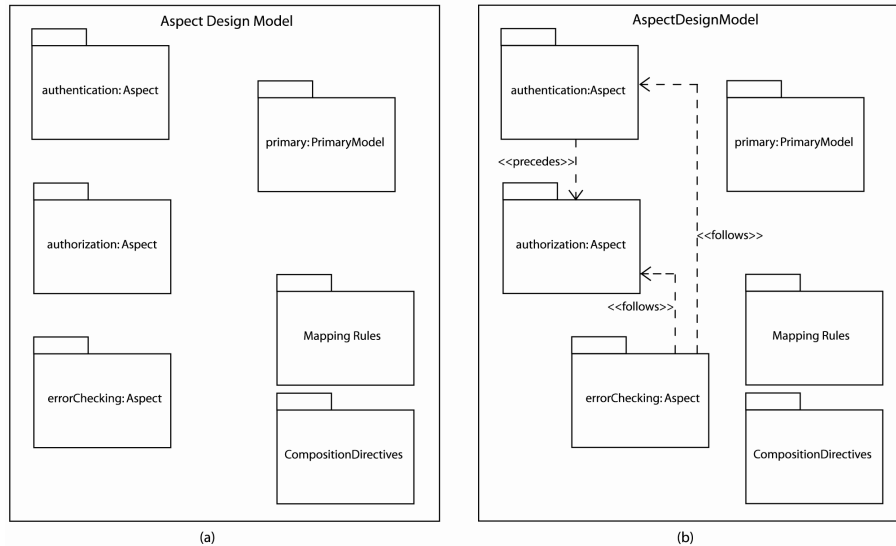
```
(1) authentication precedes authorization
```

We could have also defined a composition directive using the `follows` directive with the operands reversed to achieve the same result. Suppose we also wish to weave the *errorChecking* aspect last. The following composition directives achieve this:

```
(2) errorChecking follows authorization
```

```
(3) errorChecking follows authentication
```

The result is shown in Figure 8(b). The dependency from *authentication* to *authorization* illustrates the weave-order relationship that specifies that *authentication* must be woven before *authorization*, and the dependencies from *errorChecking* to each of the other aspects illustrates the two binary weave-order relationships that specify *errorChecking* as the last aspect to be woven.



**Fig. 8.** Example 4: Specifying Weave Order

## 5 Related Work

Clarke *et al.* describe an approach similar to AOM which is based on *subjects* [3,4,5], where a subject is a particular view of the comprehensive system. There is no primary design; instead everything is a subject and the overall system design is obtained through the composition of all subjects. The compositions of subjects include the addition or overriding of named elements in a model. One limitation of this approach is that there is no support for the merging of constraints associated with a model. There is also no support for the deletion of elements, except when an element is implicitly deleted as a result of being overridden. The operation supports conflict reconciliation through precedence and override relationships between conflicting elements, but nothing further. We describe directives that support the composition of constraints, and the deletion of model elements.

Brito and Moreira describe an aspect composition process that identifies match points in a design element and defines composition rules [2]. Rules use identified match points, a binary contribution value (either positive or negative) that quantifies the affects on other aspects, and a priority for a given aspect. In the context of AOP [10], Kienzle *et al.* describe composition rules based on dependencies between aspects [9]. Both papers [2,9] focus primarily on relationships that can exist between aspects. We describe the possible relationships between aspects as weave-order relationships and override relationships instead of priority and dependency as done by Brito and Moreira. This paper expands further on composition directives that are meant for varying the default composition behavior.

## 6 Conclusions and Future Work

This paper defines a set of composition directives that facilitate the customization of model composition. These directives can form the basis for the development of tools to support the AOM approach described in France *et al.* [6]. The directives also provide a common vocabulary for describing composition actions. Illustrated examples demonstrate the use of each directive.

The defined directives are *expressive* [1] in the sense that they possess the following two properties. First, the directives can specify common composition actions such as renaming and replacing classes and operations. Second, the directives can be used to specify creation and removal of model elements, making it possible to significantly alter how models are composed.

Empirical evaluation is needed to validate the AOM approach in real world design settings. Specifically (1) the amount of effort required to specify the kinds of compositions that are required in real world designs needs to be empirically evaluated; (2) the development of a tractable method of identifying conflicts in a composed model needs to be investigated; and (3) the currently defined composition directives need to be tried in a real design setting, and evaluated for their ability to support the kinds of composition actions that actually occur. This evaluation could result in the specification of some common composition strategies [8] to manage the complexity of specifying compositions and is an area of future work.

We are also exploring how to express the applicability and consequences of using composition directives in terms of pre and postconditions for directives. We plan to investigate the use of the Object Constraint Language [12] for this purpose.

## Acknowledgements

This material is based in part on work supported by the U.S. National Science Foundation under grants CCR-0098202 and CCR-0203285, and by the AFOSR under grant FA9550-04-1-0102. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or the AFOSR.

## References

1. R. Allen, and D. Garlan. 1997. A Formal Basis for Architectural Connection. *ACM Trans on Software Engineering and Methodology*. vol 6, no 3, pp.213-249, July 1997
2. I. Brito, and A. Moreira, Towards a Composition Process for Aspect-Oriented Requirements. In *Proceedings of the Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design*. Boston, MA, March 2003.
3. S. Clarke and J. Murphy. Developing a tool to support the application of aspect-oriented programming principles to the design phase. In *Proceedings of the International Conference on Software Engineering (ICSE '98)*, Kyoto, Japan, April 1998.

4. S. Clarke, W. Harrison, H. Ossher, and P. Tarr. Separating concerns throughout the development lifecycle. In *Proceedings of the 3rd ECOOP Aspect-Oriented Programming Workshop*, Lisbon, Portugal, June 1999.
5. S. Clarke. Extending standard UML with model composition semantics. *Science of Computer Programming*, Volume 44, Issue 1, pp. 71-100. Elsevier Science, July 2002.
6. R. B. France, I. Ray, G. Georg, and S. Ghosh. An Aspect-Oriented Approach to Design Modeling. *IEEE Proceedings - Software*, Special Issue on Early Aspects: Aspect Oriented Requirements Engineering and Architecture Design. (To Appear)
7. R. B. France, D. K. Kim, S. Ghosh, and E. Song. A UML-Based Pattern Specification Technique. *IEEE Transactions on Software Engineering*, Volume 30, No 3, March, 2004.
8. G. Georg, R. B. France and I. Ray, Composing Aspect Models. In *Proceedings of the Workshop on Aspect Oriented Modeling with UML*, San Francisco, CA, October 2003.
9. J. Kienzle, Y. Yu, and J Xiong. On Composition and Reuse of Aspects. In *Proceedings of the Foundations of Aspect-Oriented Languages Workshop*, Boston, MA, March 2003.
10. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J-M. Loingteir and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220-242, Jyvaskyla, Finland, June 1997.
11. D. K. Kim, R. France, S. Ghosh. A UML-Based Language for Specifying Domain-Specific Patterns. *Special Issue on Domain Modeling with Visual Languages, Journal of Visual Languages and Computing*, 2004. (To Appear).
12. The Object Management Group (OMG). Unified Modeling Language. OMG, <http://www.omg.org/docs/formal/03-03-01.pdf>. Version 1.5, March 2003.
13. G. Straw, G. Georg, E. Song, S. Ghosh, R. France, J. M. Bieman. Primitives of Composition Directives. *Technical Report CS 04-103, Computer Science Department, Colorado State University, 2004.*