An Aspect-Oriented Approach to Early Design Modeling

Robert France, Indrakshi Ray, Geri Georg, Sudipto Ghosh
Department of Computer Science
Colorado State University, Fort Collins, CO 80523

Abstract

Developers of modern software systems are often required to build software that address security, fault-tolerance, and other dependability concerns. A decision to address a dependability concern in a particular manner can make it difficult or impossible to address other concerns in software. Proper attention to balancing key dependability and other concerns in the early phases of development can help developers better manage product risks through early identification and resolution of conflicts and undesirable emergent behaviors that arise as a result of interactions across behaviors that address different concerns.

In this paper we describe an aspect-oriented modeling (AOM) approach that eases the task of exploring alternative ways of addressing concerns during software modeling. The paper focuses on use of the AOM approach to produce logical, aspect-oriented architecture models (AAMs) that describe how concerns are addressed in technology-independent modeling terms. An AAM consists of a set of aspect models and a base architecture model called the primary model. An aspect model describes how a dependability concern is addressed, and a primary model describes how other concerns are addressed. Composition of the aspect and primary models in an AAM produces an integrated view of the logical architecture described by the AAM. Composition can reveal conflicts and undesirable emergent properties. Resolving these problems can involve developing and analyzing alternative ways of addressing concerns. Localizing the parts of an architecture that address pervasive and non-orthogonal dependability concerns in aspect models allows developers to more easily evolve and replace the parts as they explore alternative ways of balancing concerns in the early stages of development.

1 Introduction

The pervasiveness of computer systems highlights the need to engineer software that deliver services in a dependable manner. Designs of dependable software must address multiple, possibly interdependent dependability concerns such as access control, confi dentiality, and data integrity. The manner in which a dependability concern is addressed can affect how other concerns are addressed. Balancing concerns during software development can involve developing and analyzing alternative ways of addressing the concerns. Lack of attention to balancing dependability and

other concerns in the early software development phases can lead to major rearchitecting of the design in later stages of development.

In this paper, a concern is a problem coupled with a desired goal [20, 21], where the goal determines acceptable solutions to the problem. For example, the problem of prohibiting unauthorized access to protected resources in a banking system is a dependability concern that must be addressed by banking software that manipulates the protected resources. A model that describes how a concern is addressed is called a *concern solution model*. In particular, a model that describes how a dependability concern is addressed is called a *dependability solution model*. For example, a Role Based Access Control (RBAC) model [34] can be used to describe a solution to the banking system's access control concern. A decision to address a concern in a particular manner can give rise to other concerns. For example, the RBAC solution to the access control problem gives rise to new concerns pertaining to the management of roles and permissions.

This paper focuses on addressing dependability concerns during logical architecture modeling of software. The concern solution models are expressed in high-level, technology-independent modeling terms. Current software development techniques allow developers to structure logical architectures in terms of modules that can be composite classes (i.e., classes that have an internal class structure), subsystems or interfaces. These modules typically localize solutions that address key functional concerns. Addressing non-orthogonal dependability concerns results in dependability solutions that are spread across the modules of the architecture and tangled with functionality described in the modules. These solutions are said to *crosscut* the primary structure of the architecture model.

Balancing concerns that are addressed by crosscutting solutions in the early phases of development can be challenging, primarily because of the difficulty of consistently changing or replacing the crosscutting solutions in an architecture model. A modeling approach that supports localizing

the descriptions of crosscutting dependability solutions can significantly ease the task of evolving and replacing the solution descriptions in an architecture model. In this paper we describe an aspect-oriented modeling (AOM) approach that allows developers to conceptualize, describe and communicate logical dependability solution in isolation. The dependability solution models are called aspect models. An aspect-oriented architecture model (AAM) produced by the AOM approach consists of a set of aspect models and a base architecture model called the primary model. The primary model describes concern solutions that determine the base structure of the architecture model. Each aspect model describes a dependability solution that crosscuts the primary model. An integrated view of the architecture is obtained by composing aspect and primary models to produce a composed AAM. Conflicts and undesirable emergent properties can be identified during composition of aspect and primary models and during analysis of the composed AAM. Addressing these deficiencies can lead to consideration of alternative ways of addressing concerns. Use of the AOM approach in the early stages of software development can help reduce software product risks through early identification and resolution of conflicts and undesirable behaviors that emerge as a result of integrating concern solutions.

The remainder of this paper is organized as follows. In Section 2 we discuss the major concepts underlying our AOM approach and give an overview of the approach. In Section 3 we describe a technique for representing aspect models, and in Section 4 we describe how aspect models can be composed with primary models. In Section 5 we identify some limitations of the approach and discuss issues that are not yet fully addressed in the approach. We give an overview of related work in Section 6 and we conclude in Section 7 with an outline of our plans to further develop the AOM approach.

2 Aspect-Oriented Modeling

In the aspect-oriented programming (AOP) language AspectJ, an aspect is a type that crosscuts a program structure [23]. An aspect contains information typically found in a class (i.e., data members and methods) in addition to behavior that is executed at specified point in a program's execution. The well-defined points are called join points and the specifications of join points are called pointcuts. In the modeling community there has been some work on describing aspect-oriented programs using modeling languages such as the Unified Modeling Language (UML) [38]. AOM, as described in this paper, is not concerned with describing aspect-oriented programs. Rather, the AOM approach described in this paper provides support for modeling of concern solutions in isolation and for integrating the concern solution models with models describing the primary structure of software.

Modeling languages such as the UML provide some support for multidimensional separation of concerns through the use of different diagram types that can be used to describe non-orthogonal views of a system. AOM approaches allow developers to define additional dimensions of separation based on system-specific concerns. In an AOM approach, aspects localize concern solutions that crosscut views described by different diagrams in a system model.

The separation of crosscutting elements is a characteristic that is common to AOP and AOM, but differences between the artifacts (models versus code) can give rise to differences in techniques. For example, at the code level there is a single representation of functionality (the source code), while a model can describe a system from multiple views using different diagrams. The views can be non-orthogonal, for example, a UML sequence diagram that describes how a set of class instances interact to accomplish a task crosscuts the class diagram view of a system. In the AOM approach described in this paper, aspects describe solutions that crosscut UML model

views.

Another difference between AOM and AOP is that code level aspect weaving is concerned primarily with inserting functionality at well-defined points in a program's execution. The points at which functionality can be inserted are determined by the join point model of the AOP language. Software models are typically static descriptions of structure and behavior. In the cases where the semantics of a modeling language supports execution of models one can conceivably create a join point model for the modeling language to support an AOP-like notion of weaving. In the absence of such semantics, weaving at the model level is essentially static composition of model views.

2.1 Supporting Aspect-Oriented Modeling

The AOM approach described in this paper provides support for (1) describing crosscutting concern solutions as modeling views called aspects, (2) synthesizing an integrated model by composing aspect and primary model views, and (3) identifying and resolving conflicts and undesirable emergent properties that arise as a result of integrating aspect and primary models.

Two broad types of concerns can be identified [21]: A *concrete* concern can be directly realized in a model (i.e., there are model elements that specifically address the concern), and a *qualitative* concern is based on qualities or attributes of a system. Access control and error recovery are examples of concrete concerns, while concerns pertaining to system performance and memory utilization are examples of qualitative concerns. The AOM approach described in this paper is applicable to concrete concerns only. Henceforth, a concrete concern is referred to simply as a concern.

Aspect models in our AOM approach describe crosscutting dependability solutions in logical (i.e., high-level and technology-independent) terms. A crosscutting concern solution can be isolated if its distributed elements have common structural and behavioral characteristics. A gen-

eralized form of the solution can then be represented as a pattern, where the pattern describes common characteristics of the distributed solution parts. A pattern view of crosscutting solutions screens out context-specific details and makes it possible to conceive, describe, and understand the solutions in isolation. In our AOM approach an aspect model is a pattern that characterizes a family of logical concern solutions. The patterns are described using UML model templates, as is also done in the Theme approach [5]. The template notation used in our work is an adaptation of a UML-based pattern language, called the Role-Based Metamodeling Language (RBML) [12]. Composing an aspect model with a primary model requires that one first instantiate the pattern by binding template parameters to application-specific values. An instantiated aspect model is called a *context-specific* aspect model. This approach paves the way for the development and systematic use of design patterns that capture logical solutions to dependability concerns.

Model composition technologies that automate significant parts of the AOM composition activity are needed if AOM is to scale-up to models of complex "real-world" software systems. At one extreme are composition tools that take in aspect and primary models and produce composed models without further input from developers. This fixed composition approach provides very little flexibility in how aspect models are composed with primary models. At the other extreme, developers also provide composition procedures that detail how the aspect models are to be composed with primary models. This approach is very flexible, but requires more effort from developers. More practical solutions are likely to lie between these two approaches. For example, a tool can codify a default composition procedure and allow developers to vary some aspects of the procedure using composition directives. This is the approach taken in our work.

Context-specific aspect, primary, and composed models are analyzed to uncover flaws. Analysis of the composed model can reveal conflicts and undesirable emergent properties. Analysis can also be carried out to determine the extent that dependability solutions meet their objectives when

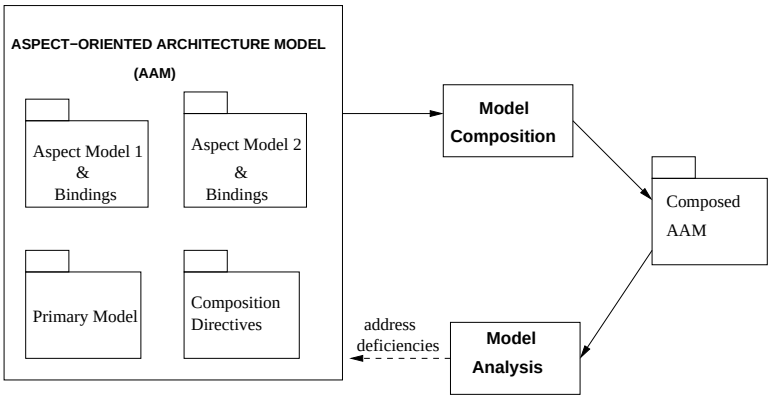


Figure 1: Components of the AOM Approach

integrated with other concerns.

2.2 An Overview of the AOM Approach

The major components of the AOM approach are shown in Fig. 1. An AAM of an application consists of (1) a primary model, (2) aspect models and the bindings used to instantiate them in the application context, and (3) composition directives that determine how the instantiated aspect models are composed with the primary model to produce a composed AAM.

A primary model consists of UML diagrams that each describes a view of the base architecture. The primary models in this paper consist of two types of diagrams: UML classifier and interaction diagrams. Aspect models describe patterns of logical dependability solutions as UML diagram templates. An AAM presents logical views of a software architecture.

Fig. 2 illustrates how an AAM consisting of two aspect models and a primary model is composed. The aspect models are instantiated by binding template parameters to application-specific values. We refer to the namespace from which binding values and names of elements in the primary model are drawn as the *application domain namespace*. An aspect model can be instantiated multiple times to produce multiple context-specific aspects. Composition of context-specific

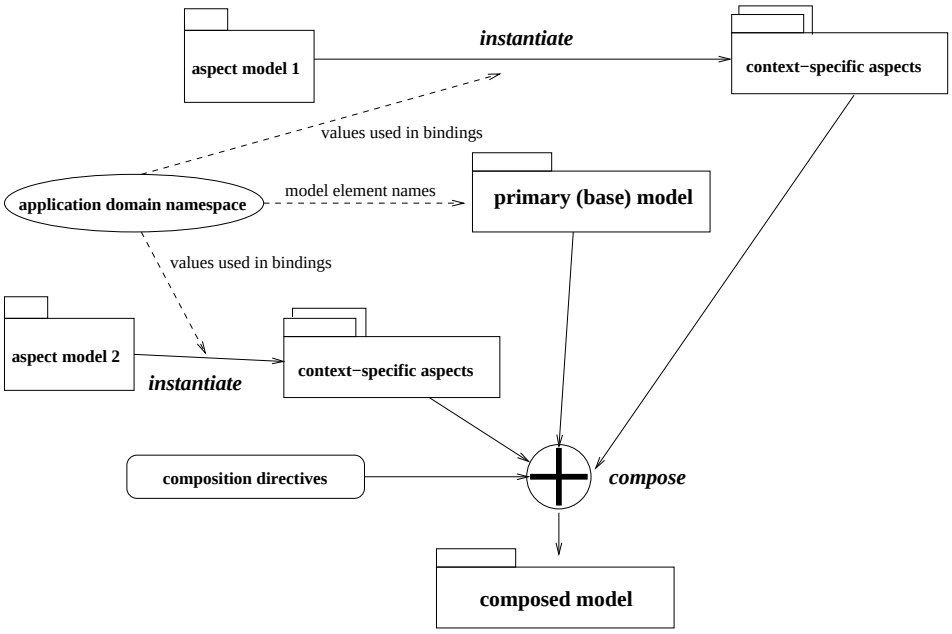


Figure 2: An Overview of Composition in the AOM Approach

aspect and primary models produces a model consisting of UML diagrams obtained by merging corresponding UML diagrams in the context-specific aspect and primary models. The AOM approach provides a basic composition procedure that can be altered in restricted ways by composition directives. For example, a composition directive can (1) specify that properties in aspect models override conflicting properties in primary models (or vice versa), (2) specify that particular primary (or aspect) 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.

The *Model Analysis* component in Fig. 1 is responsible for analyzing the composed model to identify errors and to determine the extent that dependability objectives are met. The focus of this paper is on aspect representation and model composition. We illustrate how identified conflicts can be resolved using composition directives, but a detailed account of techniques for analyzing UML models is outside the scope of this paper.

3 Representing Aspect Models

In this section we describe how aspect models can be represented as template UML diagrams representing patterns of concern solutions. The template diagrams in this paper produce UML diagrams describing logical architectural views of solutions when instantiated.

In the UML, template models are described by parameterized packages that explicitly list the parameters in the package header. We have found this notation to be unwieldy when a large number of parameters are involved. In this paper the parameters are explicitly marked in the template diagrams using the symbol "|".

Fig. 3 shows an aspect model, *Auth*, characterizing logical solutions in which access to a service is restricted to authorized clients. The aspect model consists of two diagram templates: A *class diagram template* that describes structural properties of the concern solutions and a *collaboration diagram template* that describes interactions among solution elements. Instantiating the class diagram template shown in Fig. 3(a) results in a class diagram that consists of composite classes representing logical architectural views of clients, servers with services under access control, and authorization repositories. A service under access control is represented by two operations in a server class:

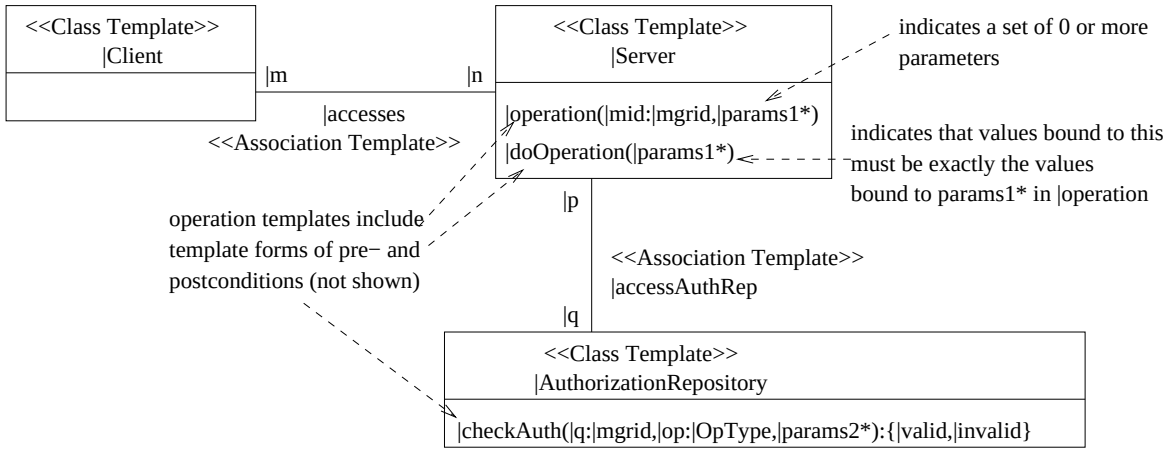
- An operation that checks whether a client that requests the service is authorized to execute the service. The operation signature is obtained by instantiating the operation template $|operation$. The operation takes in as arguments the client's identifier (represented by the operation argument template $|mid : |mgrid$) and zero or more values needed by the service (represented by the argument template $|params1*$). The template parameter $params1*$ is referred to as a *collection parameter* indicating that it must be bound to a collection of values.

- An operation that performs the required service. This operation is obtained by instantiating the operation template |*doOperation*. The use of the |*params*1∗ collection parameter in both the *operation* and *doOperation* templates indicates that the same value (i.e., the same set of arguments) must be used to instantiate the collection parameter in both of the templates.

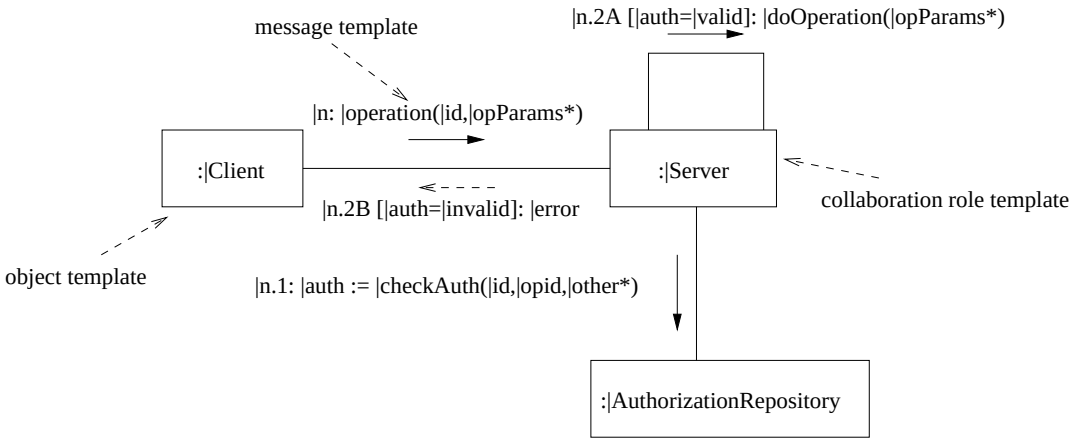
The class template |*AuthorizationRepository* consists of the operation template |*checkAuth* that produces an operation that performs authorization checks when instantiated. A |*checkAuth* operation uses the client identifier (represented by |*q* : |*mgrid*), an operation identifier (represented by |*op* : |*OpType*), and possibly other information passed in as arguments (represented by the collection parameter |*params*2∗), to determine whether the client is authorized to access the operation or not. If the client is authorized the operation returns a value that is an instantiation of |*valid*, otherwise it returns a value that is an instantiation of |*invalid*.

Operation templates may be associated with template forms of pre- and postconditions, referred to as *constraint templates*, that produce OCL specifications when instantiated. These constraint templates are presented separately from the diagrams to reduce diagram clutter. If an operation template is not associated with a constraint template then an operation produced by the template must be specified in the primary model or its behavior is to be specified or implemented in a subsequent refinement or detailing of the logical model. The operation templates |*doOperation* and |*checkAuth* do not have constraint templates associated with them. The following is the commented constraint template associated with the |*operation* template. The notation is based on the Object Constraint Language (OCL) version 2 [40]:

```
Context |Server::|operation(|mid:|mgrid,(|p:|T)*) :
Pre:
    -- This operation can be invoked at any time.
```



(a) Class Diagram Template for an Authorization Aspect Model



(b) Collaboration Diagram Template for an Authorization Aspect Model

Figure 3: An Authorization-based Access Control Aspect Model

```
        true
    Post:
        /* The service is carried out if and only if the client is
            authorized to invoke the service. */
         let authmessage : OclMessage =
            |AuthorizationRepositoryˆ|checkAuth(|mid,|opid,|p*) in
          (authmessage.hasReturned() and authmessage.result() = True
            implies |Serverˆ|doOperation(|p*)) and
          (|Serverˆ|doOperation(|p*) implies
            authmessage.hasReturned() and authmessage.result() = True)
```

The collaboration diagram template shown in Fig. 3(b) consists of template forms of participants (e.g., $:|Client$) and messages (e.g., $|n:|operation(|id,|opParams*)$. An instantiated participant template produces either a named or anonymous participant, for example, binding *UserMgmt* to the parameter *Server* in the $:|Server$ participant template produces the anonymous participant $:UserMgmt$. In a participant template, the type parameter (e.g., $|Server$ in $:|Server$) must be a classifier template in a corresponding classifier diagram template. Participant type parameters and the corresponding classifier templates must be instantiated with the same value.

Message templates consist of parameterized message sequence expressions, and parameterized message expressions. For example, $|n.1:|auth:=|checkAuth(|id,|opid,|other*)$, consists of a parameterized sequence expression, $|n.1$ in which *n* is a parameter that can be substituted by sequence expression (e.g., substituting 2.1.3 for *n* gives the sequence expression 2.1.3.1), and a parameterized message expression $|auth:=|checkAuth(|id,|opid,|other*)$ with parameters *auth*, *checkAuth*, *id*, *opid*, and an optional set of arguments indicated by the collection parameter *other*∗. The message expression $response:=IDcheck(userid,updateOp,userstatus,usersession)$ can be

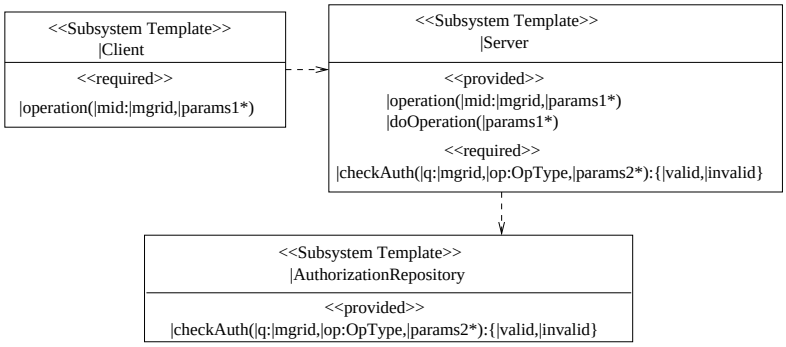
obtained from this template by binding *response* to *auth*, *IDcheck* to *checkAuth*, *userid* to *id*, *updateOp* to *opid* and $\{userstatus, usersession\}$ to $other*$.

The collaboration diagram template for the *Auth* aspect model describes the following interaction pattern:

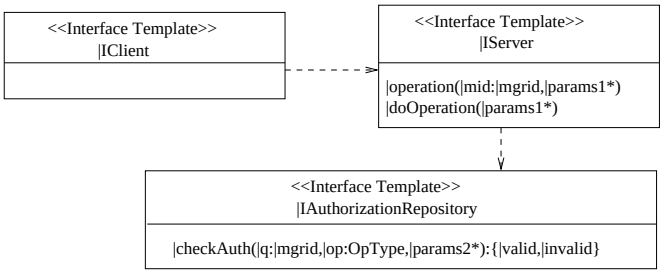
- **Message** $|n$: A client requests a service on a server by calling an instantiation of $|operation$.
- **Message** $|n.1$: An authorization check is requested by calling an instantiation of $|checkAuth$ in the authorization repository linked with the server.
- **Message** $|n.2A$: If authorization is granted then the service is performed by invoking an instantiation of $|doOperation$.
- **Message** $|n.2B$: If authorization is not granted the client is informed that access is not allowed.

The aspect model shown in Fig. 3 produces concern solution models that can be integrated with architecture models in which modules are composite classes (see [39] for more details on UML composite classes). Logical architectures can also be described using UML subsystems and interfaces as modules. The access control solution expressed in terms of subsystem templates is shown in Fig. 4(a), and the logical solution expressed in terms of interface templates is shown in Fig. 4(b). The collaboration diagram templates in these aspect models are syntactically identical to the collaboration diagram template shown in Fig. 3(b) and thus are not shown.

The three access control aspect models shown in Fig. 3 and Fig. 4 are specializations of the aspect model shown in Fig. 5. The collaboration diagram of the generalized aspect is syntactically identical to the collaboration diagram shown in Fig. 3(b). The generalized aspect model cannot be directly instantiated because it is based on abstract UML constructs (classifiers and relationships). This type of aspect model is called an *abstract* aspect model. An abstract aspect model must be

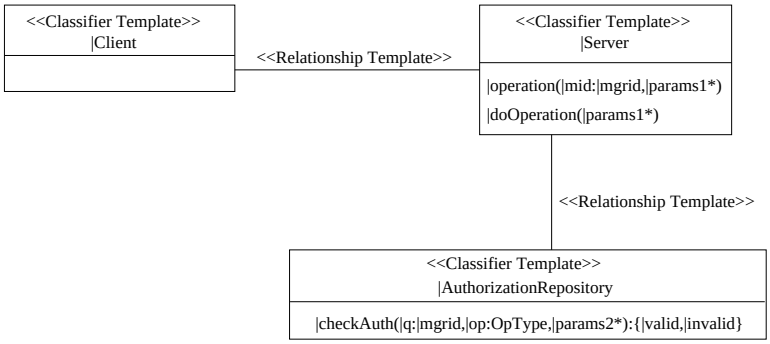


(a) Subsystem Diagram Template for an Authorization Aspect Model

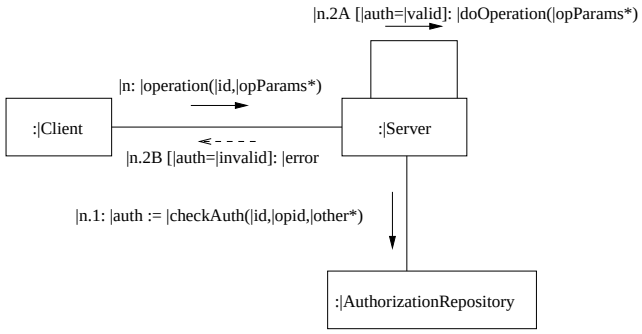


(b) Interface Diagram Template for an Authorization Aspect Model

Figure 4: Examples of Subsystem and Interface Based Access Control Aspect Models



(a) Classifier Diagram Template for an Authorization Aspect Model



(b) Collaboration Diagram Template for an Authorization Aspect Model

Figure 5: A Generalized Access Control Aspect Model

specialized to a concrete aspect model (i.e., one based on concrete UML constructs) before it can be instantiated.

The remainder of this paper uses architecture models in which modules are composite classes to illustrate the AOM approach. The internal structures of the composite classes are hidden in the architectural views presented in this paper.

4 Composing Aspect and Primary Models

Composing an aspect model with a primary model involves (1) instantiating the aspect model, using bindings, to produce a context-specifi c aspect model, and (2) integrating the context-specifi c

aspect model with the primary model. In this section we illustrate how composition can be carried out using a small example.

4.1 A Composition Example

Fig. 6(b) shows a primary model that describes a user management system in which *Manager* objects are linked to a *UserMgmt* object that controls access to a repository of user information (a *UserRepository* object). The *UserMgmt* class defines operations for adding a user to the repository (*addUser*) and for deleting a user from the repository (*deleteUser*). Access to the *addUser* and *deleteUser* operations by *Manager* objects is unrestricted in the primary model. To restrict access to these operations the instantiated *Auth* aspect model shown in Fig. 6(a) is composed with the primary model to obtain the composed model shown in Fig. 6(c).

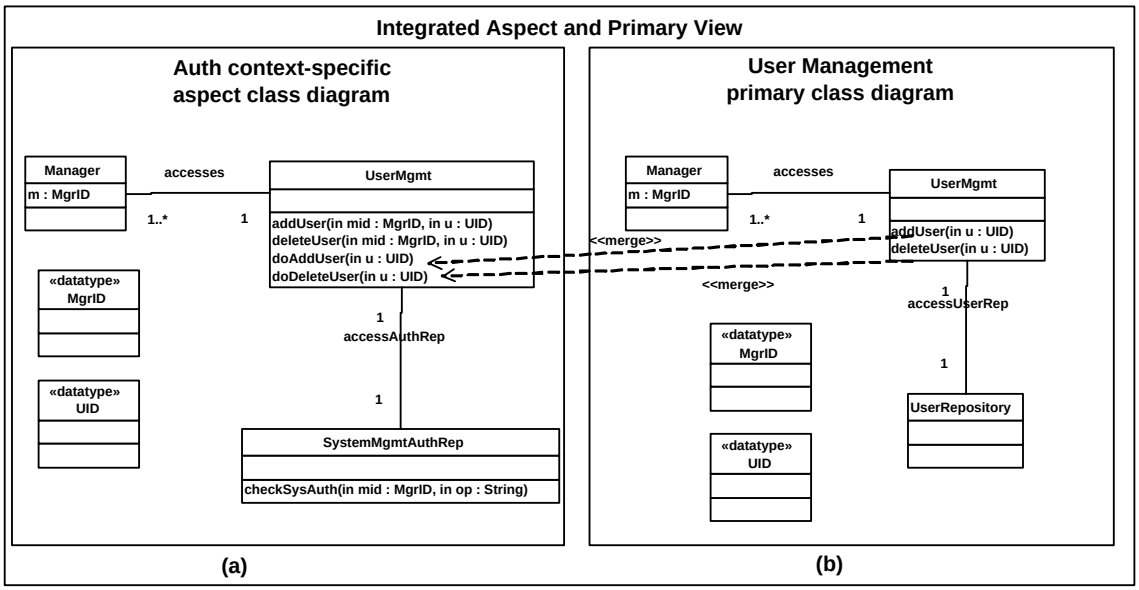
The context-specific aspect model in Fig. 6(a) is obtained by instantiating the *Auth* aspect model using bindings that define the values that are to be substituted for parameters in the *Auth* diagram templates. A binding relates an aspect model element to a model element and can be expressed as a pair of the form *(aspect element name, model element name)*. The *model element name* can be the name of a primary model element or the name of an application-specific element that is to be added to the composed model during composition. The type of the construct named by *model element name* must be the same as the parameter type, for example, a class template can only be bound to a model element that is a class. Some of the bindings used to produce the context-specific aspect model shown in Fig. 6(a) are given below:

(|Client, Manager); (|mgrid, MgrID); (|accesses, accesses); (|m, 1..*); ((|n, |p, |q),1)[1];

(|doOperation, doDeleteUser), (|doOperation, doAddUser);

(|Server,UserMgmt); (|AuthorizationRepository, SystemMgmtAuthRep).

[1]This is an abbreviated form of three pairs that respectively map *n*, *p*, and *q* to the multiplicity 1



Composition Directives

Rename Primary::UserMgmt::addUser() to doAddUser()

Rename Primary::UserMgmt::deleteUser() to doDeleteUser()

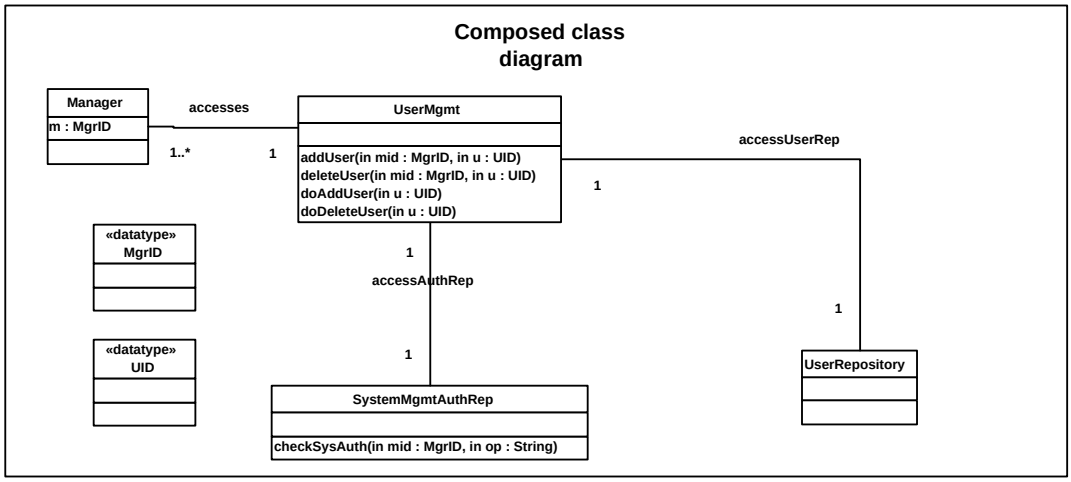


Figure 6: Example of Composing a Context-Specifi c Aspect Class Diagram and a Primary Class Diagram

Note that a single parameter may be instantiated more than once; for example, the operation template *doOperation* is instantiated twice to produce *doAddUser* and *doDeleteUser* operations. An *instantiation multiplicity* can be associated with a template to restrict the number of times a template can be instantiated; for example, a template of the form |*Temp 1..1* indicates that *Temp* can only be instantiated once. If a template is not associated with a (instantiation) multiplicity then the number of instantiations possible is not restricted (as is the case in the aspect models given in this paper).

Sometimes it is more convenient to express bindings as relationships between structures. For example, the bindings for operation templates can be expressed as follows:

((|operation,|mid,|params1*), (addUser, mid,{u:UID}));

((|operation,|mid,|params1*), (deleteUser, mid,{u:UID}));

((|checkAuth, |q, |op, |OpType, |params2*), (checkSysAuth, mid, op, String, {})).

Bindings also determine how constraint templates are instantiated. For example, the above bindings are used to produce the following OCL definitions of the *addUser* and *deleteUser* operations in the context-specific aspect model:

```
Context UserMgmt::addUser(mid:MgrID,u:UID):
Pre:
    true
Post:
  /* doAddUser() is called if and only if the Manager object
     is authorized to add users. */
   let authmessage : OclMessage =
     SystemMgmtAuthRep^checkSysAuth(mid,?:String) in
     authmessage.hasReturned() and authmessage.result() = True
```

```
        implies self^doAddUser(u) and
     self^doAddUser(u) implies
        authmessage.hasReturned() and authmessage.result() = True

Context UserMgmt::deleteUser(mid:MgrID,u:UID):
Pre:
    true
Post:
    /* doDeleteUser() is called if and only if the Manager object
        is authorized to delete users. */
     let authmessage : OclMessage =
        SystemMgmtAuthRep^checkSysAuth(mid,?:String) in
       authmessage.hasReturned() and authmessage.result() = True
         implies self^doDeleteUser(u) and
       self^doDeleteUser(u) implies
         authmessage.hasReturned() and authmessage.result() = True
```

The AOM approach uses a basic name-based composition procedure in which elements with the same name are merged to form a single diagram element in the composed model. For example, merging the aspect and primary class diagram views of the *Manager* class results in a class that integrates information from both views. Some of the rules that determine how information associated with matching elements are combined are given below (these rules can be modified using composition directives, as indicated below):

- If the matching elements are operations with operation specifications, the operation specification in the composed model is the conjunction of the operation specifications associated with the matching operations. A composition directive can be used to vary how the specifications are logically connected.

- If the matching elements are attributes (or other elements) with constraints, the constraint associated with the attribute in the composed model is the conjunction of the constraints associated with the matching attributes. A composition directive can be used to vary how the constraints are logically connected.

- If the matching elements are associations, then the stronger (more restrictive) multiplicity at an association end is used in the composed model. A composition directive can be used to override this rule.

Unmatched model elements (i.e., model elements that only occur in either the aspect model or the primary model) are included in the composed class diagram.

Using the basic composition procedure to compose the *UserMgmt* aspect and primary model views results in a conflict for the *addUser* and *deleteUser* operations because they have the same names but different specifications in the two views; the *addUser* and *deleteUser* in the context-specific aspect model carry out authorization checks, while the operations with the same names in the primary model add and delete users, respectively (the primary model specifications are not given in this paper). Furthermore, the operations *doAddUser* and *doDeleteUser* in the context-specific aspect model have the same specifications as those provided for *addUser* and *deleteUser* operations, respectively, in the primary model. Composition directives are used to resolve the conflict by renaming the *addUser* and *deleteUser* operations in the primary model to *doAddUser* and *doDeleteUser*. The renaming removes the conflict and allows the primary model operations

to be merged with the *doAddUser* and *doDeleteUser* operations in the context-specific aspect model. The result is that the *doAddUser* and *doDeleteUser* operations in the context-specific aspect model are respectively merged with the original *addUser* and *deleteUser* operations in the primary model as shown in Fig. 6.

In general, a composition directive can (1) determine the order in which multiple aspect models are composed with a primary model, (2) define precedence or override relationships between matching aspect and primary model elements with conflicting properties or definitions, and (3) determine the elements that are renamed (e.g., to resolve conflicts), added, or deleted during composition. Adding new elements or deleting existing elements may be necessary to correctly compose aspect and primary models. For example, a security access control aspect may restrict access to an object by prohibiting particular relationships between the object and other objects. This can be done by identifying undesirable relationships in the aspect models and deleting them if found in the primary model. Elements marked for deletion in an aspect model are referred to as *prohibited elements*. Later in this section we give an example of a situation that requires composition directives that add and delete model elements.

In summary, composition directives allow one to vary how aspect and primary models are composed. Consequently, aspect models do not need to capture all possible variations. In the next subsection we show how composition directives can be used to obtain variants of solutions described by aspect models.

4.2 Using Composition Directives to Obtain Variants of Composed Models

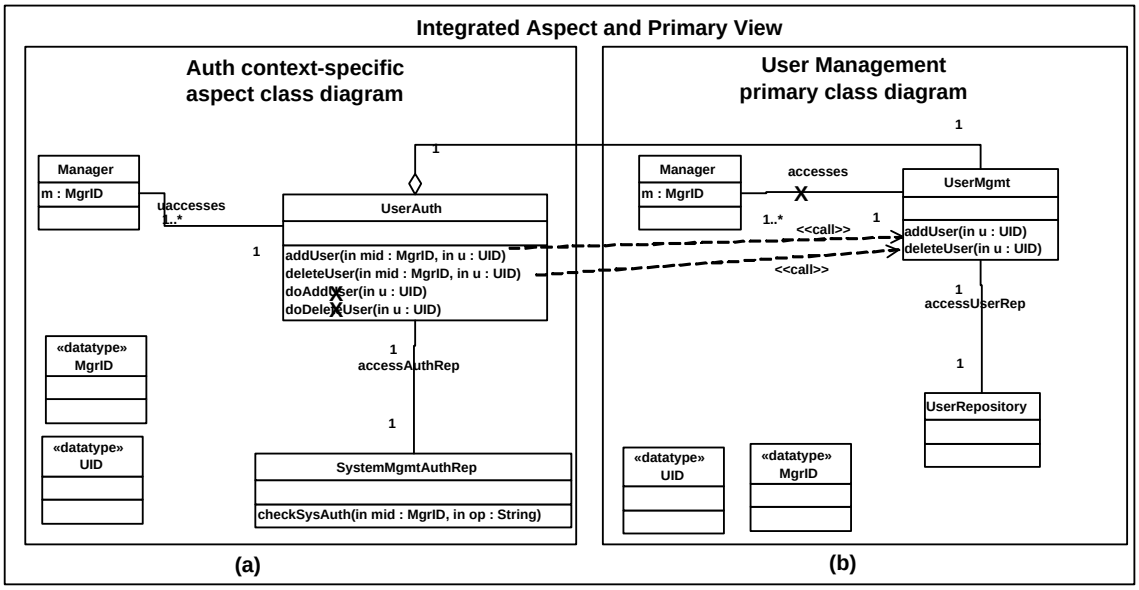
Using the same aspect and primary models, different composed models can be produced by varying the bindings and composition directives. Fig. 7 shows a composed model obtained by composing the *Auth* aspect class diagram and the User Management primary class diagram using a

different set of bindings and composition directives. We do not give the bindings for this case; they can be inferred from the context-specific aspect model shown in Fig. 7(a).

The two systems described by the composed AAMs shown in Fig. 6(c) and in Fig. 7(c) accomplish the same tasks but do so differently. In Fig. 7(c) the authorization operations and the services are located in separate objects. Rather than treating the *UserMgmt* class as a |*Server* class, a new server class is introduced by the context-specific aspect model (see Fig. 7(a)). The intent is that the *addUser* and *deleteUser* operations in the *UserAuth* class would call the corresponding operations in *UserMgmt* after a successful authorization. To create a class diagram that reflects this intent, composition directives are defined that (1) add an association between the *UserAuth* and the *UserMgmt* classes, (2) removes the association between the client and the *UserMgmt* class, (3) removes the *doAdduser* and *doDeleteUser* operations from the *UserAuth* class, and (4) replaces references to *doAdduser* and *doDeleteUser* with references to *addUser* and *deleteUser*, respectively, in *UserMgmt*. The first directive is depicted by the association between the classes in the aspect and the primary model shown in Fig. 7. The second, third, and fourth directives are captured by the following expressions:

```
Replace Primary::Manager::accesses by Aspect::Manager::uaccesses
    - removes the accesses association between Manager and UserMgmt
      in the primary model (graphically indicated by placing an X
      on the association in the class diagram) and replaces all
      references to the association in Manager to the uaccesses
      association in the aspect model.

Replace Aspect::UserAuth::doAddUser() by Primary::UserMgmt::addUser()
    - removes doAddUser (graphically indicated by placing an X on the
```



Composition Directives

Replace Primary::Manager::accesses by Aspect::Manager:uaccesees

Replace Aspect::UserAuth::doAddUser() by Primary::UserMgmt::addUser()

Replace Aspect::UserAuth::doDeleteUser() by Primary::UserMgmt::deleteUser()

prohibited elements are marked with an X

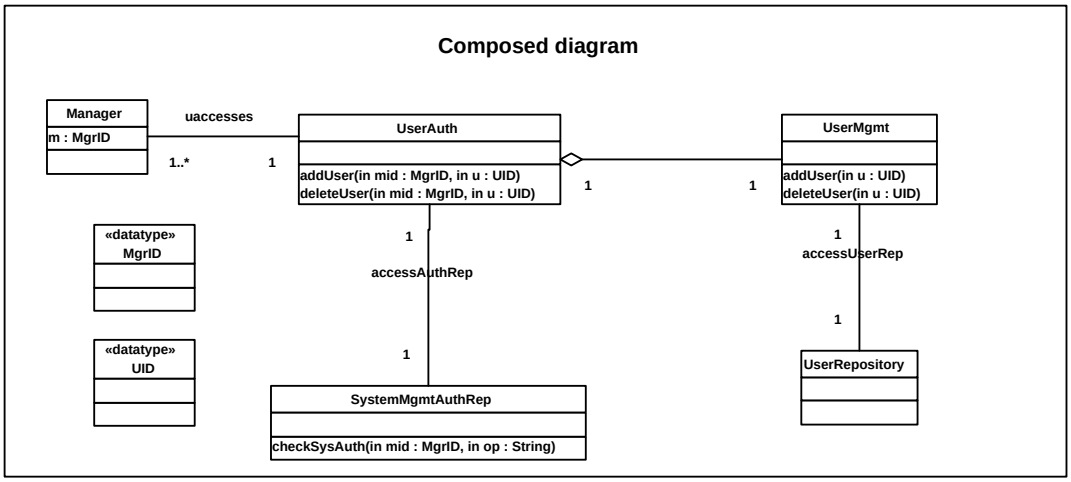


Figure 7: An Alternative Composition of the *Auth* and User Management Class Diagrams

```
        operation in the class diagram) and replaces all references
        to it by references to addUser in the primary model.

Replace Aspect::UserAuth::doDeleteUser() by Primary::UserMgmt::deleteUser()
    - removes doDeleteUser and replaces all references to it
        by references to addUser in the primary model.
```

The replacement of references is needed to ensure that the constraint definitions that refer to the deleted elements refer to their replacements in the composed model. For example, the above directives produce a composed model that include the following operation definitions for *addUser* and *deleteUser* in *UserAuth* (this is obtained by replacing references to `self^doAddUser(u)` by `UserMgmt^addUser(u)` , and `self^doDeleteUser(u)` by `UserMgmt^deleteUser(u)` in the OCL definitions of *addUser* and *deleteUser* given earlier for the primary model in Fig. 6(b)):

```
Context UserAuth::addUser(mid:MgrID,u:UID):
Pre:
    true
Post:
    /* UserMgmt.addUser() is called if and only if the Manager object
        is authorized to add users. */
   let authmessage : OclMessage =
      SystemMgmtAuthRep^checkSysAuth(mid,?:String) in
     authmessage.hasReturned() and authmessage.result() = True
        implies UserMgmt^addUser(u) and
     UserMgmt^addUser(u) implies
```

```
        authmessage.hasReturned() and authmessage.result() = True


Context UserAuth::deleteUser(mid:MgrID,u:UID):
PreCondition:
    true


PostCondition:
    /* UserMgmt deleteUser() is called if and only if the Manager
        object is authorized to delete users. */
   let authmessage : OclMessage =
      SystemMgmtAuthRepˆcheckSysAuth(mid,?:String) in
     authmessage.hasReturned() and authmessage.result() = True
        implies UserMgmtˆdeleteUser(u) and
     UserMgmtˆdeleteUser(u) implies
        authmessage.hasReturned() and authmessage.result() = True
```

In summary, we have shown how aspect and primary models can be composed and how composition directives can be used to resolve conflicts. One can view primary models and context-specific aspect models as views of an architecture, and thus their composition can be considered to be a view composition activity. We also show how composition directives and bindings can be used to produce different composed models from the same aspect and primary models. The example illustrates how bindings and composition directives can be used to reflect architectural decisions.

5 Limitations and Open Issues

In this section we discuss some of the issues that are not yet addressed by our AOM approach, and outline our plans for addressing the issues.

5.1 Identifying Aspects

It may not be desirable to model all crosscutting concern solutions as aspects. An AOM approach should provide guidelines that help developers determine the crosscutting concern solutions that can beneficially be localized in aspects. Our AOM approach targets crosscutting dependability solutions that may need to be balanced against other concern solutions, and those that are expected to evolve significantly during development. The localization of these solutions can ease evolution of the solutions and provide support for rigorous tradeoff analysis. Currently, our AOM approach does not provide a set of detailed guidelines for determining the crosscutting solutions that should be localized as aspects. Good guidelines should be based on experience and data collected on projects that utilize the AOM approach. Such experience and data are not yet available.

5.2 Developing Composition Strategies

When multiple aspect models are composed with a primary model, one has to be concerned with (1) the order in which the aspect models are composed, and (2) identifying and resolving conflicts or compromised behaviors. A conflict arises when a property in one aspect model contradicts a property in another aspect model. Composing a single aspect model with a primary model can also result in conflicts that need to be resolved (as shown in the previous section).

A behavior defined by an aspect model is compromised when it cannot be performed as specified because some of its sub-behaviors have been modified (or deleted) after merging with behaviors defined in other aspect models or the primary model. For example, (1) an aspect model may

remove a relationship between two entities that is needed by a behavior defined in another aspect model, or (2) an operation replacement introduced by a composition directive results in behavior that violates requirements previously satisfied by the operation being replaced. These problems can be resolved by making tradeoffs based on the relative importance of satisfying the conflicting requirements. Resolving problem (1) requires one to tradeoff the requirement that needs the relationship against the requirement that necessitates its deletion. Problem (2) can be resolved by restoring the overridden operation and renaming the operation replacement.

It may be possible to apply prior experience in addressing concerns to constrain composition such that the occurrences of conflicts and compromised behaviors are minimized. Such experience can be captured in *composition strategies*. A composition strategy is influenced by domain knowledge pertaining to aspects (e.g., security and fault tolerance expertise), past experiences in addressing concerns, results of tradeoff analyses, and the properties (e.g., idempotency, commutativity, associativity, and monotonicity) of the aspect models. Consider, for example, two security aspect models: one for authentication and the other for authorization. Doing authorization without authentication is meaningless. To get the desired result, an authentication aspect model must be composed with a primary model before an authorization aspect model.

In summary, a composition strategy should be based on the properties of the aspect models, the constraints imposed by the domains of the aspect models, the results of tradeoff analyses, and the past experiences based on realizing multiple, competing aspect models. A challenge is to develop a language for expressing composition strategies and techniques for obtaining composition directives from strategies. We are currently addressing these problems in our AOM research.

5.3 Analyzing Composed Models

For large complex systems, the result of composition may be a complex model that may be difficult to comprehend. On the other hand, the composed model provides the detail needed to identify conflicts and undesirable emergent properties that arise as a result of interactions between model elements described by aspect and primary models. In the AOM approach, composition is carried out primarily to support analysis that uncovers conflicts and other defects that arise as a result of integrating aspect and primary model views.

Analysis can be performed at three levels: Unit analysis is concerned with analyzing a single context-specific aspect model, integration analysis occurs when context-specific aspect models are composed sequentially with a primary model, and system analysis occurs when all aspect models have been composed with a primary model. When multiple aspect models are composed sequentially with a primary model, one must test that no existing capabilities were broken and that the capability of the newly composed aspect was preserved in the composition. System analysis is concerned with determining whether the composed AAM satisfies the requirements.

To support dynamic evaluation of composed models, an operational semantics for UML models is needed. We are currently adapting a systematic technique for testing and exercising UML designs to our AOM approach [1, 16, 29]. State exploration techniques, such as model-checking (e.g., see [2, 19]), can also be used to analyze composed models.

We are also developing a system analysis technique that involves evaluating composed models against a representative set of usage scenarios. The scenarios describe both proper and improper uses of the system. Scenarios describing improper usages are called *misuse* scenarios. For example, in order to evaluate the impact of security concern solutions on an application, misuse scenarios that describe malicious attacks can be developed. Misuse scenarios are used to deter-

mine if the mechanisms defined by the security aspect models are sufficient to prevent the attacks from compromising protected resources. Scenarios describing authorized interactions are used to determine if the authorized activities are adversely affected by behaviors described by the aspect models. Scenarios are expressed in terms of UML behavioral models (e.g., sequence diagrams) and can be based on use cases that describe authorized behaviors and on misuse cases that describe behaviors that should not be present in a correct system implementation. Analysis involves composing the scenario descriptions with the composed AAM and evaluating the result. If correct composition of a misuse scenario and an AAM produces a consistent model then the AAM has a flaw. Similarly, if correct composition of a scenario describing authorized interactions and an AAM produces an inconsistent model, then the AAM has a flaw.

5.4 Evolving Aspect-Oriented Models

Support for extracting aspect and primary model views from composed models can help ease the task of evolving AAMs. For example, changing a context-specific aspect model or primary model after composition has been carried out can be accomplished by extracting the model view from the composed model and changing the model. Reintegration of the view involves propagating the changes to the other parts of the composed model. Similarly, developers should be able to add new composition directives or modify composition directives and bindings in order to resolve conflicts and fix other defects in the composed model. Extraction of model views or composition related information, and their reintegration requires tool support if this approach is to scale-up to large system models. Automated support for view extraction and reintegration also eases exploration of solution alternatives carried out to support tradeoff analysis.

Extracting views and other information used to compose models requires maintaining relationships among aspect models, the primary model and the composed model. The problem is similar

to tracking the evolution of complex composite parts in discrete manufacturing. We are currently investigating the use of a standard framework, the Product Data Management (PDM) framework [18], that was developed for managing the evolution of complex products in the discrete manufacturing area to support storing and evolving AAMs.

5.5 Process Support for Architectural Modeling using AOM

AOM can be carried out in the context of an iterative and incremental architecture development process. In the first iteration an initial primary model is developed. This model reflects early decisions pertaining to the concerns that determine the modular structure of the architecture. The AOM approach described in this paper supports the development of architecture model in which the modules can be composite classes, subsystems, or logical components.

An initial set of context-specific aspects that describe logical solutions to a subset of dependability concerns that crosscut the primary model are also developed in the first iteration. The initial aspect and primary models are composed to produce a composed AAM, which is then analyzed. The following activities are carried out in each subsequent iteration:

- If the analysis performed in the previous iteration uncovers problems, the aspect models, primary model, or the composition directives are modified accordingly.
- New aspect models for dependability concerns not covered in previous iterations can be introduced, and the primary model can also be extended to take into account functionality not considered in previous iterations.
- If needed, new composition directives are created.
- The modified aspect and primary models are composed and analyzed.

5.6 Tool Support for AOM

We are developing a prototype integrated toolset that supports (1) creation and cataloging of aspect models, (2) composition of aspects and primary models, and (3) rigorous analysis of composed models. To date, our work on tool development has produced the following:

- An architectural design of a toolset that supports creation of aspect models, composition of aspects and primary models, and analysis of composed models has been developed [26].
- A prototype editor for creating aspect model class diagram templates has been developed. The editor was built using the Eclipse Modeling Framework (see *http://www.eclipse.org/emf*). The prototype does not support instantiation of the templates.
- A tool, built on top of Rational Rose, that generates instantiations from template forms of UML class diagrams (generic aspects) has been developed.
- A prototype model composer that takes primary model and context-specific aspect class diagrams and composes them has been developed.

We are currently integrating and extending the above tools to form an integrated AOM tool set.

6 Related Research

Aspect-oriented programming (AOP) supports multi-dimensional separation of concerns (MDSoC) at the programming level [3, 22, 23, 24, 25, 27, 28, 35, 36]. An AOP aspect is an implementation or design concern that crosscuts the primary functional units of a program (e.g., concerns that crosscut classes of an object-oriented program). A few researchers have started to address the problem of defining and composing aspects at an abstraction level higher than the programming language level (e.g., see [5, 11, 17, 32, 33, 37]).

Fiadeiro *et al.* [11] specify aspects related to system coordination using an algebraic approach. Their approach is applicable to detailed design and code, and utilizes a notation that is not widely known by system developers. Gray *et al.* [17] use aspects to represent aspects in domain-specific models. Their research is part of the Model-Integrated Computing (MIC) initiative that targets embedded software systems specifically. MIC extends the scope and usage of models such that they form the backbone of a development process for building embedded software systems. Requirements, architecture and the environment of a system is captured in the form of formal high-level models that allow the representation of concerns. Our work on MDSoC can complement the MIC efforts by providing UML-based techniques for representing and composing aspects, and making tradeoff decisions. Suzuki *et al.* [37] extend the UML so that it can be used to model code level aspects. Their approach is restricted to design aspects that can be represented as aspects in an aspect-oriented program.

In the AOM approach proposed by Clarke *et al.* [4, 6, 7], a design, called a *Subject*, is created for each system requirement. A comprehensive design is a composition of subjects. Subjects are expressed as UML model views. Composition relationships specify how models are to be composed by identifying overlapping concepts in the subjects and specifying how models are integrated. The UML metamodel is extended to support composition relationships and describe well-formedness rules for composition. Two types of integration strategies are used: Override and merge. Override integration is used when existing behavior in a subject needs to be updated to reflect new requirements. Merge integration is used when subjects for different requirements are to be integrated. Operations in related subjects may need to be merged into a unified operation. Reconciliation strategies are use to resolve conflicts between property values of corresponding subject elements. Precedence relationships, transformation functions applied to conflicting elements, explicit specification of reconciled elements, and default values may be used for reconciliation.

As part of the Early Aspects initiative, Moreira, Araujo, and Rashid have targeted multi-dimensional separation throughout the software cycle [30, 31, 32, 33]. This work supports modularization of broadly scoped properties at the requirements level to establish early tradeoffs, provide decision support and promote traceability to artifacts at later development stages.

The work described in this paper extends our previous work (e.g., see [13, 14, 15]) by refining the aspect modeling notation and the instantiation process, and refining the notion of composition directives to support conflict resolution and modeling of solution variants.

7 Conclusion

Current modeling approaches provide good support for modularizing systems along a few dimensions. AOM can significantly enhance support for separation of concerns targeted at tackling growing software complexity. The AOM approach described in this paper can help developers better manage the complexity of creating and evolving complex software that must address multiple dependability concerns.

Our research goal is to develop an AOM approach that addresses three factors that contribute to the complexity of software development: (1) the complexity inherent in the required functionality of the software system; (2) the pervasiveness and variety of interdependent concerns that must be addressed in an architecture; and (3) the need to balance forces when addressing competing system concerns. The above factors can be addressed in an AOM approach that integrates work on model-driven development, MDSoC, and value-based assessment. Model-driven development addresses factor (1) by raising the level of abstraction at which functionality is developed. Approaches that support MDSoC address factor (2) by providing the means for isolating, composing and analyzing crosscutting solutions. Value-based assessment techniques address factor (3) by

providing a base for rigorous tradeoff analysis. The work described in this paper addresses factors (1) and (2). We are currently developing support that explicitly addresses factor (3). Tradeoff analysis is desired when crosscutting solutions interact in ways that compromise the accomplishment of concern objectives. In such situations the system developer must make tradeoffs based on prioritizations of objectives. The challenge is to (1) develop systematic and quantitative tradeoff analysis techniques that allow developers to assess alternative solutions, (2) develop techniques for capturing and representing experience related to making tradeoffs across a set of aspect models, and (3) use the captured experience to guide how aspect models are composed with other models. The captured experience can take the form of composition strategies that determine the set of aspect models and composition directives that produce a composed model that best meets the requirements. Composition strategies and the decisions they drive should be based on information about value and importance of the architectural choices represented in alternative aspect models. Our ongoing work in this area involves adapting existing approaches to tradeoff analysis, for example, the DDP approach [8, 9, 10].

Acknowledgements

This material is based upon work funded by AFOSR under Award No. FA9550-04-1-0102.

References

[1] A. Andrews, R. France, S. Ghosh, and G. Craig. Test Adequacy Criteria for UML Design Models. *Journal of Software Testing, Verification and Reliability*, 13(2):95–127, April-June 2003.

[2] J. M. Atlee and J. Gannon. State-Based Model Checking of Event-Driven System Requirements. *IEEE Transactions on Software Engineering*, 19(1):24–40, Jan. 1993.

[3] L. Bergmans and M. Aksit. Composing multiple concerns using composition filters. *Communications of the ACM*, 44(10), Oct 2001.

[4] S. Clarke. "Extending Standard UML with Model Composition Semantics". *Science of Computer Programming*, 44(1):71–100, July 2002.

[5] 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.

[6] S. Clarke and R. J. Walker. Composition patterns: An approach to designing reusable aspects. In *The 23rd International Conference on Software Engineering (ICSE), Toronto, Canada*, 2001.

[7] S. Clarke and R. J. Walker. Towards a standard design language for AOSD. In *The 1st International Conference on Aspect-Oriented Software Development, Enschede, The Netherlands*, April 2002.

[8] M. Feather. A quantitative risk-based model for reasoning over critical system properties. In *Proceedings of the International Workshop on Requirements for High Assurance Systems*, pages 11–18, Essen, Germany, September 2002.

[9] M. Feather, S. Conford, J. Dunphy, and K. Hicks. A quantitative risk model for early lifecycle decision making. In *Integrated Design and Process Technology, IDPT-2002*, Society for Design and Process Science, 2002.

[10] M. Feather and S. Cornford. Quantitaive Risk-based requirements reasoning. *Requirements Engineering Journal, Springer Verlag, (accepted)*.

[11] J. L. Fiadeiro and A. Lopes. Algebraic semantics of co-ordination or what is it in a signature? In A. Haeberer, editor, *Proceedings of the 7th International Conference on Algebraic Methodology and Software Technology (AMAST'98)*, volume 1548 of *Lecture Notes in Computer Science*, pages 293–307, Amazonia, Brasil, January 1999. Springer-Verlag.

[12] R. France, D.-K. Kim, S. Ghosh, and E. Song. A UML-based pattern specification technique. *IEEE Transactions on Software Engineering*, 30(3), March 2004.

[13] G. Georg, R. France, and I. Ray. An Aspect-Based Approach to Modeling Security Concerns. In *Proceedings of the Workshop on Critical Systems Development with UML*, Dresden, Germany, 2002.

[14] G. Georg, R. France, and I. Ray. Designing High Integrity Systems using Aspects. In *Proceedings of the Fifth IFIP TC-11 WG 11.5 Working Conference on Integrity and Internal Control in Information Systems (IICIS 2002)*, Bonn, Germany, November 2002.

[15] G. Georg, I. Ray, and R. France. Using Aspects to Design a Secure System. In *Proceedings of the Interational Conference on Engineering Complex Computing Systems (ICECCS 2002)*, Greenbelt, MD, December 2002. ACM Press.

[16] S. Ghosh, R. B. France, C. Braganza, N. Kawane, A. Andrews, and O. Pilskalns. Test adequacy assessment for UML design model testing. In *Proceedings of International Symposium on Software Reliability Engineering, ISSRE 2003*, 2003.

[17] J. Gray, T. Bapty, S. Neema, and J. Tuck. Handling Crosscutting Constraints in Domain-Specific Modeling. *Communications of the ACM*, 44(10):87–93, Oct. 2002.

[18] O. M. Group. PDM enablers. Technical Report mfg/98-02-02, The Object Management Group (OMG), 2002.

[19] G. J. Holzmann. *Design and Validation of Computer Protocols*. Sotware Series. Prentice Hall, Englewood Cliffs, NJ, 1991.

[20] M. Jackson. *Problem Frames: Analyzing and Structuring Software Development Problems*. Addison Wesley Professional, 2000.

[21] M. Kande. *A Concern-Oriented Approach to Software Architecture*. PhD thesis, EPFL, Lausanne, Switzerland, 2003.

[22] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, Oct. 2001.

[23] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '01)*, pages 327–353, Budapest, Hungary, June 2001.

[24] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, 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.

[25] K. Kieberherr, D. Orleans, and J. Ovlinger. Aspect-oriented programming with adaptive methods. *Communications of the ACM*, 44(10):39–41, Oct. 2001.

[26] F. Mekerke, G. Georg, R. France, and R. Alexander. Tool Support for Aspect-Oriented Design. In *Advances in Object-Oriented Information Systems: OOIS2002 Workshops*. Springer-Verlag, 2002.

[27] H. Ossher and P. Tarr. Using multidimensional separation of concerns to (re)shape evolving software. *Communications of the ACM*, 44(10):43–50, Oct. 2001.

[28] J. A. D. Pace and M. R. Campo. Analyzing the role of aspects in software design. *Communications of the ACM*, 44(10):66–73, Oct. 2001.

[29] O. Pilskalns, A. Andrews, S. Ghosh, and R. B. France. Rigorous testing by merging structural and behavioral UML representations. In *Proceedings of 6th International Conference on the Unified Modeling Language, UML 2003, San Francisco, USA, October 20-24, 2003*, 2003.

[30] A. Rashid. A Hybrid Approach to Separation of Concerns: The Story of SADES. In *3rd International Conference on Meta-Level Architectures and Separation of Concerns (Reflection), Springer-Verlag Lecture Notes in Computer Science 2192*, pages 231–249, Kyoto, Japan, September 25–28 2001.

[31] A. Rashid and R. Chitchyan. Persistence as an Aspect. In *2nd International Conference on Aspect-Oriented Software Development, ACM*, pages 120–129, Boston, March 2003.

[32] A. Rashid, A. Moreira, and J. Araujo. Modularization and Composition of Aspectual Requirements. In *2nd International Conference on Aspect-Oriented Software Development, ACM*, pages 11–20, Boston, March 2003.

[33] A. Rashid, P. Sawyer, A. Moreira, and J. Araujo. Early Aspects: A Model for Aspect-Oriented Requirements Engineering. In *IEEE Joint International Conference on Requirements Engineering, IEEE Computer Society Press*, pages 199–202, Essen, Germany, September 9–13 2002.

[34] R. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role Based Access Control Models. *IEEE Computer*, 29(2):38–47, 1996.

[35] A. R. Silva. Separation and composition of overlapping and interacting concerns. In *OOPSLA '99 First Workshop on Multi-Dimensional separation of Concerns in Object-Oriented Systems*, Denver, Colorado, November 1999.

[36] G. T. Sullivan. Aspect-oriented programming using reflection and metaobject protocols. *Communications of the ACM*, 44(10):95–97, Oct. 2001.

[37] J. Suzuki and Y. Yamamoto. Extending UML with Aspects: Aspect Support in the Design Phase. In *Proceedings of the 3rd ECOOP Aspect-Oriented Programming Workshop*, Lisbon, Portugal, June 1999.

[38] The Object Management Group. Unified Modeling Language. Version 1.5, OMG, formal/2003-03-01, 2003.

[39] The Object Management Group. Unified Modeling Language: Superstructure. Version 2.0, OMG, ptc/03-07-06, 2003.

[40] J. Warmer and A. Kleppe. *The Object Constraint Language, Second Edition*. Addison-Wesley, 2003.

List of Tables

List of Figures