An Aspect-Based Approach to Modeling Security Concerns

Geri Georg
Agilent Laboratories, Agilent Technologies, Fort Collins, USA
geri_georg@agilent.com

Robert France, Indrakshi Ray
Department of Computer Science, Colorado State University
Fort Collins, USA
france@cs.colostate.edu, iray@cs.colostate.edu

Abstract. In this paper we show how design-level aspects can be used to encapsulate security concerns that can be woven into models of software designs. The aspect-oriented design technique described in this paper allows one to analyze the impact of security concerns on other functional concerns. We illustrate our technique using a small example.

1 Introduction

Aspect-oriented programming (AOP) is concerned with encapsulating cross-cutting concerns (e.g., exception handling, data logging), or *aspects*, to better manage program evolution and to facilitate reuse of the cross-cutting concerns (e.g., see [6, 7]). AOP can be viewed as an outgrowth of efforts concerned with developing programming language support for the separation of concerns design principle. From a design perspective one can view an aspect as a concern that cross-cuts the primary design modularization. An aspect approach to design allows one to encapsulate cross-cutting concerns that can be woven in designs of complex systems.

In our approach, an *aspect-oriented design* (AOD) consists of a primary model (i.e., a model based on the primary modularization of the design) and one or more aspect models. In this paper, we focus on aspects that reflect security concerns (specifically, access control). An integrated model of the system can be obtained by "weaving" (composing) the primary model with the aspect models.

In treating security and other critical concerns as aspects during design we aim to reap the following benefits:

- Aspects allow one to understand and communicate concerns as patterns, rather than in terms of a specific application's behavior.
- Changes to concerns are made in one place (in the aspect), and effected by weaving the aspects into primary models.

- Enables analysis of the impact of security concerns on design elements. System architects can use the results of the analysis to make trade-offs and to refactor designs in order to decrease complexity.

In our work aspects are patterns of structures and behaviors. These patterns can be applied to a set of elements in a model of a system. An aspect is defined in terms of structures of meta-roles called (meta-)*Role Models* [3,4]. A meta-role[1] defines the properties that conforming syntactic elements of the modeling language (model elements) must possess. Applying (weaving) an aspect into a primary model involves modifying specified elements of the model such that they conform to the properties expressed in a Role Model. Weaving (1) adds model elements to, (2) deletes model elements from, and (3) modifies model elements in the primary model so that the resulting model conforms to the Role Model. The models used in this paper are expressed in the UML (Unified Modeling Language [9]).

In Section 2 we give an overview of Role Models and describe how they can be used to represent aspects. We illustrate our approach to modeling aspects by developing Role Models for a simple security mechanism. In Section 3 we illustrate our approach to AOD by weaving the security aspect into UML static and interaction models of a simple user management system. We conclude in Section 4 with an overview of our plans to further develop this work.

2 Modeling Aspects using Role Models

A number of authors have tackled the problem of defining and weaving aspects at the design level (e.g., see [1,2,10]). The goal of our work is to provide a highly flexible environment for weaving aspects into models that leverages our work on pattern-based model refactoring [3]. Aspects are expressed as patterns and weaving is a special case of transforming a UML model using design patterns.

Viewing an aspect as a pattern allows us to use a technique we developed for representing patterns in terms of meta-roles. A meta-role (henceforth referred to simply as a role) is a collection of properties that can be applied to model elements of a specific type (the model element type is the *base* of the role). For example, a class role (i.e., a role with the UML metamodel element *Class* as a base) defines properties that determine a set of UML class constructs. A model element conforms to a role if it possesses the properties defined in the role. A role specifies the structure of conforming model elements (i.e., syntactic elements) and the features that must be present in conforming syntactic elements. For example, a class role can specify that conforming UML classes must be abstract or concrete and must have certain attributes and behaviors (operations or methods) that have specified properties (expressed as OCL constraints).

---

[1] A meta-role is **not** the same as a collaboration role. A collaboration role is a view of a UML class, while a meta-role determines a subset of instances of a UML metamodel class.

A Role Model describes the structural and behavioral views of a pattern in terms of a structure of roles. The UML is used as the modeling notation in our work, thus the Role Models we developed define properties of conforming UML models [4]. A UML model element (e.g., a class or an association) that has the properties defined in a role can *play the role*, that is, it *conforms to* (or realizes) the role. A UML model is said to conform to (or realize) a Role Model (i.e., is a realization) if it consists of model elements that conform to the roles in the Role Model.

Establishing that a UML model conforms to a Role Model, with respect to a mapping between elements in the UML model and roles in the Role Model, is accomplished by verifying that the properties specified in the model imply the properties specified in the Role Model. We use a template form of Role Models that allows one to generate conforming model elements from the Role Models to facilitate weaving of aspects into models. One can view the Role Models described in this paper as a specialized form of the more general Role Models developed in our previous work on specifying patterns of structures and behaviors [3,4].

Weaving an aspect defined by Role Models is a model transformation process in which a non-conforming model is transformed to a conforming model (i.e., a model that incorporates the aspect). Weaving a Role Model into a target UML design model, *M*, can involve (1) merging roles with specified model elements in *M*, that is, modifying (including deleting) specific model elements in *M* so that they conform to the roles and (2) for roles that are not associated with any model element in *M*, generating new model elements from the roles and inserting them into *M*.

A design aspect can be modeled from a variety of perspectives. In this paper we focus on two aspect views: static and interaction views. An aspect's static view defines the structural properties of the aspect. The interaction view specifies the interaction patterns associated with the aspect. We developed two types of Role Models: *Static Role Models* (SRMs) and *Interaction Role Models* (IRMs) [4]. SRMs define patterns of UML static structural models (e.g., Class Diagram patterns), while IRMs define UML interaction diagram patterns (e.g., Collaboration Diagram patterns). In the following subsections we show how SRMs can be used to define aspects from a static perspective and how IRMs can be used to define aspects from an interaction perspective. An aspect definition typically consists of a single SRM and one or more IRMs.

2.1 An Overview of SRMs

In this subsection we give an overview of SRMs and the next subsection shows how they can be used to define aspects. An example of a SRM defining an Observer-Subject static model pattern, and a conforming UML model are shown in Figure 1. The *base* of a role determines the type of UML model elements that can play the role. *Observer* and *Subject* are class roles (i.e., conforming elements are UML classes), while *Observes* is an association role (i.e., conforming elements are UML associations). This SRM characterizes class diagrams in which an observer class (i.e., a class that conforms to the *Observer* role) is linked, via an association that conforms

to the *Observes* role, to a subject class (i.e., a class that conforms to the *Subject* role). Association roles have multiplicity constraints expressed in template form (e.g.*[[n]],* where *n* is a range). If a multiplicity (an explicit set of ranges) is shown on an end of the association role then conforming associations must have this multiplicity on their corresponding association ends. In situations where the multiplicities can vary across associations realizing the association role, the association role has a multiplicity constraint expressed as templates of the form *[[p]]*, where *p* is constrained to be a range. Multiplicities in a realization of an association role containing these template multiplicities can be obtained by substituting values of *p* that satisfy the constraints. In Figure 1, *p* and *q* shown in the multiplicity constraint templates associated with the role *Observes* are not constrained, thus the association *obsTemp* conforms to this role.

A role can consist of two types of properties: *Metamodel-level constraints* and *Feature roles*. Feature roles are associated only with classifier roles in our work. Metamodel-level constraints are well-formedness rules that constrain the form of UML constructs that can realize the role. They are expressed as predicates (stated in the Object Constraint Language (OCL) [9,11]) over the UML metamodel elements. The metamodel-level constraint of *Subject* states that classes that conform to the role must not be abstract.

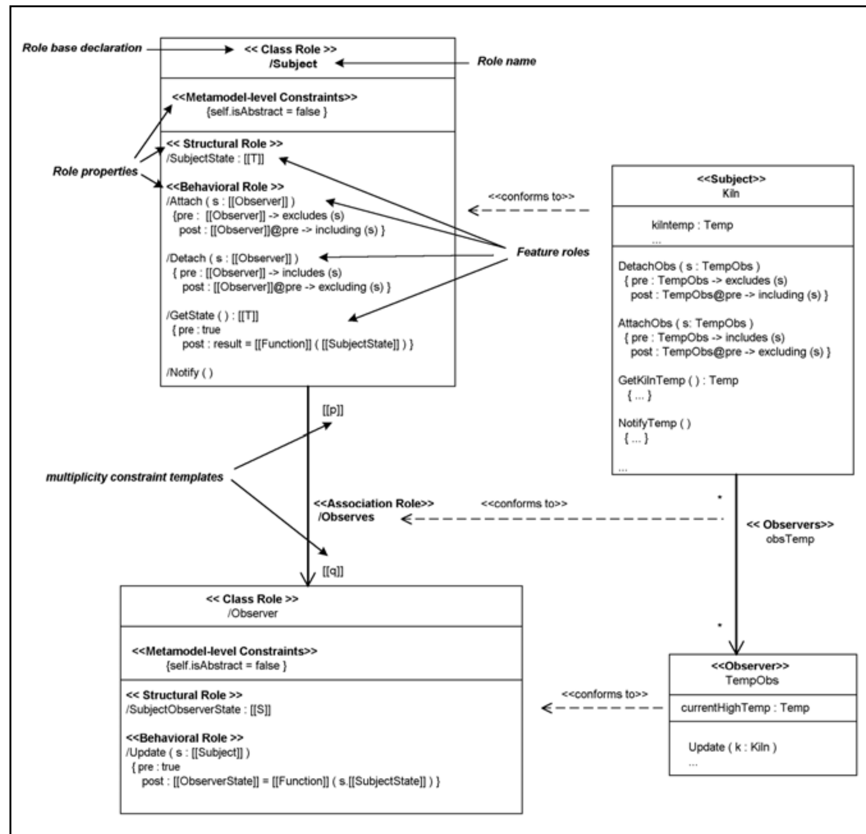

Figure 1. A SRM Characterizing Observer-Subject Static Models

Feature roles determine the features that must be present in the model elements that conform to the enclosing SRM role. A feature role consists of a name, and an optional property specification expressed as a *constraint template*. There are two types of feature roles: *Structural roles* (e.g., *SubjectState* in *Subject*) specify state-related features that are realized by attributes or value-returning operations in a SRM role realization, while *behavioral roles* (e.g., *Attach* and *Detach* in *Subject*) specify behaviors that can be realized by a single operation or method, or by a composition of operations or methods in a SRM role realization.

Checking that a UML class diagram conforms to the Role Model shown in Figure 1 can proceed as follows:

1. Relate UML model elements to roles: Conformance is always with respect to a particular mapping of model elements to roles. The mapping reflects that the model elements are intended to play the mapped roles. In Figure 1, we want to establish that the UML class diagram conforms to the Role Model under the following mapping:

a. *Kiln* to *Subject* (i.e., *Kiln* is intended to play the role of *Subject*): One must also specify what elements of *Kiln* are intended to play the feature roles specified in *Subject*. In this case the mapping is as follows: (1) *kilntemp* to *SubjectState* (results in *T* being instantiated to *Temp*), (2) *DetachObs* to *Detach* (results in *Observer* being instantiated to *TempObs*), (3) *AttachObs* to *Attach*, (4) *GetKilnTemp* to *GetState*, and (5) *NotifyTemp* to *Notify*.

b. *obsTemp* to *Observes* (results in *p* and *q* being instantiated to the range 0 to many).

c. *TempObs* to *Observer*: Like the mapping between *Kiln* and *Subject* this includes a mapping between the elements of *TempObs* that are intended to play feature roles in *Observer*. We do not give these mappings in this paper.

2. For each mapped model elements, establish that the model element conforms to the mapped role. This involves (1) checking that the metamodel-level constraints are true for the model element, (2) for class roles with features roles; checking that elements mapped to feature roles satisfy the feature role properties, and (3) for association roles; establishing that associations mapped to association roles have the properties specified by the corresponding role, that is, (i) the association end multiplicities can be obtained by instantiating the constraint templates with values that satisfy stated constraints; in this case p and q are not constrained and (ii) the classifiers at the association ends play the roles shown at the corresponding ends of the association role.

In the conforming model, *Kiln* conforms to *Subject* and *TempObs* conforms to *Observer*. The attribute *kilntemp* in *Kiln* conforms to the structural role *SubjectState* in *Subject* after substituting *Temp* for *T*. *AttachObs* and *DetachObs* conform to *Attach* and *Detach* respectively because their specifications imply the specifications obtained by substituting *TempObs* for *Observer* in the constraint templates of *Attach* and *Detach*. In this paper, a stereotype with a role name (e.g., << Subject >>) in a model construct is used to indicate that the model element plays the role.

The above process is carried out when one wants to determine whether an independently created model conforms to a Role Model. Role Models can also be used to generate a subset of its conforming models as outlined below:

- Create an instance of the role's base (e.g., for a class role, an instance is a UML class construct) that satisfies the metamodel-level constraints.
- For each structural role, generate an attribute and associated constraints by substituting a name for the role name, and instantiating type and constraint templates.

- For each behavioral role, generate an operation and associated constraints by substituting a name for the role name, and appropriately instantiating the pre- and post-condition constraint templates.

In this paper we use the generative capability of Role Models to weave aspects into primary models, as will be discussed in Section 3.

2.2 Modeling Aspects as SRMs

The static view of a security aspect is shown Figure 2 (only some behavioral role properties are shown in this SRM). This aspect is concerned with access control. Requests to a target object (*TargetComponent*) for the execution of an operation results in the following sequence of access authorization activities: (1) the target object determines whether the calling object (*ComponentOperator*) has permission to access the target object, (2) if the calling object has the right to access the object, then a check is made to determine whether the calling object has access to the called operation, (3) if the calling object has access to the operation then the operation is performed, else the operation is not executed (i.e., access to the operation is denied).

The static structure that supports the behavioral pattern described in the previous diagram is shown as an SRM in Figure 2. The SRM shows five class roles:

- *ComponentOperator*: Client classes play this role.
- *TargetComponent*: Server classes play this role.
- *AuthorizationRepository*: Classes representing repositories of client-server authorizations play this role.
- *OpAuthorizationRepository*: Classes representing repositories of client-operation authorizations can play this role.
- *AuthorizationEntry*: Classes defining the structure of authorization data play this role.

The class roles are related by association roles, indicating that associations that play the association roles connect the classes that play the class roles. Figure 2 shows multiplicity constraints as OCL statements. Some of the feature roles defined in the SRM are informally described below:

- *TargetComponent* has the following behavioral roles (the full definitions of the roles are not given for the sake of brevity): (1) *authID* characterizes behaviors that verify that a client (*ComponentOperator*) is authorized to access a target object, (2) *lockOut* characterizes behaviors that prevent client (*ComponentOperator*) objects from accessing target objects further if a corresponding *authID* behavior fails, and (3) *doOperation* is a behavior whose access is controlled by its containing object (*TargetComponent*), that is only authorized clients can execute the operation).

- *AuthorizationRepository* contains the behavior role *getAuth* that characterizes behaviors that search for a particular client-target authorization (*AuthorizationEntry*). Similarly, *OpAuthorizationRepository* has a behavior role *getOpAuth* that characterizes searches for a particular client-operation authorization.

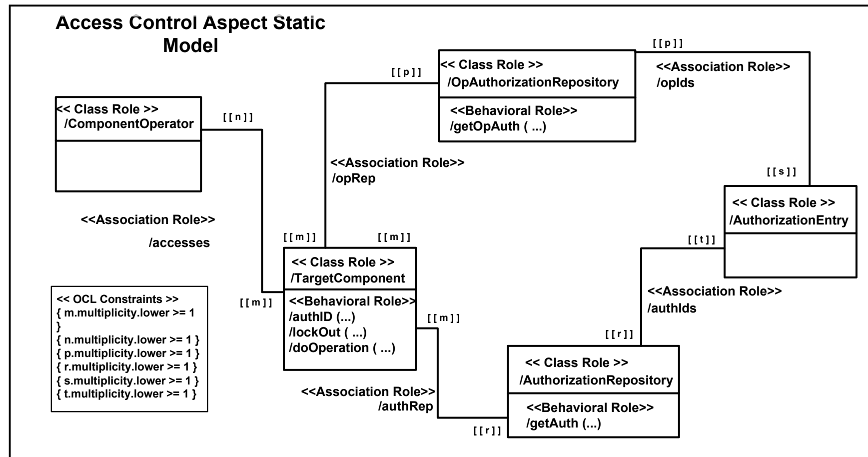

Figure 2. Static view of an access control aspect.

2.3 IRMs: Modeling the Dynamic View of Aspects

UML interaction models (e.g., Collaboration and Sequence Diagrams) are used to specify or describe interactions between system parts. Interactions characterized by an aspect are defined using a template form of interaction models called *Interaction Role Models* (IRMs) [4]. An IRM consists of template forms of UML collaboration roles, and messages. Instantiating these templates results in an Interaction Diagram that reflects the context in which the pattern of interaction will be used, for example, substituting values for message template parameters will result in sequence numbers that indicate the order of the interactions with respect to the context it is woven into. In this paper we describe only the Collaboration Diagram template form of IRMs.

One can describe the interactions that take place when a client object needs to interact with a server object under service access control as shown in Figure 3. In the *Authorization Interaction Aspect*, the client or *user* (an instance of a class that conforms to *ComponentOperator*) sends an operation request to the target (server object). This interaction is associated with the parameterized sequence specification *[[n]]*. On receipt of the message, the target must first determine whether *user* can access the target by searching the authorization repository for the authorization (sequence specification *[[n]].1*). If none is found (sequence specification *[[n]].2A*) then *user* does not have permission to access any operation on the target and further access is denied (sequence specifications *[[n]].2A.1,[[n]].2A.2*) . If *user* has access to the target (sequence specification *[[n]].2B*) then the target determines if *user* is

authorized to access the requested operation (sequence specification *[[n]].2B.1*). If the user's authorization to access the operation fails (sequence specification *[[n]].2B.2A*), *user* is locked out of further access to the target (sequence specification *[[n]].2B.2A.1*) and access is denied (sequence specification *[[n]].2B.2A.2*). If *user* is authorized to execute the operation, the operation is performed (sequence specification *[[n]].2B.2B*). Weaving this IRM into an interaction model involves substituting values for the template parameters, thus tying the IRM to a context.

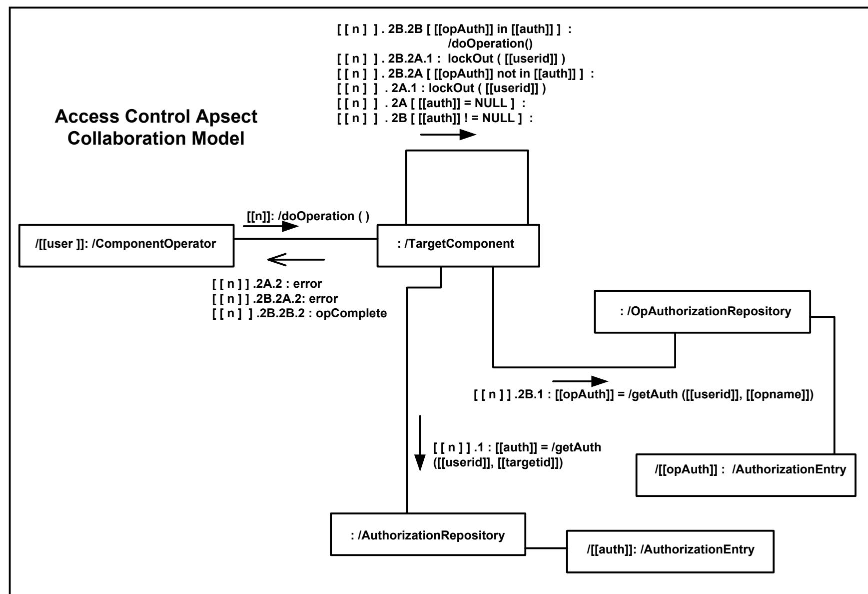

Figure 3. The Interaction View of the Access Control Aspect

3 Weaving Aspects into a Design Model: An Example

Weaving of an aspect into a primary model involves:

- Mapping primary model elements to the roles they are intended to play: Before the weaver can incorporate the pattern (aspect) into a primary model the modeler must first indicate the parts of the model the aspect is to be woven into. This can be accomplished by the modeler explicitly indicating the model elements that are intended to play the roles. Alternatively, the aspect can characterize the points into which it will be woven (as is done with pointcuts in AspectJ). For this paper we assume that the former approach is being used. We are currently developing support for the second approach. Note that not all model elements need be mapped to roles. Also, not all roles need be associated with a primary model element. Roles note associated with primary model elements indicate that new model elements must be created and added to the primary model as described later.

- Merging roles with primary model elements: Each model element that is mapped to a role has its properties matched with the properties contained in the aspect, and additional properties are generated from the role if deficiencies in the model element are found. For example, a class that is mapped to a class role that does not have attributes or operations that play structural and behavioral roles defined in the mapped class role is extended with attributes and operations generated from the role.
- Adding new elements to the primary model: Each role that is not mapped to a model element is used to generate a model element that is then added to the model.
- Deleting existing elements from the primary model: If a model element is mapped to a <<delete>> role (a <<delete>> role indicates that conforming elements must not exist in the model), then the model element is removed from the primary model.

In this section we illustrate the weaving of the access control aspect into a simple user management system.

3.1 Weaving Aspect into Static Models using SRMs

The static structure of the user management system is modeled by the Class Diagram shown in Figure 4. The user management system consists of (1) *Managers* that direct actions on user information, (2) *SystemMgmt* that carries out the action, (3) *userList* that contains information about users and (4) *userInfo* that contains information about individual users. We assume that there is a single *SystemMgmt* and a single *userList*.

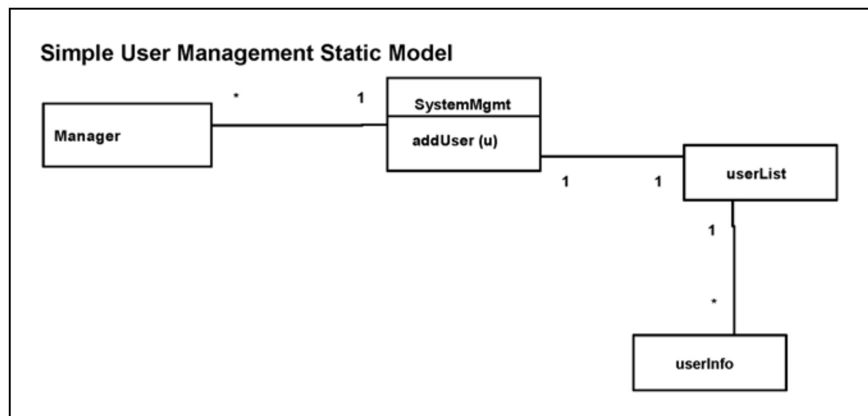

Figure 4. Design Class Diagram for the User Management System

Figure 5 shows the result of weaving the static view of the Access Control aspect into the original User Management Class Diagram. Weaving is accomplished as follows:

1. ***Mapping primary model elements to roles and merging roles with associated primary model elements:*** *Manager* is mapped to *ComponentOperator* and *SystemMgmt* is mapped to *TargetComponent*. *AuthorizationEntry* will be played by *userInfo*, and *userList* will play the *AuthorizationRepository* role. There is no primary model element that is intended to play the *OpAuthorizationRepository* role. Merging of roles to mapped primary model elements can result in modifications to the mapped primary model elements. For example, the *SystemMgmt* class must be augmented to play the role of *TargetComponent* by adding operations *authId* and *lockout*. The *userList* class is augmented to play the *AuthorizationRepository* role by adding a *getAuth* operation. The multiplicity of the association between *Manager* class (playing the *ComponentOperator* role), and *SystemMgmt* class (playing the *TargetComponent* role), must be changed from * to *1..**, that is, the aspect template *n* must be substituted with *. Other multiplicity substitutions are (1) *1..1* for *m*, (2) *1..1* for *r*, and (3) 1..* for *t*. The aspect roles played by model elements are shown in the woven static diagram in Figure 5 using stereotypes.

2. ***Adding model elements:*** In the user management system there is no existing model element that can play the role of *OpAuthorizationRepository*, so a model element that plays this role must be added to the primary model. In addition, new associations that can play the role of *opRep* and *opIds* must be added to the model between the new model element and *userInfo*, respectively.

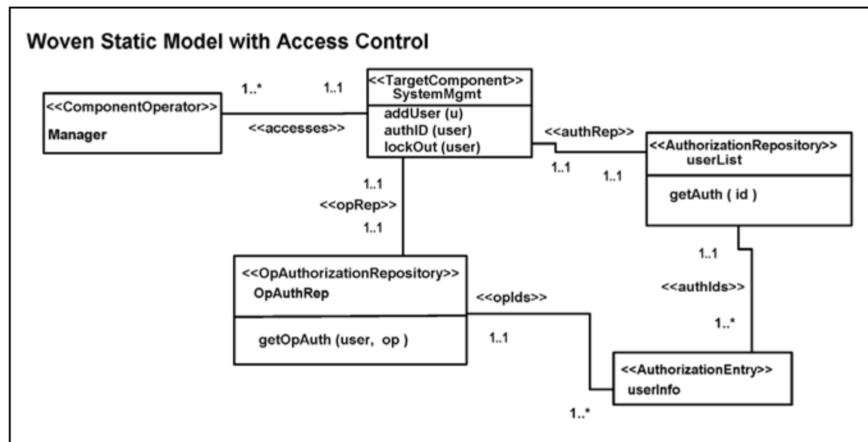

Figure 5. Result of Weaving the Access Control Aspect into the User Management System Class Diagram.

3.2 Weaving Aspects into Interaction Models using IRMs

Figure 6 shows the Collaboration Diagram describing the main design flow for adding a user to the system, and Figure 7 shows the result of weaving the Access Control aspect interaction view into this model. The weaving is accomplished as follows:

- The *doOperation* role is played by the *addUser* operation.
- The *auth* parameter is substituted with *uID*, and the *user* parameter is substituted with *m*.
- Interaction *1* in Figure 6 corresponds to interaction *[[n]]* in the IRM, thus *n* is set to 1. The nested interactions in Figure 6 are renumbered by replacing the outer sequence number (i.e., 1) by *1.2B.2B* (*1.2B.2B* is the last interaction specified in the IRM after substituting 1 for *n*). For example, sequence *1.1* in Figure 6 becomes sequence *1.2B.2B.1* in the woven model.

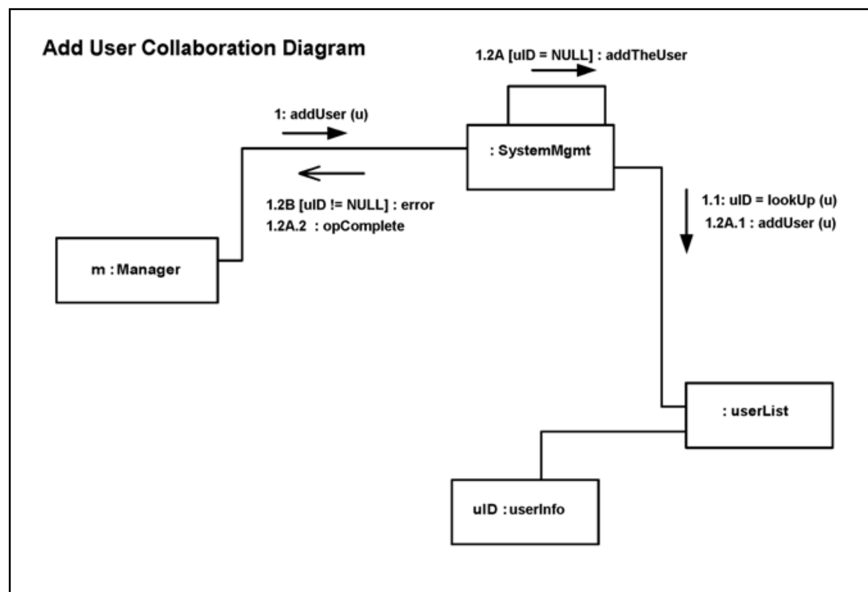

Figure 6. Add a User Collaboration Diagram

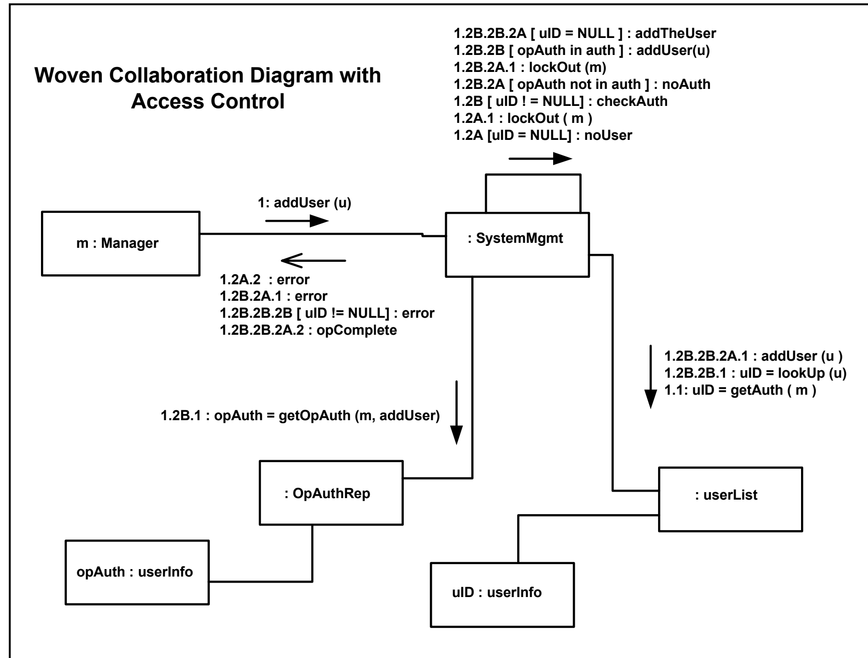

Figure 7. Add User Woven with Access Control Aspect

4 Conclusion

In this paper we (1) give an overview of how aspects can be expressed as Role Models that define patterns of design structures, and (2) illustrate, using a simple example, the weaving of aspects into UML models. We are currently evaluating the scalability of this approach on industrial strength software systems. Thus far we have used this approach to define and (manually) weave fault tolerance, and other security aspects into UML models of small applications from industry.

A system can be associated with multiple aspects – each aspect modeling a cross-cutting concern. There can be dependencies among the aspects and the order of weaving plays an important role. Consider, for example, auditing and authentication. Auditing is concerned with logging methods invoked by clients. Authentication is concerned with determining the identity of a user. Authentication invokes methods to verify a user. Thus, if auditing is woven before authentication, and the auditing roles are not associated correctly with authentication roles, the methods corresponding to the authentication aspect may not be logged. Thus, weaving an auditing aspect prior to an authentication aspect can result in an incorrectly woven model if the intent is to log authentication activities. Weaving the authentication aspect prior to the auditing aspect results in a correct woven model. (See [5] for details.) We are currently

investigating these types of dependencies among aspects in order to define weaving strategies.

Our experience also indicates that flexible tool support for weaving will greatly enhance the practicality of our approach. Flexibility in the weaving process is necessary because the manner in which aspects are woven is highly dependent on the form of the source models and the type of aspect being woven. Thus far we have been able to algorithmically describe the weaving of the types of aspects we have developed. We are currently developing a prototype tool that will support flexible weaving, by providing users with a language for describing (reusable) weaving strategies and weaving procedures ([8]).

References

1. **S. Clarke, W. Harrison, H. Ossher, and P. Tarr** 1999. Separating concerns through the development lifecycle. In *Proceedings of the third ECOOP Aspect-Oriented Programming Workshop*.
2. **S. Clarke and J. Murphy** 1998. 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)*.
3. **R. B. France, D. K. Kim, and E. Song**. 2002. Patterns as precise characterizations of designs. *Technical Report 02-101*, Computer Science Department, Colorado State University.
4. **R. B. France, D. K. Kim, E. Song, and S. Ghosh**. 2001. Using roles to characterize model families. *Proceedings of the Tenth OOPSLA Workshop on Behavioral Semantics: Back to the Basics*.
5. **G. Georg, R. B. France, I. Ray**. 2002. Designing high integrity systems using aspects. To appear in Proceedings of IICIS 2002.
6. **G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold**. 2001. Getting started with AspectJ. *Communications of the ACM* **44**(10, October):59-65.
7. **G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin**. 1997. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume LNCS 1231, June.
8. **F. Mekerke, G. Georg, R. B. France, R. Alexander**. 2002. Tool support for aspect-oriented design. To appear in OOIS 2002 Workshop on MDA.
9. **Object Management Group**. 2001. Unified Modeling Language V. 1.4. *http://www.omg.org*, September.
10. **J. Suzuki and Y. Yamamoto**. 1999. Extending UML with aspects: Aspect support in the design phase. *Proceedings of the third ECOOP Aspect-Oriented Programming Workshop*.
11. **J. Warmer and A. Kleppe**. 1999. The Object Constraint Language, Addison Wesley Longman, Inc.