Checking Policy Enforcement in an Access Control Aspect Model

Eunjee Song
Dept. of Computer Science
Baylor University
Waco, TX, USA
eunjee_song@baylor.edu

Robert France
Dept. of Computer Science
Colorado State University
Ft. Collins, CO, USA
france@cs.colostate.edu

Indrakshi Ray
Dept. of Computer Science
Colorado State University
Ft. Collins, CO, USA
iray@cs.colostate.edu

Hanil Kim
Dept. of Computer Education
Cheju National University
Jeju, Korea
hikim@cheju.ac.kr

ABSTRACT

From a software design perspective, access control policies are requirements that must be addressed in a design. For example, access control policies are constraints that determine the type of access authorized users have on information resources. In this paper, we show how one can formulate access control policies as a policy model, formulate an access control aspect model that enforces policies as an aspect, and verify whether the aspect model enforces the given policies or not. As an access control policy example, we use Role-Based Access Control (RBAC).

Categories and Subject Descriptors

D.2.1 [**Requirements/Specifications**]: Languages, Methodologies; D.2.2 [**Design Tools and Techniques**]: Object-oriented design methods; D.2.4 [**Software/Program Verification**]: Validation

General Terms

Design, Security, Languages, Verification

1. INTRODUCTION

Rigorously establishing that a design enforces specified access control policies requires checking that the design elements describing the access control feature enforce the policies. If the elements are scattered across a design and tangled with other design elements then verification of the policy enforcement becomes difficult. Aspect-oriented modeling (AOM) techniques have been advocated as solutions for making the above tasks easier by isolating crosscutting access control features from other features in the design.

In our approach, access control features are modeled as aspect models using the RBML template notation. Access control policies are expressed in terms of UML class diagrams with the Object Constraint Language (OCL) [26]constraints. Verifying access control aspect models against given policy requirements required us to define realization mapping pairs to show the relationship of concepts in two models. These mapping rules were used to transform the OCL invariants in the policy model into invariants expressed in terms of aspect model concepts. A difficulty arose when relating generic and domain-specific concepts: An aspect model describes generic concepts while a policy model expresses domain-specific concepts. To tackle this problem we obtained a prototypical UML model from the generic aspect model. The prototypical model is one of the most-general context-specific models that can be instantiated from the aspect model. It is obtained by simply replacing parameters with the parameter names. The multiplicity parameters are replaced with the weakest form of multiplicities allowed by the aspect model. This prototypical model is used to establish that an aspect model enforces access control policies expressed as UML class models.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CTIC'07 Anaheim, California, USA
Copyright 2002 ACM X-XXXXX-XX-X/XX/XX ...$5.00.

2. BACKGROUND

In the AOM approach, features that crosscut a design can be described by aspect models if their distributed parts have common characteristics. In these cases the cross-cutting features can be isolated and described as patterns. Aspect models are descriptions of patterns and composition of aspect and primary models involves incorporating instantiations of the patterns into specified parts of the primary model. An overview of composition in the AOM approach is shown in Fig. 1. An AOM design model consists of the following artifacts: (1) A *primary model* which describes application features not described by aspect models, (2) A set of *aspect models*, where each model describes a pattern that is a generic (parameterized) description of a cross-cutting feature, (3) A set of *bindings* that determines the pattern instantiations that will be produced and composed with the primary model, and (4) A set of *composition directives* that determines how aspect models are composed with the primary model.

Before an aspect model can be composed with a primary model, the aspect model must be instantiated in the context of the application domain. An instantiation is obtained by binding elements in the aspect model to elements in the application domain. The result is called a *context-specific aspect model*. A context-specific aspect model is produced for each part of the primary model into which the aspect feature is to be incorporated. (For further details, refer

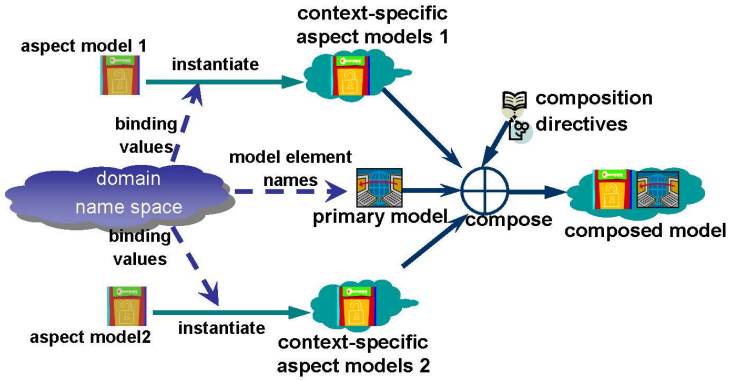

Figure 1: An overview of the AOM approach

to [12].)

In our work, primary and context-specific aspect models are expressed in the Unified Modeling Language (UML) [24]. The UML is an Object Management Group (OMG) standard modeling language. A system is described in the UML using multiple diagrams that present different views of the system. In this paper we use only two types of diagrams: Class diagrams specify static structure and sequence diagrams describe how objects interact to accomplish tasks. A UML class diagram consists of a set of classifiers (for example, classes, interfaces) and their relationships (for example, association, generalization). Classes may have attributes and operations. In this paper, operation specifications and constraints on attributes are expressed using the OCL [26]. A UML sequence diagram presents a behavioral view that focuses on the interactions that take place between class objects when they collaborate to accomplish a specific task. The interactions are expressed in terms of lifelines representing objects and messages that are passed between objects.

Aspect models consist of template forms of UML diagrams. In this paper, aspect models consist of class and sequence diagram templates. Instantiating an aspect model involves binding template parameters to names in an application domain namespace (see Fig. 1). A class diagram template consists of parameterized elements, such as, relationship templates and class templates that consist of attribute templates and operation templates. Attribute templates can be associated with OCL constraint templates that produce constraints that restrict attribute values when instantiated. Similarly, operation templates can be associated with OCL pre- and postcondition templates that produce operation specifications when instantiated.

Role-Based Access Control (RBAC) [9] is used to protect information targets (henceforth referred to simply as targets) from unauthorized users. To achieve this goal, RBAC specifies and enforces different kinds of constraints. Core RBAC defines the properties that must be present in any RBAC application. Core RBAC requires that users be assigned to roles, roles be associated with permissions (approval to perform an operation on a target), and that users acquire permissions through their associated roles. For example, in a banking application, users can be assigned to roles such as loan officer and teller, where a loan officer has permission to issue loans to customers.

Sandhu *et al.* [22] have specified four conceptual RBAC models. Core RBAC (RBAC_0) is the most basic model. In core RBAC, a user can establish a session to activate a subset of roles to which the user is assigned. Hierarchical RBAC (RBAC_1) includes RBAC_0 and introduces *role hierarchies*. Hierarchies structure roles to reflect an organization's lines of authority and responsibility and they are specified using inheritance of roles. RBAC_2 includes RBAC_0 and introduces constraints to restrict the assignment of users or permissions to roles, or the activation of roles in sessions. Constraints are used to specify application dependent conditions, such as, separation of duties. RBAC_3 combines both RBAC_1 and RBAC_2, thus providing role hierarchies as well as constraints.

Core RBAC does not place any constraint on the cardinalities of the user-role assignment relation or the permission-role association. In core RBAC each user can activate multiple sessions; however, each session is associated with only one user. The operations that a user can perform in a session depend on the roles activated in that session and the permissions associated with those roles.

Hierarchical RBAC adds role hierarchies to Core RBAC. Role hierarchies define inheritance relation among the roles in terms of permissions and user assignments. If role *r1* inherits role *r2* then all permissions of *r2* are also permissions of *r1* and all users of *r1* are also users of *r2*. There are no cardinality constraints on the inheritance relationship. The inheritance relationship is reflexive, transitive and anti-symmetric.

Static Separation of Duty (SSD) relations are used to define conflicting roles: If a user is assigned to roles that conflict then there is a conflict of interest with respect to permissions assigned to the user via the roles. SSD relations between roles constrain how users are assigned to roles: Membership in one role that takes part in an SSD relation prevents the user from being a member of the other role. The SSD relationship is symmetric, but it is neither reflexive nor transitive. SSD relations may exist in the absence of role hierarchies (referred to as SSD RBAC or RBAC_2), or in the presence of role hierarchies (referred to as hierarchical SSD RBAC or RBAC_3). The presence of role hierarchies complicates the enforcement of the SSD relations: Before assigning users to roles not only should one check the direct user assignments but also the indirect user assignments that occur due to the presence of the role hierarchies.

Fig. 2 shows a model of hierarchical SSD RBAC that consists of: (1) a set of users ($USERS$) where a user is an intelligent autonomous agent , (2) a set of roles ($ROLES$) where a role is a job function , (3) a set of sessions ($SESSIONS$) where a user establishes a session during which he/she activates a subset of the roles assigned to him/her, (4) a set of targets ($TGTS$), where a target is an entity that contains or receives information, (5) a set of opera-

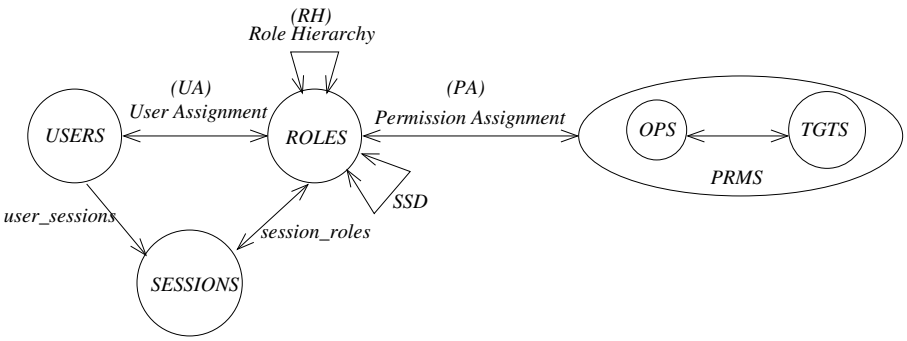

Figure 2: Hierarchical SSD Role-Based Access Control

tions types (OPS) where an operation describes a service provided by the application, and (6) a set of permissions ($PRMS$) where a permission is an approval to perform an operation on targets. The cardinalities of the relationships are indicated by the absence (denoting one) or presence of arrow heads (denoting many) on the corresponding associations. For example, the association of user to session is one-to-many. All other associations shown in the figure are many-to-many. The association labeled *Role Hierarchy* defines the inheritance relationship among roles. The association labeled *SSD* specifies conflicting roles.

3. FORMULATING ACCESS CONTROL POLICIES

Policies are expressed in terms of UML class diagrams with OCL constraints. We define this form of diagrams with constraints as a policy model. A policy model is obtained by analyzing the given policy statement. For example, the following shows how the Role-Based Access Control (RBAC) policy specified in [9] can be expressed in a policy model. The access control policy statement must describe under what condition a user does or does not have permission to access a target to perform a certain type of operations in a session. The RBAC policy is stated as follows:
[P$_{RBAC-1}$]: If a user u has permission to access a target t to perform operations of type *op* in a session s, then there exists a role r with the following properties:

- r has permission to access t to perform operations of type *op*,
- r is an authorized role for u, and
- r is currently activated in s.

[P$_{RBAC-2}$]: Roles activated in a session must be a subset of the roles assigned to the user of the session.

A class model that is required to describe the RBAC policy is shown in Fig. 3. The *User* class and *Session* class in the policy model represent a set of users and a set of sessions respectively. The *Role* class in the policy model represents a set of roles that a user can play. The *Permission* class in the policy model represents pairs of sets where one part of a pair is a set of *Target* instances and the other is a set of *OperationType* instances.

To formally specify the first part of the policy statement, P$_{RBAC-1}$, we define the derived *hasPermission* relationship between *Session* and *Permission* that links sessions with their permissions as stated in the RBAC policy P$_{RBAC-1}$. The OCL statements that define the RBAC policy P$_{RBAC-1}$ are given below:

P$_{RBAC-1}$:
context Session **inv**:
hasPermission→forAll(p:Permission|
activatedRole→exists(r|r.allowedPermission→includes(p))
and sessionUser.authorizedRole→exists(r|
r.allowedPermission→includes(p)))
where a derived association *hasPermission* is defined as follows:

context Session:: hasPermission : Set(Permission)
derive: activatedRole.allowedPermission

The policy statement P$_{RBAC-2}$ is expressed in the OCL as follows:

context Session **inv**:
sessionUser.authorizedRole→includesAll(self.activatedRole)

These two policy properties must be true in the model that enforces the RBAC policy. The policy model described above is used for showing the policy enforcement that will be described later in Section 5.

4. FORMULATING ACCESS CONTROL MODELS AS ASPECT MODELS

An access control model is formulated as an aspect model using the UML model template notation [11]. For example, Fig. 4 shows the class diagram template of the hierarchical SSD RBAC aspect model that was illustrated in Fig. 2. The class diagram template of the hierarchical SSD RBAC aspect model consists of a set of users, a set of roles, a set of user sessions, a set of targets, a set of operation types, and a set of permissions as described earlier in Section 2. Users are assigned to roles, roles are associated with permissions, and users acquire permissions by being members of roles. Association templates, such as *UserAssignment* and *UserSessions* produce associations between instantiations of the class templates they connect. The multiplicity "1" on the *User* end of the *UserSessions* template is strict: a session can only be associated with one user. Only roles that are assigned to the user of a session can be activated for that user in the same session. The following invariant shown in Fig. 4 specifies the above constraint:

context Session **inv**:
self.UserSession.GetAuthorizedRoles()→
includesAll(self.GetAllActiveRoles())

The operations that a user can perform in a session depend on the roles activated in that session and the permissions associated with

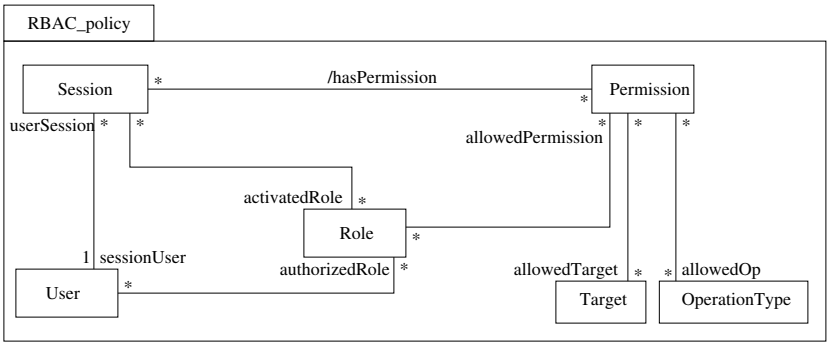

Figure 3: An RBAC policy model

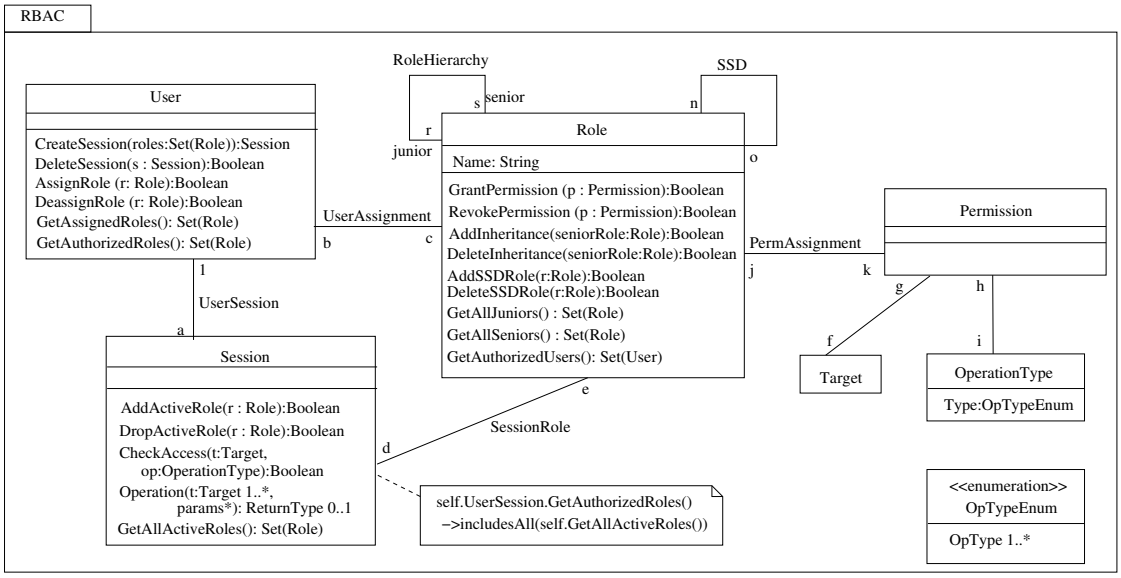

Figure 4: The class model template view of the RBAC aspect model

those roles. The operation template $Operation$ in the $Session$ template represents operations under access control. Instantiating $Operation$ produces an operation that describes a behavior that is performed on target elements (e.g., a withdraw operation on an account element in a banking system). The class template $OperationType$ contains an attribute template with a type parameter ($Type$). Instances of $Type$ may be any of the user-defined enumeration literals instantiated from $OpType$ which is an attribute template of the enumeration template *OpTypeEnum*. Table 1 gives an overview of operation templates in each class template shown in Fig. 4.

All the user-defined names[1] in the diagram are template parameters that must be bound to values when instantiating the aspect model. Note that we used the symbol "|" to indicate template parameters in [23]. In this paper, however, we do not show the template parameter indicator symbol "|" for the readability. For example, the parameter template $params$ in the $Operation$ template shown in $Session$ of Fig. 4 must be bound to a list of zero or more operation arguments as indicated by the "$*$" following the parameter name.

An instantiation of $Operation$ can have one or more arguments representing target elements (*t:Target 1..**) and zero or more other arguments (*params **). For example, the operation *transfer (from: Account, to:Account, amount:Integer)* can be obtained from $Operation$ using the following bindings for template parameters: $<transfer \mapsto Operation,\ Account \mapsto Target,\ amount\ :\ Integer \mapsto params>$, where $(value \mapsto parameter)$ represents a binding that is done by providing a value for a parameter. For more detailed specification of $Operation$ and its instantiation examples, refer to [23]. Table 1 gives an overview of operation templates in each class template.

The $CheckAccess$ operation template in $Session$ is intended to enforce the RBAC policy P_{RBAC}. The OCL specification associated with $CheckAccess$ is given below:

context Session::CheckAccess(t:Target,
op:OperationType):Boolean
pre: true
post: result =

[1]In Fig. 4, for example, the following values are examples of ones that are not template parameters: UML keywords such as Boolean, String, Set, and enumeration, OCL keywords such as self and includesAll , and all variable names such as s for a Session object, t for a Target object.

```
self.GetAllActiveRoles().Permission → exists (p |
    p.Target → includes(t) and
    p.OperationType → includes(op))
```

The postcondition of $CheckAccess$ states the following: *If there exists an activated role with the required permission, the CheckAccess operation returns true, otherwise it returns false.*

The behavior described by $CheckAccess$ is called whenever an operation under access control (represented by $Operation$) is invoked. When the behavior described by $Operation$ is invoked it first checks whether the caller has permission to perform the operation on the target objects (the set of targets represented by the parameter t in the $Operation$ template) by invoking the behavior described by $CheckAccess$ for each target object. Invariants and operation specifications associated with other elements in the RBAC aspect model are given in the Appendix A.

5. VERIFYING POLICY ENFORCEMENT

In this section, we illustrate how one can apply our approach to verifying the policy enforcement of an aspect model using an RBAC aspect model example. Fig. 5 shows the policy enforcement verification process we developed.

policy model and aspect model : These models are two inputs to the verification process. Note that an aspect model is a generic description of model families and it is specified using UML diagram template notation while a policy model is stated in terms of UML class diagram concepts with constraints that express restrictions given in a policy statement. [2]

realization mapping rules : To verify that an aspect model describing an access control feature enforces targeted access control policies, we define realization mapping rules. A realization mapping in our approach is a set of pairs that defines how elements in the policy model are realized in the generalized aspect model. These mapping rules are used to transform the OCL invariants in the policy model into invariants expressed in terms of aspect model concepts.

the most general context-specific aspect model : We obtain a context-specific aspect model that is described at the M1 level so that OCL transformations can be performed between two M1-level models (i.e., one is a policy model and the other is a context-specific aspect model). Note that a context-specific aspect model is obtained by providing one or more domain-specific parameter values for each parameter in an aspect model (refer to Fig. 1). For example, for a banking application domain, $BankSession$ and $BankRole$ can be provided for the class templates $Session$ and $Role$ in Fig. 4 respectively. Since no domain-specific value set is available yet, we obtain the most general context-specific aspect model of an aspect model by providing a general set of parameter values for its template parameters as described below:

- provide a current parameter name as a parameter value for each template (e.g., class, attribute, operation, association templates) except multiplicity parameters on ends of association templates.
- provide a multiplicity indicator "$*$" that represents "zero or more" for each multiplicity template in an alphabet letter (i.e., the weakest form of multiplicity for an unconstrained multiplicity parameter).

[2]In other words, an aspect model in this paper is defined at the M2 (metamodel) level of the four UML layers while a policy model is defined at the M1 (model) level.

transformed policy model invariants : Using realization mapping rules, invariants of an policy model are transformed into invariants expressed in terms of aspect model concepts.

enforcement verification : Verifying that the aspect model enforces the policy model involves establishing that transformed invariants hold in the aspect model.

When an RBAC policy model in Fig. 3 and an RBAC aspect model in Fig. 4, for example, are given as inputs to our verification procedure, realization mapping rules are defined in the form of set of realization pairs. A realization pair can be defined explicitly or implicitly. An explicit pair has the form {`policyElem`, *aspectElem*}, indicating that the policy model element `policyElem` is realized by an element $aspectElem$ in the aspect model. The explicit realization pairs used in the verification of the RBAC aspect model reflect the simple one-to-one relationship between the major concepts shown in the policy model and the aspect model.

The following shows explicitly defined in realization pairs in the RBAC realization mapping:
{`Session`, *Session*}, {`User`, *User*}, {`Role`, *Role*}, {`Permission`, *Permission*}, {`Target`, *Target*}, {`OperationType`, *OperationType*}.

In the implicit form, one or both items in the pair are expressions that are evaluated. For example, the following is the realization pair that describes how the `UserSession` association in the RBAC policy model is realized in the aspect model:

{`u:User.userSession`, *u:User.UserSession*}.

In the above `u:User.userSession` represents the set of session objects associated with a user u in the policy model, and *u:User.UserSession* represents the set of sessions associated with the realization of u in the RBAC aspect model.

The following are implicitly defined in realization pairs in the RBAC realization mapping:

{`u:User.authorizedRole`,
    u:User.GetAuthorizedRoles()},
{`r:Role.allowedPermission`,
    r:Role.Permission},
{`s:Session.sessionUser`,
    s:Session.UserSession},
{`s:Session.activatedRole`,
    s:Session.GetAllActivatedRoles()},
{`p:Permission.allowedTarget`,
    p:Permission.Target},
{`p:Permission.allowedOp`,
    p:Permission.OperationType}.

The permissions associated with a session via the `hasPermision` association end in the policy model, is realized by the set of permissions associated with roles activated in a session. If we model the relationship between permissions and sessions in the RBAC aspect model as a derived association between *Session* and *Permission* that is named *SPermission*, then the following realization pair maps `hasPermission` links to *SPermission* links:
{`s:Session.hasPermission`, *s:Session.SPermission*}.

Fig. 6 shows the most general context-specific RBAC class model that is obtained from the RBAC aspect model given in Fig. 4. *Session* and *Role*, for example, are provided as parameter values to instantiate the class templates *Session* and *Role* respectively and *CheckAccess* is provided as a parameter value for an operation template *CheckAccess*. Therefore, the most general context-specific RBAC aspect model has an operation named *CheckAccess* in a

Table 1: The list of operation templates defined in RBAC.

Operation Template	Description
$User$ Class Template	
$CreateSession$	creates a new session and activates a default role set; creates a $UserSession$ link between the user and the session
$DeleteSession$	deactivates roles that are activated in the given session and deletes that session; deletes a $UserSession$ link
$AssignRole$	creates a $UserAssignment$ link between the user and the given role
$DeassignRole$	deletes a $UserAssignment$ link
$GetAssignedRoles$	returns the set of roles directly assigned to the user as well as those roles that are inherited by the directly assigned roles
$GetAuthorizedRoles$	returns the set of roles directly assigned to the user as well as those roles that are inherited by the directly assigned roles (junior roles)
$Session$ Class Template	
$AddActiveRole$	creates a new $SessionRole$ link between the session and the role that is one of roles in the authorized role set.
$DropActivatedRole$	deletes a $SessionRole$ link between the session and the given role.
$GetAllActiveRoles$	returns a set of all roles which are activated for that session and all their junior roles
$CheckAccess$	determines whether an access should be granted or not
$Operation$	invokes the operation under access control if the caller has permission to each target
$Role$ Class Template	
$GrantPemission$	creates a new $PermAssignment$ link between the role and the given
$RevokePemission$	permission deletes a new $PermAssignment$ link
$AddInheritance$	adds a link *RoleHierarchy* to a senior role
$DeleteInheritance$	deletes a link *RoleHierarchy* to a senior role
$AddSSDRole$	creates a new SSD link between roles
$DeleteSSDRole$	deletes an SSD link
$GetAllJuniors$	returns a set of roles which consists of direct junior roles and all other junior roles acquired by the transitive closure relation in a role hierarchy
$GetAllSeniors$	returns a set of roles which consists of direct senior roles and all other senior roles acquired by the transitive closure relation in a role hierarchy
$GetAuthorizedUsers$	returns the set of users that are assigned to the role and its senior
$CheckAccess$	roles determines whether an access should be granted or not
$Permission$ Class Template	
$CheckAccess$	determines whether an access should be granted or not

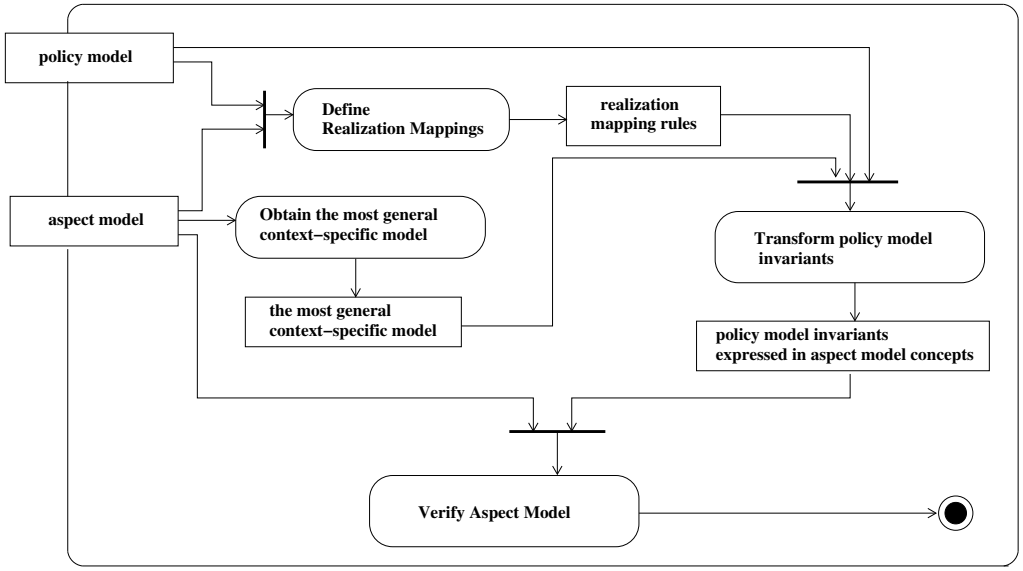

Figure 5: Verifying a design aspect model against its policy model.

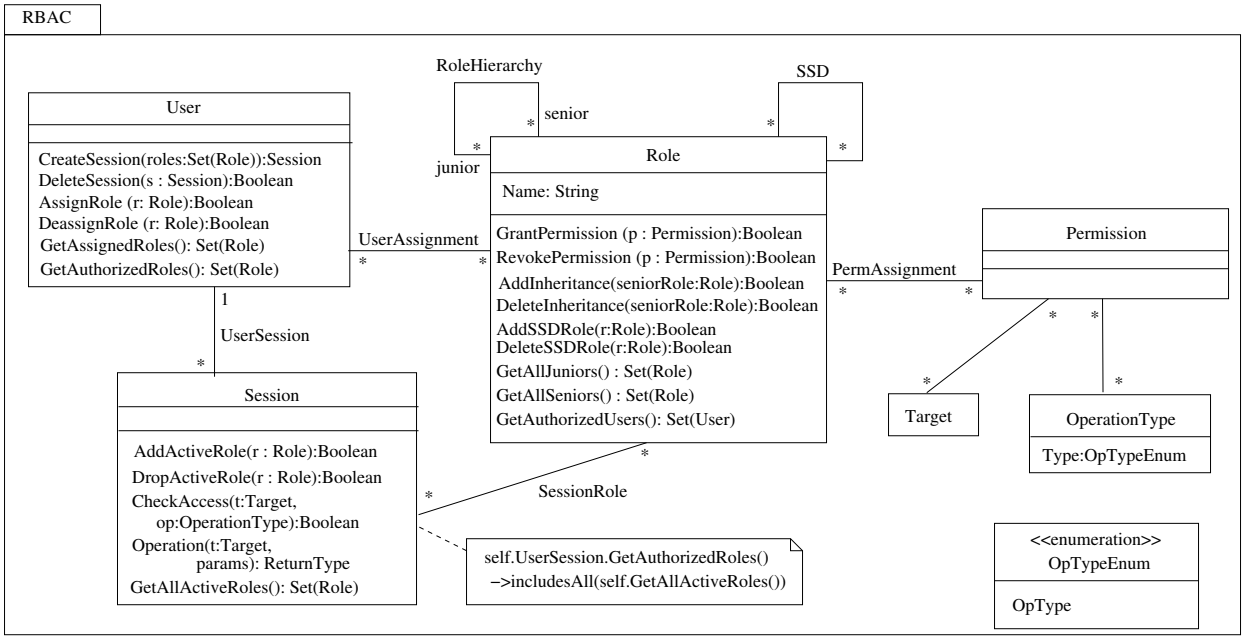

Figure 6: The most general context-specific RBAC aspect model

class *Session*. For each multiplicity parameter expressed in an alphabet letter (e.g., the multiplicity parameter a on $Session$ end of $UserSession$ association of Fig. 4), we provide the multiplicity indicator "$*$".

Using realization mappings shown above, the RBAC policy constraints P_{RBAC-1} and P_{RBAC-2} are transformed to the following:

P$_{RBAC-1-transformed}$:
context Session **inv**:
SPermission→forAll(p:Permission|
GetAllActiveRoles()→exists(r|
r.Permission→includes(p)) and
UserSession.GetAuthorizedRoles()→exists(r|
r.Permission→includes(p)))

where a derived association $SPermission$ is defined as follows:

context Session:: SPermission : Set(Permission)

derive: self.GetAllActiveRoles().Permission

P$_{RBAC-2-transformed}$:
context Session **inv**:
UserSession.GetAuthorizedRoles()→
includesAll(GetAllActivatedRoles())

The behavior described by the $CheckAccess$ template must enforce the transformed RBAC policy properties, $P_{RBAC-1-transformed}$ and $P_{RBAC-2-transformed}$. We show the policy enforcement below:

- From the definition of $SPermission$, if there is a permission in the set of permissions represented by $SPermission$ that grants the access, an invocation of a $CheckAccess$ operation with an argument t representing the target operation and an argument *op* representing the operation type, must return true; otherwise it returns false.
- Note that $s.CheckAccess(t, op) = true$ when the following expression in the postcondition of $CheckAccess$ is true:
  self.GetAllActiveRoles().Permission → exists (p |
  p.Target → includes(t) and
  p.OperationType → includes(op))
- $P_{RBAC-2-transformed}$ holds because of the invariant shown in Fig. 6.
- Showing that $P_{RBAC-2-transformed}$ holds in the RBAC aspect model requires one to show that the following expression is true for all permissions in the set of permissions represented by $SPermission$ (i.e., permissions that grant the access):
  GetAllActiveRoles()→exists(r|
  r.Permission→includes(p)) $--$ (1)
  and
  UserSession.GetAuthorizedRoles()→exists(r|
  r.Permission→includes(p)) $--$ (2)
  The postcondition of $CheckAccess$ must return true for each permission p that grants the access. Therefore, the following expression holds:
  GetAllActiveRoles().Permission→exists(p) $--$ (3)
  which is equivalent to
  GetAllActiveRoles().exists(r|
  r.Permission→includes(p))
  Therefore, (1) holds.
  From the expression (3) and $P_{RBAC-2-transformed}$, we know the following expression holds:
  UserSession.GetAuthorizedRoles().Permission→exists(p)
  which is equivalent to
  UserSession.GetAuthorizedRoles()→exists(r|
  r.Permission→includes(p))
  Therefore, (2) holds.
- Therefore, two transformed RBAC policy properties, $P_{RBAC-1-transformed}$ and $P_{RBAC-2-transformed}$, holds in RBAC aspect model.

6. RELATED WORK

Approaches to specifying and analyzing access control features that are based on sophisticated mathematical concepts (e.g., [4, 5, 6, 8, 16]), formally stated, allow one to check that developed access control features enforce required polices. In practice, however, applying mathematically-based formal specification techniques can be difficult because of the high degree of mathematical skill needed. Therefore, a representation that can be analyzed without sacrificing understandability and usability is desirable.

In this regard, Tidswell and Jaeger [25] propose an approach to visualizing access control constraints. They point out the need for visualizing constraints and the limitations of previous work (e.g., [2, 20, 21]) on expressing constraints. Another effort to graphical specification of RBAC is proposed by Koch *et al.* [18]. In their approach, RBAC policies are represented by graph transformations. Verification of RBAC policies is carried out by showing that graphical constraints do not occur in the graph specifying RBAC policies. The drawback of these two approaches is that they created a new notation for specifying constraints and it is not clear how the new notation can be integrated with other widely-used design notations. The approach described in this research utilizes notations from a standardized modeling language and also integrates the policy specification activity with design modeling activities. In our approach, access control policies and access control models are expressed using UML-based notations. Therefore, UML tools can be used to specify them.

There has been some work on using the UML to model security features (e.g., see [1, 7, 10, 15, 17, 19]. Chan and Kwok [7] model a design pattern for security that addresses asset and functional distribution, vulnerability, threat, and impact of loss. Lodderstedt *et al.* [19] propose SecureUML and define a vocabulary for annotating UML-based models with information relevant to access control. Jürjens [17] models security mechanisms based on the multi-level classification of data in a system using an extended form of the UML called UMLsec. The UML tag extension mechanism is used to denote sensitive data. Statechart diagrams model the dynamic behavior of objects, and sequence diagrams are used to model protocols. Deployment diagrams are also used to model links between components across servers. UMLsec is fully described in a UML profile. These approaches mainly focus on extending the UML notation to better reflect security concerns. The approach described in this research complements the UMLsec by capturing access control policies in patterns that can be reused by developers of secure systems. France *et al.* [10] and Georg *et al.* [15] have shown how concerns can be modeled as aspects, expressed as structural and behavioral patterns specifications, and composed with designs expressed in the UML (e.g., security concerns [13, 15], and authentication and auditing [14]). Our work complements their work by providing a technique to check the policy enforcement in an aspect model. Ahn *et al.* [1] also proposes a framework that can be utilized for representing and validating the RBAC design model using UML and OCL. In their work, they translate RBAC authorization constraints that are formally specified in Role-based Constraints Language 2000 (RCL2000) [3] into OCL constraints for their model validation. While their framework is proposed for validating the UML design models against formally specified authorization constraints, our work has more focuses on design verification against policy requirements that are represented by policy models in this paper.

7. CONCLUSION

In this paper, we have given an AOM approach to modeling access control features that enforce access control policies separately as aspects and checking whether modeled aspects enforces given policies. In our approach, access control features are modeled as aspect models using the UML model template notation [11]. To check that an aspect model enforces expected access control policies, one needs to express access control policies in the forms that access control aspects are specified. In our approach, we propose to create a *policy model* that is a set of policies stated in terms of UML class diagram concepts with constraints and to obtain a context-specific aspect model that is the most general form of instantiation from the aspect model.

Verifying access control aspect models against given policy requirements required us to define realization mapping pairs to show the relationship of concepts in two models. These mapping rules

were used to transform the OCL invariants in the policy model into invariants expressed in terms of aspect model concepts. A difficulty arose when relating generic and domain-specific concepts: An aspect model describes generic concepts while a policy model expresses domain-specific concepts. To tackle this problem we obtained a prototypical UML model from the generic aspect model. The prototypical model is one of the most-general context-specific models that can be instantiated from the aspect model. It is obtained by simply replacing parameters with the parameter names. The multiplicity parameters are replaced with the weakest form of multiplicities allowed by the aspect model. This prototypical model is used to establish that an aspect model enforces access control policies expressed as UML class models. We are currently working on applying the suggested approach to a small on-line shopping cart example and plan to extend our case study to more complex system examples.

Security policies including access control policies may change during a mission and evolving systems must keep pace with those changes. Our approach can be extended to evolving security policies. By encapsulating security concerns (in aspects), changes can be made in the aspect, and the effects can be incorporated into the models through composition. Therefore, we are currently investigating how aspects can be extracted from the composed model and modified so that the required changes can be applied to aspects and then eventually applied to the application itself using a verifiable aspect composition approach proposed in [23].

8. ACKNOWLEDGEMENTS

The work was supported in part by AFOSR under Award No. FA9550-04-1-0102 and by the MIC(Ministry of Information and Communication), Korea under the ITRC(Information Technology Research Center) support program supervised by the IITA(Institute of Information Technology Advancement) (IITA-2007-C1090-0701-0040).

9. REFERENCES

[1] G.-J. Ahn and H. Hu. Towards realizing a formal rbac model in real systems. In *SACMAT '07: Proceedings of the 12th ACM symposium on Access control models and technologies*, pages 215–224, New York, NY, USA, 2007. ACM Press.

[2] G. J. Ahn and R. Sandhu. The RSL99 Language for Role-Based Separation of Duty. In *Proceedings of the 4th ACM Workshop on Role-Based Access Control*, pages 43–54, Fairfax, VA, 1999.

[3] G.-J. Ahn and R. S. Sandhu. Role-based authorization constraints specification. *ACM Trans. Inf. Syst. Secur. (TISSEC)*, 3(4):207–226, 2000.

[4] S. Barker. Security Policy Specification in Logic. In *Proceedings of the International Conference on Artificial Intelligence*, pages 143–148, Las Vegas, NV, 2000.

[5] S. Barker and A. Rosenthal. Flexible Security Policies in SQL. In *Proceedings of the 15th Annual IFIP WG 11.3 Working Conference on Data and Applications Security*, Niagara-on-the-Lake, Canada, 2001.

[6] E. Bertino, P. Bonatti, and E. Ferrari. TRBAC: A Temporal Role-Based Access Control Model. In *Proceedings of the 5th ACM Workshop on Role-Based Access Control*, pages 21–30, Berlin, Germany, 2000.

[7] M. T. Chan and L. F. Kwok. Integrating Security Design into the Software Development Process for E-commerce Systems. *Information Management and Computer Security*, 9(2-3):112–122, 2001.

[8] F. Chen and R. Sandhu. Constraints for Role-Based Access Control. In *Proceedings of the 1st ACM Workshop on Role-Based Access Control*, Gaithersburg, MD, 1995.

[9] D. Ferraiolo, R. Sandhu, S. Gavrila, and D. R. K. an d R. Chandramouli. Proposed NIST Standard for Role-Based Access Control. *ACM Transactions on Information and Systems Security*, 4(3):224–274, August 2001.

[10] R. B. France and G. Georg. Modeling fault tolerant concerns using aspects. Technical Report 02-102, Computer Science Department, Colorado State University, 2002.

[11] R. B. France, D. Kim, S. Ghosh, and E. Song. A UML-based pattern specification technique. *IEEE Transactions on Software Engineering*, 30(3):193–206, March 2004.

[12] R. B. France, I. Ray, G. Georg, and S. Ghosh. An aspect-oriented approach to design modeling. *IEE Proceedings - Software, Special Issue on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design*, 151(4):173–185, August 2004.

[13] G. Georg, R. B. 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. B. 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. B. France. Using Aspects to Design a Secure System. In *Proceedings of the Interational Conference on Engineering Complex Computing Systems (ICECCS 2002)*, pages 117–126, Greenbelt, MD, December 2002. ACM Press.

[16] S. Jajodia, P. Samarati, and V. S. Subrahmanian. A Logical Language for Expressing Authorizations. In *IEEE Symposium on Security and Privacy*, pages 31–42, Oakland, CA, May 1997.

[17] J. Jurjens. Towards development of secure sytems using umlsec. In *Proc. of the 4th Int'l. Conf. on Fundamental Approaches to Software Engineering*, pages 187–200, Genova, Italy.

[18] M. Koch, L. V. Mancini, and F. Parisi-Presicce. A graph-based formalism for RBAC. *ACM Trans. Inf. Syst. Secur.*, 5(3):332–365, 2002.

[19] T. Lodderstedt, D. Basin, and J. Doser. Secureuml: A uml-based modeling language for model-driven security. In *5th Int'l. Conf. on the Unified Modeling Language, 2002*, pages 426–441, 2002.

[20] M. Nyanchama and S. Osborn. The Role Graph Model and Conflict of Interest. *ACM Transactions on Information Systems Security*, 2:3–33, 1999.

[21] S. Osborn and Y. Guo. Modelling Users in Role-Based Access Control. In *Proceedings of the 5th ACM Workshop on Role-Based Access Control*, pages 31–37, Berlin, Germany, July 2000.

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

[23] E. Song, Y. R. Reddy, R. B. France, I. Ray, G. Georg, and R. Alexander. Verifiable composition of access control and application features. In *SACMAT '05: Proceedings of the tenth ACM symposium on Access control models and technologies*, pages 120–129, New York, NY, USA, 2005.

ACM Press.

[24] The Object Management Group (OMG). Unified Modeling Language: Superstructure. Version 2.0, Final Adopted Specification, OMG, http://www.omg.org, August 2003.

[25] J. E. Tidswell and T. Jaeger. An Access Control Model for Simplifying Constraint Expression. In *Proceedings of the 7th ACM conference on Computer and communications security*, pages 154–163, Athens, Greese, November 2000.

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