Modeling Role-Based Access Control Using Parameterized UML Models

Dae-Kyoo Kim, Indrakshi Ray, Robert France, Na Li

Computer Science Department
Colorado State University
Fort Collins, CO 80523, USA
Fax: +1 970 491 2466
(dkkim, iray, france, na)@cs.colostate.edu

Abstract. Organizations use Role-Based Access Control (RBAC) to protect computer-based resources from unauthorized access. There has been considerable work on formally specifying RBAC policies but there is still a need for RBAC policy specification techniques that can be integrated into software design methods. This paper describes a method for incorporating specifications of RBAC policies into UML design models. Reusable RBAC policies are specified as patterns and are expressed using UML template diagrams. Incorporating RBAC policies into an application specific model involves instantiating the patterns and composing the instantiations with the model. The method also includes a technique for specifying patterns of RBAC violations. Developers can use the patterns to identify policy violations in their models. The method is illustrated using a small banking application.

1 Introduction

Access control policies are constraints that protect computer-based information resources from unauthorized access. Role-Based Access Control (RBAC) [8] is used by many organizations to protect their information resources from unauthorized access. RBAC policies are defined in terms of permissions that are associated with roles assigned to users. A permission determines what operations a user assigned to a role can perform on information resources.

Work on formalization of RBAC policies has resulted in the development of new specification notations (e.g., see [1]), but there is still a need for policy specification approaches that can be integrated with design techniques used in industry. The Unified Modeling Language (UML) is considered to be the industry de-facto standard for modeling software-based systems. Use of the UML to specify RBAC policies eases the task of incorporating the policies into UML application models.

This paper describes a method that integrates RBAC policy specification and UML design modeling. The method includes (1) a technique for specifying generic RBAC policies as patterns that can be instantiated to produce application-specific design structures that specify the RBAC constraints, (2) techniques for systematically incorporating design structures produced by RBAC policy patterns into UML design models, and (3) a technique for specifying design structures that violate RBAC constraints as patterns.

Generic RBAC policies are specified by patterns expressed as UML diagram templates. Instantiating a RBAC pattern to produce an application-specific RBAC design structure involves binding the template parameters to application-specific design elements.

The RBAC patterns can be used to support at least two approaches to incorporating RBAC constraints into a UML design model: (1) The templates can be used to produce an initial design structure that is then extended to address other design concerns; and (2) the design structures produced by the templates can be merged with a previously developed application design model, referred to as the *primary model*, to obtain a design model that specifies RBAC constraints.

The method also provides a technique for specifying RBAC constraint violations as patterns. The violation patterns are expressed as template object diagrams, and can be used to check for the presence of violations in designs. To ease the task of checking for violations using the patterns we have developed an approach to visualizing application-specific RBAC constraints as object diagrams.

An overview of RBAC is given in Section 2. In Section 3 we present a generic RBAC model expressed as a class diagram template, and give examples of object diagram templates that describe patterns of policy violations. Section 4 describes how the RBAC pattern can be incorporated into a primary model. Section 5 gives examples of application-specific RBAC policies expressed as object diagrams. Section 6 illustrates how the violation patterns described by object diagram templates can be used to detect violations in application-specific RBAC policies. An overview of related work is provided in Section 7. The paper concludes with a discussion of current limitations of the approach and our plans to address the limitations.

2 Overview of RBAC

RBAC constraints can be organized as follows: *Core RBAC*, *Hierarchical RBAC*, *Static Separation of Duty Relations*, and *Dynamic Separation of Duty Relations*.

Core RBAC embodies the essential aspects of RBAC. The constraints specified by Core RBAC are present in any RBAC model. The Core RBAC requires that users be assigned to roles (job function), roles be associated with permissions (approval to perform an operation on an object), and users acquire permissions by being assigned to roles. The Core RBAC does not place any constraint on the cardinalities of the user-role assignment relation or the permission-role association. Core RBAC also includes the notion of user sessions. A user establishes a session during which he activates a subset of the roles assigned to him. 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 features supporting role hierarchies. Hierarchies are used to describe a structure of roles in an organization. Role hierarchies define an inheritance relation among the roles. Role $r1$ inherits from role $r2$ only if all permissions of $r2$ are also permissions of $r1$ and all users of $r1$ are also users of $r2$. The inheritance relationship is reflexive, transitive and anti-symmetric.

Static Separation of Duty (SSD) relations are necessary to prevent conflict of interests that arise when a user gains permissions associated with conflicting roles (roles that cannot be assigned to the same user). SSD relations are specified for any pair of roles that conflict. The SSD relation places a constraint on the assignment of users to roles, that is, assignment to a role that takes part in an SSD relation prevents the user from being assigned to the related conflicting role. The SSD relationship is symmetric, but it is neither reflexive nor transitive. SSD may exist in the absence of role hierarchies (referred to as SSD RBAC), or in the presence of role hierarchies (referred to as hierarchical SSD RBAC). 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.

Dynamic Separation of Duty (DSD) relations aim to prevent conflict of interests as well. The DSD relations place constraints on the roles that can be activated in a user's session. If one role that takes part in a DSD relation is activated, the user cannot activate the related (conflicting) role in the same session. A model of RBAC is shown in Fig. 1.

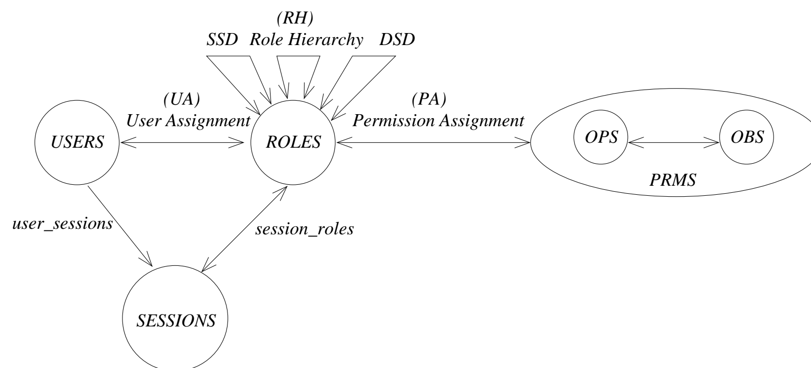


Fig. 1. RBAC

The RBAC in Fig. 1 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 objects (*OBS*) where an object is an entity that contains or receives information, 4) a set of operations (*OPS*) where an operation is an executable image of a program, and 5) a set of permissions (*PRMS*) where a permission is an approval to perform an operation on objects. The cardinalities of the relationships are indicated by the absence (denoting one) or presence of arrows (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 the roles that conflict with each other. The association labeled *DSD* specifies the roles that cannot be activated within a session by the same user.

3 A Reusable RBAC Model

In this section a RBAC pattern is described as a UML template class diagram. A class diagram is obtained from a template diagram by binding the parameters to values. Fig. 2 shows a class diagram template describing hierarchical RBAC with SSD and DSD. The symbol "|" is used to indicate parameters. We use this notation when there is a large number of parameters because the standard UML parameter notation is cumbersome.

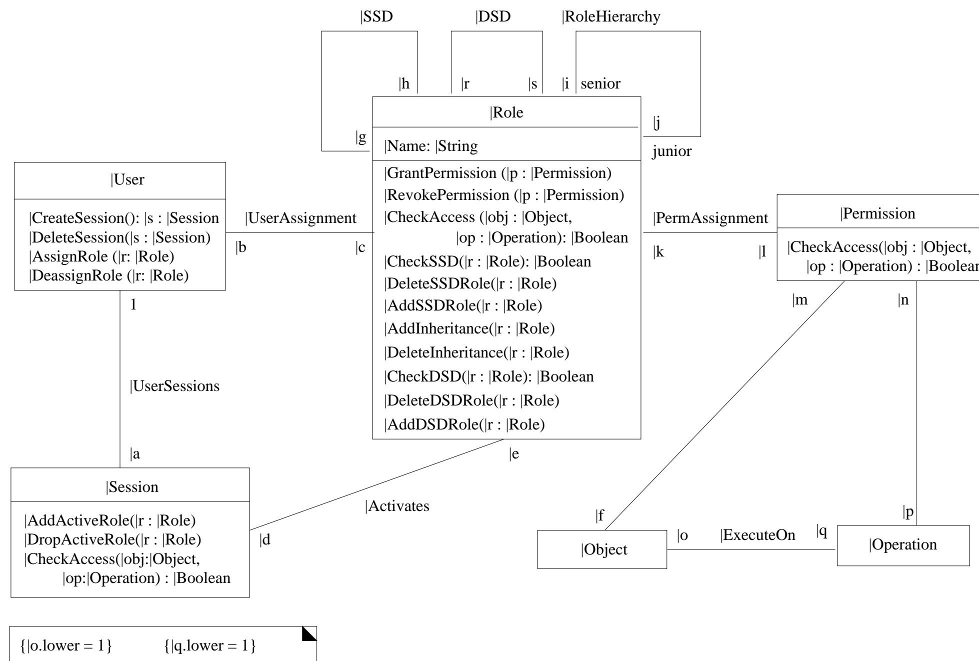


Fig. 2. A RBAC Class Diagram Template

The class diagram template shown in Fig. 2 consists of class and association templates. A class template is a class descriptor with parameters. Class templates are associated with attribute templates (e.g., |*Name* : *String* in *Role*) and operation templates (e.g., |*GrantPermission* in *Role*). Association templates (e.g., |*UserAssignment*) consist of parameters for association names and association-end multiplicities. The OCL constraints in Fig. 2 restrict the values that can be bound to association-end multiplicity parameters. For example, {|o.lower = 1} restricts the multiplicities that can be bound to the parameter *o* to ranges that have a lower bound of 1. The multiplicity "1" on the *UserSessions* association-end attached to *User* is strict: a session can only be associated with one user.

The *User* class template defines classes that describe users. A user can create a new session (*CreateSession*), delete a session (*DeleteSession*), associate self with a new role

(*AssignRole*) and remove an associated role (*DeassignRole*). A *UserSessions* link (i.e., an instance of an association obtained by binding the parameters of *UserSessions* to values) is created by a *CreateSession* operation (i.e., an operation obtained by binding the operation template parameters to values) and deleted by a *DeleteSession* operation. The operation *AssignRole* creates a *UserAssignment* link; the *DeassignRole* removes a *UserAssignment* link.

The class template *Role* is used to produce classes representing roles with behavior that (1) associates a new permission with the role (*GrantPermission*), (2) deletes an existing permission associated with the role (*RevokePermission*), (3) adds an immediate inheriting role (*AddInheritance*), (4) deletes an immediate inheriting role (*DeleteInheritance*), (5) adds a role to the set of conflicting roles (*AddSSDRole*), (6) deletes a role from the existing set of conflicting roles (*DeleteSSDRole*), (7) checks whether the role is in an SSD relationship with a given role in the presence of hierarchies (*CheckSSD*), (8) checks whether the role has a given permission (*CheckAccess*), (9) checks whether the role is in a DSD relation with a given role (*CheckDSD*), (10) deletes a DSD relation between the role and a given role (*DeleteDSDRole*), and (11) adds a DSD relation with a given role (*AddDSDRole*). The class template *Session* is associated with the template operations: *AddActiveRole* (activates a role in a session), *DropActiveRole* (deactivates a role in a session), and *CheckAccess* (checks whether the role has the permission to perform an operation on an object).

The class template *Permission* is associated with an operation template, *CheckAccess*, that checks whether the role has the permission to perform the operation on the object.

Each operation template is associated with an OCL template expression that produces OCL pre- and post-conditions when the template parameters are bound to values. Pre- and post-condition templates associated with the *CreateSession* and *GrantPermission* operation templates are given below:

```
context |User::|CreateSession():(|s:|Session)
  post: result = |s and
   |s.oclIsNew() = true and self.|Session → includes(|s)

context |Role::|GrantPermission (|p:|Permission)
  post: self.|Permission → includes(|p)
```

We express RBAC constraints that restrict SSD and DSD relationships as OCL template expressions. Examples of these constraints are given below:

– SSD constraint. A user cannot be assigned to two roles that are involved in an SSD relation.

```
context |User inv:
   self.|Role → forAll(r1, r2 | r1.|SSD → excludes(r2))
```

– Hierarchical SSD constraint. There cannot be roles in an SSD relation which have the same senior role.

context |Role **inv**:
let allSenior(r1) = r1.senior → union(r1.senior → collect(r2 | allSenior(r2)))
in
  self.|SSD → forAll(r1 | allSenior(r1) → excludesAll(allSenior(self))

– DSD constraint. A user cannot activate two roles in DSD relation within a session.
context |User **inv**:
  |self.|Session.|Activates → forAll(r1, r2 | r1.|DSD → excludes(r2))

4 Applying the RBAC Model

To illustrate our approach we use a simple banking application taken from [5]. The application is used by various bank officers to perform transactions on customer deposit accounts, customer loan accounts, ledger posting rules, and general ledger reports. The transactions include 1) create, delete, or modify customer deposit accounts, 2) create, delete, or modify customer loan accounts, 3) modify the ledger posting rules, and 4) create general ledger report. A class diagram (the primary model) for the application is shown in Fig. 3. Class attributes and operations are not shown in the diagram.

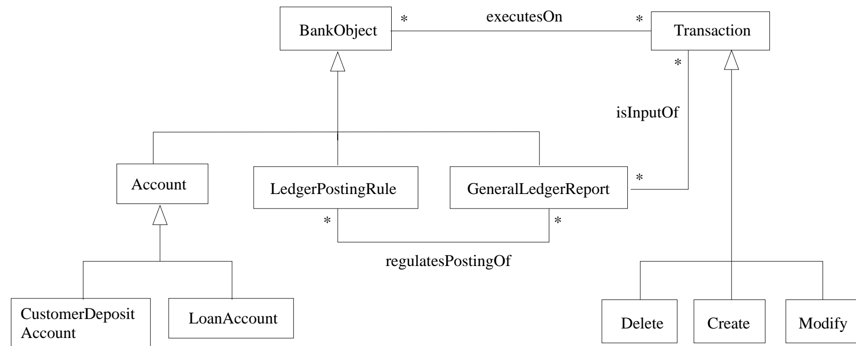


Fig. 3. A Partial View of a Banking System Primary Model

Access control policies are not specified in the primary model. RBAC features can be incorporated into the primary model by composing an instantiation of the RBAC template in Fig. 2 with the primary model. The composition is carried out as follows:

1. Instantiating the RBAC template : To incorporate RBAC features into a primary model, the modeler must first instantiate the RBAC template model by binding parameters to elements representing concepts in the domain of the primary model. Some

of these model elements may be elements in the primary model. Class diagrams obtained from the RBAC class diagram template are referred to as *context-specific RBAC diagrams*. Fig. 4 shows a context-specific RBAC class diagram for the banking application. In the diagram, *BankRole*, *BankObject*, and *Transaction* are respectively bound to *Role*, *Object*, and *Operation* parameters in the RBAC template diagram.

Fig. 4. A Context-Specific RBAC Class Diagram

2. Merging the context-specific diagram with the primary model : The view defined by the context-specific RBAC diagram is merged with the view defined in the primary model to obtain a composed model. Elements in the instance and the primary model are merged if and only if they have the same syntactic type (i.e., UML metamodel class) and name. Model elements in the context-specific RBAC diagram that do not exist in the primary model are added to the primary model. For example, if the RBAC diagram has a class with an operation that has the same name, but different signature as a primary model operation in a matching class, the operation is included in the composed model, resulting in an overloaded operation. However, if the RBAC diagram has a class with an operation with the same name and signature as a primary model operation in a matching class, either one of the operation names must be changed or the operation specifications must be logically composed to produce a consistent operation specification. In the first case, the developer must specify the new name of the operation and in the second case

the developer must specify the logical operator to be used in the composition. These developer inputs are expressed as *composition directives*: A directive allows a developer to vary how RBAC and primary model elements are merged. For more on merging rules and composition directives see [9, 10].

The result of the composition is a composed model in which access control features specifi ed by the context-specifi c RBAC model are incorporated into the primary model. The composed model for the banking system is shown in Fig. 5. The *BankObject* and *Transaction* classes in the context-specifi c RBAC diagram are merged with *BankObject* and *Transaction* classes in the primary model, and *BankUser*, *BankRole*, *BankSession*, and *Permission* are the RBAC classes that are included in the composed model.

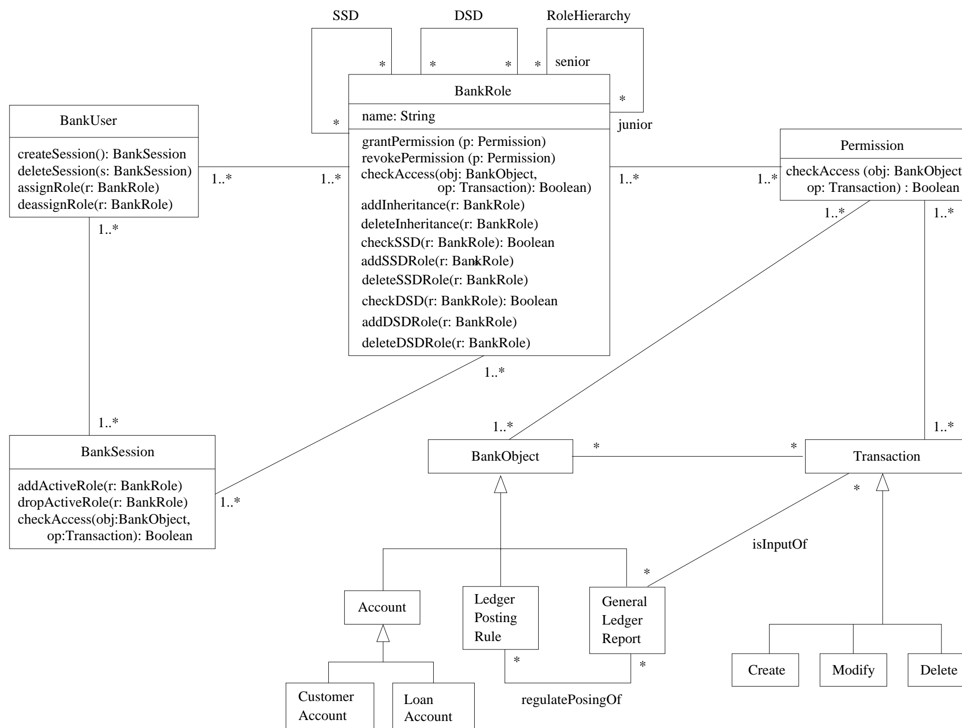


Fig. 5. Composed Model

5 Describing Application-Specific RBAC Policies Using Object Diagrams

Application-specifi c RBAC policies constrain how system users access system resources. They determine 1) the assignment of roles to system users, 2) the permissions associ-

ated with roles in the systems, 3) the inheritance relationships between roles, and 4) the SSD and DSD relationships between roles. In this section we illustrate how application-specific RBAC policies can be described by object diagrams.

The RBAC model supports the specification of four types of policies: 1) *core policies* that conform to core RBAC, that is, policies that determine user-role and role-permission assignments, 2) *hierarchical policies* that conform to hierarchical RBAC, that is, policies that determine inheritance relationships between roles, 3) *SSD policies* that conform to SSD RBAC, that is, policies that determine what roles are conflicting, and 4) *DSD policies* that conform to DSD RBAC, that is, policies that determine what roles to be activated in a session. A set of application-specific RBAC policies for the banking system is given below:

Core policies: The roles of the banking system (instances of *BankRole*) are *teller*, *customerRerviceRep*, *accountant*, *accountingManager* and *loanOfficer*. The permissions assigned to these roles are given below:

P1 A teller can modify customer deposit accounts.
P2 A customer service representative can create or delete customer deposit accounts.
P3 An accountant can create general ledger reports.
P4 An accounting manager can modify ledger-posting rules.
P5 A loan officer can create and modify loan accounts.

Fig. 6 shows the object diagrams describing policies P1 to P5 respectively.

Hierarchical policies: A role hierarchy defines inheritance relationships between roles. Through the inheritance relationship, a senior role inherits the permissions of its junior roles and any user assigned to the senior role is also assigned to the junior roles. The hierarchical policies in the banking application are stated below:

H1 Customer service representative role is senior to the teller role.
H2 Accounting manager role is senior to the accountant role.

Fig. 7(a),(b) describe policies H1 and H2 respectively.

SSD policies: SSD policies prevent a user from being assigned to two conflicting roles. For the banking system the following pairs of roles are conflicting:
{*(teller, accountant), (teller, loanOfficer),*
(loanOfficer, accountant), (loanOfficer, accountingManager),
(customerServiceRep, accountingManager)} The object diagram in Fig. 8 describes the SSD RBAC policies.

DSD policies: DSD policies prevent a user from playing a role in a session, if another role in a DSD relation has been activated. For the banking system the following pair of roles are in DSD relation:
{*(customerServiceRep, loanOfficer)*} The object diagram in Fig. 9 describes the DSD RBAC policy.

6 Identifying Conflicts in Application-Specific RBAC Policies

In this section we show how RBAC violation patterns expressed as object diagram templates can be used to identify conflicts in application-specific policies. If a violation

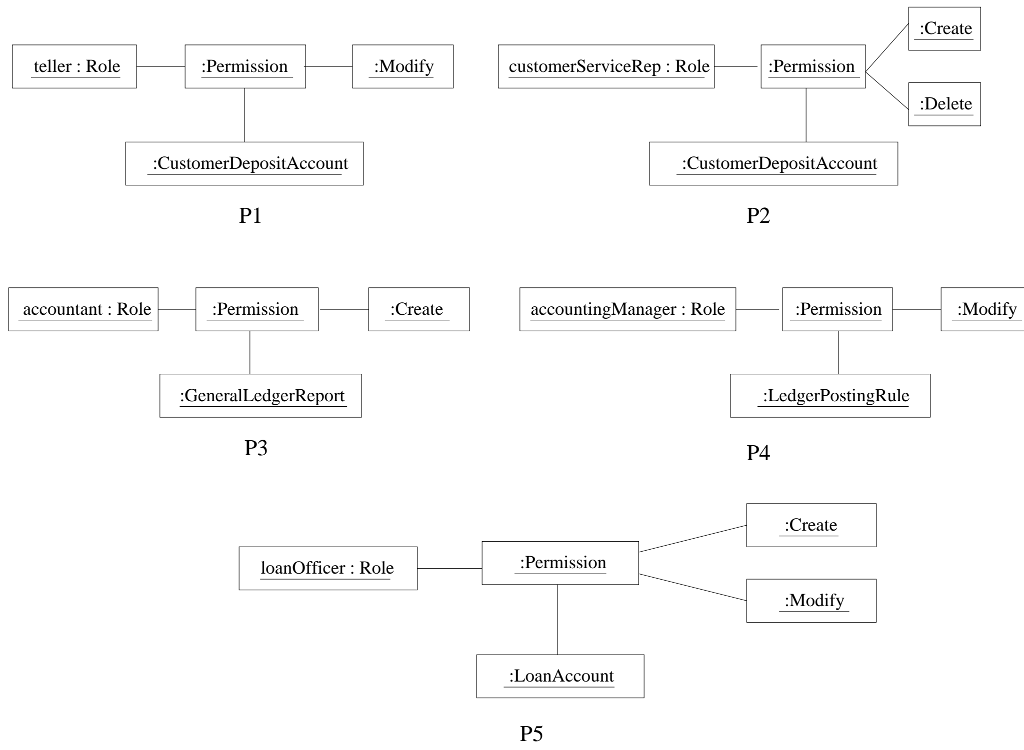


Fig. 6. Object Diagrams describing Core RBAC Policies

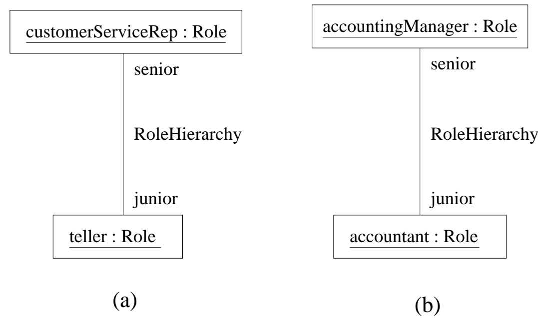


Fig. 7. Object Diagrams for Hierarchical Policies

pattern exists in an object diagram describing an application-specifi c policy, then a conflict exists.

Fig. 10 shows object diagram templates that when instantiated produce object structures that violate RBAC constraints. Fig. 10(a) describes structures in which a user is

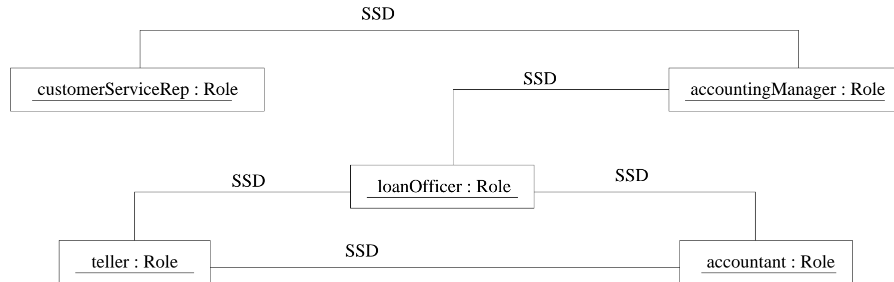


Fig. 8. Object Diagram for SSD Policies

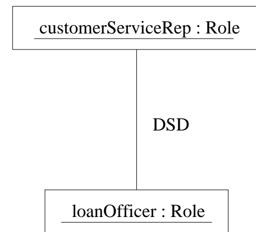
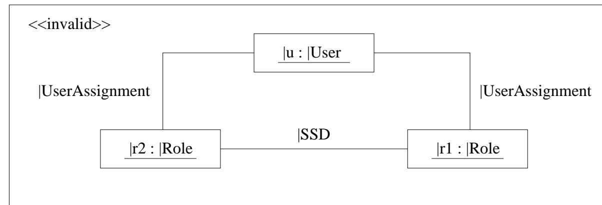


Fig. 9. Object Diagram for DSD Policy

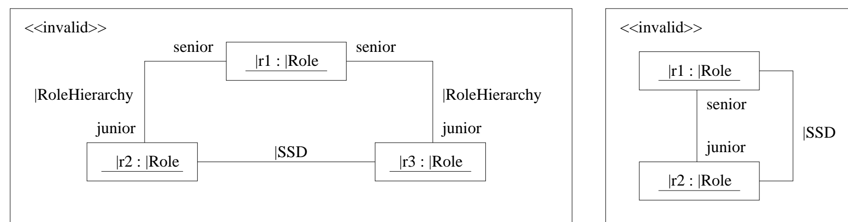
assigned to roles in an SSD relationship (violation of the SSD constraint). Fig. 10(b) describes structures in which two roles in an SSD relationship have a common senior role and structures in which a senior role is in an SSD relationship with a junior role (both are violations of the hierarchical SSD constraint). Fig. 10(c) describes structures in which a user in a session activates two roles that are in a DSD relationship (a violation of the DSD constraint). We illustrate how these object diagram templates can be used to identify conflicts in application models later in this paper.

Fig. 11 shows the object diagram that integrates the policies shown in Fig. 7, Fig. 8, and Fig. 9. The reader can visually check that the pattern described by object diagram template in Fig. 10(b) does not occur in Fig. 11.

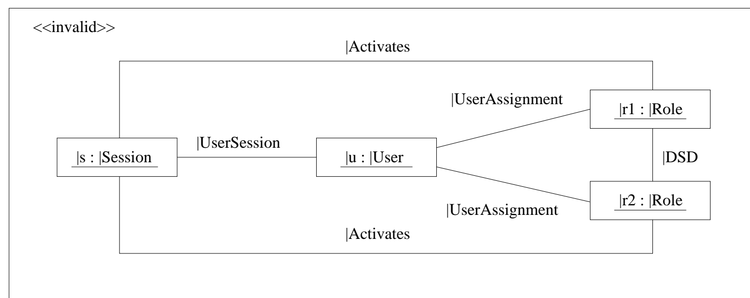
Formally, an object diagram has the violation described by a violation pattern if there exists a binding that produces an object structure contained in the object diagram. To illustrate how conflicts can be identifi ed, consider the case in which the following policy is added to the set of policies described in the previous section: "The branch manager role is senior to all the other roles in the bank." Fig. 12 shows the result of including this policy in the banking application's policy set. A number of occurrences of the pattern described in Fig. 10(b) can be found in Fig. 12. For example, if we assign a user to the branch manager role, the user is also assigned to the roles *customerServiceRep* and *accountingManager* through inheritance. However, the roles *customerServiceRep* and *accountingManager* are in an SSD relation.



(a) Violation of SSD Constraint



(b) Violations of Hierarchial SSD Constraint



(c) Violation of DSD Constraint

Fig. 10. RBAC Constraints

7 Related Work

Tidswell and Jaeger [21] propose an approach to visualizing access control constraints. They point out the need for visualizing constraints and the limitations of previous work on expressing constraints. A drawback of their work 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 paper utilizes a popular standardized modeling language (the UML) and also integrates the policy specification activity with UML design modeling activities.

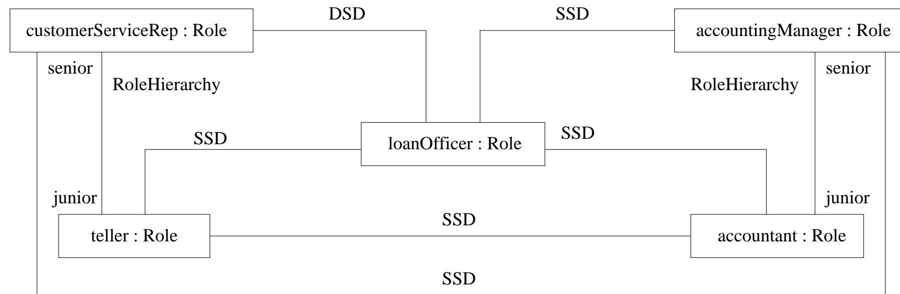


Fig. 11. Combined Object Diagram

A large volume of research (e.g., see [2–4, 6, 7, 11, 12, 14] exists in the area of access control policy specification. Formal logic-based techniques (e.g., see [2–4, 6, 11, 14]) are often used to specify security policies. The use of mathematical concepts and notation that are not familiar to software developers makes them difficult to use and understand. Other researchers have used high-level languages to specify policies [12, 13, 19, 20]. Although high-level languages are easier to understand than formal logic-based approaches, they are not analyzable.

Some work has been done on modeling system security using UML. Jurjens [15] proposes UMLsec, a UML profile for modeling and evaluating security aspects based on the multi-level security model. Lodderstedt *et al.* propose SecureUML [17], an extension of the UML that defines security concepts based on RBAC. These approaches mainly focus on extending the UML notation to better reflect security concerns. The approach described in this paper tackles the complementary task of capturing RBAC policies in patterns that can be reused by developers of secure systems.

8 Conclusion

The work described in this paper focuses on specifying only the static structure of RBAC. A complete RBAC model should also include descriptions of the patterns of behavior supported by RBAC. In previous work (e.g., see [9, 16]) we developed template forms of interaction diagrams that can be used to specify interaction patterns. The interaction patterns can be used to characterize families of allowed and prohibited behaviors. We are also developing template forms of other UML behavioral models.

The use of violation patterns to identify policy conflicts, while useful, has its limitations. Checking for the presence of a pattern in an object diagram specifying a set of policies is essentially a search for a subgraph in an object diagram, which is known as *subgraph isomorphism problem*. Detecting subgraph isomorphism can be a difficult task [18]. Our work in this area focuses on identifying alogorithms to support practical application of this technique.

Validation of the method is needed. To support planned validation activities we are developing a tool set that allows developers to create and instantiate UML diagram templates, and to compose template instantiations with UML design models.

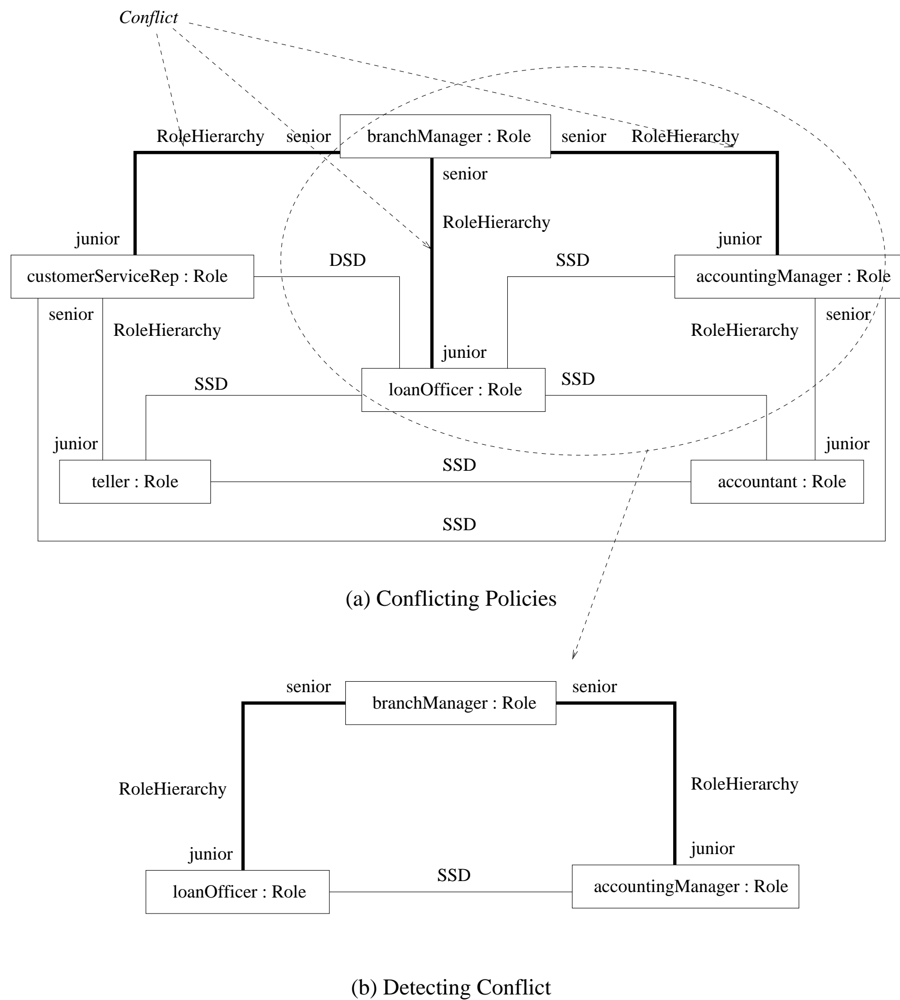


Fig. 12. Violation Pattern Occurrence: Hierarchical SSD

References

1. G.J. Ahn and R. Sandhu. Role-based Authorization Constraints Specification. *ACM Transactions on Information and Systems Security*, 3(4):207–226, November 2000.
2. S. Barker. Security Policy Specification in Logic. In *Proceedings of the International Conference on Artifi cial Intelligence*, pages 143–148, Las Vegas, NV, 2000.
3. 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.

4. 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.
5. R. Chandramouli. Application of XML Tools for Enterprise-Wide RBAC Implementation Tasks. In *Proceedings of 5th ACM workshop on Role-based Access Control*, Berlin, Germany, July 2000.
6. 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.
7. N. Damianou and N. Dulay. The Ponder Policy Specification Language. In *Proceedings of the Policy Workshop*, Bristol, U.K., 2001.
8. D.F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST Standard for Role-Based Access Control. *ACM Transactions on Information and Systems Security*, 4(3), August 2001.
9. Geri Georg, Robert France, and Indrakshi Ray. An Aspect-Based Approach to Modeling Security Concerns. In *Proceedings of the Workshop on Critical Systems Development with UML*, Dresden, Germany, 2002.
10. Geri Georg, Indrakshi Ray, and Robert 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.
11. R. J. Hayton, J. M. Bacon, and K. Moody. Access Control in Open Distributed Environment. In *IEEE Symposium on Security and Privacy*, pages 3–14, Oakland, CA, May 1998.
12. M. Hitchens and V. Varadarajan. Tower: A Language for Role-Based Access Control. In *Proceedings of the Policy Workshop*, Bristol, U.K., 2001.
13. J. A. Hoagland, R. Pandey, and K. N. Levitt. Security Policy Specification Using a Graphical Approach. Technical Report CSE-98-3, Computer Science Department, University of California Davis, July 1998.
14. 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.
15. J. Jurjens. UMLsec: Extending UML for Secure Systems Development. In *Proceedings of Fifth International Conference on the Unifi ed Modeling Language, pp. 412-425*, pages 412–425, Dresden, Germany, October 2002.
16. Dae-Kyoo Kim, Robert France, Sudipto Ghosh, and Eunjee Song. Using Role-Based Modeling Language (RBML) as Precise Characterizations of Model Families. In *Proceedings of the Interational Conference on Engineering Complex Computing Systems (ICECCS 2002)*, Greenbelt, MD, December 2002. ACM Press.
17. T. Lodderstedt, D. A. Basin, and J. Doser. SecureUML: A UML-Based Modeling Language for Model-Driven Security. In *Proceedings of Fifth International Conference on the Unifi ed Modeling Language*, pages 426–441, Dresden, Germany, October 2002.
18. B.T. Messmer and H. Bunke. Subgraph Isomorphism in Polynomial Time. In *Lecture Notes in Computer Science Graph Theory - ECCV'98, Springer-Verlag*, Berlin, 1998.
19. OASIS. XACML Language Proposal, Version 0.8. Technical report, Organization for the Advancement of Structured Information Standards, January 2002. Available electronically from http://www.oasis-open.org/committees/xacml.
20. C. Ribeiro, A. Zuquete, and P. Ferreira. SPL: An Access Control Language for Security Policies with Complex Constraints. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, February 2001.
21. J. E. Tidswell and T. Jaeger. An Access Control Model for Simplifying Constraint Expression. In *Proceedings of 7th ACM conference on Computer and communications security*, pages 154–163, Athens, Greese, November 2000.