Describing Access Control Patterns Using Roles

Dae-Kyoo Kim, Pooja Mehta, and Priya Gokhale
Department of Computer Science and Engineering
Oakland University
Rochester, MI 48309
{kim2,pmehta,pvgokhal}@oakland.edu

Abstract

Access control patterns describe access control mechanisms at a high level of abstraction. An access control pattern provides a general solution to a class of access control problems for the confidentiality, integrity and availability of the information resources of software systems. While there has been much literature describing the general solution as to how these patterns enforce access control voluminously, there is little work that describes the patterns in a pattern template using appropriate notations for an easy and quick reference. In this paper, we present pattern descriptions of three commonly used access control patterns (DAC, MAC, RBAC) described in the template of pattern-oriented software architecture (POSA). We use an extension of the UML for representing the structure and behaviors of the patterns to capture variations of pattern realizations. We also attempt to give more details on the problem domain of the patterns to help developers choose an suitable pattern.

1 Introduction

An access control mechanism enforces access control policies which specify high-level requirements of who can access what information under what circumstances. An access control pattern is an abstraction of an access control mechanism, capturing the system-independent concepts in the mechanism. An access control pattern provides a general solution to a class of access control problems for the confidentiality, integrity and availability of the information resources of software systems.

There has been a huge volume of literature (e.g., [4, 5, 8, 15, 29]) that describes access control mechanisms. However, there is only a little work describing them in a pattern template for easy and quick reference. This includes the work by Fernandez and Pan [6] which presents Authorization, RBAC and Multilevel Security models in a pattern template. Their work primarily focuses on the solution domain of the patterns described through a typical example.

In this work, we present pattern descriptions for the three widely used access control patterns, Discretionary Access Control (DAC) [15], Mandatory Access Control (MAC) [29] and Role-Based Access Control (RBAC) [8] in the template of the pattern-oriented software architecture (POSA) [3]. The goal of this work is to provide an easy and quick reference of these patterns for developers to help them choose an appropriate pattern for a given problem and understand the solution of the chosen pattern.

An issue in writing a pattern is the lack of suitable notations to describe the structure and behaviors of the pattern [9, 14, 20]. Most (if not all) of the commonly used pattern descriptions e.g., see [3, 11, 13, 24, 31]) uses the OMT notation [27] or the Unified Modeling Language [36]

"as it is" to describe pattern's structure and collaborations. A problem with this is that the structure and collaborations represented in these notations are just a typical realization of the pattern [9, 14]. In practice, pattern realizations vary depending on the application context, and even in the same application domain, there can be multiple realizations having different structure for the same pattern. As such, we argue that the pattern descriptions should be described using a notation that can capture such variations of pattern realizations.

In this work, we use an extension of the UML developed in our previous work [10, 16] to describe the structure and behaviors of pattern participants for capturing variations of pattern realizations. The extension describes a pattern in terms of pattern roles [17] where a pattern role can be played by model elements (e.g., classes) in a specific application domain. By the nature of roles, a pattern role can be played by multilple elements which takes into account possible variations. The properties of a pattern role constrain the eligibility of model elements to play the role. Our work specifically aims at facilitating the use of the three patterns for developing models of secure software systems described in the UML.

Based on our study, we also found that most of the literature of the patterns mainly describes the solutions of the patterns focusing on how they enforce access control, but little attention has been paid to the problem domain. From the practical point of view, it is equally (or more) important to know which pattern is suitable to use under what conditions, the applicability of the pattern, and the pros and cons of applying the pattern. Such informaiton in the problem domain helps developers choose an appropriate pattern to use. In the proposed descriptions, we attempt to give more details on the problem domain of the three patterns.

The rest of the paper is organized as follows. Section 2 describes the notion of pattern roles. Section 3 presents the descriptions of of the DAC, MAC and RBAC pattern. Section 4 concludes the paper.

2 Pattern Roles

Commonly used pattern descriptions (e.g., [3, 11, 13, 24]) use a typical realization of the pattern to describe the structure and collaborations of the pattern solution. However, we believe that they should be described in a way that captures various realizations. For instance, one common participant in many security patterns is *User*. In one application domain, there may be several classes (e.g., Teller, Accountant, Loan Officer) that play the *User* role, but in another, there might be only one class (just Client) playing the role. Such variations are very common, and should be considered in pattern descriptions. In our work, we employ the notion of "pattern roles" [17] to describe such variations in the structure and collaborations of the problem and solution domain of DAC, MAC and RBAC patterns. A pattern role represents a pattern participant, and it can be played by one or more elements in an application domain.

Fig. 1 shows the relationships between a pattern role and the UML infrastructure [36]. In our work, pattern roles are defined at the metamodel-level (M2) in the UML, and played by model elements at the model-level (M1) (e.g., classes, associations). In the figure, the *MyRole* pattern role, which is denoted by "|" symbol, defines a subset of instances of the *Class* metaclass which is the base metaclass of the role as indicated by the bold text label above the role name. Every role must have one base metaclass in the UML, and only the instances of the base metaclass can play the role. For example, in the diagram only the instances of the *Class* metaclass can play the |*MyRole* pattern role. Then, a question might arise, can any instance play the role? No. In the formal definition [17], pattern roles are defined by a set of constraints, and only those instances

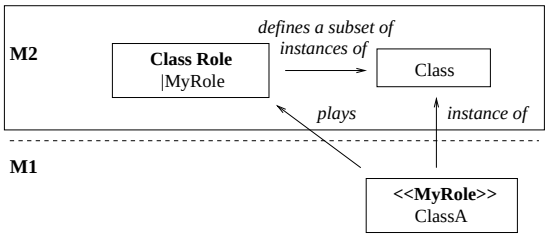


Figure 1: Relationship between Model Role and UML Infrastructure

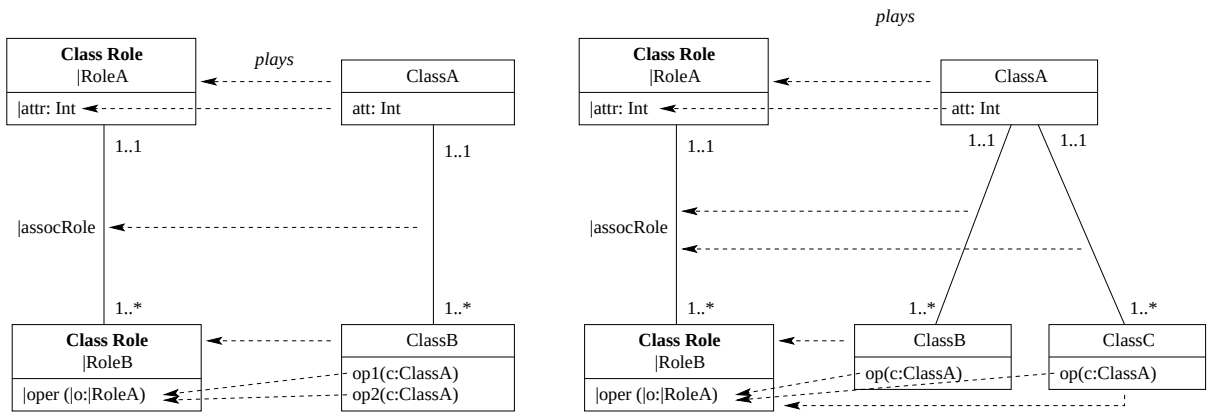
that satisfy the constraints can play the role. These constraints are used to define precise and rigorous pattern specifications mainly to support the development of pattern tools. We leave out the discussion as to how to define the constraints since the patterns being described here are very informal. However, we believe that the use of the pattern role concept in this work is still beneficial in the light of taking into account variations in realizations. We use the "|" notation to indicate pattern roles in this paper, and they should be interpreted as being possibly played by multiple classes in an application domain.

A class role, which has the *Class* metaclass as its base, may have feature roles (i.e., attribute roles and operation roles whose base is the *Attribute* and *Operation* metaclass, respectively, in the UML), and several features in a model may play the feature roles. A class role may be connected with another class role via an association role whose base is the *Association* metaclass.

Figure 2: An Example of a Pattern Structure

Fig. 2 shows an example of a pattern structure described in terms of pattern roles. The structure has two class roles *RoleA* and *RoleB* where *RoleA* has an attribute role |*attr* that can be played by integer attributes, and *RoleB* has an operation role |*oper* that can be played by operations whose parameter type is *RoleA*. The class roles are connected by the association role |*assocRole*. The multiplicities at the ends of the |assocRole constrain that classes playing the class roles must have that multiplicities. The formal definition of pattern roles [17] has a more complicated way of constraining multiplicities, but in this work, we simply constrain that class playing a role must have the specified multiplicity.

Fig. 3 shows examples of two different model structures that have model elements playing the pattern roles in Fig. 2. In Fig. 3(a), there are two classes *ClassA* and *ClassB*, each of which plays |*RoleA* and |*RoleB*, repsectively. *ClassA* can play |*RoleA* because it has an attribute whose type is integer, playing the |*attr* role in |*RoleA*. Similarly, *ClassB* can play |RoleB since it has two operations that can play the |*oper* role in |*RoleB*. The two operations can play the |*oper* role because they have a parameter whose type is *ClassA* which plays the |*RoleA* as required in |*oper*. The arrows show the mapping between the model elements and the pattern roles played by them. Fig. 3(b) shows another application domain that has two classes (*ClassA* and *ClassB*) playing one class role (|*RoleB*), which is very possible, and there might be other applications having different structures to play the pattern roles. These examples show the benefits of using pattern roles.



(a) Application Domain 1

(b) Application Domain 2

Figure 3: Examples of Model Elements Playing Pattern Roles

It should be noted that the notion of pattern roles is different from the object roles in the UML at the level which they are defined. Pattern roles are defined at the metamodel level played by model elements, while object roles are defined at the model level played by objects as shown in Fig. 4.

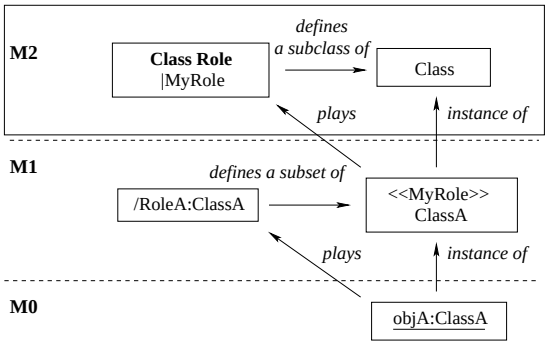


Figure 4: Pattern Roles and Object Roles

In the figure, the object role */RoleA* is defined at the model-level (M1), and played by an instance (*objectA* at the object-level (M0)) of *ClassA*, while the pattern role |*MyRole* is defined at the metamodel-level (M2), and played by an instance (*ClassA* at the model-level(M1)) of its base *Class* metaclass.

3 Access Control Patterns

This section presents the DAC pattern based on Harrison *et al.*'s work[15], the MAC pattern based on the Bell-La Padula model for the MAC pattern [15], and the RBAC pattern based on the NIST proposal [8] and the book by Ferraiolo *et al.* [7].

3.1 Discretionary Access Control

The DAC pattern enforces access control based on the identity of the requestor and explicit access rules that define who can or cannot execute which actions on which resources. A user (subject) who has access to an object requested may delegate the permission of the object to another user. The DAC pattern is also known as access control matrix [19].

Example

Consider an environment where access control is solely managed by the security administrator. A problem in such an environment is that it requires much efforts for the administrator to maintain access control for every single user and to deal with daily-basis requests for permission changes. Also, related to confidentiality, the administrator may give a permission to a person who is not supposed to be authorized to access the information. For example, in a medical care system where patient information should be kept in confidential, access to the information should be limited to only the doctor who handles the case, or other doctors who have permission given by the handling doctor. When the administrator is requested for permission to be given to a clerk to access a patient file, obviously it should not be allowed. However, the request may contain disguised information about the requestor, deceiving the administrator. Related to availability, such environment may cause a situation where no one can access information. For example, in the medical case system above, if *DoctorA* who handles *CaseFile1* has left for a vacation, without making a permission request for other doctors to access the file, no one can access the file in case of emergency. Fig. 5 illustrates these problems.

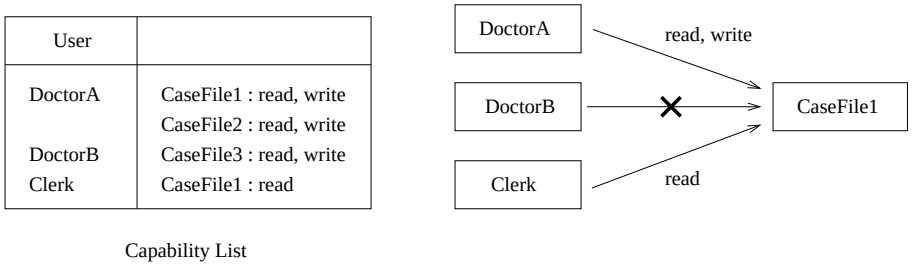


Figure 5: A Motivating Example

Context

Development of access control systems that allow user-controlled administration of access rights to objects.

Problem

In general, the problem environments suitable for use of the DAC pattern usually have access control that is managed solely by administrators for granting and revoking a user or group an access permission to an object without concerning the discretion of the users of the object. Use the DAC pattern

- where users own objects.
- when permission delegation is needed.

- when resolution of conflicting privilege is needed. For example, there might be inconsistencies between the permissions given to a user as an individual and a member of a group to which the user belongs. The user may be allowed to access an object as a member of a group, but not allowed in individual permissions. An ordered evaluation of a permission list can be used to resolve such authorization conflicts. The evaluation stops either when all requested access rights have been granted by one or more permission entries, or when any one of the requested access rights has been denied by one of the permission entries.
- when security mechanisms are needed in a heterogeneous environment to control access to groups of different kinds of resources.
- where Multi-user Relational Database is used.

Solution

The DAC pattern can address the problems above by using the concept of "permission delegation" which allows a user of an object to give away permission to other users to access the object at her/his discretion without the intercession of the administrator. Using the DAC pattern, the burden on the administrator is shared with the users of objects since the users are capable of giving away permissions. Also, permission delegation can mitigate the confidentiality issue since a user can give permissions directly to related people in the area, and the availability issue since a user can give a permission at any time whenever needed.

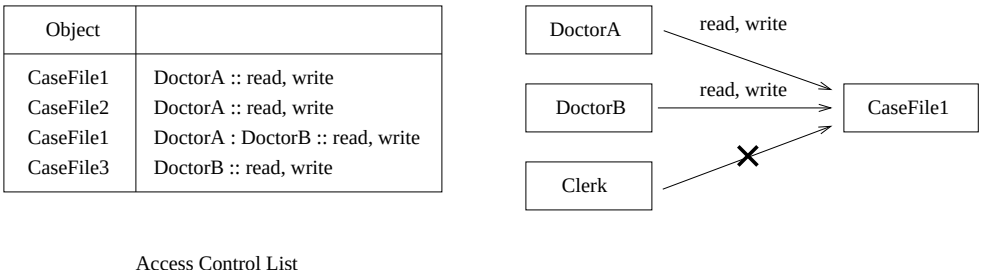


Figure 6: DAC Solution

Fig. 6 shows an Access Control List (ACL) that implements the DAC pattern, addressing the above problems. In the ACL, a delegating user of the object is represented in *user::rw* and a named user to which permission is delegated is represented in *user:namedUser:rw*. For example, *DoctorA* is a user of *CaseFile1*, and has read and write access to the file, and *DoctorB* is a named user who is granted to access *CaseFile1* by *DoctorA*, and has read and write access to the file.

Structure

Fig. 7 shows the solution structure of the DAC pattern where the user who owns an object may grant or deny permissions (privileges) to other users or groups.

- *User* represents a user or group who has access to an object, or a named user or group who are granted access to an object by the user or group. The owner or owning group of an object has full access to the object, and can grant or revoke an access permission to other users or groups at their discretion.
- *Object* represents any information resource (e.g., files, databases) to be protected in the system.

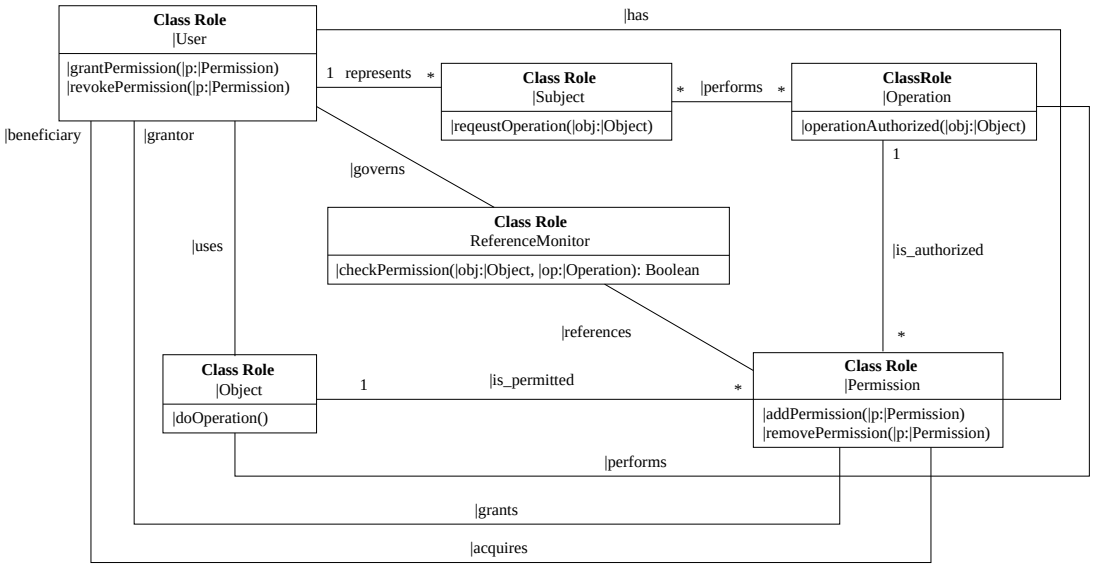


Figure 7: Solution Structure

- *Operation* represents an active process invoked by a user.
- *Subject* represents a process acting on behalf of a user in a computer-based system, another computer system, a node, or a set of attributes.
- *Permission* represents an authorization to carry out an action on the system. In general, Access Control Lists (ACLs) are used to describe DAC policies for its easiness in reviewing. An ACL shows permissions in terms of objects and a combination of users and their access rights.
- *ReferenceMonitor*: A user requests an operation on an object, and the request is checked for permission. If the user has a permission to the object, the operation may be performed, otherwise, access is denied.

Dynamics

Fig. 8 shows the collaboration for requesting an operation. Where there is a request for an operation, the reference monitor checks for permission of the user for the operation using a permission list, which is typically described in ACLs. If the permission is allowed, the operation is performed, otherwise, the request is denied.

Variants

Based on the underlying concept of the DAC pattern, there have been several variants proposed [28]. The variants differ by the degree of the strictness in owner's discretion, and they can be categorized into strict DAC, liberal DAC and DAC with change of ownership. Strict DAC is the most strict form in which only the owner of an object can grant access to the object. Liberal DAC allows the owner to delegate access granting authority to other users. Liberal DAC can be further divided into one-level grant, two-level grant and multilevel-grant, depending on the level at which access granting authority can be passed on. DAC with change of ownership allows the owner to delegate ownership to other users.

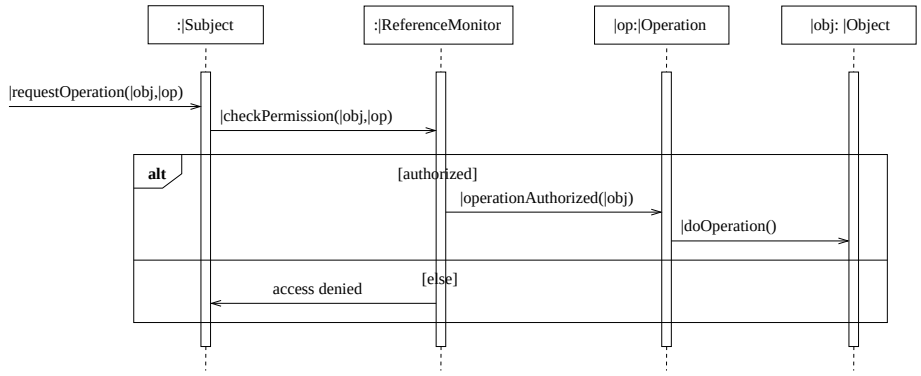


Figure 8: Requesting Operation

Known Uses

Standard Oracle9i [22] uses the DAC pattern to mediate user access to data through database privileges such as SELECT, INSERT, UPDATE and DELETE. The TOE [37], a sensitive data protection product developed by The Common Criteria Evaluation and Validation Scheme (CCEVS), uses the DAC pattern to mediate access to cryptographic keys against unauthorized access to data. Windows NT implements the DAC pattern to control generic access rights such as No Access, Read, Change, and Full Control for different types (e.g., Everyone, Interactive, Network, Owner) of groups.

Consequences

The DAC pattern has the following advantages:

- Users can self manage the privileges.
- The burden of security administrators is significantly reduced, as resource users and administrators jointly manage permissions.
- Per-user granularity for individual access decisions as well as coarse-grained access for groups are supported.
- It is easy to change privileges.
- Supporting new privileges is easy.

The DAC pattern has the following disadvantages:

- It is not appropriate for multilayered systems where information flow is restricted.
- There is no mechanism of restricting rights other than revoking the privilege.
- It becomes quickly complicated and difficult to maintain access rights as the number of users and resources increases.
- It is difficult to know the "reasonable rights" for a user or group.
- There can be inconsistencies in policies since individual users can give away access permissions to others.
- Read access can be given to unknown users to the owner of the object since the user granted by the owner can give away read access to other users.

See Also

The Authorization pattern [6] - is an Object Oriented pattern. It also has the concept of delegation as in th e DAC pattern. Unlike the DAC pattern, the request to access an object may not need to specify the particular object in the rule. It may be implied by an existing protected object.

3.2 Mandatory Access Control

The MAC pattern governs access based on the sensitivity of subjects and objects which is given according to a hierarchy of security levels. The MAC pattern is also known as multilevel security model and lattice-based access control [30].

Example

The DAC pattern controls access based on permissions that describe who should be allowed to access what object. There are two problems with the DAC pattern. First, there is nothing that prevents a user granted read access to a file by the owner of the file from copying the content of the file and granting read access to other users. Second, a user who wants to access a file for which he has access can write a Trojan horse program to copy the content of the file while the owner of the file performs some function of the program. For example, suppose a user *John* wants to access a file for which he does not have permission to access. Of course, the DAC system will not allow him to access the file since he has no permission to the file. However, if another user, who is granted read access to the file by the owner *Jane* of the file, grants *John* read access to the file, *John* can read the file without *Jane* being aware of. In another case, *John* could write a program for *Jane* that provides some useful function, and while *Jane* performs the function, the program can read and copy the file to a location where *John* can access. This is illustrated in Fig. 9.

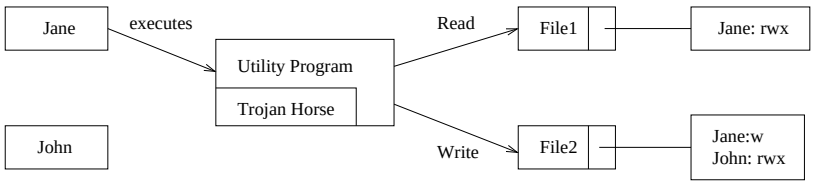


Figure 9: A Trojan Horse Problem in the DAC Pattern

Context

Development of access control systems that handle classified objects, and need to limit users actions according to the classifications.

Problem

The MAC pattern can solve the problems with the DAC pattern in multi-layered environment (e.g., military and government systems) by assigning security levels to users and objects. Thus, the solution of the DAC pattern can be considered as a problem of the MAC pattern. That is, if a system using the DAC pattern is placed in a multi-layered environment, the MAC pattern can be applied to improve the confidentiality. The MAC pattern can also be applied to the models that

have no access control enforced. In such cases, for the absence of access control, any user can access any object irrespective of security classification. Use the MAC pattern

- when the environment is multi-layered. A multi-level environment is the one where in users and objects are arranged. For example in the military domain users and files are classified into distinct levels of hierarchy like Unclassified, Public, Secret and Top Secret. User access to files is restricted based on the classification.
- when security policies need to be defined centrally. The access control decisions are to be imposed by a mediator (e.g., security administrator), and users should not be able to manipulate them.

Solution

The MAC pattern solves the above issues in classified environments by assigning security levels to users and objects as shown Fig. 10.

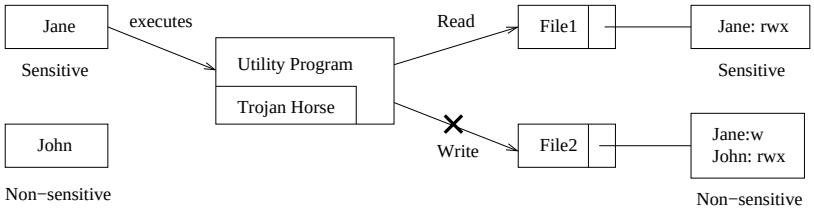


Figure 10: A Trojan Horse Solution in the MAC Pattern

In the MAC pattern, a user can read a file, but cannot write to the file if the security level of the file is lower than the security level of the user. If the security level of the file equals to or higher than that of the user, the user can write to the file, but cannot read the file.

Classification	Category
UNCLASSIFIED	U.S.
CONFIDENTIAL	U.S.
SECRET	U.S.
TOP SECRET	U.S.
UNCLASSIFIED	Allies
CONFIDENTIAL	Allies
SECRET	Allies
TOP SECRET	Allies

User	Classification	Category
John	SECRET	US
Jane	TOP SECRET	US
Smith	UNCLASSIFIED	Allies
Bill	UNCLASSIFIED	US

File	Classification	Category
File1	Secret	US

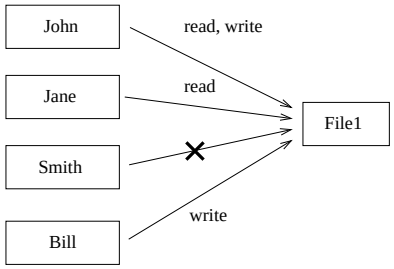


Figure 11: MAC Example

For example, consider an environment where documents are classified as shown in Fig. 11 In the diagram, *John* has read and write access to *File1* since his Classification and Category are same as that of the file. *Jane* can only read the file since her classification dominates the classification of *File1*. Writing to the file for *Jane* is prohibited by the MAC pattern. *Smith* cannot read as well as write to *File1* since his category is different than the category of *File1*. *Bill* can write to the file since the classification of the file dominates his classification, but cannot read the file.

Structure

The MAC pattern enforces access control based on the level of the user in a hierarchy. Fig. 12 shows the solution structure of the MAC pattern [18].

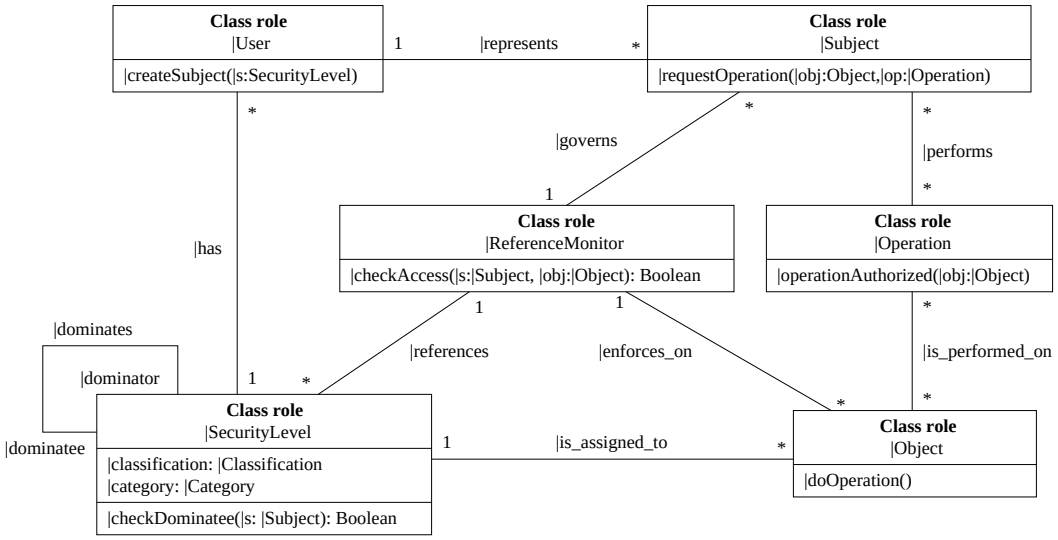


Figure 12: MAC Solution Structure

- *User* represents a user or group of users who interacts with the system. A user is assigned a hierarchical security level (e.g., SECRET, CONFIDENTIAL) and non-hierarchical category (e.g., U.S., Allies) to which the user belongs. A user may have multiple login IDs which can be active simultaneously. A user also may create and delete one or more subjects.
- *Subject* represents a computer process that acts on behalf of a user to request an operation on an object. For example, an ATM machine being used by a user can be viewed as a subject.
- *Object* represents any information resource (e.g., files, databases) in the system that can be accessed by user. An object is assigned a hierarchical security level and non-hierarchical category to which the object belongs.
- *Operation* is an action invoked by a subject to be performed on an object,
- *SecurityLevel* represents a classification assigned to users (subjects) and objects. The classifications are arranged in a hierarchy. Category represents any value from a non-hierarchical set.
- *ReferenceMonitor*: A user requests an operation on an object, and the request is checked for accessibility based on the following constraints.
  - Simple security property - A subject S is allowed a read access to an object O only if $L(S) \geq L(O)$.
  - Star property - A subject S is allowed a write access to an object O only if $L(S) \leq L(O)$.

  If the both constraints are satisfied, the operation may be performed, otherwise, access is denied. For example, in Fig. 11, a user with the category U.S. and classification as CONFIDENTIAL should be able to read an object with category U.S. and classification as UNCLASSIFIED. However, the same user should not be able read an UNCLASSIFIED document from the category Allies.

Dynamics

Fig. 13 shows the collaboration for requesting an operation [18]. The subject tries to invoke an operation on an object. The Security Level determines the dominatee and donimator. Also, it determines whether the access should be allowed or not. If the access is allowed, the subject performs the desired operation.

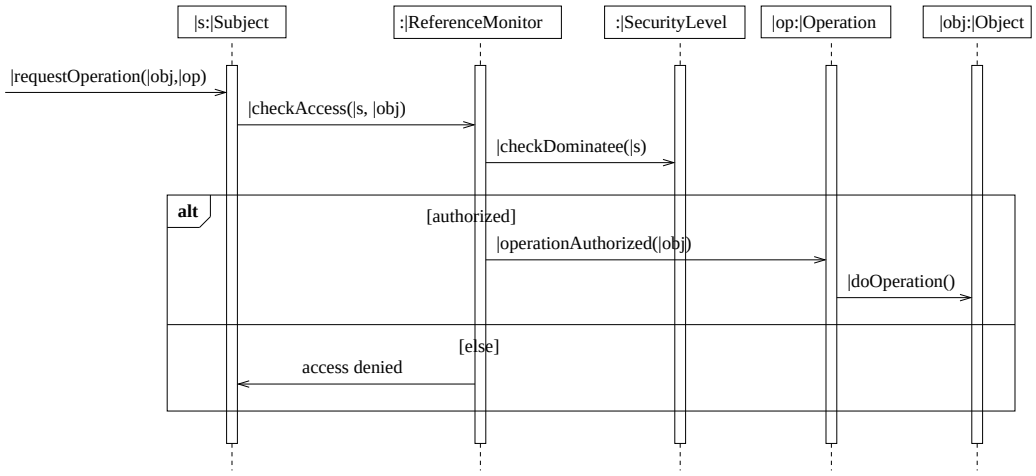


Figure 13: Requesting Operation

Variants

Biba integrity model [1] can be viewed as a variant of the MAC pattern, emphasizing on integrity rather than confidentiality. Biba model has access control properties (simple integrity property and integrity star property) similar to those in the MAC pattern, but their domination relations for read and write access are reversed.

Known Uses

Security-Enhanced Linux (SELinux) kernel [33], developed by NSA, MITRE Corporation, NAI labs and Secure Computing Corporation (SCC), enforces the MAC pattern to implement a flexible and fine-grained MAC architecture called Flask. Flask operates independently of the traditional Linux access control mechanisms. TrustedBSD [38], developed by the FreeBSD foundation, provides a set of trusted operating system extensions to the FreeBSD operating system, an advanced operating system for x86, amd64, and IA-64 compatible architectures. TrustedBSD contains modules that implement MLS (Multi-Level Security) and fixed-label Biba integrity policies (a variant of MAC). GeSWall (General Systems Wall) [12] developed by GentleSecurity is the Windows security project that implements the MAC pattern to provide OS integrity and data confidentiality transparently and invisibly to user.

Consequences

The MAC pattern has the following advantages:

- The system using the MAC pattern is secure to Trojan horse attacks.

- The assignment of a classification and category to users and objects is centralized by a mediator.
- The MAC pattern is appropriate for organizations with distinct levels of hierarchy.

The MAC pattern has following disadvantages:

- Introducing a new object or user requires a careful assignment of a classification and category.
- The mediator who assigns the classifications to users and objects should be a trusted person.

See Also

The Biba's Integrity pattern [1] - addresses integrity issues rather than confidentiality. Read and write accesses are based on integrity levels of subjects and objects. The Chinese Wall pattern [2] - is similar to the MAC pattern in that it has reading and writing rules, and the writing rule takes into account the Trojan horse problem. However, unlike the MAC pattern, but it has no distinction between users and subjects. Subjects include both users and processes acting on behalf of the user.

3.3 Role-Based Access Control

The RBAC pattern enforces access control based on roles to facilitate authorization management for a large number of users and resources.

Example

In a small organization where the application of an information system is narrow and the number of users and objects are low, access control such as the DAC pattern could be used by direct mapping between the users and objects. However, in a large organization, such a direct mapping becomes infeasible, requiring significant time and efforts to maintain the mapping. As an example, suppose a person is hired as a secretary and given permissions to read commercial letters, advertising letters, admission letters and conference letters. Once she is removed from the position, all permissions allocated to her has to be revoked. That is, the administrator has to change all respective entries in the access control list. If there are many such cases, it would require significant efforts for the administrator to maintain the access list.

Context

Development of access control systems that handle a large number of users and objects, and are expected to have frequent changes of access rights.

Problem

The MAC pattern is known to solve the Trojan horse problem (reading an unauthorized file and writing it to a new file) in the DAC pattern by using security levels. However, the MAC pattern still allows some security breaches known as "Covert Channels" (e.g., storage channels, timing channels) which can reveal certain information of the system by a Trojan horse program. For example, the Trojan horse program is able to transmit information such as when the program runs or waits, or the usability of shared resources (e.g., by creating and deleting bogus print jobs).

Another issue in traditional access control patterns (DAC, MAC) is that their use is limited to a specific domain. The DAC pattern rose from small and autonomous environments, and thus its

use is limited to environments like academics or small organizations. Similarly, the MAC pattern rose from rigid environments where users and information are classified. Thus its use is limited to environments like government and military. The RBAC pattern emerged as an alternative to traditional access control for other domains. Particularly, the RBAC pattern has gained great attention from the commercial domain where applications are large and networked for its economics in security administration. Unlike the domain supported by the DAC pattern where users own objects, in the commercial domain, the end users generally do not "own" the information to which they are allowed access.

The RBAC pattern can be used for a model that has concepts of roles, users and objects. A role represents a job function with certain authority and responsibility in an organization, and a user assigned to the role acquires the authority and responsibility given to the role. Use the RBAC pattern

- where control of data and application is restricted to the enterprise.
- where there are a large number of users and data objects to be managed for access control (e.g., e-commerce applications in cross-enterprise distributed networks).
- when only few security administrators are available.
- when the organization structure is stable, that is, there is infrequent change of job definitions.
- when there is frequent change of job responsibility or high job turnover, requiring dynamic response to enterprise policy changes.

Solution

The RBAC pattern overcomes the above problem by using the concept of "role" which is an abstraction of users. Instead of mapping directly users to objects, the RBAC pattern maps roles (e.g., Secretary, Manager) to permissions, and assigns users to the roles for which the users are authorized. The users acquire permissions given to the role. Because the access control in the RBAC pattern is defined in terms of relatively static entities of roles and permissions, access control becomes much simpler and more efficient to manage.

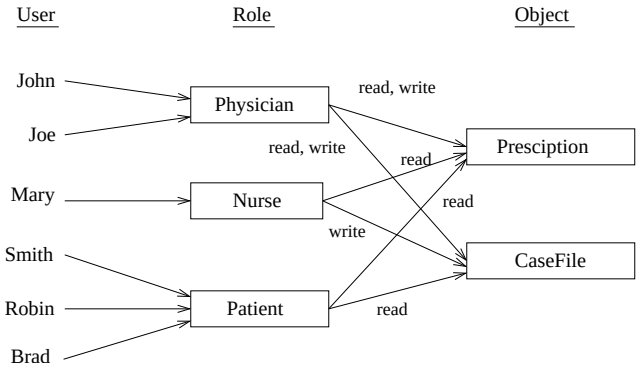


Figure 14: RBAC Example

Structure

Use of the RBAC pattern introduces concepts of Role, User, Session, Object, Permission and Operation as shown in Fig. 15.

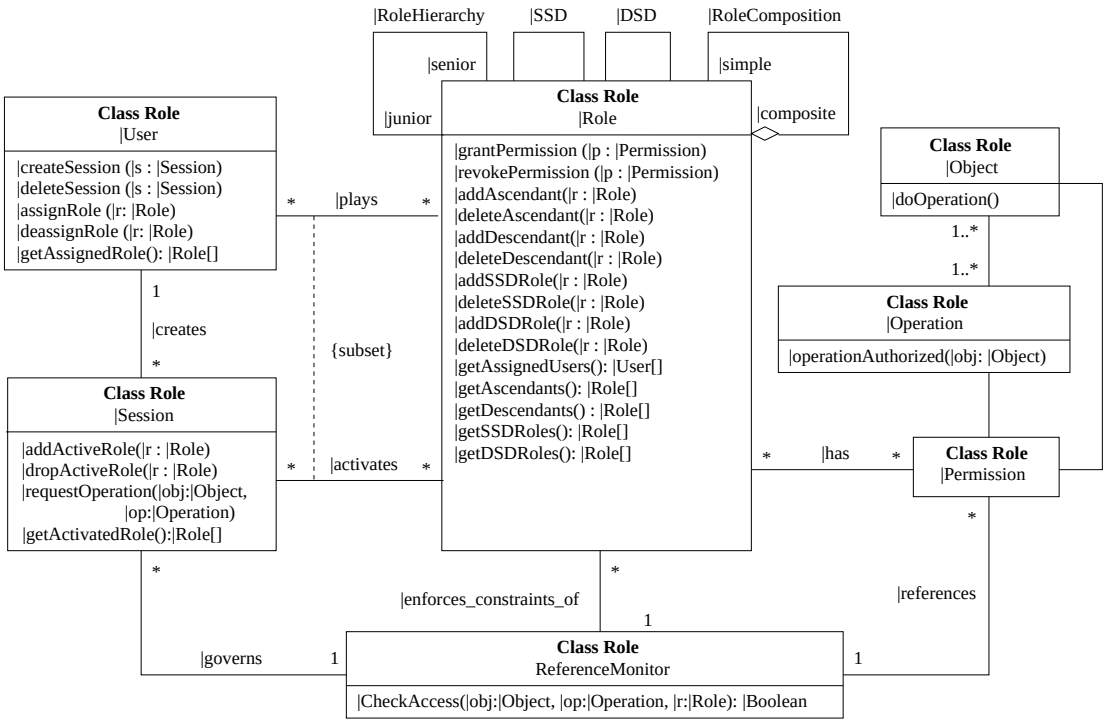


Figure 15: Solution Structure

- *Role* represents a job function with certain authority and responsibility in an organization. A role is represented as a relation of users and permissions, and a user assigned to the role acquires the permissions given to the role. Roles can have overlapping responsibilities and rights.
  Roles may be structured in a hierarchy to reflect an organization's lines of authority and responsibility. Role hierarchies define an inheritance relation among the roles in terms of permissions and user assignments. That is, role *r1* inherits role *r2* only if all permissions of *r2* are also permissions of *r1* and all users of *r1* are also users of *r2*.
  Two roles may have conflict of interests each other that prevents a user to be assigned to both roles. That is, a user whose membership in one role cannot be a member of the other conflicting role. Static Separation of Duties (SSD) constraints are defined to prevent assigning conflicting roles to a single user. SSD constraints are enforced during user assignment.
  Two roles may also have another type of conflict of interests that they cannot be activated within the same user session. Dynamic Separation of Duties (DSD) constraints are defined for such conflicting roles, and enforced during role activation within a session. If one role in a DSD constraint is activated, the user cannot activate the other conflicting role in the same session.
- *User* is a person who interacts with a computer system. A user may have multiple login IDs which can be active simultaneously. A user can create and delete a session.
- *Session* represents an instance of a user's dialog with a system. A session is a mapping of a user and a set of activated roles assigned to the user. A session can activate and deactivate a role, and a user may have multiple sessions running simultaneously.
- *Object* represents any information resource (e.g., files, databases) to be protected in the

system.

- *Operation* is an action to be performed on an object, invoked within a session. Examples of operations in a file systems are read, write and execute, and in a database management systems, insert, delete, append and update.
- *Permission* represents an authorization to perform an operation on an object or multiple objects. A permission is composed of an operation and an object on which the operation is performed. Permission checks whether or not the requested operation can be performed on the target object.
- *ReferenceMonitor*: A user requests an operation on an object, and the request is checked for accessibility based on SSD, DSD and role hierarchy constraints for the roles that the user plays.

Dynamics

Access is checked when a user is trying to perform an operation over an object. The role assigned to the user is checked whether the role has a permission to carry out the requested operation on the object, otherwise access is denied. Fig. 16 shows a collaboration for checking access.

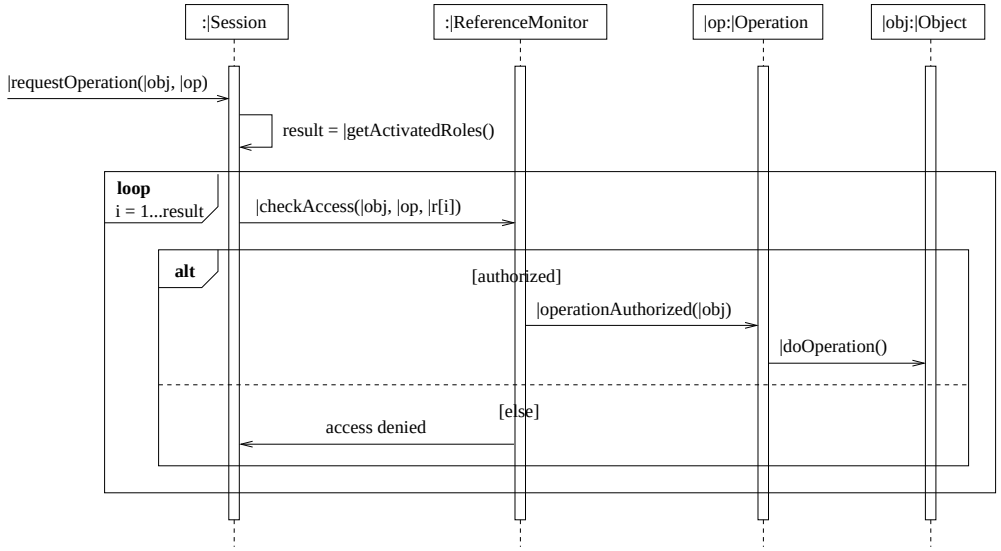


Figure 16: Requesting Operation

Variants

There are fours variants (RBAC0, RBAC1, RBAC2, RBAC3) depending upon the different properties of the RBAC pattern. First and most underlying is RBAC0. It has basic rules of role assignment, role authorization and transaction assignment. Second is RBAC1 which is an add-on to RBAC0, has property of hierarchies. Third variant is RBAC2 which provides the support of SoD (Separation of Duty). The last form, RBAC3 is a combination of RBAC0, RBAC1 and RBAC2, and the RBAC pattern described in this paper addresses RBAC3. The RBAC pattern can be used with the DAC pattern [28] and MAC pattern [23, 26].

Known Uses

The Sun ONE Identity Server [34] uses the RBAC pattern to map business functions to a logical group of users by employing roles to define logical group of users. Sun's J2EE [34] uses the RBAC pattern for the authorization service for E-Commerce applications by plugging the RBAC pattern into existing password management systems as provided by the UNIX and Windows [35]. The RBAC pattern is implemented in Solaris 8 for user login and restricting access to tools and utilities. IBM uses RBAC for security within WebSphere Portal [39] separating users into guests, users, administrators and super users. In Oracle applications, roles are defined to determine what data and functions within an application a user has access to [21].

Consequences

The RBAC pattern has the following advantages:

- The RBAC pattern greatly reduces the complexity of access control for a large number of users and objects by using roles instead of users since in general, there are much more users than roles. This also facilitates updating access rights of users.
- Organization policies about job functions can be dynamically reflected in the definition of roles. If there is a change of access rights for a role, the change can be made via role without interrupting user's work .
- A user can activate multiple sessions at a time, and a session can activate multiple roles assigned to the user. This improves the functional flexibility,
- Users are given only the necessary access privileges to perform their duties on an assigned role. This is also known as *least privilege*.
- Users can be easily given a different set of permissions by reassigning them to a different role.

The RBAC pattern has the disadvantage:

- The additional concepts (e.g., roles, sessions) and their related constraints (e.g., SSD, DSD) add complexity to implementation.

See Also

The Abstract Session pattern [25] - is similar to the RBAC pattern, but gives more focus on network sessions, In general, both patterns are seen and could be implemented together in a networking environment. The Abstract Session pattern provides a way for an object to store per-client state without sacrificing type-safety or efficiency through creating sessions

The Thread Specific Storage pattern [32] - allows multiple threads to use one logically global access point to retrieve thread-specific data without incurring locking overhead. This is similar to RBAC where multiple users can play a single role, reducing the overhead of administrator.

4 Conclusion

We have presented pattern descriptions for DAC, MAC and RBAC patterns described in an extension of the UML. The use of pattern roles from the extension for describing the structure and behaviors of the patterns faciliates capturing varitions of pattern realizations. Pattern roles are played by model elements in an application domain, and a single pattern role can be played by multiple elements. The model elements playing a role must satisify the constraints defined for the role. One can remove or add more constraints to a role to adjust the eligibility of model elements to play that role.

The proposed pattern descriptions also provide more details on the problem domain of the patterns that can help developers choose a suitable pattern for a given problem. The pattern descriptions can also be used as a basis for formalizing the problem and solution domain of the access control patterns to support the pattern-based model development for secure systems [18].

5 Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. CCF-0523101. Any opinions, findings and conclusions or recomendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).

References

[1] K. J. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372, Bedford, MA. Air Force Electronic Systems Division, 1977.

[2] D. Brewer and M. Nash. The Chinese Wall Security Policy. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 206–214, Oakland, CA, 1989. IEEE Computer Society Press.

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *A System of Patterns: Pattern-Oriented Software Architecture*. Wiley, 1996.

[4] 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.

[5] T. Doan, S. Demurjian, T.C. Ting, and A. Ketterl. MAC and UML for Secure Software Design. In *Proceedings of 2nd ACM Workshop on Formal Methods in Security Engineering: From Specifications to Code*, Washington D.C., 2004.

[6] E. B. Fernandez and Pan R. A Pattern Language for Security Models. In *Proceedings of the 8th Conference on Pattern Language of Programs (PloP)*, Monticello, IL, 2001.

[7] D. Ferraiolo, D. K, and R. Chandramouli. *Role-Based Access Control*. Artech House, 1993.

[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] G. Florijn, M. Meijers, and P. van Winsen. Tool Support for Object-Oriented Patterns. In *Proceedings of the 11th European Conference on Object Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 472–495. Springer-Verlag, 1997.

[10] R. 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.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[12] General System Wall. http://www.securesize.com/GeSWall/.

[13] M. Grand. *Patterns in Java-A catalog of reusable design patterns illustrated with UML*. Wiley, 1999.

[14] A.L. Guennec, G. Sunye, and J. Jezequel. Precise Modeling of Design Patterns. In *Proceedings of the 3rd International Conference on the Unified Modeling Language (UML)*, pages 482–496, York, UK, 2000. Springer-Verag, LNCS 1939.

[15] M.H. Harrison, W.L. Ruzzo, and J.D. Ullman. Protection in Operating Systems. *Communications of the ACM*, 19(8):461–471, August 1976.

[16] D. Kim. *A Meta-Modeling Approach to Specifying Patterns*. PhD thesis, Colorado State University, Fort Collins, CO, 2004.

[17] D. Kim, R. France, S. Ghosh, and E. Song. A Role-Based Metamodeling Approach to Specifying Design Patterns. In *Proceedings of the 27th IEEE Annual International Computer Software and Applications Conference (COMPSAC)*, pages 452–457, Dallas, USA, 2003. IEEE Computer Society Press.

[18] D. Kim and P. Gokhale. A Pattern-Based Technique for Developing UML Models of Access Control Systems. In *Submission to the 30th Annual International Computer Software and Applications Conference (COMPSAC)*, Chigaco, IL, 2006. To be published.

[19] B.W. Lampson. Protection. In *Proceedings of the 5th Princeton Conference on Information Sciences and Systems*, pages 437–443, Princeton, 1971. Reprinted in ACM Operating Systems Rev. 8, 1 (Jan. 1974), pp 18-24.

[20] A. Lauder and S. Kent. Precise Visual Specification of Design Patterns. In *Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP)*, pages 114–136. Springer-Verlag, LNCS 1445, 1998.

[21] L. Notargiacomo. Role-Based Access Control In ORACLE7 And Trusted ORACLE7. In *Proceedings of the 1st ACM Workshop on Role-Based Access Control*, page 17, Gaithersburg, MD, 1995.

[22] Oracle9i. http://www.oracle.com/technology/products/oracle9i/datasheets/ols/OLS9iR2_ds.html.

[23] S. L. Osborn, R. Sandhu, and Q. Munawer. Configuring Role-Based Access Control to Enforce Mandatory and Discretionary Access Control Policies. *ACM Transactions on Information and System Security*, 3(2):85–106, 2000.

[24] W. Pree. *Design Patterns for Object-Oriented Software Development*. Addison Wesley, 1995.

[25] N. Pryce. Abstract Session: an object structural pattern. In L. Rising, editor, *Design Patterns inCommunications Software*, pages 191–208. Cambridge University Press, New York, USA, 2001.

[26] I. Ray, N. Li, D. Kim, and R. France. Using Parameterized UML to Specify and Compose Access Control Models. In *Proceedings of the 6th IFIP TC-11 WG 11.5 Working Conference on Integrity and Internal Control in Information Systems (IICIS)*, Lausanne, Switzerland, 2003.

[27] James Rumbaugh, Michael R. Blaha, William Premerlani, Frederik Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[28] R. Sandhu and Q. Munawer. How To Do Discretionary Access Control Using Roles. In *Proceedings of the 3rd ACM Workshop on Role-Based Access Control (RBAC-98)*, Fairfax, VA, 1998. ACM Press.

[29] R. Sandhu and P. Samarati. Access Control: Principles and Practice. *IEEE Communications*, 32(9):40–48, September 1994.

[30] R. S. Sandhu. Lattice-Based Access Control Models. *IEEE Computer*, 26(11):9–19, 1993.

[31] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. Wiley, 2000.

[32] D. C. Schmidt, T.H. Harrison, and N. Pryce. Thread-Specific Storage - An Object Behavioral Pattern for Accessing per-Thread State Efficiently. *In the C++ Report*, 9(10), 1997.

[33] SELinux. http://www.nsa.gov/selinux/.

[34] Sun One Identity Server. http://sunflash.sun.com/articles/62/4/iplanet/9662.

[35] M. M. Swift, A. Hopkins, P. Brundrett, C. Van Dyke, P. Garg, S. Chan, M. Goertzel, and G. Jensenworth. Improving the granularity of access control for Windows 2000. *ACM Trans. on Information and System Security*, 5(4):398–437, November 2002.

[36] The Object Management Group (OMG). Unified Modeling Language: Superstructure. Version 2.0 Formal/05-07-04, OMG, http://www.omg.org, August 2005.

[37] TOE. http://niap.nist.gov/cc-scheme/st/ST_VID4048.html.

[38] TrustedBSD Project. http://www.trustedbsd.org/.

[39] WebSphere Portal for Multiplatforms. http://www-306.ibm.com/software/genservers/portal/.