A Rule-Based Framework for Role-Based Delegation

Longhua Zhang
College of Information Technology, UNC Charlotte
Charlotte, NC 28223, USA
lozhang@uncc.edu

Gail-Joon Ahn
Dept. of Computer Science, UNC Charlotte
Charlotte, NC 28223, USA
gahn@uncc.edu

Bei-Tseng Chu
Dept. of Software Information System, UNC Charlotte
Charlotte, NC 28223, USA
billchu@uncc.edu

ABSTRACT

In current role-based systems, security officers handle assignments of users to roles. However, fully depending on this functionality may increase management efforts in a distributed environment because of the continuous involvement from security officers. The emerging technology of role-based delegation provides a means for implementing RBAC in a distributed environment with empowerment of individual users. The basic idea behind a role-based delegation is that users themselves may delegate role authorities to other users to carry out some functions on behalf of the former. This paper presents a role-based delegation model called RDM2000 (role-based delegation model 2000), which is an extension of RBDM0 by supporting hierarchical roles and multi-step delegation. The paper explores different approaches for delegation and revocation. Also, a rule-based language for specifying and enforcing the policies based on RDM2000 is introduced.

Keywords: Role, Access Control, Delegation, Rule-Based

1. INTRODUCTION

In current role-based systems, security officers handle assignments of users to roles. This function may increase management efforts in a large-scale, highly decentralized environment because of the continuous involvement from security officers. Delegation is a necessary approach to enhance the scalability of a distributed system since it enables decentralization of administration tasks. The emerging technology of role-based delegation provides a means for implementing RBAC in a fully distributed environment with empowerment of individual users.

Delegation is an important factor for secure distributed computing environment. In the simplest case, *Alice* delegates her role to *Bob*. Upon *Bob*'s request, a service will be granted if the requested service is already granted to *Alice*. Naturally, access decisions need to take this delegation notion into account. Other related issues include revocation of delegated roles as well as how to specify and enforce security policies regarding delegation and revocation. For example, *Alice* may want to revoke *Bob* from a delegated role. A revocation mechanism must be provided and security policies must specify this action.

In this paper, a rule-based framework for role-based delegation is presented. Generally speaking, a rule-based system is a system where behaviors are governed by a set of explicit rules. The framework includes a role-based delegation model called RDM2000 and a rule-based language for specifying policies based on RDM2000. The enforcement of policies is also discussed.

The rest of this paper is organized as follows. Section 2 describes motivations of this work and related works. Section 3 defines the basic components of RDM2000 including delegation and revocation. In section 4, we describe the semantics of rule-based specification language for expressing and enforcing delegation and revocation policies. Possible extensions are discussed in section 5. Section 6 concludes this paper.

2. MOTIVATIONS AND RELATED WORKS

In this section, we introduce motivations behind this work and give an overview of related works.

2.1 Motivation

Delegation requirements arise when a user may need to act on another user's behalf for accessing resources. There are many definitions in the literature. In general, delegation is referred to as one active entity in a system delegates its authority to another entity to carry out some functions on behalf of the former. Different types of delegation have been proposed. The most common delegation types include user to machine, user to user, and machine-to-machine delegation [2, 7, 8, 10, 11].

Two views of delegation were summarized [15]: In administratively directed delegation, an administrative infrastructure outside the direct control of a user mediates delegation, e.g. a security officer must mediate all delegations. In user directed delegation, any user's system may mediate delegation to resources under the user's control. However, in both situations it is necessary to enforce predefined delegation policies to prevent power abuses by individual users.

Barka and Sandhu further identified a set of characteristics related to delegation including *permanence*, *totality*, and *levels of delegation* [8]: *permanence* refers to types of delegation in terms of their time duration; *totality* refers to how completely

SACMAT'01, May 3-4, 2001, Chantilly, Virginia, USA.

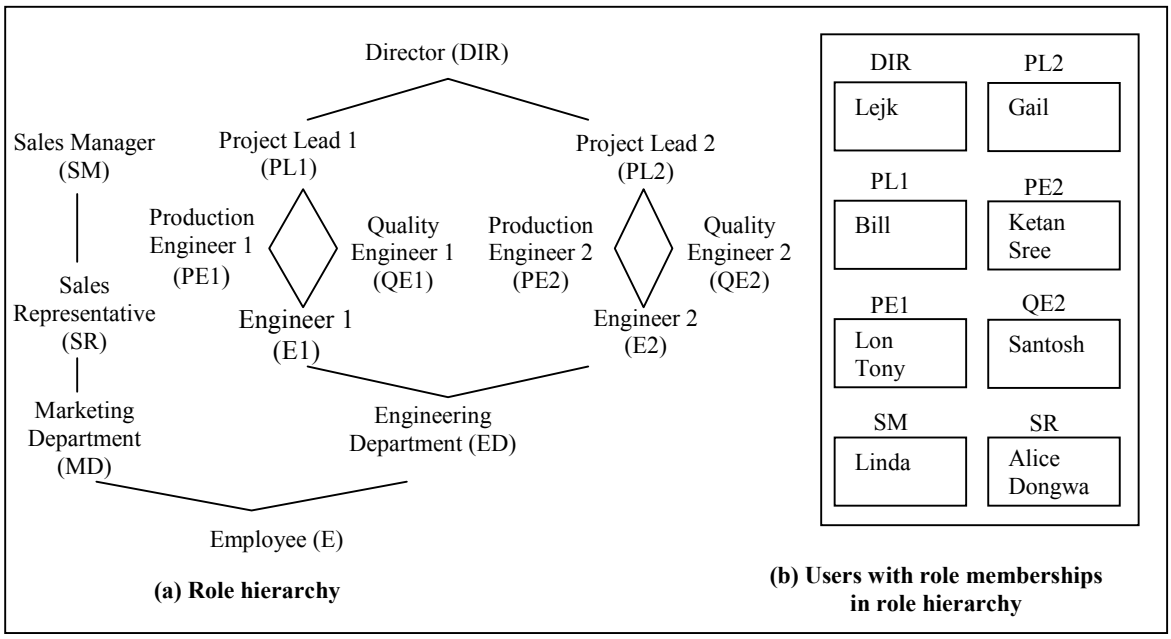


Figure 1: An example of organizational role hierarchy and users

the permissions assigned to the role are delegated; and *levels of delegation* defines whether or not each delegation can be further delegated and for how many times.

We focus on user-to-user delegation; especially a user delegates his role to another user. In our approach, we also deal with user directed delegation including multi-step delegation in a role hierarchy.

2.2 Related Works

Role-based access control is an enabling technology for managing and enforcing security in large-scale and enterprise-wide systems. In the past few years, researchers and vendors have proposed many enhancements of RBAC models. A general model, commonly called RBAC96 [18, 16] has become widely accepted by the information security community. In RBAC96, the central notation is that permissions are associated with roles, and users are assigned to appropriate roles. Users can be easily reassigned from one role to another. Roles can be granted new permissions. And permissions can be easily revoked from roles as needed. This greatly simplified management of permissions [18]. Our framework is based on the RBAC96 model.

A work closely related to ours is RBDM0 model proposed by Barka and Sandhu [7, 8]. They developed a simple role-based delegation model. They explored some issues including revocation, delegation with hierarchical roles, partial delegation, and multi-step delegation. And they formalized the delegation model with total delegation and flat roles. One limitation of RBDM0 is that this work does not address the relationships among each component of a delegation, which is a critical notion to the delegation model.

In ARBAC97 [17], Sandhu et al. developed URA97 for user role assignment. This assignment is handled by security officers. The basic concept in ARBAC97 implies that we can use RBAC for managing RBAC itself. This provides great administrative convenience and scalability. However, it may increase management efforts in a distributed environment because of the continuous involvement from security officers. Our work borrows the role assignment mechanisms in URA97 to support role-based delegation.

A number of researchers have looked at the semantics of authorization, delegation, and revocation. Li et al. proposed a logic for authorizing delegation in large-scale, open, distributed systems [13, 14]. But in their logic, role-based concepts were not fully adopted; neither did they address revocation adequately. In [12], Jajodia et al. proposed a logical language (ASL) for expressing authorization. ASL supports multiple access control policies. ASL is not role-oriented framework while we focus exclusively on a language that can specify and enforce policies for authorizing role-based delegation and revocation. This kind of language for role-based delegation has not been studied in the literature.

3. A FRAMEWORK FOR ROLE-BASED DELEGATION AND REVOCATION

In this section we propose a delegation model called RDM2000. This model supports role hierarchy and multi-step delegation by introducing the delegation relation. Our work is based on the framework of the RBAC96 model [18, 16] and the RBDM0 model [7, 8]. Figure 1 illustrates an organizational role hierarchy and users' role memberships. To illustrate each functional component in this model, we use this example in the rest of this paper.

Figure 1(a) shows the regular roles hierarchy in an organization. There is a junior-most role *E* to which all employees in the organization belong. There are two departments in this organization: engineering department and marketing department. Within the engineering department there is a junior-most role *ED* and senior-most role *DIR*. There are two projects in this department; each project has a senior-most project lead role (*PL1*, *PL2*) and junior-most engineering role

(*E1*, *E2*). Between these two roles there are two incompatible roles: production engineering (*PE1*, *PE2*) and quality engineering role (*QE1*, *QE2*). Within the marketing department there are only three roles: senior-most sales manager role (*SM*), junior-most marketing department role (*MD*), and sales representative role (*SR*) in the middle. Figure 1(b) shows users and their role memberships after user-role assignment by security officers. Security officers assign these roles to users.

3.1 Assumptions

The scope of our model is to address user-to-user delegation based on role hierarchies. In RBAC96, security officers handle the user-role assignment. In our model, the delegation from one user to another is actually assigning the delegated role to a user. Thus, the delegating user needs to do user-role assignment too. However, it is necessary to distinguish clearly between user-role assignment and role delegation. In a user-role assignment, the security officer must activate the administrative role(s), while in role delegation, the delegating user may need to activate his/her regular role(s). Although it is possible to delegate an administrative role, we only consider the regular role delegation in this paper. Enabling administrative role delegation may lose control of the regular role proliferation.

We have the following assumptions in the RDM2000 model. We assume that each user-role assignment is unique, so that a delegation can be identified by a unique user role assignment by delegated users. We also assume that a user who is a member of role *r* (whether it is explicit or implicit membership) cannot be delegated the same role *r*, thus preventing cycles in delegations. For example, project leader *Bill* with role *PL1* cannot be delegated the role *PE1* or *QE1* since he has been a member of these roles implicitly (these roles are junior to role *PL1*). However, the role *DIR* can be delegated to *Bill* since he is not a member of this role.

In some cases, we may need to define whether or not each delegation can be further delegated and for how many times, or up to the maximum delegation depth. We introduce two types of delegation: single-step delegation and multi-step delegation. Single-step delegation does not allow the delegated role to be further delegated; multi-step delegation allows delegated user to further delegate the delegated role until it reaches the maximum delegation depth. The maximum delegation depth is a natural number defined to impose restriction on the delegation (It will be discussed in section 3.2.2). Single-step delegation can be treated as a special case of multi-step delegation with maximum delegation depth equal to one.

3.2 Basic Elements and System Functions

Figures 2 defines the basic elements on which our framework is based and system functions that are used in this paper. It also shows the original RBAC96 and RBDM0 components.

[1]. In order to keep it consistent with RBAC96 model, we do not include the duration constraint in this relation. However, we believe duration is an important component in the delegation, and we will address the duration issue in our future work.

3.2.1 Basic elements and system functions: from RBAC96 and RBDM0

We consider, essentially, the following entity sets in our framework: users *U*, roles *R*, permissions *P*, sessions *S*, and constraints (see definition 1 in figure 2).

A user in this model is a human being, a role is a job function or job title and permission is an approval of executing an object method (access to one or more objects, or privileges to carry out a particular task). There are two sets of users involved with a role *r (We extend RBDM0 flat roles to support role hierarchy)*:

- Original users *Users_O(r)* are the users who are assigned to role *r* (or a role(s) senior to *r*) by original user-role assignment;
- Delegated users *Users_D(r)* are the users who are assigned to role *r* (or a role(s) senior to *r*) by delegated user-role assignment.

A user can be an original user of one role and a delegated user of another role. Also it is possible for a user to be both an original user and a delegated user of the same role. For example, if *Lejk* delegates his role *DIR* to *Bill*, then *Bill* is both an original user (explicitly) and a delegated user (Implicitly) of role *PL1*.

A session is a mapping between a user and a set of activated roles. The function *User(s)* returns the user associated with a session and *Roles(s)* returns the roles activated in a session.

Hierarchies are a natural means for structuring roles to reflect an organization's lines of authority and responsibility. The Hierarchies are partial orders. A partial order is a reflexive, transitive, and anti-symmetric relation.

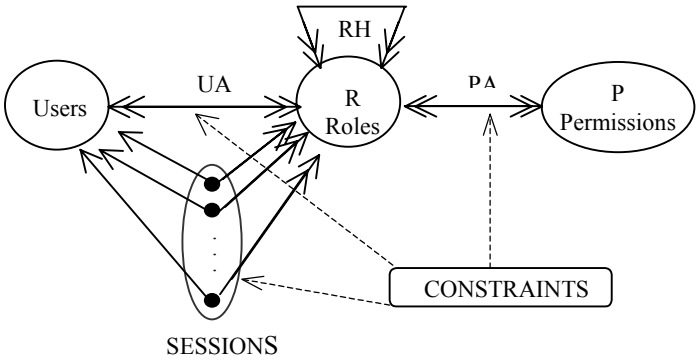
The user assignment *UA* is a many-to-many relation between users and roles. The original user assignment *UAO* is a many-to-many relation between original users and roles. The delegated user assignment *UAD* is a many-to-many relation between delegated users and roles. The permission assignment *PA* is a many-to-many relation between permissions and roles. Users are authorized to use the permissions of roles to which they are assigned.

3.2.2 Basic elements and system functions: beyond RBAC96 and RDBM0

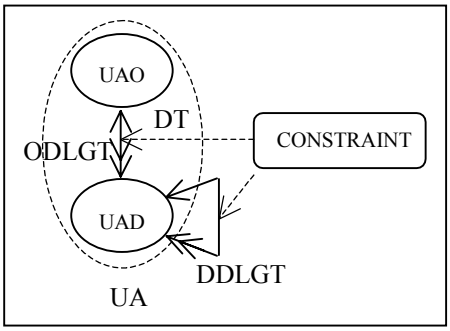
Definition 2 in figure 2 defines additional elements and system functions used in RDM2000.

We define a new relation in RDM2000 called delegation relation (*DLGT)* (see figure 2(b)). The motivation behind this relation is to address the relationships among different components involved in a delegation. In a user-to-user delegation, there are altogether four components[1]: a delegating user, the delegating role, a delegated user, and the delegated role. For example, *(Gail, PL2, Dongwa, QE2)* means *Gail* acting in role *PL2* delegates role *QE2* to *Dongwa*. A delegation relation further divides into original user delegation (*ODLGT*) and delegated user delegation (*DDLGT*) in a multi-step delegation.

Based on this relation we build a set of functions. For example, function *Prior* maps one *UA (u1, r1)* to *another UA (u2, r2)* or *Ø*, so that *(u2, r2, u1, r1)* $\in$ *DLGT* or *(u1, r1)* $\in$ *UAO*; function *Path* maps a *UA* to a delegation path; and function *Depth* returns the depth of the delegation path.



(a) RBAC96 model



(b) A delegation relation

Definition 1 The following is a list of original RBAC96 and RBDM0 components:

- P, R, U, and S are sets of permissions, roles, users, and sessions respectively
- $UA \subseteq U \times R$ is a many to many user to role assignment relation
- $PA \subseteq P \times R$ is a many to many permission to role assignment relation
- $RH \subseteq R \times R$ is a partial order on R called the role hierarchy or role dominance relation
- $Users\_O(r) = \{u \mid (\exists r' \geq r')(u, r') \in UAO\}$
- $Users\_D(r) = \{u \mid (\exists r' \geq r')(u, r') \in UAD\}$
- $Users(r) = Users\_O(r) \cup Users\_D(r)$, note that it is possible $Users\_O(r) \cap Users\_D(r) \neq \emptyset$
- Users: $R \rightarrow 2^U$ is a function derived from UA mapping each role to a set of users, where $Users(r) = \{u \mid (u, r) \in UA\}$
- User: $S \rightarrow U$ is a function that maps each session to a single user, $user(s_i) = u \mid (u, r) \in s_i\}$
- Roles: $S \rightarrow 2^R$ is a function that maps each session s_i to a set of roles, where $Role(s_i) \subseteq \{r \mid (\exists r \leq r')\ [(user(s_i), r') \in UA]\}$
- Permissions: $S \rightarrow 2^P$ is a function derived from PA mapping each session s_i to a set of permissions where $Permissions(s_i) = \{p \mid [(\exists r \leq r')\ (p, r') \in PA, r' \in Role(s_i)]\}$
- $UAO \subseteq U \times R$ is a many to many original user to role assignment relation
- $UAD \subseteq U \times R$ is a many to many delegated user to role assignment relation
- $UA = UAO \cup UAD$

Definition 2 The following is a list of components added in our delegation model:

- N is a set of natural numbers
- $DLGT \subseteq UA \times UA = U \times R \times U \times R$ is a many to one delegation relation
- $ODLGT \subseteq UAO \times UAD$ is an original user delegation relation
- $DDLGT \subseteq UAD \times UAD$ is a delegated user delegation relation
- $DLGT = ODLGT \cup DDLGT$
- $DT \subseteq UA \times UA$ is a delegation tree
- Prior: $U \cup R \rightarrow U \times R$ is a function that maps Each UA to another UA or Ø
  $Prior(u, r) = \{(u', r') \mid (u, r) \in UAD, (u', r', u, r) \in DLGT\}$
  $Prior(u, r) = \{ \emptyset \mid (u, r) \in UAO \}$
- A delegation path is a set of ordered user role assignment
  $Path(u_0, r_0) = \{(u_0, r_0), (u_1, r_1), \ldots, (u_i, r_i), \ldots, (u_n, r_n) \mid (u_i, r_i) = Prior(u_{i-1}, r_{i-1}) \in UA \text{ when } i>0\}$
  $Path(u, r) = \{\emptyset \mid (u, r) \in UAO \}$
- Depth: $U \cup R \rightarrow N$. A delegation depth is the number of elements in a delegation path minus one
  $Depth(u, r) = \{n \mid n = |Path(u, r)|, (u, r) \in UAD \}$
  $Depth(u, r) = \{0 \mid (u, r) \in UAO \}$

Figure 2: Basic elements and system functions for RDM2000

Table 1. Example of can_delegate with prerequisite roles

Delegating Role	Prerequisite Condition	Max. Depth	Candidate Delegated Role Set
PL1	E1	2	{PE1, QE1, PL1}
PL1	SR	1	{ED, PE1, QE1, PL1}
QE2	SR&(-QE1)	1	{ED, E2, QE2}

A delegation path is a set of ordered user-role assignment relations. A delegation path is generated when a multi-step delegation is applied. A delegation path always starts from an original user-role assignment. Delegation paths starting with the same original user-role assignment can further construct a delegation tree. A delegation tree is a user-role assignment/delegation hierarchy. Each node in the tree refers to a user-role assignment, each edge to a delegation relation. The layer of a user-role assignment in the tree is referred as the delegation depth. For example, we have the following set of delegation relations:

D1: (Lejk, DIR, Linda, PL1) ∈ DLGT
D2: (Linda, PL1, Alice, PE1) ∈ DLGT
D3: (Linda, PE1, Dongwa, PE1) ∈ DLGT
D4: (Lejk, DIR, Tony, QE2) ∈ DLGT

From above delegations, we can get delegation paths *P1*, *P2*, *P3*, and *P4* by applying *Path* function. Then we can construct a tree from these paths (See figure 3).

DLGT	Delegation Path
D1	P1: (Linda, PL1), (Lejk, DIR)
D2	P2: (Alice, PE1), (Linda, PL1), (Lejk, DIR)
D3	P3: (Dongwa, PE1), (Linda, PL1), (Lejk, DIR)
D4	P4: (Tony, QE2), (Lejk, DIR)

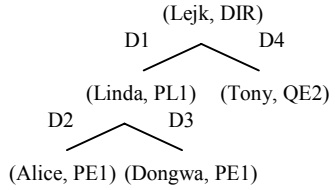


Figure 3: An example for delegation paths and delegation tree
D1, D2, D3, and D4 stand for a delegation relation;
P1, P2, P3, and P4 denote a delegation path.

To impose restrictions on such a hierarchy, we need limit the depth as well as the width of the delegation tree. There are three issues to control depth of a delegation: *no control*, *boolean control*, and *integer control* [20]. Using no control imposes no restriction on role proliferation. Boolean control can impose restriction on the depth as well as the width of the delegation tree; the delegating user decides whether or not the delegated user can further delegate the delegated role. The role proliferation depends totally on users themselves using the boolean approach. There is no high level restriction, say a policy, to limit the maximum depth of a delegation. Using integer control can limit the maximum depth, but the drawback is that it has no control on the width of the delegation tree, so it is not a tight control on role proliferation. A better solution may choose integer control at a high level (delegation policy) to restrict the maximum depth as well as boolean control at a low level (individual delegation) to restrict the width of a delegation.

3.3 Delegation

We adopt the notation of prerequisite condition from ARBAC97. The prerequisite conditions can impose restrictions on which users can be delegated to a role.

Definition 3 A prerequisite condition CR is a boolean expression using the usual "&" (and) and "|" (or) operators on the term of cr, cr can be x (membership) or –x (non-membership), where x is a role term, for example, CR = cr_1 & cr_2 | (-cr_3).

Definition 4 The following relation authorizes user-to-user delegation in this framework:

- $can\_delegate \subseteq R \times CR \times N$

where R, CR, N are sets of roles, prerequisite conditions, and maximum delegation depth respectively.

The meaning of *(r, cr, n) ∈ can_delegate* is that a user, say *Bill*, who is a member of role *r* (or a role senior to *r*) can delegate role r (or a role junior to role *r*) to any other user, say *Alice*, whose current membership or non-membership, in roles satisfies the prerequisite condition *cr* without exceeding the maximum delegation depth *n*. For example, *(PL1, SR, 1) ∈ can_delegate*, then *Bill* can delegate role *PE1* to *Alice*, so that *(Bill, PL1, Alice, PE1) ∈ DLGT*. The meaning of *(r, ∅, n) ∈ can_delegate* is that a user who is a member of role *r* (or a role senior to *r*) can delegate role r (or a role junior to role *r*) to any other user. Table 1 shows examples of the *can_delegate* relation for the delegation tree in figure 3.

3.4 Revocation

Revocation is an important process that must accompany the delegation. For example, *Linda* delegated role *PE1* to *Alice*; however, *Alice* abuses her empowerment by leaking production secrecy to an external party. Thus she must be revoked from the delegated role *PE1* immediately.

In this section, our focus is exclusively on delegation revocation. There may exists two ways to revoke a delegation:

- using duration-restriction constraint
- and allowing user revocation.

3.4.1 Revocation using duration-restriction constraint

In this approach, a duration constraint is attached to each delegation relation so that when the assigned time expires, the delegation also expires. Duration-restriction revocation is a simple self-triggered process that ensures the automatic revocation of role membership. It is extremely useful when the attached duration is a small time period. It can eliminate the overhead of administrative effort of manually revoking a delegation. However, duration-restriction by itself is not enough to ensure security; and the time period must be set carefully since we might overset or under-set the time. Since we do not formalize the duration constraint in RDM2000, the duration-restriction revocation remains for our future work.

3.4.2 User Revocation

We consider two types of revocation in this category: grant-dependent and grant-independent revocation. Grant-dependent revocation means only users in the delegation path prior to a user can revoke his role membership. Grant-independent revocation

DLGT	Delegation Path before Revocation	Delegation Path after Revocation
D1	P1: (Linda, PL1), (Lejk, DIR)	N/A
D2	P2: (Alice, PE1), (Linda, PL1), (Lejk, DIR)	P2: (Alice, PE1), (Bill, PL1)
D3	P3: (Dongwa, PE1), (Linda, PL1), (Lejk, DIR)	P3: (Dongwa, PE1), (Bill, PL1)
D4	P4: (Tony, QE2), (Lejk, DIR)	P4: (Tony, QE1), (Lejk, DIR)

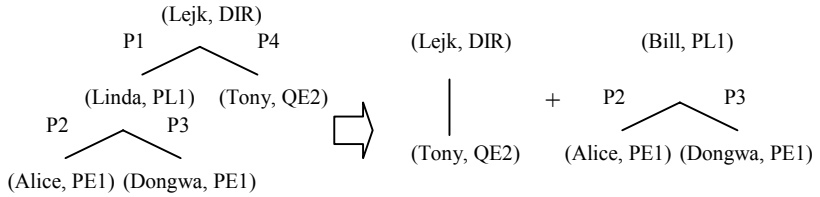


Figure 4: An example for Non-cascading revocation

means any original user of a role can revoke a delegated user from the role.

User revocation has two options: non-cascading and cascading revocation. Using the delegation tree in figure 4, suppose *Bill* is going to revoke *Linda* from role *PL1*. Cascading revocation implies that when *Linda* is revoked from role *PL1*, *Alice* and *Dongwa* are revoked from *PE1* subsequently. The non-cascade revocation means only *Linda* is revoked from *PL1*. Cascading revocation revokes a node from the delegation tree together with the sub-tree below the node, while non-cascading revocation only revokes the node. If the revoked node is not a leaf node, non-cascading revocation may leave a "*hole*" in the delegation tree, thus leaving conflicts in some delegation path. A possible solution might be *Bill* takes over the delegating user's responsibility from *Linda*, and changes all delegations: *(Linda, PL1, u', r') ∈ DLGT* to *(Bill, PL1, u', r') ∈ DLGT*. See figure 4.

We introduce two approaches for implementation of a cascading revocation: an *instant* implementation and a *dilatory* implementation. The instant implementation revokes all related delegations with the authorization of a cascading revocation. That is, if *Linda* is revoked from *PL1*, *Alice* and *Dongwa* will be revoked from *PE1* immediately. This can be achieved by browsing over the entire database table for all delegations: *{(u, r, u', r')|(Linda, PL1)∈ Path(u', r')}* and revoking them. The instant approach is difficult to implement in a distributed environment because of the computational complexity. The dilatory implementation adopts a run-time revocation. Only *Linda* will be revoked from role *PL1* with the authorization of a cascading revocation. After the revocation, *assignment (Linda, PL1)* does not belong to *UAD*. When *Alice* presents her *UAD* to activate the delegated role, system will check the status of each element in the delegation path: since *(Linda, PL1)* does not belong to *UAD*, the delegation path P2: *{(Alice, PE1), (Linda, PL1), (Lejk, DIR)}* is no longer valid. And this *UAD* will be revoked as well. This dilatory revocation does not lead to timing attacks since the delegation path must be validated when a user activates the delegated role.

Definition 5 The following relations authorize delegation revocation:

- *can_revokeGI* $\subseteq R$
- *can_revokeGD* $\subseteq R$

The meaning of *(b) ∈ can_revokeGI* is that a user, say *Bob*, whose current membership includes a delegated role *b*, can be revoked from the role by original user of role *b*. The meaning of *(b) ∈ can_revokeGD* is that all users who are prior to him in the delegation path can revoke a user, say Bob, who has current membership in delegated role b from the role. Table 2 and 3 show examples of these relations for the delegation tree in figure 3.

Table 2. Example of can_revokeGI

Revoked Role	Revoking Role Set	Revoked User Set	Candidate Revoking User Set
PL1	{PL1, DIR}	{Linda}	{Bill, Lejk}
PE1	{PE1, PL1, DIR}	{Alice, Dongwa}	{Lon, Tony, Bill, Lejk}
QE2	{QE2, PL2, DIR}	{Tony}	{Santosh, Gail, Lejk}

Table 3. Example of can_revokeGD

Revoked Role	Revoking Role Set	Revoked User Set	Candidate Revoking User Set
PL1	{PL1, DIR}	{Linda}	{Lejk}
PE1	{PE1, PL1, DIR}	{Alice, Dongwa}	{Linda, Lejk}
QE2	{QE2, PL2, DIR}	{Tony}	{Lejk}

Table 4: Mapping system functions

Mapping functions	RDM2000 system functions	Semantics
depth(u, r)	Depth: $U \cup R \rightarrow N$	Return the delegation depth of a (delegated) user-role assignment
juniorEQ(r, r')	$\leq$	Role r is junior to role r'
path(u, r)	Path: $U \cup R \rightarrow \{(u_0, r_0), \ldots (u_n, r_n)\}$	Return the delegation path of a (delegated) user-role assignment (u, r)
prior(u, r)	Prior: $U \cup R \rightarrow U \times R$	Return the user-role assignment previous to (u, r) in the delegation path
permissions(s)	Permissions: $S \rightarrow 2^P$	Return all activating permissions in a session
roles(s)	Roles: $S \rightarrow 2^R$	Return all activating roles in a session
seniorEQ(r, r')	$\geq$	Role r is senior to role r'
user(s)	User: $S \rightarrow U$	Maps each session to a single user
users(r)	Users: $R \rightarrow 2^U$	Return all users who are members of role r
users_o(r)	Users_O: $R \rightarrow 2^U$	Return all original users who are members of role r
users_d(r)	Users_D: $R \rightarrow 2^U$	Return all delegated users who are members of role r

In RDM2000, the delegation and revocation policies are embedded in its components (the delegation and revocation relations). Next, we propose a rule-based language for explicitly specification of these policies. In this language, policies are defined as a set of basic authorization rules. The authorization of delegation and revocation can be computed using a finite set of rules.

4. THT RULE-BASED POLICY SPECIFICATION LANGUAGE

RDM2000 defines policies that allow regular users to delegate their roles. It also specifies the policies regarding which delegated roles can be revoked. In this section we describe a rule-base language to enforce these policies. There are two reasons that we choose a rule-based language: first, the delegation and revocation relations defined in RDM2000 lead naturally to declarative rules; second, an individual organization may need to add local policies to further control delegation and revocation, a declarative rule-base system allows individual organization to easily incorporate such local policies. We emphasize the use of functions. We also show how these constructs can be used to express delegation and revocation policies. We demonstrate the enforcement of these polices as well.

4.1 The Language

The main purpose of the rule-based specification language is to enforce authorizations of delegation and revocation based on the RDM2000 model. A rule-based language is a declarative language in which binds logic with rules [19]. An advantage is that it is entirely declarative so it is easier for security administrator to define policies. The proposed language is a rule-based language with a clausal logic.

Definition 6 A *clause*, also known as a *rule*, takes the form:

$H \leftarrow F1\&F2\&\ldots\&Fn$

where H, F1, F2, …, Fn are boolean functions. The rule can be read as:

Table 5: Utility functions

RSPL Functions	Return value	Semantics
in(a, b)	Truth Value	Describe the membership between a and b. b is a member of a.
active(u, r, s)	Truth Value	Return true if users u have role r activated in session s
revoked(u, r)	Truth Value	Return true if any one of the user-role assignment in the delegation path of (u, r) was revoked
delegatable(u, r)	Truth Value	Return true if user u has the authority to further delegate role r. This function always returns true if (u, r) is an original user-role assignment.

Table 6: Authorization Functions

Basic authorization functions	Derived authorization functions	Semantics
	allow(u, p, s)	Refer to rule 4
can_delegate(dr, cr, n)	der_can_delegate(u1, u2, r, dlg_opt)	Refer to rule 1 and rule 5
can_revokeGD(r)	der_can_revokeGD(u1, u2, r, rvk_opt)	Refer to rule 2 and 7
can_revokeGI(r)	der_can_revokeGD(u1, u2, r, rvk_opt)	Refer to rule 3 and 8

to deduce H,
deduce F1 and
deduce F2 and
...
deduce Fn.

The fundamental element of our language is a set of functions. A function has a name, a set of arguments and a return value. Function itself can be an argument of another function. A function returning truth-value is also called a boolean function. There are three categories of functions: functions mapped from RDM2000 model, utility functions, and authorization functions.

4.2 Functions

There is a set of system functions defined in RBAC96, RBDM0 and RDM2000 models. We map these system functions to functions in the language (see table 4). Utility functions are general-purpose boolean functions including *in, active*, *revoked*, *expired, and delegatable* (see table 5). Authorization functions define authorization policies and enforcement of these policies. They further divide into basic authorization functions and derived authorization functions (see table 6). The semantics of mapping functions and utility functions are defined in their table respectively.

4.3 Basic Authorization Rules

Basic authorization rules take form $H \leftarrow$. They are predefined security policies embedded within each RDM2000 components.

Rule 1 **A user-user delegation authorization rule** is a rule of the form:

can_delegate(dr[1], cr, n) ← .
where dr, cr, and n are elements of roles, prerequisite conditions, and maximum delegation depths respectively.

This rule is the basic user to user delegation authorization policy extracted from *can_delegate relation* in RDM20000. It means that a member of the role *dr* (or a member of any role that is senior to *dr*) can assign a user whose current membership satisfies prerequisite condition *cr* to role *dr* (or a role that is junior to *dr*) without exceeding the maximum delegation depth *n*.

Rule 2 **A cascading grant-dependent revocation authorization rule** is a rule of the form:

can_revokeGD(r) ← .
where r is element of roles.

This rule is the basic cascading grant-dependent revocation authorization policy extracted from *can_ revokeGD relation* in RDM2000. It means that a member of the delegated role *r* (or a member of a delegated role that is junior to *r*) can be revoked membership of role *r* by all users who are prior to him in the delegation path.

Rule 3 **A cascading grant-independent revocation authorization rule** is a rule of the form:

can_revokeGI(r) ← .
where r is element of roles.

[1]. *dr is used* to indicate a delegating role and *r* to indicate a delegated role.

This rule is the basic cascading grant-independent delegation revocation policy extracted from *can_ revokeGI relation* in RDM2000. It means that a member of the delegated role *r* (or a member of a delegated role that is junior to *r*) can be revoked membership of role *r* by any original users of role *r*.

4.4 Authorization Derivation Rules for Enforcing Policies

The basic authorization specifies the policies defined in RDM2000. However, a user cannot be authorized delegation or revocation through basic authorization rules since these basic rules are specified based on roles instead of individual users. Further derivations are needed for enforcement of these policies.

4.4.1 Enforcement of Access Control Policies

Rule 4 An **access control rule** forms:

allow(u, r, p, s) ← *active(u, r, s) &*
in(p, permissions(s))
where u, r, p, and s are elements of users, roles, permissions, and sessions respectively.

This rule implies that permission *p* is granted user *u* with role *r* activated in session *s*.

Access control rule is essential since it says that whether user *u* belongs to original users of *r* or delegated users of *r*. This is a basic assumption of our delegation model.

4.4.2 Enforcement of Delegation Policies

Rule 5 **A user-user delegation authorization derivation rule** forms:

der_can_delegate(u1, u2, r, dlg_opt) ←
can_delegate(dr, cr, n)&
active(u1, dr1, s)&
delegatable(u1, dr1)&
seniorEQ(dr1, dr)&
in(u2, cr)&
juniorEQ(r, dr)&
in(depth(u1, dr1), n).
where u1 and u2 are elements of users; dr, dr1, and r are elements of roles; cr, and s are elements of prerequisite condition and sessions respectively; dlg_opt is a Boolean term, if it is true, then delegatable(u2, r) is true. This argument is used as Boolean control of delegation proliferation.

This rule means that user *u1* with a membership of a role senior to *dr* can assign user *u2* whose current membership satisfies prerequisite condition *cr* to role *r* (*r* is junior to role *dr*) without exceeding the maximum delegation depth *n*.

For example, if the security officer specified the following delegation policies:

Policy 1: can_delegate(PL1, SR, 2) ←.
Policy 2: can_delegate(PL1, E2, 1) ←.

Lejk needs to delegate role *PL1* to *Linda* (*s*uppose the rule engine will search delegation policies from *policy 1* to *policy 2*).

To deduce der_can_delegate(Lejkl, Linda,PL1, true),
deduce can_delegate(PL1, SR, 2)= true and
deduce delegatable (Lejkl, DIR) = true and
deduce active(Lejkl, DIR, s) = true and
deduce seniorEQ(DIR, PL1) = true and

deduce = true and
...
to deduce in(depth(Lejk, DIR), 2),
deduce depth(Lejk, DIR) return 0
deduce in(0, 2)=
true.

The delegation is authorized by applying *policy 1*. However, if *Bill* wants to delegate *QE1* to *Sree*, rule engine tries both delegation policies.

To deduce der_can_delegate(Bill,Sree,QE1),
deduce can_delegate(PL1, SR, 2)= true and
deduce active(Bill, PL1, s) = true and
...
deduce in(Sree, SR) = false, next available delegation policy
deduce can_delegate(PL1, E2,1)= true and
…
true

If deduction results for all available policies are false, then this delegation request will be denied. For example, delegation from *Gail* to *Linda* with role *PL2* is denied since there does not exist a policy for this case.

4.4.3 Enforcement of Delegation Revocation Policies

Rule 6 **Automatic revocation authorization rules** forms:

der_can_revoke_auto_cascade(u, r) ← revoked(u, r).
where u, r are elements of users and roles.

This rule means if any of user role assignments in the delegation path *Path(u, r)* is revoked excluding the last user role assignment in the delegation depth, user *u* is revoked automatically from role *r*.

Rule 7 **A cascading grant-dependent revocation authorization derivation rule** forms:

der_can_revokeGD(u1, u, r, rvk_opt) ← can_revokeGD (r)& active(u1, r1, s) & in((u1,r1), Path(u, r)).
where u1and u are elements of users, u1≠u; r is an element of role respectively. The rvk_opt is a Boolean term; it decides whether this revocation is cascading or non-cascading revocation. If it is true, the revocation is cascading, otherwise non-cascading.

This rule means that the user *u* can be revoked from role *r* by any other user *u1* with role *r1* activated, where *(u1, r1)* ∈ *Path(u, r)*.

Rule 8 **A cascading grant-independent revocation authorization derivation rule** is a rule of the form:

der_can_revokeGI(u1, u, r, rvk_opt)← can_revokeGI(r) & in(u1, users_o(u, r)).
where u1and u are elements of users, u1≠u; r is a element of role respectively. The rvk_opt is a Boolean term; it decides whether this revocation is cascading or non-cascading revocation. If it is true, the revocation is cascading, otherwise non-cascading.

This rule means that the user *u* can be revoked from role *r* by any original user *u1*.

In a revocation process, the authorization of a cascading revocation subsequently authorizes a set of automatic revocations by applying *rule 6,* no matter which implementation (dilatory or instant) is used; and the authorization of a non-cascading revocation subsequently coalesces a set of delegation path.

5. DISCUSSION

We have defined basic rules and derivation rules for specification and enforcement of policies embedded in RDM2000. We discuss possible extensions for the rule-based framework in this section. There are many possible extensions we may consider. First, a delegating user may need to delegate a role to all members of another role at the same time. For example, sales manager *Linda* may want to delegate role *PE1* to all sales representatives in marketing department. This type of delegation is more effective if we adopt a role-to-role delegation. In another case, Director *Lejk* may need to delegate his particular permissions to role *PL1*. This kind of permission-centric delegation may be useful for certain cases. However, integration of these two types of delegation into our framework will dramatically increase the complexity of model. To reduce the complexity of the integration of different kinds of delegation, it should be restricted within proper domain.

Second, constraints are an important aspect of RBAC96. They can be used as a power mechanism for laying out higher-level organization policies. Major examples include incompatible role assignment, separation of duties (SOD), and Chinese wall policy. We can always add new rules, such as an integrity rule: *error() ← F1&,…., Fn.* It derives an error every time the conditions in rule body *F* are satisfied. For example, two incompatible roles *PE1* and *QE1* in figure 1 can be specified using the following rule: *error()← in(u, users(PE1))& in(u, users(QE1)).* This also demonstrates one of the advantages of the rule-based languages: the rules are easily extended to include new features. This constraint representation will be studied in the future.

Also, it is important to ensure the correctness and convergence of rule derivations. The notion of correctness has several interpretations. First, there is an issue whether the rules impose a desired discipline on the defined policies. Second, one may wonder whether some rules would always terminate, and whether it may be free from internal conflicts. There are no formal techniques at this stage that would allow us to answer these questions in a complete general case. However, this lack of formal answers does not diminish the importance of the language itself. We can intuitively articulate this issue. The derivation and evaluation of authorization policies can be computable in polynomial time, and the computing of access decisions is conclusive: they are either authorized or denied.

6. CONCLUSION

In this paper, we have proposed a rule-based framework for role-based delegation including RDM2000 model and rule-based specification language. An important contribution of this work is that our delegation model supports regular role delegation in role hierarchy and multi-step delegation. A rule-based declarative language has been proposed to specify and enforce policies. There are many further issues that need to be explored. We plan to extend the RDM2000 model to include the role to role delegation and permission to role delegation. Also, the

language needs to be enhanced to include constraint representation and to study the convergence problem. As a part of the on-going research efforts, we are implementing the prototype of the proposed framework for law-enforcement agencies on a distributed environment.

7. REFERENCES

[1] Gail-Joon Ahn, Ravi Sandhu: The RSL99 Language for Role-Based Separation of Duty Constraints. ACM Workshop on Role-Based Access Control 1999: 43-54

[2] Martin Abadi, Michael Burrows, Butler Lampson and Gordon Plotkin. A calculus for Access Control in Distributed Systems. ACM Transaction on Programming Languages and Systems, Vol.15 No. 4, Sept 1993, pages 706-734

[3] Tuomas Aura. Distributed access-rights management with delegation certificates. Security Internet Programming. J. Vitec and C. Jensen(Eds.) pp.211-235 Springer: Berlin, 1999.

[4] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. The role of trust management in distributed system security. Security Internet Programming. J. Vitec and C. Jensen(Eds.) pp.185-210 Springer: Berlin, 1999.

[5] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. IEEE Symposium on Security and Privacy. Oakland, CA. May 1996.

[6] Venkata Bhamidipati and Ravi Sandhu. Push Architectures for USER ROLE Assignment. Proceedings of 23rd National Information Systems Security Conference, pages 89-100, Baltimore, Oct. 16-19, 2000

[7] Ezedin Barka and Ravi Sandhu. A Role-based Delegation Model and Some Extensions. Proceedings of 16th Annual Computer Security Application Conference, Sheraton New Orleans, Dec. 11-15, 2000

[8] Ezedin Barka and Ravi Sandhu. Framework for Role-Based Delegation Model. Proceedings of 23rd National Information Systems Security Conference, pages 101-114, Baltimore, Oct. 16-19, 2000

[9] David Ferriaolo, Janet Cugini, and Richard Kuhn. Role-based access control (RBAC): Features and Motivations. Proceeding s of 11th Annual Computer Security Application Conference, pages 241-248, New Orleans, LA, Dec 11-15 1995.

[10] Henry M. Glad. Access Control for Large Collections. ACM Transactions on Information Systems, Vol.15, No.2, April 1997, pages 154-194

[11] Morrie Gasser, Ellen McDermott. An Architecture for Practical Delegation a Distributed System. 1990 IEEE Computer Society Symposium on Research in Security and Privacy. Oakland, CA, May 7-9,1990

[12] Sushil Jajodia, Pierangela Samarati and V.S. Subrahmanian. A Logical Language for Expressing Authorizations. IEEE Symposium on Security and Privacy. May 1997.

[13] Ninghui Li, Joan Feigenbaum, and Benjamin N. Grosof. A logic-based knowledge representation for authorization with delegation (extended abstract). Proceeding 12th intl. IEEE Computer Security Foundations Workshop, 1999. (extended version is IBM Research Report RC 21492)

[14] Ninghui Li and Benjamin N. Grosof. A practically implementation and tractable delegation logic. IEEE Symposium on Security and Privacy. May 2000.

[15] J. Linn, M. Nyström. Attribute Certification: An Enabling Technology for Delegation and Role-Based Controls in Distributed Environments. ACM Workshop on RBAC 1999: 121-130

[16] Ravi Sandhu. Rational for the RBAC96 Family of Access Control Models. In Proceedings of 1st ACM Workshop on Role-based Access control, 1997

[17] Ravi Sandhu, Venkata Bhamidipati, and Qamar Munawer. The ARBAC97 Model for Role-based Administration of Roles. ACE Transactions on Information and System Security. Vol.2, No.1, Feb. 1999, pages 105-135

[18] Ravi Sandhu, E. Coyne, H. Feinstein and C. Youman. Role-based access control model. IEEE Computer, 29(2), Feb. 1996.

[19] Serge Abiteboul, Stéphane Grumbach. A Rule-Based Language with Functions and Sets. ACM Transactions on Database Systems, 16(1): 1-30 (1991)

[20] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, T. Ylonen. SPKI Certificate Theory, RFC2693, http://www.ietf.org/rfc/rfc2693.txt, 1999