A Rule-Based Framework for Role-Based Delegation and Revocation

LONGHUA ZHANG, GAIL-JOON AHN, and BEI-TSENG CHU
University of North Carolina at Charlotte

Delegation is the process whereby an active entity in a distributed environment authorizes another entity to access resources. In today's distributed systems, a user often needs to act on another user's behalf with some subset of his/her rights. Most systems have attempted to resolve such delegation requirements with ad-hoc mechanisms by compromising existing disorganized policies or simply attaching additional components to their applications. Still, there is a strong need in the large, distributed systems for a mechanism that provides effective privilege delegation and revocation management. This paper describes a rule-based framework for role-based delegation and revocation. The basic idea behind a role-based delegation is that users themselves may delegate role authorities to others to carry out some functions authorized to the former. We present a role-based delegation model called RDM2000 (role-based delegation model 2000) supporting hierarchical roles and multistep delegation. Different approaches for delegation and revocation are explored. A rule-based language for specifying and enforcing policies on RDM2000 is proposed. We describe a proof-of-concept prototype implementation of RDM2000 to demonstrate the feasibility of the proposed framework and provide secure protocols for managing delegations. The prototype is a web-based application for law enforcement agencies allowing reliable delegation and revocation. The future directions are also discussed.

Categories and Subject Descriptors: K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms: Management, Security

Additional Key Words and Phrases: Role, access control, delegation, revocation, rule-based

1. INTRODUCTION

The objective of delegation is to get the job done by sending it to someone else, for example, a subordinate. In McNamara [2002], effective delegation is defined as "the hallmark of good supervision." Effective delegation not only develops people who are ultimately more fulfilled and productive, but also frees the delegating users up to more important issues. In access control systems,

Portions of this paper appeared in preliminary form under the title "A Rule-Based Framework for Role-Based Delegation" in Proceedings of the 6th ACM Symposium on Access Control Models and Technologies (SACMAT), Chantilly, VA, May 3–4, 2001, pp. 153–162.

Authors' addresses: Longhua Zhang, Gail-Joon Ahn, and Bei-Tseng Chu, Laboratory of Information Integration, Security and Privacy (LIISP), Department of Software and Information Systems, College of Information Technology, University of North Carolina at Charlotte, Charlotte, NC 28223; email: {lozhang,gahn,billchu}@uncc.edu; URL: http://www.sis.uncc.edu/LIISP.

the delegation requirement arises when a user needs to act on another user's behalf for accessing resources. This might be for a limited time, for example, a vacation, sharing resources temporarily with others, and so on. Otherwise users may perceive security as a hindrance and bypass it. With delegation, the delegated user has the privileges to react to situations or access information without referring back to the delegating user.

To operate such a delegation system successfully, an access control system must be established to enable the flow of information for both parties to have full and rapid access to relevant information. 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. Also, it must be possible to revoke a delegated role. For example, Alice may want to revoke Bob from the delegated role later. A revocation mechanism must be provided and security policies must specify this action.

Delegation is an important factor for secure distributed computing environment. There are many definitions of delegation in the literature [Abadi et al. 1993; Barka and Sandhu 2000a; Gasser and McDermott 1990; Gladney 1997]. In general, it is referred to as one active entity in a system that delegates its authority to another entity to carry out some functions on behalf of the former. The most common delegation types include user-to-machine, user-to-user, and machine-to-machine delegation. Although the entities involved in these delegation types are different, they all have the same consequence—the propagation of access rights. Propagation of access rights in decentralized collaborative systems presents difficult problems for traditional access mechanisms, where authorization decisions are made based on the identity of the resource requester. Unfortunately, access control based on identity may be ineffective when the requester is unknown to the resource owner. Over the years, researchers have proposed a variety of distributed access control mechanisms. Delegation has been recognized as one of mechanisms to support access management in a distributed computing environment [Aura 1999]. Blaze et al. [1996, 1999] introduced trust management for decentralized authorization. Some trust management systems, such as KeyNote and SPKI/SDSI, use credentials to delegate permissions. Lampson et al. [1992] addressed how one principal can *delegate* some of its authority to another one. They showed a way to express delegation with access control calculus [Abadi et al. 1993] and also used timeouts to revoke delegations.

Role-based access control (RBAC) has been widely accepted as an alternative to traditional discretionary and mandatory access controls [Ferraiolo et al. 1995; Sandhu et al. 1996; Sandhu 1997]. RBAC is an enabling technology for managing and enforcing security in large-scale and enterprise-wide systems. Researchers and vendors have proposed many enhancements of RBAC models in the past decade. A general model, commonly referred as RBAC96 [Sandhu et al. 1996], has become widely accepted by the information security community. In RBAC96, the central notion is that permissions are associated with roles, users are assigned to appropriate roles, and users acquire permissions by being members of roles. Users can be easily reassigned from one role to another. Roles can be granted new permissions. Permissions can be easily revoked from roles as needed. Hence, RBAC provides a means for empowering individual users through role-based delegation in a fully distributed environment.

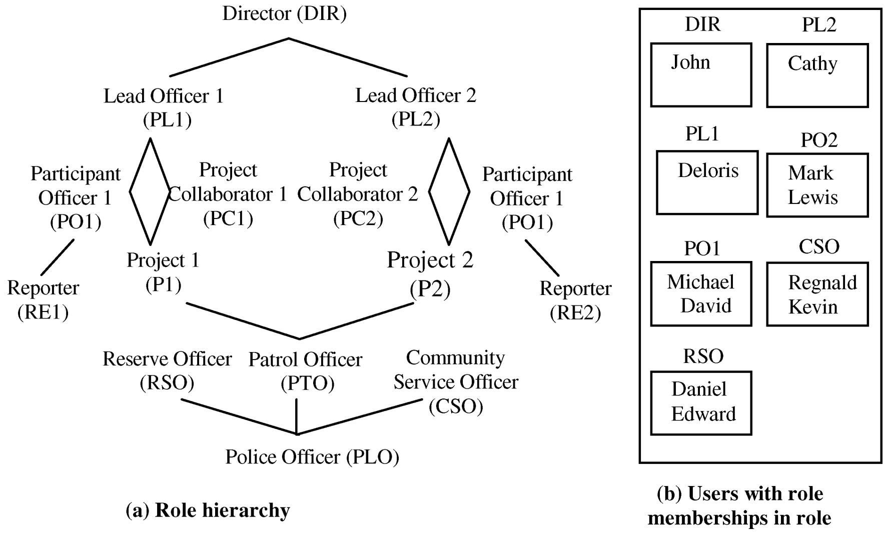
Although the importance of delegation has been recognized for a long time, the concept has not been supported in current role-based systems [Ferraiolo et al. 1995; Sandhu et al. 1996]. To delegate a role, the delegating user has to request a security officer to assign the role to the delegated user. Such a model would significantly increase the management efforts in a large-scale, highly decentralized environment because of the dynamic nature of delegations and the continuous involvement from security officers. In order to support effective delegation, management of user assignment could not realistically be centralized to a small group of security officers. The emerging technology of role-based delegation [Barka and Sandhu 2000a; Zhang et al. 2001] provides a means for decentralizing user assignment with empowerment of individual users. It enables decentralization of administrative tasks.

In this paper, we focus on user-to-user delegation, where a user delegates his/her role to another user. A preliminary version of our previous work is appeared in Zhang et al. [2001], which is further explored by Liebrand et al. [2002]. In our approach, we deal with administrative-directed delegation [Linn and Nyström 1999] including multistep delegation in role hierarchies. We propose a rule-based framework for role-based delegation. A rule-based system is a system where all behaviors are governed by a set of explicit rules. The framework includes a role-based delegation model called RDM2000 (role-based delegation model 2000) [Zhang et al. 2001] and a rule-based language for specifying policies on RDM2000. The enforcement of policies is also discussed. We present a proof-of-concept prototype implementation of RDM2000 to demonstrate the feasibility of the proposed delegation model and provide secure protocols for managing delegations.

The rest of this paper is organized as follows. Section 2 describes background and motivates our work. Section 3 defines components of RDM2000 including role-based delegation and revocation. In Section 4, we describe the semantics of rule-based specification language for delegation and revocation policies. A proof-of-concept implementation is presented in Sections 5 and 6. We compare our work with previous related work in Section 7. Section 8 concludes this paper and outlines future directions.

2. MOTIVATION

We give examples in a law-enforcement organization to clarify the problem. The police department's *community-oriented problem solving* is an effort to develop partnerships between law enforcement agencies and members of the community to address issues of concern. Working closely with the people in local areas, police officers are able to prevent crime from happening rather than dealing with it after its occurrence. The *community problem-oriented policing system* (*CPOPS*) is proposed to improve the service as a part of the police department's ongoing community policing efforts. In CPOPS, problem-oriented policing projects are developed to identify potential problems and resolve them



(a) Role hierarchy

(b) Users with role memberships in role

Fig. 1. An example of organizational role hierarchy and users.

before they become significant. By putting all related information together and allowing delegation, it enables officers to respond quickly to urgent calls and increases the proactive time.

In a simple problem-oriented policing project, police officers can be involved in many concurrent activities, for example, conducting initial investigations, investigating and analyzing crimes, preparing police reports, and assessing projects. The specific level of access and permissions a user can have will be determined by his/her responsibilities in the organization. In order to achieve this, users are identified to the system as having one or more roles in each project, such as lead officer, participant officer, policing reporter, and so on.

Figure 1 illustrates the organizational role hierarchy and users' role memberships in the organization. There is a junior-most role *PLO* to which all police officers in the organization belong. The senior most role director (*DIR*) assesses and coordinates projects. There are two policing projects in this organization; each project has a senior-most project lead role (*PL1*, *PL2*) and junior-most Project role (*P1, P2*). Between them are participant officer roles (*PO1*, *PO2*) and their junior project reporter roles (*RE1, RE2*), as well as project collaborator role (*PC1*, *PC2*). Project lead officers have major administrative duties for the project. Participant officers maintain twenty-four hours a day, seven days a week service for the project. Collaborators are people from other projects or external organizations that need access to project information. Reserve Officers (*RSO*) work on special assignment as needed. Community service officers (*CSO*) are nonsworn personnel who conduct a variety of nonhazardous law enforcement duties, for example, preparing policing report, performing traffic control, and so on. Figure 1(b) shows users and their role memberships after the user role assignment by security officers.

Delegation is an important feature in CPOPS. In this example, *John*, a director, needs to coordinate two problem-oriented policing projects under his supervision. As one of the projects, *Project 1*, confronts problems that are particularly perplexing, collaborations are necessary for information sharing with members from *Project 2*. Since *John* believes in delegating responsibility, he would like to delegate certain responsibilities to *Cathy* and her staff to collaborate closely. Of course, he wants all of this to happen securely and to monitor the progress of the delegation. This scenario requires role-based delegation in a variety of places: *John* needs to delegate his role to *Cathy* as well as the rights to further delegate the delegated role; *Cathy* needs to delegate the delegated role to her staff as necessary; further delegation is possible when the staff members are performing the collaboration tasks. If there is no delegation support, security officers have to be involved in every single collaborative activity. Such an approach makes real-time collaboration difficult if not impossible, given the large number of users that participate in these collaborations and the twenty-four hours a day, seven days a week working environment.

From this example, we identified the major requirements of role-based delegation as follows:

- *Support for multistep delegation*. This feature defines whether or not a delegation can be further delegated. Single-step delegation does not allow the delegated role to be further delegated. Sometimes, it would be desirable to further delegate the delegated role. Multistep delegation allows a delegated user to further delegate the delegated role.
- *Support for different revocation schemes*. Revocation is an important process that must accompany the delegation. It refers to the process to take away the delegated privileges, or the desire to go back to the state before privileges were delegated. There are different revoking schemes, among them are strong and weak revocations, cascading and noncascading revocations, as well as grant-dependent and grant-independent revocations.
- *Support for constraints*. Constraints can be imposed on other components of RBAC for laying out higher-level organizational policies. Delegation and revocation constraints specify restrictions on when the delegation or revocation performed is valid, or on when a cascaded delegation or revocation is valid. In this paper, we focus on those constraints applied to the users, roles, and their assignments, for example, separation of duty (SOD), maximum role cardinality, and so on. We require that constraints be enforced while carrying out delegations and revocations.
- *Support for partial delegation*. This feature is also referred to as totality of a delegation. It means how completely the permissions assigned to the delegating role can be delegated. Total delegation means all permissions are delegated; partial delegation means only subsets of the permissions are delegated. Both types of delegation are important since they allow users to selectively delegate permissions such that no more permission than necessary is delegated. This supports the well-known least privilege security principle.

In the subsequent sections, we formalize the role-based delegation model RDM2000. We then present an implementation of RDM2000 in a law-enforcement organization.

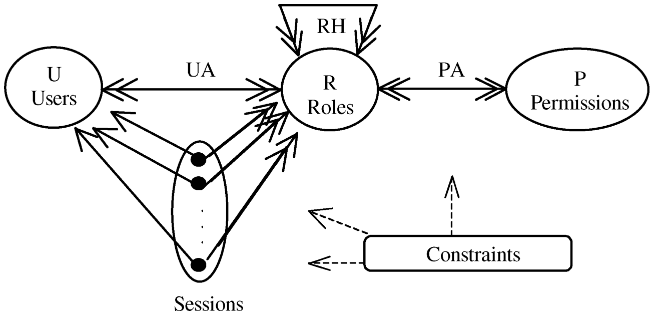


Fig. 2. RBAC96 model.

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 multistep delegation by introducing the delegation relation. Our work is built upon RBAC96 model [Sandhu et al. 1996] and RBDM0 model [Barka and Sandhu 2000a].

3.1 Basic Elements and System Functions—from RBAC96 and RBDM0

RBAC96 elements and relations are depicted in Figure 2. They include sets of five basic elements: users U, roles R, permissions P, sessions S, and constraints. The fundamental definition is that individual users are assigned to roles and permissions are assigned to roles. A role is a means for naming many-to-many relationships among individual users and permissions.

A user in this model is a human being, a role is 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). Although the concept of a user can be extended to include intelligent autonomous agents, machines, even networks, we limit a user to a human being in our model for simplicity.

RBAC96 has two relations: *user assignment (UA)* and *permission assignment (PA)* as illustrated in Figure 2. The user assignment is a many-to-many relation between users and roles. The permission assignment is a many-to-many relation between permissions and roles. Users are authorized to use the permissions of roles to which they are assigned. This arrangement provides great flexibility and granularity of assigning permissions to roles and users to roles. There are two sets of users associated with role *r*:

- *Original users* are those users who are assigned to the role *r*.
- *Delegated users* are those users who are delegated to the role *r*.

The same 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 *John* delegates his role *DIR* to *Deloris*, then *Deloris* is both an original user (explicitly) and a delegated user (implicitly) of role *PL1* because the role *DIR* is senior to the role *PL1*. The original user assignment (*UAO*) is a many-to-many user assignment relation between original users and roles. The delegated user assignment (*UAD*) is a many-to-many user assignment relation between delegated users and roles. The function *Users_O(r)* returns set of original users of the role r and *Users_D(r)* returns set of delegated users of the role r.

A session is a mapping between a user and possibly many roles. For example, a user may establish a session by activating some subset of assigned roles. A session is always associated with a single user and each user may establish zero or more sessions. The function *Sessions(u)* returns the sessions associated with a user, *Roles(s)* returns the roles activated in a single session, and *Permissions(s)* returns the permissions in each session.

Role hierarchies are a natural means for structuring roles to reflect an organization's lines of authority and responsibility, and are organized in partial order $\geq$, so that if $x \geq y$ then role x inherits the permissions of y. A member of x is also implicitly a member of y. in such case, x is said to be senior to y. A partial order is a reflexive, transitive, and antisymmetric relation. Role hierarchies provide a powerful and convenient means to enforce the principle of least privilege since only required permissions to perform a task are assigned to the role.

We summarize the above discussions as follows.

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

RBAC96 components:

- U, R, P, and S are sets of users, roles, permissions, 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 partially ordered role hierarchy.
- Sessions: U $\rightarrow 2^{\mathrm{S}}$ is a function that maps a user to a set of sessions.
- Roles: S $\rightarrow 2^{\mathrm{R}}$ is a function that maps a session s to a set of roles.
- Permissions: S $\rightarrow 2^{\mathrm{P}}$ is a function derived from PA mapping each session s to a set of permissions.

RBDM0 components:

- 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.
- Users: R $\rightarrow 2^{\mathrm{U}}$ is a function mapping each role to a set of users.
  $\text{Users}(r) = \{u \mid (u, r) \in \text{UA}\}$
- $\text{Users}(r) = \text{Users\_O}(r) \cup \text{Users\_D}(r)$
  $\text{Users\_O}(r) = \{u \mid (\exists r' \geq r)(u, r') \in \text{UAO}\}$
  $\text{Users\_D}(r) = \{u \mid (\exists r' \geq r)(u, r') \in \text{UAD}\}$

3.2 Role-Based Delegation

The scope of our model is to address user-to-user delegation supporting role hierarchies. To simplify the discussion of delegation, we assume a user cannot be delegated to a role if the user is already a member of that role. For example,

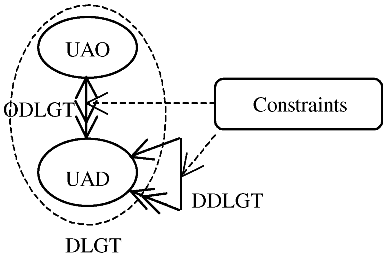


Fig. 3. RDM2000 model.

project leader *Deloris* with role *PL1* cannot be delegated the role *PO1* or *PC1* since she has already been an implicit member of these roles. Another assumption we made in RDM2000 is that we consider only the regular role delegation in this paper. Although it is possible and desirable to delegate an administrative role, it is difficult to control the regular role propagation in such a manner.

We first define a new relation in RDM2000 called *delegation relation* (*DLGT*). It includes sets of three elements: original user assignments *UAO*, delegated user assignment *UAD*, and constraints. 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 five components: a delegating user, a delegating role, a delegated user, a delegated role, and associated constraints. To simplify our discussion, we only include the first four components in the representation of a delegation relation in RDM2000. For example, ((*Deloris, PL1*), (*Cathy, PL1*)) means *Deloris* acting in role *PL1* delegates role *PL1* to *Cathy*. We assume each delegation is associated with zero or more constraints. A delegation relation is one-to-many relationship on user assignments. The delegation relation supports partial delegation in a role hierarchies: a user who is authorized to delegate a role r can also delegate a role r' that is junior to r. For example, ((*Deloris, PL1*), (*Lewis, PC1*)) means *Deloris* acting in role *PL1* delegates a junior role *PC1* to *Lewis*. A delegation relation is one-to-many relationship on user assignments. It consists of *original user delegation* (*ODLGT*) and *delegated user delegation* (*DDLGT*). We will revisit these concepts after we define delegation tree. Figure 3 illustrates components and their relations in RDM2000.

RDM2000 has the following components and theses components are formalized from the above discussions.

Definition 2. The following are additional components introduced in the RDM2000 model:

- DLGT $\subseteq$ UA × UA is one-to-many delegation relation. A delegation relation can be represented by $((u, r), (u', r')) \in DLGT$, which means the delegating user u with role r delegated role r' to user u'.
- ODLGT $\subseteq$ UAO × UAD is an original user delegation relation.
- DDLGT $\subseteq$ UAD × UAD is a delegated user delegation relation.
- DLGT = ODLGT $\cup$ DDLGT.

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 *multistep delegation*. Single-step delegation does not allow the delegated role to be further delegated; multistep delegation allows multiple delegations until it reaches the maximum delegation depth. The maximum delegation depth is a natural number defined to impose restriction on the delegation. Single-step delegation is a special case of multistep delegation with maximum delegation depth equal to one.

A *delegation path* (*DP*) is an ordered list of user assignment relations generated through multistep delegation. A delegation path always starts from an original user assignment. We use the following notation to represent a delegation path.

$$uao_0 \rightarrow uad_1 \rightarrow \cdots \rightarrow uad_i \rightarrow \cdots \rightarrow uad_n$$

Delegation paths starting with the same original user assignment can construct a delegation tree. A *delegation tree* (*DT*) expresses the delegation paths in a hierarchical structure. Each node in the tree refers to a user assignment and each edge to a delegation relation. The layer of a user assignment in the tree is referred to as the delegation depth. The function *Prior* maps one delegated user assignment to the delegating user assignment; function *Path* returns the path of a delegated user assignment; and function *Depth* returns the depth of the delegation path.

We give the following example to illustrate the concepts of delegation path and delegation tree. Suppose we have a set of delegation relations as follows:

D1: ((John, DIR), (Cathy, PL1)) $\in$ *DLGT*
D2: ((Cathy, PL1), (Mark, PC1)) $\in$ *DLGT*
D3: ((Cathy, PL1), (Lewis, PC1)) $\in$ *DLGT*
D4: ((John, DIR), (David, PC2)) $\in$ *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, as shown in Figure 4. The new components defined in RDM2000 are clearly illustrated in the delegation tree. For example, each parent–child relation in the tree is a DLGT; ODLGT is the first delegation in tree, DDLGT are subsequent delegations; function prior gives prior delegation node; delegation path is the path originated from root; and delegation depth is the depth of the node in tree.

We summarize the above discussions as follows.

Definition 3. Elements and functions in multistep delegation:

- N is a set of natural numbers.
- DP $\subseteq$ UA $\times$ UA is an ordered list of user assignments representing a delegation path.
- DT $\subseteq$ UA $\times$ UA is a user assignment hierarchy representing a delegation tree.
- Prior: UA $\rightarrow$ UA is a function that maps a user assignment to another subsequent user assignment that forms a delegation relation.

DLGT	Delegation Path
D1	P1: (John, DIR) → (Cathy, PL1)
D2	P2: (John, DIR) → (Cathy, PL1) → (Mark, PC1)
D3	P3: (John, DIR) → (Cathy, PL1) → (Lewis, PC1)
D4	P4: (John, DIR) → (David, PC2)

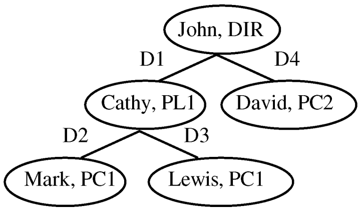


Fig. 4. An example for delegation paths and a delegation tree. *D1, D2, D3*, and *D4* stand for delegation relations; *P1, P2, P3*, and *P4* denote delegation paths.

$\text{Prior(ua)} = \{\text{ua}' \mid \text{ua} \in \text{UAD}, (\text{ua}', \text{ua}) \in \text{DLGT}\}$, where $\text{ua} = (u, r)$
$\text{Prior(ua)} = \{\emptyset \mid \text{ua} \in \text{UAO}\}$

- Path: UA → DP is a function that maps a UA to a delegation path.
  $\text{Path}(\text{ua}_0) = \{\text{ua}_n \rightarrow \cdots \rightarrow \text{ua}_i \rightarrow \text{ua}_{i-1} \cdots \rightarrow \text{ua}_0 \mid \text{ua}_i = \text{Prior}(\text{ua}_{i-1})\}$, where $\text{ua}_0 = (u, r)$.
- Depth: UA → N is a function that returns the delegation depth in the delegation path.

3.3 Delegation Authorization

In delegation authorization, our goal is to impose restrictions on *which role* can be delegated to *whom*. We partially adopt the notion of prerequisite condition from ARBAC97 [Sandhu et al. 1999] to introduce delegation authorization in RDM2000.

Definition 4. A *prerequisite condition CR* is a Boolean expression using the usual "&" (and) and "|" (or) operators on terms of form x and $\bar{x}$ where x is a regular role, for example, $CR = r_1 \ \& \ r_2 \mid \bar{r}_3$.

Definition 5. 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 $(r, cr, n) \in can\_delegate$ 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 r) to any user whose current entitlements in roles satisfy the prerequisite condition cr without exceeding the maximum delegation depth n. For example, (*PL1, PO2, 1*) ∈ *can_delegate*, then *John* can delegate role *PC1* to *Mark who is a member of PO2 role*, so that (*John*, *PL1, Mark, PC1*) ∈ *DLGT*. The meaning of

Table I. Example of $can\_delegate(r, cr, n)$ with Prerequisite Roles

Delegating Role (r)	Prerequisite Condition (cr)	Max. Depth (n)	Candidate Delegated Role Set
DIR	*PTO*	2	$\{RE1, P1, PO1, PC1, PL1\} \cup$ $\{RE2, P2, PO2, PC2, PL2\}$
PL1	$PLO\&\overline{PO2}$	2	$\{PTO, RE1, P1, PO1, PC1, PL1\}$
RE1	*CSO*	1	$\{RE1\}$

$(r, \emptyset, n) \in can\_delegate$ 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 r) to any other user. Table I shows examples of *can_delegate* relations.

3.4 Role-Based Revocation

Revocation is an important process that must accompany the delegation. For example, *Cathy* delegated role *PC1* to *Mark*; however, if *Mark* is transferred to another division of the organization, he should be revoked from the delegated role *PC1* immediately. Several different semantics are possible for user revocation. Hagstrom et al. [2001] categorized revocations into three dimensions in the context of owner-based approach: global and local (*propagation*), strong and weak (*dominance*), and deletion or negative (*resilience*). Barka and Sandhu [200b] identified user grant-dependent and grant-independent revocation (*grant-dependency*). Since negative authorization is not considered in RDM2000, we articulate user revocation in the following dimensions: *grant-dependency*, *propagation*, and *dominance*.

Grant-dependency refers to the legitimacy of a user who can revoke a delegated role. *Grant-dependent* (*GD*) revocation means only the delegating user can revoke the delegated user from the delegated role. *Grant-independent* (*GI*) revocation means any original user of the delegating role can revoke the user from the delegated role.

Dominance refers to the effect of a revocation on implicit/explicit role memberships of a user. A *strong* revocation of a user from a role requires that the user be removed not only from the explicit membership but also from the implicit memberships of the delegated role. A *weak* revocation only removes the user from the delegated role (explicit membership) and leaves other roles intact. Strong revocation is theoretically equivalent to a series of weak revocations. To perform strong revocation, the implied weak revocations are authorized based on revocation policies. However, a strong revocation may have no effect if any upward weak revocation in the role hierarchy fails [Sandhu et al. 1999].

Propagation refers to the extent of the revocation to other delegated users. A *cascading* revocation directly revokes a delegated user assignment in a delegation relation and also indirectly revokes a set of subsequent propagated user assignments. A *noncascading* revocation only revokes a delegated user assignment.

Our framework supports all eight types of user revocation categorized by the above dimensions. The semantics of different revocation types are addressed as follows. Our examples mainly show grant-dependent revocation for brevity. Among different types of revocations, weak noncascading grant-dependent

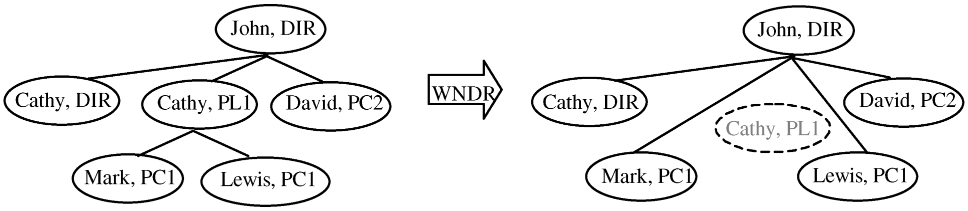


Fig. 5. An example for weak noncascading revocation.

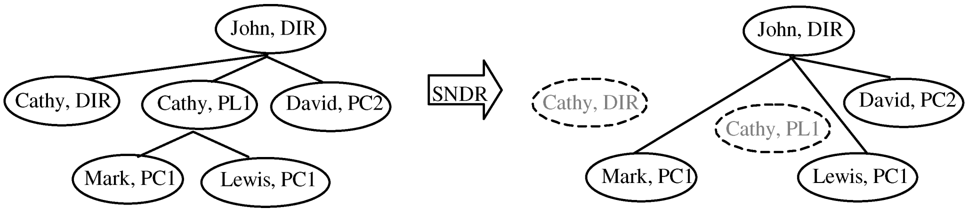


Fig. 6. An example for strong noncascading revocation.

(*WNDR*) and weak noncascading grant-independent (*WNIR*) revocations are the simplest yet the most basic revocation schemes. They can be used as the basic building blocks for other types of revocation schemes in our framework. Suppose the revocation in Figure 5 is weak noncascading, for *John* to revoke *Cathy* from role *PL1*, it is important to note that only *Cathy*'s membership of role *PL1* is changed; other role memberships of *Cathy* and all the delegated user assignments propagated by *Cathy* are still valid. If the revoked node is not a leaf node, noncascading revocation may leave a "*hole*" in the delegation tree. A solution might be the revoking user takes over the delegating user's responsibility. In this example, *John* takes over the delegating user's responsibility from *Cathy*, and changes all delegation relations: $((Cathy, PL1), (u, r)) \in$ DLGT to $((John, DIR), (u, r)) \in DLGT$. In this case, *John* takes over *Cathy*'s delegating responsibility for *Mark* and *Lewis*.

A strong noncascading GD/GI revocation of $(u, r) \in UAD$ can be seen as an explicit weak noncascading GD/GI revocation of (u, r) combined with a set of implicit weak noncascading GI revocations of $(u, r') \in UAD$ where r' is a role senior to r. Suppose we have a delegation tree as shown in Figure 6. For *John* to strongly, noncascadingly revoke *Cathy* from $PL1$, *Cathy* is removed not only from membership of $PL1$, but also from roles that are senior to $PL1$, in this example *DIR*. We use *SNDR* and *SNIR* to indicate strong noncascading GD revocation and strong noncascading GD revocation, respectively.

A weak cascading GD/GI revocation of $(u, r) \in UAD$ can be seen as an explicit weak noncascading GD/GI revocation of (u, r) combined with a set of implicit weak noncascading GI revocations of $(u', r') \in UAD$ where (u', r') is further propagated by (u, r). Suppose *John* is attempting to revoke *Cathy* from role *PL1*. Cascading revocation implies that when *Cathy* is revoked from role *PL1*, *Mark* and *Lewis* are revoked from *PC1* subsequently as shown in Figure 7. We use *WCDR* and *WCIR* to indicate weak cascading GD revocation and weak cascading GD revocation, respectively.

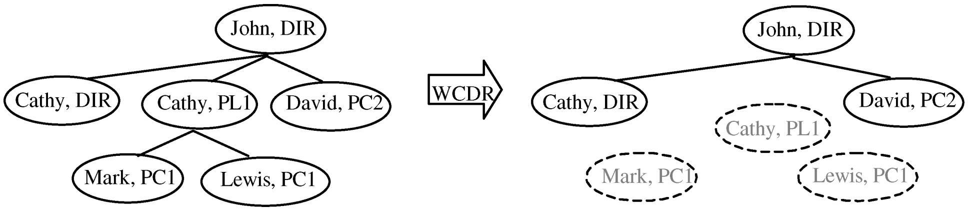


Fig. 7. An example for weak cascading revocation.

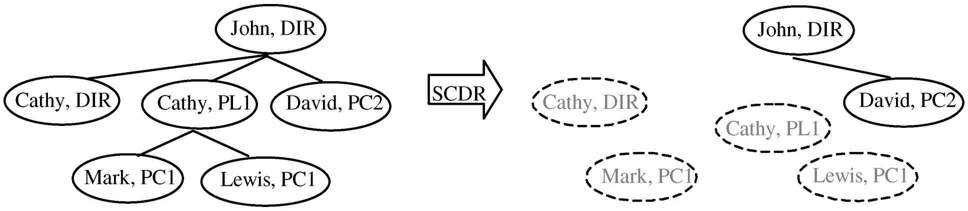


Fig. 8. An example for strong cascading revocation.

A strong cascading GD/GI revocation of (u, r) can be seen as a combination of a strong noncascading GD/GI revocation of (u, r) and a weak cascading GD/GI revocation of (u, r). In Figure 8, for *John* to strongly, cascadingly revoke *Cathy* from *PL1*, not only *Cathy* is strongly revoked from membership of *PL1*, but also *Mark* and *Lewis* are revoked from role *PC1*. We use *SCDR* and *SCIR* to indicate weak cascading GD revocation and weak cascading GD revocation, respectively.

It is equally important that users understand the semantics of above revocation schemes, since users themselves determine the scheme that a revocation actually follows.

3.5 Revocation Authorization

Definition 6. The following relations authorize delegation revocation:

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

The meaning of $(b) \in$ *can_revokeGD* is that only the delegating user who has current membership in b can revoke a delegated user from the delegated role that is junior to b. The meaning of $(b) \in$ *can_revokeGI* is that any user whose current membership includes a delegated role$b$ in the delegation path that is prior to a delegated user whose current membership includes a delegated role junior or equal to$b$, can revoke the delegated user from role b. Tables II and III show examples of these relations for the delegation tree in Figure 4. (*DIR*) $\in$ *can_revokeGI* means that a user (*John*) with role *DIR* prior to a delegated user (*Cathy, Mark, Lewis, David*) in a delegation path is authorized to revoke the delegated user from the delegated role (*PL1*, *PC1*, *PC2*). *can_revokeGD* is the default setting for a delegating role. If a user can delegate a role, the same user can also revoke the delegated role.

Table II. Example of *can_revokeGI*

Revoking Role	Candidate Revoked Role Set	Revoking User Set	Candidate Revoked User Set
DIR	{DIR, PL1, PC1, PC2}	{John}	{Cathy, Mark, Lewis, David}
PL1	{PL1, PC1}	{John, Cathy}	{Mark, Lewis}

Table III. Example of *can_revokeGD*

Revoking Role	Candidate Revoked Role Set	Revoking User Set	Candidate Revoked User Set
DIR	{DIR, PL1, PC2}	{John}	{Cathy, David}
PL1	{PC1}	{Cathy}	{Mark, Lewis}

Certain types of revocation requests may result in a set of related user role revocations. Any revocation scheme can be constructed through basic revocation schemes. Thus, the implementation of a revocation actually supports two types of enforcement: an *explicitly enforced revocation* (simply called an *explicit revocation*) and an *implicitly enforced revocation* (simply called an *implicit revocation*). An explicit revocation is directly from the user revocation request. An implicit revocation is a revocation resulting from a strong revocation or a cascading revocation.

We introduce two approaches to implement an implicit revocation: the *eager* approach and the *lazy* approach. The eager implementation revokes all explicit and implicit revocations immediately after the authorization of a revocation request. For example, in a weak cascading revocation, if *Cathy* is revoked from *PL1*, *Mark* and *Lewis* will be revoked from *PC1* immediately. This can be achieved by browsing over all the delegation paths and revoking users from delegated roles. The eager approach is difficult to implement in a distributed environment because of the computational complexity. The lazy implementation adopts a run-time revocation. Only *Cathy* will be revoked from role *PL1* with the authorization of a weak cascading revocation. When *Mark* activates the delegated role, system will check the status of each element in the delegation path: since (*Cathy*, *PL1*) is no longer valid, the delegation path *P2:* {(*Mark*, *PC1*), (*Cathy*, *PL1*), (*John*, *DIR*)} is not allowed. *Mark* will be finally removed from *PC1*. This lazy approach will not lead to timing attacks caused by the delay since the status of a user assignment is validated when a user activates the delegated role.

4. THE RULE-BASED POLICY SPECIFICATION LANGUAGE

RDM2000 defines policies that allow regular users to delegate their roles. It also specifies the policies regarding how delegated roles can be revoked. In this section we describe a rule-based language to enforce these policies. There are two reasons to 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 local policies to further control delegation and revocation. A declarative rule-based system allows individual organizations to easily incorporate such local policies. We emphasize the use of functions. We show how our construction 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 specify and enforce authorizations of delegation and revocation based on the RDM2000 model. A rule-based language is a declarative language, which binds logic with rules [Abiteboul and Grumbach 1991]. 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 7. A *clause*, also known as a *rule*, takes the form:

$$H \leftarrow B.$$

where H stands for rule head and B stands for rule body.

A successful inference of B will trigger H to be true. This provides exactly the mechanism for authorization specification and enforcement. An authorization is similar to an assertion. If the condition defined in the rule body is true, then it will trigger some actions (e.g., authorizations). Thus, the condition of an authorization policy can be encoded in a rule body; and the authorization can be encoded in the rule head.

4.2 Functions

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: specification functions, utility functions, and authorization functions, as shown in Tables IV–VI, respectively. Specification functions express information of the RBAC and RDM2000 components. We have a set of system functions defined in RBAC96 and RDM2000 models. We map these system functions to specification functions. Utility functions are general-purpose Boolean functions providing supportive functionalities, for example, comparison, aggregation. Authorization functions define authorization policies and enforcement of these policies. They further divide into basic authorization functions and derived authorization functions.

4.3 Basic Authorization Rules

Basic authorization rules take form $H \leftarrow$. Bodies of basic authorization rules are empty, which means they are always true. Basic authorization rules are predefined security policies and facts specified within RBAC and RDM2000 components.

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

$$can\_delegate(r, cr, n) \leftarrow .$$

Table IV. Specification Functions

Mapping Functions	RDM2000 System Functions	Semantics
active(u, r, s)		Return true if the user u has role r activated in a session s.
conflicting(x, y)		Return true if x conflicts with y, where x and y can be users or roles.
delegatable(u, r)		Return true if a user u has the authority to further delegate a role r. This function always returns true if (u, r) is an original user assignment.
depth(u, r)	Depth: $\mathrm{U} \cup \mathrm{R} \rightarrow \mathrm{N}$	Return the delegation depth of a (delegated) user assignment.
duration(u, r)	Duration: $\mathrm{UA} \rightarrow \mathrm{T}$	duration(u, r) returns the assigned duration-restriction constraint of the delegated user assignment (u, r).
expires(t)		expires(t) returns true if the duration t expires.
junior(r, r')	$\leq$	Role r is junior to role r'.
path(u, r)	Path: $\mathrm{U} \cup \mathrm{R} \rightarrow \mathrm{DP}$	Return the delegation path of a (delegated) user assignment (u, r).
prior(u, r)	Prior: $\mathrm{U} \cup \mathrm{R} \rightarrow \mathrm{U} \times \mathrm{R}$	Return the user assignment previous to (u, r) in the delegation path.
permissions(s)	Permissions: $\mathrm{S} \rightarrow 2^{\mathrm{P}}$	Return all activated permissions in a session.
revoked_cascade (u, r)		Return true if any one of the user assignment in the delegation path of (u, r) was revoked.
revoke_stronge (u, r, u', r')		Return true if (u', r') was strongly revoked by (u, r).
roles(s)	Roles: $\mathrm{S} \rightarrow 2^{\mathrm{R}}$	Return all activated roles in a session.
senior(r, r')	$\geq$	Role r is senior to role r'.
sessions(u)	Sessions: $\mathrm{U} \rightarrow \mathrm{S}$	Map a user to a set of sessions.
users(r)	Users: $\mathrm{R} \rightarrow 2^{\mathrm{U}}$	Return all users who are members of role r.
users_o(r)	Users_O: $\mathrm{R} \rightarrow 2^{\mathrm{U}}$	Return all original users who are members of role r.
users_$d(r)$	Users_D: $\mathrm{R} \rightarrow 2^{\mathrm{U}}$	Return all delegated users who are members of role r.

where r, 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 RDM2000. It means that a member of the role r (or a member of any role that is senior to r) can assign a user whose current membership satisfies prerequisite condition cr to role r (or a role that is junior to r) without exceeding the maximum delegation depth n.

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

$$can\_revokeGD(r) \leftarrow .$$

where r is element of roles.

Table V. Utility Functions

RSPL Functions	Return Value	Semantics
in(x, y)	Truth Value	Describe the membership between a and b: that is, x is a member of y.
equals(x, y)	Truth Value	Return true if $x = y$.
lt(x, y)	Truth Value	Return true if $x < y$.
not(x)	Truth Value	not(x) = !x, where x is a Boolean term.

Table VI. Authorization Functions

Basic Authorization Functions	Derived Authorization Functions	Semantics
	allow(u, r, p, s)	Refer to rule 4
	der_can_revoke_auto_cascade(u, r)	Refer to rule 8
	der_can_revoke_auto_strong(u, r)	Refer to rule 9
	der_can_revoke_auto_expire(u, r, rvk_opt)	Refer to rule 10
can_delegate(r, cr, n)	der_can_delegate(u, r, u', r', dlg_opt)	Refer to rules 1 and 5
can_revokeGD(r)	der_can_revokeGD(u, r, u', r', rvk_opt)	Refer to rules 2 and 6
can_revokeGI(r)	der_can_revokeGI(u, r, u', r', rvk_opt)	Refer to rules 3 and 7
	error(u, r, u', r')	Refer to rule 11

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 a role r only by the delegating user.

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

$$can\_revokeGI(r) \leftarrow .$$

where r is element of roles.

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 a role r by any user who is prior to him in the delegation path.

4.4 Authorization Derivation Rules for Enforcing Policies

The basic authorization specifies the policies and facts defined in RDM2000. Further derivations are needed for authorization and their enforcement. An authorization derivation rule expresses authorization on an individual user. The rule body describes an inference logic that consists of basic authorization, specification and utility functions. The result can be either true (authorized) or false (denied).

4.4.1 *Enforcement of Access Control Policies.*

Rule 4. An *access control rule* is a rule of the form:

$$allow(u, r, p, s) \leftarrow active(u, r, s) \,\&\, in(p, permissions(s))$$

where u, r, p, and s are elements of users, roles, permissions, and sessions, respectively.

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

Access control rule says that a user with a role r activated in a session s will be granted a permission p that is assigned to r, whether r is assigned by security officer or delegated by another user.

4.4.2 *Enforcement of Delegation Policies.*

Rule 5. *A user-user delegation authorization derivation rule* is a rule of the form:

$$\begin{aligned} &\textit{der\_can\_delegate}(u, r, u', r', \textit{dlg\_opt}) \leftarrow \\ &\quad \textit{active}(u, r, s)\& \\ &\quad \textit{delegatable}(u, r)\& \\ &\quad \textit{can\_delegate}(r'', cr, n)\& \\ &\quad \textit{senior}(r, r'')\& \\ &\quad \textit{in}(u', cr)\& \\ &\quad \textit{junior}(r', r'')\& \\ &\quad \textit{lt}(\textit{depth}(u, r), n). \end{aligned}$$

where u and u' are elements of users; r, r', 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 (u', r') is true. This argument is used as Boolean control of delegation propagation.

This rule means that a user u with a membership of a role r senior to r''activated in session s can delegate a user u' whose current role membership satisfies prerequisite condition cr to role r' (r' is junior to role r'') without exceeding the maximum delegation depth n.

For example, the security officer specifies the following delegation policies addressed in Table I:

$$\begin{aligned} &\textit{Policy 1: can\_delegate}(DIR, PTO, 2) \leftarrow . \\ &\textit{Policy 2: can\_delegate}(PL1, PLO\&\overline{PO2}, 2) \leftarrow . \\ &\textit{Policy 3: can\_delegate}(RE1, CSO, 1) \leftarrow . \end{aligned}$$

John needs to delegate a role *PL1* to *Cathy* with further delegation option (we assume that a policy engine optimizes the selection of possible policies for a derivation rule based on the delegation request, in this case, *policy 1*).

$$\begin{aligned} &\textit{To deduce der\_can\_delegate}(John, DIR, Cathy, PL1, true), \\ &\quad \textit{deduce active}(John, DIR, s) = \textit{true and} \\ &\quad \textit{deduce can\_delegate}(DIR, PTO, 2) = \textit{true and} \\ &\quad \textit{deduce delegatable}(John, DIR) = \textit{true and} \\ &\quad \textit{deduce senior}(DIR, DIR) = \textit{true and} \\ &\quad \ldots \end{aligned}$$

$$\begin{aligned}
&\textit{to deduce in(depth(John, DIR), 2),}\\
&\quad \textit{deduce depth(John, DIR)} = 0\\
&\quad \textit{deduce in(0, 2)} = \textit{true}.\\
&\textit{true}.
\end{aligned}$$

The delegation is authorized. If the deduction results for all available policies are false, then this delegation request is denied. For example, delegation from *Gail* to *Cathy* with role *PL2* is denied since there is not a policy for this case. Note that there may be more than one basic authorization rules that can be applied to infer the derivation rule. The policy engine only needs one of them to authorize the delegation request and the choice depends on the rule-optimizing algorithm. The enforcement of a delegation is straightforward. After authorization, the delegated user is assigned to the delegated role.

4.4.3 *Enforcement of Revocation Policies.*

Rule 6. *A grant-dependent revocation authorization derivation rule* is a rule of the form:

$$\begin{aligned}
&\textit{der\_can\_revokeGD}(u, r, u', r', \textit{rvk\_opt}) \leftarrow\\
&\qquad \textit{active}(u, r, s)\&\\
&\qquad \textit{can\_revokeGD}(r')\&\\
&\qquad \textit{equals}((u, r), \textit{prior}(u', r')).
\end{aligned}$$

where u and u are elements of users, $u \neq u'$; r and r' are elements of roles. The *rvk_opt* indicates the revocation scheme, which can be one of strong cascading grant-dependent (SCDR), weak cascading grant-dependent (WCDR), strong noncascading grant-dependent (SNDR), and weak noncascading grant-dependent (WNDR).

This rule means that a user u' can be revoked from a role r' by the delegating user u with a role r activated where $(u, r) = \textit{prior}(u', r')$.

Rule 7. *A grant-independent revocation authorization derivation rule* is a rule of the form:

$$\begin{aligned}
&\textit{der\_can\_revokeGI}(u, r, u', r', \textit{rvk\_opt}) \leftarrow\\
&\qquad \textit{active}(u, r, s)\&\\
&\qquad \textit{can\_revokeGI}(r')\&\\
&\qquad \textit{in}((u'', r''), \textit{path}(u', r')).
\end{aligned}$$

where u and u' are elements of users, $u \neq \textit{du}; r$ and $\textit{dr}$ are elements of role respectively. The *rvk_opt* indicates the revocation scheme which can be any one of strong cascading (*SCIR*), weak cascading (*WCIR*), strong noncascading (*SNIR*), and weak noncascading (*WNIR*).

This rule means that a user u' can be revoked from a role r' by any original user u.

Rule 8. *An automatic cascading revocation authorization rule* is rule of the form:

$$\begin{aligned} & \mathit{der\_can\_revoke\_auto\_cascade}(u, r) \leftarrow \\ & \qquad in((u', r'), path(u, r))\& \\ & \qquad \mathit{revoked\_cascade}(u'', r'', u', r'). \end{aligned}$$

where u, u' are elements of users; r, r' are elements of roles.

This rule means if any of user role assignment (u', r') in the delegation path $path(u, r)$ is revoked cascadingly, a user u is revoked from a role r.

Rule 9. *An automatic strong revocation authorization rule* is rule of the form:

$$\begin{aligned} & \mathit{der\_can\_revoke\_auto\_strong}(u, r) \leftarrow \\ & \qquad \mathit{revoked\_strong}(u'', r'', u, r')\& \\ & \qquad senior(r, r')\& \\ & \qquad \mathit{der\_can\_revokeGI}(u'', r'', u, r', WNIR). \end{aligned}$$

where u, u'' are elements of users, r, r', and r'' are elements of roles.

This rule means if user role assignment (u, r') is strongly revoked by a user u'' with a role r'', then u is weakly revoked from any role r' which is senior to r by the same user u''.

In RDM2000, we assume each delegation relation may have a duration constraint associated with it. Once the assigned duration expires, the delegation is automatically revoked. If there is no duration specified for a delegation, the delegation is permanent unless another user revokes it. When a user assigns duration to a delegation, the user needs to explicitly specify the scheme that the duration–restriction revocation will follow as well. So actually duration–restriction revocation is a scheduled user revocation. Revocation using duration constraint was proposed by Barka and Sandhu [2000a, 2000b]. 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 or predetermined. 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 it might be overset or underset.

Rule 10. *An automatic duration–restriction triggered revocation authorization rule* is rule of the form:

$$\mathit{der\_can\_revoke\_auto\_expire}(u, r, \mathit{rvk\_opt}) \leftarrow expires(duration(u, r)).$$

where u is element of users, r is element of roles.

This rule means if the duration assignment to user assignment (u, r) is expired, *(u, r) will be revoked*.

Derivation rules 6 and 7 authorize user revocation requests. Derivation rules 8 and 9 authorize revocations resulting from cascading or strong revocations. Derivation rule 10 authorizes duration–restriction triggered revocation.

The enforcement of an authorized revocation could be quite complicated in our framework.

In a revocation enforcement process, the authorization of a cascading revocation subsequently authorizes a set of automatic weak noncascading revocations by applying *rule* 8; the authorization of a strong revocation subsequently authorizes a set of automatic weak noncascading revocations by applying *rule* 9; and if necessary the enforcement of a weak noncascading revocation will coalesce a set of delegation path.

4.4.4 *Enforcement of Role-Based Constraints.* Constraints are an important component of RBAC since it can be used for laying out higher-level organizational policies in role-based systems [Ahn and Sandhu 2000; Sandhu et al. 1996]. Major examples include incompatible role assignment, separation of duties (SOD), and Chinese wall policy. We mainly focus on those constraints that impose restrictions on delegations.

Representation of constraints is critical. In a centralized role-based system, the effect of constraints can be achieved by judicious care on the part of the security officer(s). However, in a distributed system, there must be a way to represent these constraints consistently. Rules are extremely suited for constraints specification and enforcement. We introduce *an integrity rule* to represent constraints.

Rule 11. *An integrity rule* is a rule of the form:

$$error(u, r, u', r') \leftarrow B.$$

An integrity rule takes a delegation relation as its arguments, and checks if the proposed delegation violates security policies. For example, a *static separation of duty (SSOD)/incompatible roles assignment constraint* states that no user can be assigned to two conflicting roles (r_1, r_2). This constraint can be represented as

$$error(u, r, u', r') \leftarrow conflicting(r', r'')\,\&\,in(u', r'').$$

This rule says if user u is already a member of role r'', then u cannot be delegated the conflicting role r'. Suppose *PO1 and CSO* are incompatible roles. The delegation of role *PC1* from *Deloris* to *Kevin* will be denied even if the delegation authorization derivation rule is true, since the integrity rule indicates that this delegation violates the SSOD constraint.

$$error(Deloris, PL1, Kevin, PO1) \leftarrow conflicting(PO1, CSO)\,\&\,in(Kevin, CSO).$$

Similarly, we can define and enforce other role-based constraints.

An *incompatible users constraint* states that two conflicting users (u_1, u_2) cannot be assigned to the same role. This constraint can be represented as

$$error(u, r, u', r') \leftarrow conflicting(u', u'')\,\&\,in(u'', r').$$

Integrity rules are extremely suited for constraint specification and enforcement. A constraint is similar to an assertion. If the condition defined in the constraint is true, then it will trigger some actions (restrictions). However, to fully address the constraint issue is beyond the scope of this paper. It is our future work to explore the enforcement of role-based constraints through different rules.

5. SYSTEM DESIGN

As part of our on-going research efforts, we have implemented a prototype of the proposed delegation framework for law-enforcement agencies to demonstrate the feasibility of the proposed delegation model. Our prototype is a web-based application supporting secure role-based delegation and revocation. In this section, we describe the design issues that we consider and introduce the system architecture in general. We also demonstrate the delegation and revocation procedures in cpops.

5.1 Design Issues

The working environments for law-enforcement agencies are highly distributed; they can be at any place where a problem occurs. So are the systems that process the crime-related information. Delegation is one of the most important security features in such systems. It enables officers to share authority to other individuals for collaboration purposes. This eliminates the participation of security administrators and empowers the user population. RDM2000 model provides us with several alternatives to implement delegation and revocation in such a critical environment. We first address the design issues.

5.1.1 *Resource Definition.* As with all software development, good design and engineering practices are important for information and system security. This point is particularly true for security-critical software such as CPOPS. Rather than thinking of delegation and revocation as an add-on feature to CPOPS, it was designed into the system from its early stages of requirements gathering through development.

CPOPS is a secure web-based application that enables police officers to access problem-oriented policing projects' information at real time. We selected *CodeFusion* (CF) 4.0 [Forta 1998, 2001] as the development platform since it provides a full range of database interaction functions to create dynamic, data-driven web pages. In CPOPS, objects of access control in the system are mainly views of project tables; operations are access methods to these views, for example, select, insert, update, and delete. Permissions represent access methods to one or more views, for example, a link that can be viewed by users in certain roles, a code segment that displays records of a specific table, a report that members of some role can update while others can only read, and so on. One design decision we made for securing CPOPS is the definition of the protected resources. Instead of laying protection directly on views of CPOPS tables, we abstract a protected resource as any block of code that needs to be secure. Most of these code blocks actually define access methods to views of tables. Thus, permission for a role to access tables is simplified as the privilege to execute the code blocks. There are several reasons we made this abstraction. First, the protected resources are defined at a higher degree of logical abstraction than physical data and views, so they are easily understood and efficiently managed by developers. Second, since the code blocks can be predefined reusable components, it can hide the implementation details of permissions from application developers. And lastly, one of the most powerful features of the CF security framework is the capability to secure individual sections of code at runtime [Forta 1998]. This feature allows developers to control access to resources on a user-by-user basis. We extend this feature to support role-based access control. This feature ensures that only users with certain roles are allowed to request delegation and revocation.

5.1.2 *Delegation Administration* . Two views of delegation administration were summarized in Linn and Nyström [1999]: *administrative-directed delegation* and *user-directed delegation*. In administrative-directed delegation, an administrative infrastructure outside the direct control of a user mediates delegation, for example, a delegation agent must mediate all delegations. In user-directed delegation, any user may mediate delegation to resources under the user's control. In both situations it is necessary to enforce predefined delegation policies to prevent privilege abuses by individual users.

Although RDM2000 can be implemented as either user-directed or administrative-directed delegation, our implementation takes the latter approach. That is, an administrative service outside the direct control of a user mediates delegation and revocation. Users initiate delegation and revocation by sending requests to the administrative infrastructure, then the administrative service component processes the inquiries. The administrative-directed delegation has several advantages. One advantage is that it is easy to specify and modify delegation and revocation policies. Since the mediation of delegation and revocation is centralized, the specification and modification of policies need to be done only once. Another advantage is that it is easy to keep the records of all delegations and revocations in the role-based system for auditing. However, the cost of handling delegation requests for such a centralized server may be rather expensive. Another concern is how to handle the emergency if the server is not available since CPOPS requires providing service continuously. To address these problems, we may need additional server(s) to increase the throughput and availability. In this paper, we choose the administratively directed approach for brevity.

Another design decision for administration is how to impose restriction on multistep delegation. As we mentioned before, delegation paths that start from the same original user assignment will form a user assignment hierarchy called delegation tree. To impose restrictions on such a hierarchy, decisions must be made to limit the depth as well as the width of the delegation tree. There are three solutions to control depth of a delegation: no control, Boolean control, and integer control [Ellison et al. 1999]. 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. However, 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

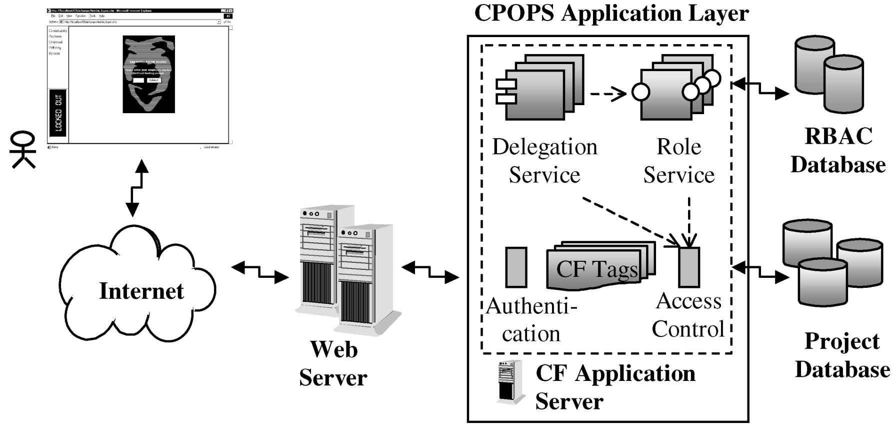


Fig. 9. Overview of system architecture.

drawback is that it has no control on the width of the delegation tree, so it is not a tight control on role proliferation. We choose integer control at a high level (delegation policy) to restrict the maximum depth, as well as Boolean control at a low level (each delegation) to restrict the width of a delegation.

5.2 System Architecture for Delegation and Revocation in CPOPS

In this section, we describe the system architecture in general to demonstrate the feasibility of the proposed delegation model and provide secure protocols for managing delegation and revocation.

RDM2000 extends existing RBAC models. It is necessary to implement RBAC components first. The mechanism of applying RBAC on the web has been discussed in several papers [Bhamidipati and Sandhu 2000; Ferraiolo et al. 1999; Linn and Nyström 1999]. Our prototype is implemented on Windows NT with Internet information server (IIS) and CF application server. It is important to note that the web server and the application server have the responsibility of authenticating user identification information and providing confidential data transmission through secure socket layer (SSL) connection.

Since delegation and revocation services are only part of a security infrastructure, we choose a modular approach to our architecture that allows the delegation and revocation service to work with current and future authentication and access control services. The modularity enables future enhancements of our approach. An overview of the architecture is shown in Figure 9. It consists of a front-end web server, a business application layer providing a number of services which include delegation/revocation service, role service, and so on, and back-end databases for the problem-oriented policing projects as well as RBAC and RDM2000 components to be managed. We briefly describe these components as follows:

Web server is the application entry point. It has the responsibility of providing confidential data transmission through secure socket layer (SSL) connection. *CF application server* implements the business application logic through CF tags that are component-based software for accessing databases and providing custom-developed business logic. We implemented three services in the application server: authentication and access control service, role service and delegation/revocation service.

Role service is a façade for other services (e.g., delegation/revocation service, authentication and access control service) to interact with the RBAC database. It provides methods to create, retrieve, and update database elements, for example, user credentials, role memberships, associated permissions, delegation relations, and so on. These elements are created and maintained using a set of graphical administration tools. These tools can also be used to maintain the integrity of database elements by checking and enforcing integrity rules. In this paper, we provide administration tools for managing RDM2000 elements, for example, authorization rules, delegation relations, and so on. The administration tools for RBAC components are beyond the scope of this paper. We can simply adopt an existing tool for that purpose such as Ferraiolo et al. [1999]. *Delegation / revocation service* implements RDM2000 as an administrative service, which authorizes and processes delegation/revocation requests. The core of the service is a rule-processing engine, which is used to decide if a delegation/revocation request can be authorized. The rule engine is an implementation of the rule-based policy language that provides an environment in which the basic authorization rules and other credentials can cooperate to produce a proof that the request complies with the authorization policies (or fail to produce such a proof). Rule processing is as follows: the rule engine accepts facts (user's role memberships and other credentials) and basic authorization rules as a set of axioms, and the user's request as a conjectured theorem; then it tries to prove this theorem, that is, to demonstrate that it can be logically derived from those axioms.

The delegation/revocation service is supported by *Authentication and access control services*. The authentication service is used to authenticate users during their initial sign-on and supply them with an initial set of credentials. Access control service makes access control decisions based on information supplied by role service. In CPOPS, the authentication and access control services are provided by CF tags. One reason that CF application server was chosen as the deployment environment of CPOPS is for its advanced security features. ColdFusion provides authentication tags and functions to determine the authentication status and the authority of each user. It allows the definition of a group of users from an LDAP server or a Windows NT domain. Although user groups are different from roles, as they do not have assigned permissions, they can be configured to implement roles.

The *project database* stores information related to problem-oriented policing projects, such as tables of investigations, crime analysis, police reports, project assessments, and so on. Although we abstract the protected resources in CPOPS as code segments, ultimately, objects of access control in the system are views of project tables. The *RBAC database* is an implementation of the RBAC96 and RDM2000 components. It stores tables specifying RBAC96 and RDM2000 components, such as users, roles, permissions, constraints, user assignments, permission assignments, role hierarchies, delegation tree, basic delegation/revocation rules, and so on. For example, the delegation tree is stored

Table VII. CPOPS components

Web servers, CF application server and CF tags	Web server is the application entry point. It has the responsibility of authenticating users and providing confidential data transmission through secure socket layer (SSL) connection together with CF application server which implements the business application logic through CF tags that are component-based software for accessing databases and providing custom-developed business logic.
Authentication and access control services	Authenticates users during their initial sign-on, supplies them with an initial set of credentials and makes access control decisions.
Role service	Acts as a façade for other services. It provides methods to create, retrieve, and update elements for RBAC database.
Delegation/revocation service	Authorizes and enforces delegation and revocation requests from users. The core of delegation/revocation service is a rule engine to optimize rule search, interpret rules and authorize user requests.
RBAC and project database	RBAC database stores tables specifying RBAC96 and RDM2000 components, such as users, roles, permissions, constraints, user assignment, permission assignment, role hierarchy, delegation/revocation rules, and so on. Project database stores tables of investigations, crime analysis, police reports, project assessments, and so on.

as table represented by the one-to-many relationships among the delegating user assignment and the delegated user assignments.

The descriptions of these main components are summarized in Table VII.

5.3 Delegation and Revocation Procedures

In this section, we describe how secure delegation and revocation are managed in our framework as shown in Figure 10.

1. A user initiates a session by connecting to the web server.
2. A SSL handshake is performed, which results in the link encryption between the browser and the web server through a session key.
3. The user is authenticated through username/password by the authentication service. If the user is successfully authenticated, role service will retrieve the user's role memberships (from both original user assignment and delegated user assignment) and display a list of assigned roles to the user.
4. The user selects role or role set to activate in current session. The role service will consult delegation/revocation service to verify that the user's role membership has not been revoked by cascading revocation. If the user's role membership is successfully verified, a RBAC session is then established. The user can then request permission to access protected resources. We omit the access control procedure deliberately in the diagram, as it is irrelevant

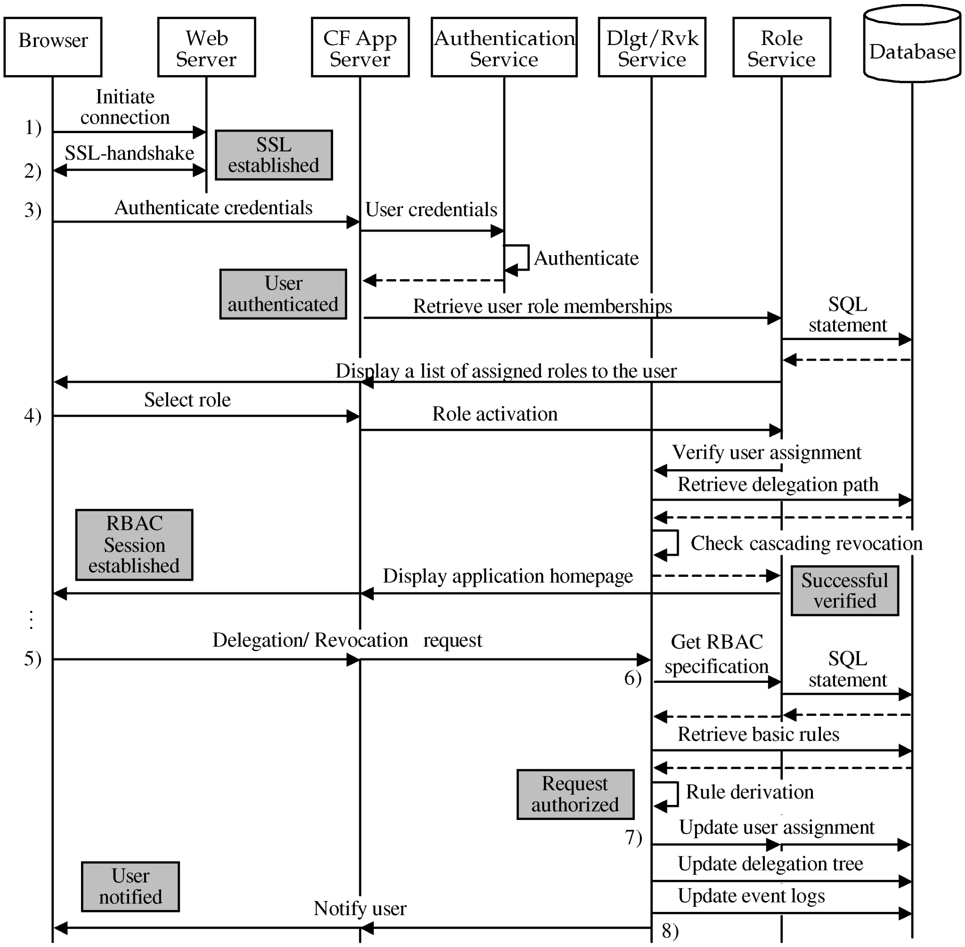


Fig. 10. Secure delegation/revocation data flow.

to our delegation framework. If the user needs to delegate or revoke a role, he proceeds to next step.

5. The user composes a delegation/revocation request. In the case of delegation, based on user's current active role, the user may choose the role or a junior role as well as the delegated user. He can also define an optional duration–restriction constraint for the delegation. A delegation request has the format: *request(type=DLGT, project, delegating_user, delegating_role, delegated_user, delegated_role, option, duration, revoke_option)*. In the case of revocation, a user needs to select the proper revocation scheme, for example, weak noncascading GD revocation (WNDR), weak cascading GD revocation (WCDR), and so on. A revocation request has the format: *request(type=RVK, project, revoking_user, revoking_role, revoked_user, revoked_role, option, null, null)*. Requests are sent to the CF application server. The delegation/revocation service is then invoked.

6. The delegated user's role membership information is retrieved from RBAC through role service as well as other RBAC specifications (role hierarchies, constraints, and so on) and basic authorization rules if necessary. The preprocessor transforms rules, users' credentials, and other RBAC/RDM2000 facts into a logic program. A query is created based on the delegation request. A delegation query has the format: *der_can_delegate(deleating_user, delegating_role, delegated_user, delegated_role, option)*, which implements the user–user delegation authorization derivation rule. A revocation query has several formats that implement revocation authorization derivation rules, for example, *der_can_revokeGD(revoking_user, revoking_role, revoked_user, revoked_role, option)*. The output and user's request (a query) is then fed to the inference engine for authorization.
7. If the request is authorized, the postprocessor in the delegation/revocation service updates the database for the RBAC and the RDM2000 elements. If the request is a delegation, a delegated user role assignment is created and the delegation tree is updated. Next time when the delegated user logs in, he will see the delegated role displayed in the list of roles assigned to him. If the request is a revocation, the delegated user role assignment is removed from the delegated user's role list. If the user selects strong revocation in Step 5, delegation/revocation service will repeat Step 6 and 7 for all roles that are delegated to the delegated user senior to the role in the revocation request. If the delegation is a cascading revocation, the delegation/revocation service changes the status of the delegated user role assignment in the delegation tree. The transaction is then recorded in event logs.
8. Both users in this transaction are notified. If the delegated user is not online, he will be notified next time when he logs in to the CPOPS.

Note that Step 3 to authenticate the user using username/password is optional since the web server can accept client certificates as a means for user authentication. In addition, CPOPS uses Oracle 8i as its database Server. The connection between CF application server and Oracle database server is secured by IIOP/SSL. Although we demonstrated the functionality of delegation and revocation using the CPOPS example in this section, the procedures are not limited to a particular application.

6. SYSTEM IMPLEMENTATION

Our proof-of-concept has two parts: the implementation of RDM2000 and the implementation of the rule-based policy language. We implement them as the delegation/revocation service: users' delegation/revocation requests are interpreted, authorized, and processed by the service; it creates RDM2000 elements based upon users' requests and maintains the integrity of the database by checking and enforcing consistency rules [Ferraiolo et al. 1999; Zhang et al. 2001, 2002]. The core of this service is a rule engine. Currently, we implement the rule inference engine by extending SWI-prolog [wielemaker] using its C++ interface. The rule engine has three functional units: a preprocessor, an inference engine, and a postprocessor. The preprocessor transforms rules, users' credentials, and other facts into a logic program. It is defined as

- *preprocess(request : class, roleservice : class)*
  It takes a user request object and a role service object as input and returns a Prolog program.

The output and user's request (a query) is then fed to the inference engine for answer. The inference engine is defined as

- *inference(request : class, program : string)*
  It takes a user request object and a Prolog program as input and returns the result of the program.

The postprocessor saves the result of an authorized delegation or revocation to the RBAC database and logs the transaction. It is defined as

- *postprocess(request : class, result : boolean, roleservice : class)*
  It takes a user request object, the inferred result and a role service object as input.

A functional definition of the rule engine is

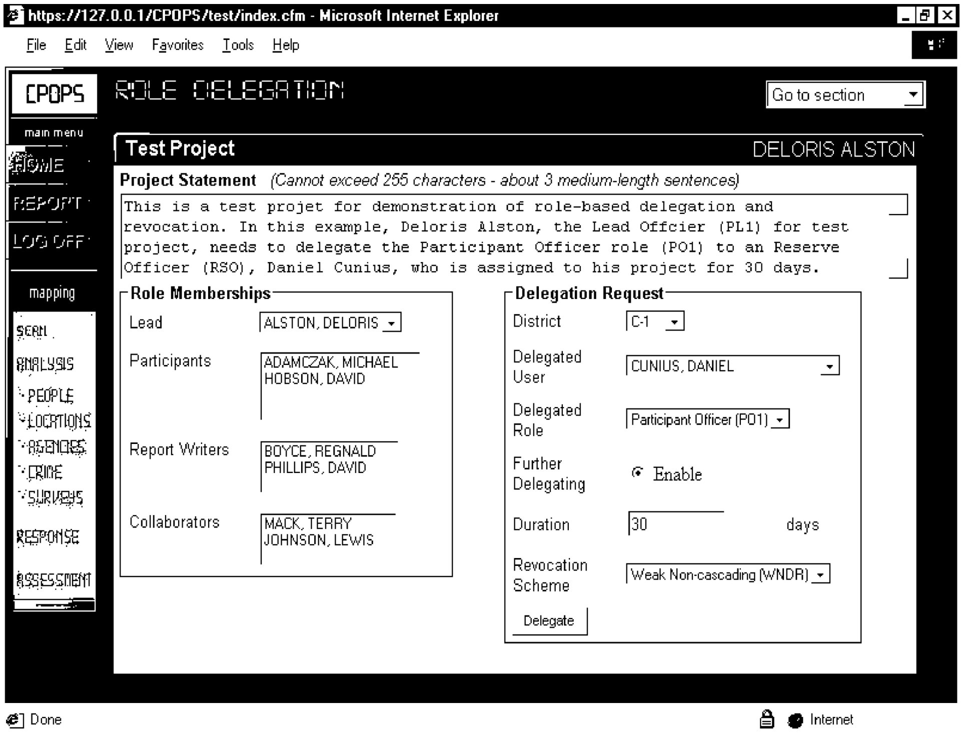
- *postprocess(request : class, inference(request : string, preprocess(request : string, roleservice : class)), roleservice : class)*

Note that the request data can be either delegation request or revocation request. It is handled through *polymorphism*. The definition of other related classes, *CRoleService* and *CDelegationRevocationService*, is illustrated in Appendix A.

We have developed graphic user interfaces (GUIs) to handle delegation and revocation requests in CPOPS. One example GUI for lead officers to manage project user role memberships is illustrated in Figure 11. In this example, an officer, *Deloris Alston*, acting as lead officer (LED) in project *test*, delegates one junior role participant officer (PRT) in the project to *Daniel Cunius*, a reserve officer (RES) from distinct C-1 who is assigned to her team temporarily (for 30 days). The delegation request form is shown in bottom-left part of Figure 11. The bottom left part of the diagram shows roles and users in the test project. *Deloris Alston* is in the project lead officer role. The bottom right part is a delegation request form. It consists of the delegated user, delegated role, further delegation option, and the optional duration constraint. Based on the form, a delegation request is created:

> *request : type=DLGT, project=test, delegating_user=Deloris_Alston, delegating_role=PL1, delegated_user=Daniel_Cunius, delegated_role=PO1, option=TRUE, duration=30, revoke_option=WNDR.*

Note that a revocation scheme needs to be specified for the duration–restriction triggered revocation. The delegation request is then forwarded to the delegation/revocation service, shown as Step 5 to Step 8 in Figure 10. The rule engine will process the request and different function calls in class *CDelegationRevocationService* are invoked, for example, *PreProcess*, *Inference*, *PostProcess*, and so on. The delegation request and rule engine log for this example are presented in Appendix B.



Test Project DELORIS ALSTON

Project Statement *(Cannot exceed 255 characters - about 3 medium-length sentences)*

```
This is a test projet for demonstration of role-based delegation and
revocation. In this example, Deloris Alston, the Lead Offcier (PL1) for test
project, needs to delegate the Participant Officer role (PO1) to an Reserve
Officer (RSO), Daniel Cunius, who is assigned to his project for 30 days.
```

Role Memberships

Lead	ALSTON, DELORIS
Participants	ADAMCZAK, MICHAEL HOBSON, DAVID
Report Writers	BOYCE, REGNALD PHILLIPS, DAVID
Collaborators	MACK, TERRY JOHNSON, LEWIS

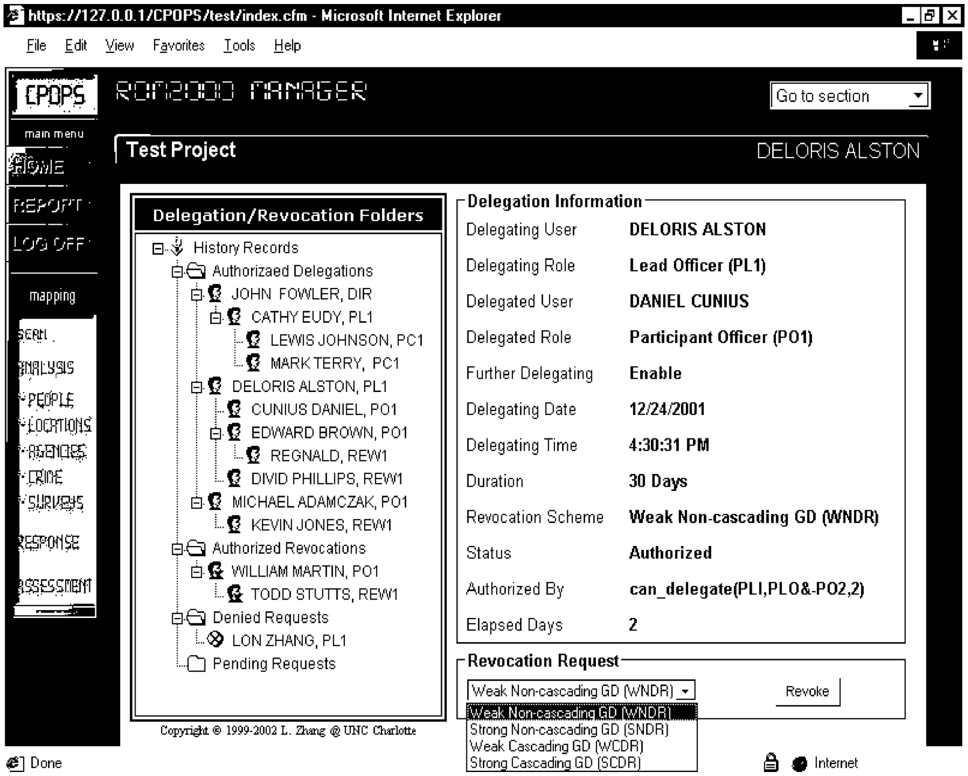
Delegation Request

District	C-1
Delegated User	CUNIUS, DANIEL
Delegated Role	Participant Officer (PO1)
Further Delegating	● Enable
Duration	30 days
Revocation Scheme	Weak Non-cascading (WNDR)

Fig. 11. GUI for delegation request in CPOPS.

A more complicated RDM2000 management interface is illustrated in Figure 12. Only directors and project leaders have access to the RDM2000 manager. This diagram clearly demonstrates the delegation trees. The tree view contains the history of the delegations/revocations and their status in the test project. The history is categorized into four folders: authorized delegation folder containing the delegation trees, authorized revocation folder, pending requests folder, and denied requests folder. The authorized delegation folder consists of not only the directly authorized delegations but also those further propagated delegated user assignments by multistep delegations. It can see clearly the RDM2000 constructs in these trees. Each node in the authorized delegation tree refers to a user assignment and each edge to a delegation relation. The depth of a user assignment in the tree is referred to as the delegation depth. *Deloris* can select the delegated user-role assignment to view the detailed information of each delegation, as shown in the right of the diagram. The bottom right includes a revocation form. After selecting a delegated user-role assignment, *Deloris* can select a revocation scheme and request to revoke the delegated user-role assignment. For example, if she selects weak noncascading GD revocation, a revocation request is created as

request : type=RVK, project=test, revoking_user= Deloris_Alston, revoking_role=PL1, revoked_user=Daniel_Cunius, revoked_role=PO1, option=WNDR, null, null).


https://127.0.0.1/CPOPS/test/index.cfm - Microsoft Internet Explorer
File Edit View Favorites Tools Help
CPOPS
RDM2000 MANAGER
Go to section
main menu
HOME
REPORT
LOG OFF
mapping
SCAN
ANALYSIS
PEOPLE
LOCATIONS
AGENCIES
CRIME
SURVEYS
RESPONSE
ASSESSMENT
Test Project
DELORIS ALSTON
Delegation/Revocation Folders
Revocation Request
Weak Non-cascading GD (WNDR)
Strong Non-cascading GD (SNDR)
Weak Cascading GD (WCDR)
Strong Cascading GD (SCDR)
Revoke
Copyright © 1999-2002 L. Zhang @ UNC Charlotte
Done
Internet

History Records
- Authorizaed Delegations
  - JOHN FOWLER, DIR
    - CATHY EUDY, PL1
      - LEWIS JOHNSON, PC1
      - MARK TERRY, PC1
  - DELORIS ALSTON, PL1
    - CUNIUS DANIEL, PO1
    - EDWARD BROWN, PO1
      - REGNALD, REW1
    - DIVID PHILLIPS, REW1
  - MICHAEL ADAMCZAK, PO1
    - KEVIN JONES, REW1
- Authorized Revocations
  - WILLIAM MARTIN, PO1
    - TODD STUTTS, REW1
- Denied Requests
  - LON ZHANG, PL1
- Pending Requests

Delegation Information

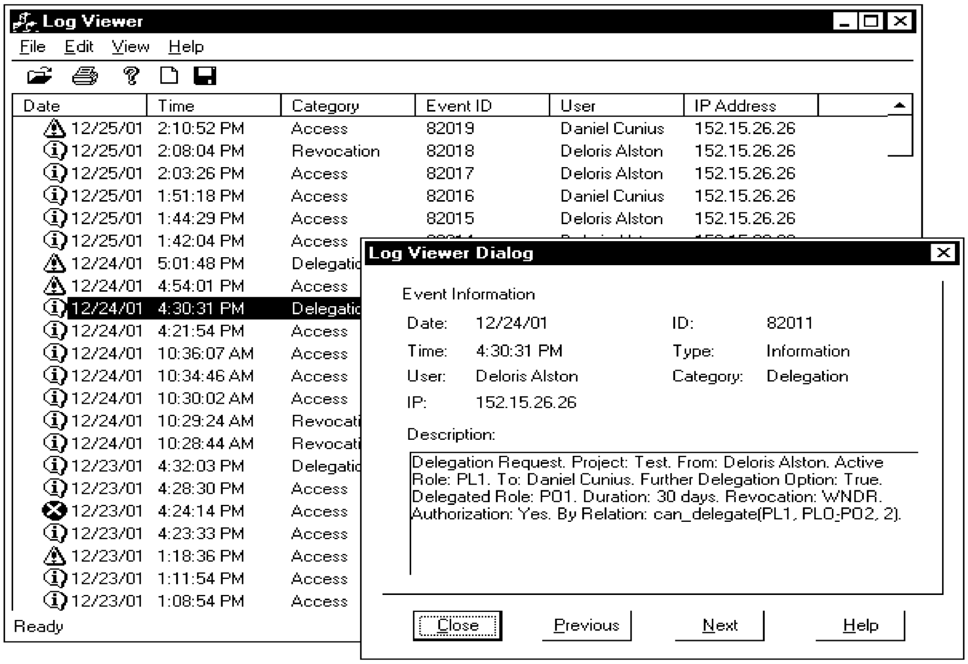
Delegating User	DELORIS ALSTON
Delegating Role	Lead Officer (PL1)
Delegated User	DANIEL CUNIUS
Delegated Role	Participant Officer (PO1)
Further Delegating	Enable
Delegating Date	12/24/2001
Delegating Time	4:30:31 PM
Duration	30 Days
Revocation Scheme	Weak Non-cascading GD (WNDR)
Status	Authorized
Authorized By	can_delegate(PLI,PLO&-PO2,2)
Elapsed Days	2

Fig. 12. GUI for RDM2000 management and revocation request in CPOPS.

The benefits of allowing users to administer user-role assignments at some points will come into the risk of exposing problem-oriented policing project information to unauthorized people. Although we assume that users can be trusted to exercise discretion in how they delegate, we cannot simply neglect the possibilities of security breaches. Designers of secure access control have traditionally emphasized audit capabilities [Aura 1999]. That is, every action should be traced back to an entity that can be held responsible for it. CPOPS has severe audit requirements. This is for both safety and legal reasons. Access to problem-oriented policing projects should be logged with the user's name, as well as date and time; all delegation and revocation actions should be marked on the audit trail. A security officer can review these access records and audit trails periodically, so that breaches can be traced and detected. The audit records of delegation and revocation in above GUI examples are shown in Figure 13. The diagram shows a list of recorded delegation/revocation and other system events. The detailed information of delegation from *Deloris* to *Daniel* can be viewed in the event viewer dialog.

7. RELATED WORK

Although the concept and McDermott of delegation is not new in authorizations [Abadi et al. 1993; Aura 1999; Barka and Sandhu 2000a; Gasser and McDermott



Date	Time	Category	Event ID	User	IP Address
⚠ 12/25/01	2:10:52 PM	Access	82019	Daniel Cunius	152.15.26.26
ⓘ 12/25/01	2:08:04 PM	Revocation	82018	Deloris Alston	152.15.26.26
ⓘ 12/25/01	2:03:26 PM	Access	82017	Deloris Alston	152.15.26.26
ⓘ 12/25/01	1:51:18 PM	Access	82016	Daniel Cunius	152.15.26.26
ⓘ 12/25/01	1:44:29 PM	Access	82015	Deloris Alston	152.15.26.26
ⓘ 12/25/01	1:42:04 PM	Access	[illegible]	[illegible]	[illegible]
⚠ 12/24/01	5:01:48 PM	Delegatic			
⚠ 12/24/01	4:54:01 PM	Access			
ⓘ 12/24/01	4:30:31 PM	Delegatic			
ⓘ 12/24/01	4:21:54 PM	Access			
ⓘ 12/24/01	10:36:07 AM	Access			
ⓘ 12/24/01	10:34:46 AM	Access			
ⓘ 12/24/01	10:30:02 AM	Access			
ⓘ 12/24/01	10:29:24 AM	Revocati			
ⓘ 12/24/01	10:28:44 AM	Revocati			
ⓘ 12/23/01	4:32:03 PM	Delegatic			
ⓘ 12/23/01	4:28:30 PM	Access			
⊗ 12/23/01	4:24:14 PM	Access			
ⓘ 12/23/01	4:23:33 PM	Access			
⚠ 12/23/01	1:18:36 PM	Access			
ⓘ 12/23/01	1:11:54 PM	Access			
ⓘ 12/23/01	1:08:54 PM	Access			

Event Information

Date:	12/24/01	ID:	82011
Time:	4:30:31 PM	Type:	Information
User:	Deloris Alston	Category:	Delegation
IP:	152.15.26.26		

Description:

Delegation Request. Project: Test. From: Deloris Alston. Active Role: PL1. To: Daniel Cunius. Further Delegation Option: True. Delegated Role: P01. Duration: 30 days. Revocation: WNDR. Authorization: Yes. By Relation: can_delegate(PL1, PLO-PO2, 2).

Fig. 13. Event logs in CPOPS.

1990; Gladney 1997; Li et al. 1999], role-based delegation received attention only recently [Barka and Sandhu 2000a, 2000b; Linn and Nyström 1999; Zhang et al. 2001, 2002]. Delegation is the main mechanism for access rights management in new distributed discretionary access control [Aura 1999]. In their approach, the principal entities are the cryptographic keys and delegation of access rights is signed with public key cryptography. With a certificate, one cryptographic key delegates some of its authority to another key. The main advantage of certificates lies in decentralization. However, their approach is a form of discretionary access control. For example, separation of duty policies cannot be expressed with only certificates. They need some mechanism to maintain the previously granted rights and the histories must be updated in real time when new certificates are issued. Delegation is also an important concept in decentralized trust management [Blaze et al. 1996, 1999; Li 1999, 2000]. Trust management deals with authorization in highly distributed systems, for example, the Internet. Other researchers have investigated the problem of delegation between machine to machine and human to machine [Abadi et al. 1993; Gasser and McDermott 1990; Gladney 1997]. Some of these delegation models had included the role concept. But they considered roles as a special kind of access rights that are same as permissions. Many important role-based concepts, for example, role hierarchies, constraints, were not addressed.

A work closely related to ours is RBDM0 (role-based delegation model zero) proposed by Barka and Sandhu (2000a). Their work was cast within RBAC0, the simplest form of RBAC96. RDBM0 is a simple delegation model supporting only flat roles and single step delegation. They distinguished the *delegated user assignment (UAD)* from *original user assignment (UAO)*. UAO is an original user to role assignment relation, while UAD is a delegated user to role assignment. Unfortunately, RBDM0 failed to formalize the relationships among UAO and UAD. Thus they did not give the definition of role-based delegation relation, which is a critical notion to the delegation model. Barka and Sandhu also discussed some advanced features of RBDM0, for example, grant-dependent revocation, two-step delegation, and so on. They mentioned possibility to extend RBDM0 to support role hierarchies. However, without defining the delegation relation, one cannot formalize these features. The lack of formality restricts the applicability of RBDM0 in practice. Barka and Sandhu (2000b) identified some critical role-based delegation features such as delegation with hierarchical roles, partial delegation, multistep delegation, temporary delegation and different revocation schemes. They reduced the large number of possible cases to a few cases that can be useful in practice. Moreover, their focus is on the theoretical aspect of delegation framework without any implementation concern. A major difference between our work and theirs is that we not only give the formal definitions of most useful delegation features but also illustrate how they can be applied in current role-based systems. In Hayton et al. [1998] and Yao et al. [2001], a role-based access control architecture, which is designed to facilitate access control in distributed systems, is introduced. Central to the OASIS model is the idea of credential-based role activation. They defined the notion of appointment so that a user may issue an appointment certificate for privilege propagation. Even though they claimed that delegation could be viewed as a special case of appointment, some important delegation features such as delegation tree, partial delegation, delegation revocation, and so on have not been addressed in their work. In addition, the appointment model lacks a notion of role hierarchy, which is an essential component of RBAC. Our work explored those issues including totality of delegation and delegation relation.

In ARBAC97, Sandhu et al. [1999] developed URA97 for security officers to handle the user assignment. In our approach [Zhang et al. 2001, 2002], the delegation from one user to another is actually assigning the delegated role to a user. Thus, the delegating user needs to perform user assignment too. It is necessary to differentiate between user assignment and delegation. In a user assignment, the security officer must activate the administrative role; while in user delegation, the delegating user activates his/her regular role. Although it is possible to delegate an administrative role, we only consider the regular role delegation in this paper.

A number of researchers have looked at the semantics of authorization, delegation, and revocation. Abadi et al. [1993] provided explicit support of role authorizations for users. In their approach, roles are used to restrict users' privileges for particular execution. However, as a language for authentication and access control, this approach is rather limited. They have included delegation concept in their work, but there is no multistep delegation control mechanism—every delegation can be freely redelegated. Li et al. [1999] proposed delegation logic (DL) for authorization in large-scale, open, distributed systems. Their focus is to develop a trust-management language to represent policies and credentials. In their logic, role-based concepts were not fully adopted; neither did they address revocation. Hagstrom et al. [2001] categorized revocation in owner-based framework into different schemes by post conditions. However, their attempt is not sufficient to model all the revocations required in role-based delegation, for example, grant-independent and duration-restricted revocations. Jajodia et al. [1997] proposed a logical language called authorization specification language (ASL) for expressing authorization. Although ASL supports multiple access control policies, it is not role-oriented framework. It cannot specify authorizations in role hierarchies. In addition, their framework does not address delegation authorization. Unlike ASL, we focus exclusively on how to specify and enforce policies for authorizing role-based delegation and revocation using a rule-based language. This kind of language for role-based delegation has not been studied in the literature.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we have proposed a rule-based framework for role-based delegation called RDM2000. We introduced rule-based specification language to specify and enforce policies. A proof-of-concept prototype implementation of the proposed framework was described. This web-based tool for law-enforcement agencies on a distributed environment supports reliable delegation and revocation. Major contributions of this work include the identification of delegation relation, comprehensive delegation model, systematic role-based delegation policy specification using rule-based language that has not been addressed in the literature, role delegation in role hierarchy and multistep delegation. We demonstrate the feasibility of the proposed framework and provide secure protocols for managing delegations through a proof-of-concept prototype implementation of RDM2000 in CPOPS.

There are many challenges that remain to be explored. A delegating user may need to delegate a role to all members of another role at the same time. For example, project lead officer *Cathy* may want to delegate role *PC1* to all participant officers in her team. This type of delegation is more effective if we adopt a group-based delegation. We only consider deletion of regular roles in this paper; we need to explore administrative role delegation in the future. Another issue is that how do original role assignment changes impact delegations. For example, what are the consequences to the delegation tree if the original user role assignment is revoked? In addition, we are now experimenting with representing rules with XML-based languages. The future work would embrace the extension of RDM2000 model to incorporate these challenges and enhancement of the rule-based policy language.

APPENDIX A

```
class CRoleService {
public:
  // @cmember Retrieves the roles assigned to a user.
  virtual CTypedPtrArray<CObArray,CRole*>* GetUserRoleAssignmentList
    (CString lpszName) = 0;
    // @cmember Retrieves the permissions assigned to a role.
    virtual CTypedPtrArray<CObArray,CPermission*>*
      GetPermissionRoleAssignmentList(CString lpszName) = 0;
   // @cmember Retrieves the permissions assigned to a role.
   virtual CTypedPtrArray<CObArray,CRole*>* GetActivateRoles() = 0;
    // @cmember Retrieves all immediate senior roles to a role.
   virtual CTypedPtrArray<CObArray,CRole*>* GetSenior Roles(CRole*
       pRole) = 0;
    // @cmember Retrieves all immediate junior roles to a role.
   virtual CTypedPtrArray<CObArray,CRole*>* GetJunior Roles(CRole*
      pRole) = 0;
    //@cmember Retrieves all conflicting roles to a role.
   virtual CTypedPtrArray<CObArray,CRole*>* GetConflictRoles(CRole*
      pRole) = 0;
    //@cmember Retrieves all conflicting user to a user.
   virtual CTypedPtrArray<CObArray,CRole*>* GetConflictUsers(CString
      lpszName) = 0;
    //@cmember Assign role to a user.
   virtual void Assignr(CString lpszName, CRole* pRole) = 0;
    ......
    };
class CDelegationRevocationService {
public:
    //@cmember PreProcess.
   virtual CString PreProcess(CString request, CRoleService* pService) = 0;
    //@cmember Inference.
   virtual BOOL Inference(CString request, CString program) = 0;
    //@cmember PostProcess.
   virtual void PostProcess(CString request, CString program) = 0;
    //@cmember Retrieves delegation depth.
   virtual int GetDelegationDepth(CRole* pRole) = 0;
    //@cmember Retrieves delegation rules.
   virtual CTypedPtrArray<CObArray,CRule*>* GetDelegationRule(CRole*
      pRole) = 0;
    //@cmember Retrieves revocation rules.
   virtual CTypedPtrArray<CObArray,CRule*>* GetRevocationRule(CRole*
      pRole) = 0;
    //@cmember Validate cascading revocation.
   virtual BOOL ValidateCascadeRevocation(CString lpszName, CRole*
      pRole) = 0;
    //@cmember Revoke.
   virtual void Revoke(CString lpszName, CRole* pRole, CRoleService*
      pService) = 0;
    //@cmember Delegate.
   virtual void Delegate(CString lpszName, CRole* pRole, CRoleService*
      pService) = 0; ......
};
```

APPENDIX B

```
%% Delegation Engine Log
%% Log ID: 82011
%% Date: 12/24/01
%% Time: 4:30:31 PM
%% From: Deloris Alston
%% Request: Type="DLGT"
  Project="test"
    From="Deloris Alston"
ActiveRole="PL1"
    To="Daniel Cunius"
DelgatedRole="PO1"
    opt="TRUE" duration="30"
rvk="WNDR"
%% Preprocessing Output - Prolog
  Program
%% Template applies to all
  delegation requests
der_can_delegate(U1, R1, U2, R2,
  Opt) :-
    active(U1, R1, _),
    delegatable(U1, R1),
    senior(R1, R),
    can_delegate(R, CR, N),
    in(U2, CR),
    junior(R2, R),
    depth(U1, R1, D),
    lt(D, N),
    not(error(U1, R1, U2, R2)).
%% basic authorization rules,
  optimized by preprosessor
can_delegate(pl1, rso, 2).
can_delegate(pl1, plo, 1).
. . .
%% representing role hierarchies
parent(dir, pl1).
parent(dir, pl2).
parent(pl1, po1).
. . .
senior(X, X).
senior(X, Y) :-
    parent(X, Y).
senior(X, Z) :-
    parent(X, Y),
    senior(Y, Z).
junior(X, Y) :-
    senior(Y, X).
%% current RBAC specifications
  for Deloris Alston
active(deloris_alston, pl1, _).
delegatable(deloris_alston, pl1).
depth(deloris_alston, pl1, 0).
in(deloris_alston, pl1).
%% current RBAC specification for
  Daniel Cunius
in(daniel_cunius, rso).
%% Support role hierarchy
in(X, Y) :-
    in(X, Z),
    senior(Z, Y).
%% constraints
%% conflicting roles
conflictingr(rso, cso).
%% conflicting users
conflictingu(daniel_cunius,
  kevin_jones).
%% enforce constraints
error(U1, R1, U2, R2) :-
    conflictingr(R2, R),
    in(U2, R).
error(U1, R1, U2, R2) :-
    conflictingu(U2, U),
    in(U, R2).
%% utility functions
lt(X, Y) :-
X<Y.
%%End Program
%% inferring...
der_can_delegate(deloris_alston,
  pl1, daniel_cunius, po1, true).
%% result: AUTHORIZED.
%% Postprocessing...
%% Update database
. . .
```

ACKNOWLEDGMENTS

We wish to thank Trent Jaeger at IBM T. J. Watson Research Center and the anonymous referees for their careful reading of the paper, constructive criticisms, and insightful comments. This work was partially supported by the National Science Foundation.

REFERENCES

Abadi, M., Burrows, M., Lampson, B., and Plotkin, G. 1993. A calculus for access control in distributed systems. *ACM Trans. Program. Lang. Syst. 15*, 4(Sept.), 706–734.

Abiteboul, S. and Grumbach, S. 1991. A rule-based language with functions and sets. *ACM Trans. Database Syst. 16*, 1–30.

Ahn, G. and Sandhu, R. 2000. Role-based authorization constraints specification. *ACM Transactions on Information and System Security 3*, 4, ACM (November) 207–226.

Aura, T. 1999. Distributed access-rights management with delegation certificates. *Security Internet programming. J. Vitec and C. Jensen Eds.* Springer, Berlin, 211–235.

Barka, E. and Sandhu, R. 2000. A role-based delegation model and some extensions. In *Proceedings of 16th Annual Computer Security Application Conference*, Sheraton New Orleans, December 11–15, 2000a.

Barka, E. and Sandhu, R. 2000. Framework for role-based delegation model. In *Proceedings of 23rd National Information Systems Security Conference*, Baltimore, October 16–19, 2000b, 101–114.

Bhamidipati, V. and Sandhu, R. 2000. Push Architectures for USER ROLE assignment. In *Proceedings of 23rd National Information Systems Security Conference*, Baltimore, October 16–19, 2000, 89–100.

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

Blaze, M., Feigenbaum, J., Ioannidis, J., and Keromytis, A. 1999. The role of trust management in distributed system security. *Security Internet Programming*. J. Vitec and C. Jensen, eds. Springer, Berlin, 185–210.

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

Ferraiolo, D., Barkley, J., and Kuhn, D. R. 1999. A role-based access control model and reference implementation within a corporate intranet. *ACM Transactions on Information and System Security 2*, 1(February), 34–64.

Ferraiolo, D., Cugini, J., and Kuhn, D. R. 1995. Role-based access control (RBAC): features and Motivations. In *Proceedings of 11th Annual Computer Security Application Conference*. New Orleans, LA, December 11–15 1995, 241–241.

Forta, B. (ed), 1998. *Nate Weiss. Advanced ColdFusion 4.0 Application Development*. MacMillan Company.

Forta, B. (ed). 2001. *Certified ColdFusion Developer Study Guide*. 1st edn. Macromedia Press.

Gasser, M. and McDermott, E. 1990. An architecture for practical delegation a distributed system. *IEEE Computer Society Symposium on Research in Security and Privacy*. Oakland, CA, May 7–9, 1990.

Gladney, H. M. 1997. Access control for large collections. *ACM Transactions on Information Systems 15*, 2(April), 154–194.

Hagstrom, A., Jajodia, S., Presicce, F. P., and Wijesekera, D. 2001. Revocations—a classification. In *Proceedings of 14th IEEE Computer Security Foundations Workshop*, Nova Scotia, Canada, June 2001, 44–58.

Hayton, R., Bacon, J., and Moody, K. 1998. OASIS: access control in an open, distributed environment. In *Proceedings of 1998 IEEE Symposium on Security and Privacy*. Oakland, CA, May 3–6. IEEE Computer Society Press, Los Alamitos, CA, 3–14.

Jajodia, S., Samarati, P., and Subrahmanian, V. S. 1997. A Logical language for expressing authorizations. *IEEE Symposium on Security and Privacy*. May 1997.

LAMPSON, B. W., ABADI, M., BURROWS, M. L., AND WOBBER, E. 1992. Authentication in distributed systems: theory and practice. *ACM Transactions on Computer Systems 10*, 4, 265–310, November 1992.

LI, N., FEIGENBAUM, J., AND GROSOF, B. N. 1999. A logic-based knowledge representation for authorization with delegation (extended abstract). In *Proceeding 12th intl. IEEE Computer Security Foundations Workshop*, (extended version is IBM Research Report RC 21492).

LI, N. AND GROSOF, B. N. 2000. A practically implementation and tractable delegation logic. *IEEE Symposium on Security and Privacy*. May 2000.

LIEBRAND, M., ELLIS, H. J., PHILLIPS, C., AND TING, T. C. 2002. Role delegation for a distributed, unified RBAC/MAC. In *Proceedings of Sixteenth Annual IFIP WG 11.3 Working Conference on Data and Application Security King's College*, University of Cambridge, UK July 29–31, 2002.

LINN, J. AND NYSTRÖM, M. 1999. Attribute certification: an enabling technology for delegation and role-based controls in distributed environments. *ACM Workshop on Role-Based Access Control* 121–130.

MCNAMARA, C. 1997. Basics of delegating. http://www.mapnp.org/library/guiding/delegate/basics.htm.

SANDHU, R. 1997. Rational for the RBAC96 family of access control models. In *Proceedings of 1st ACM Workshop on Role-based Access Control*.

SANDHU, R., BHAMIDIPATI, V., AND MUNAWER, O. 1999. The ARBAC97 model for role-based administration of roles. *ACM Transactions on Information and System Security 2*, 1(February), 105–135.

SANDHU, R., COYNE, E., FEINSTEIN, H., AND YOUMAN, C. 1996. Role-based access control model. *IEEE Computer 29*, 2(February).

WIELEMAKER, J. SWI-Prolog. http://www.swi.psy.uva.nl/projects/SWI-Prolog/

YAO, W., MOODY, K., AND BACON, J. 2001. A model of OASIS role-based access control and its support for active security. In *Proceedings of ACM Symposium on Access Control Models and Technologies* (*SACMAT*), Chantilly, VA, May 3–4, 2001, 171–181.

ZHANG, L., AHN, G., AND CHU, B. 2001. A Rule-based framework for role-based delegation. In *Proceedings of ACM Symposium on Access Control Models and Technologies* (*SACMAT 2001*), Chantilly, VA, May 3–4, 2001 153–162.

ZHANG, L., AHN, G., AND CHU, B. 2002. A role-based delegation framework for healthcare information systems. In *Proceedings of ACM Symposium on Access Control Models and Technologies* (*SACMAT 2002*). Monterey, CA, June 3–4, 2002, 125–134.

Received October 2001; revised July 2002; November 2002; January 2003; accepted April 2003