Authorization Algorithms for Permission-Role Assignments

Lili Sun, Hua Wang, Jianming Yong
(University of Southern Queensland, Queensland, Australia
{sun, wang, yong}@usq.edu.au)

Abstract: Permission-role assignments (PRA) is one important process in Role-based access control (RBAC) which has been proven to be a flexible and useful access model for information sharing in distributed collaborative environments. However, problems may arise during the procedures of PRA. Conflicting permissions may assign to one role, and as a result, the role with the permissions can derive unexpected access capabilities. This paper aims to analyze the problems during the procedures of permission-role assignments in distributed collaborative environments and to develop authorization allocation algorithms to address the problems within permission-role assignments. The algorithms are extended to the case of PRA with the mobility of permission-role relationship. Finally, comparisons with other related work are discussed to demonstrate the effective work of the paper.

Keywords: access control, conflicts, authorization
Categories: H.1.1, H.2.8, I.1.2

1 Introduction

Information security currently has vast impact to information industry, but also to individuals and our daily life [Bella, 2008, Chen et al. 2008]. Access control, as a significant method to secure the information in distributed environments, has been studied by many researchers [Harrer et al. 2008, Wang et al. TKDE]. The National Institute of Standards and Technology developed the role-based access control (RBAC) prototype [Feinstein H. L. 1995] and published a formal model [Ferraiolo and Kuhn 1992]. RBAC has been widely used in database system management and distributed environments since it enables managing and enforcing security in large-scale and enterprise-wide systems. RBAC involves individual users being associated with roles as well as roles being associated with permissions (Each permission is a pair of objects and operations). As such, a role is used to associate users and permissions. A user in this model is a human being. A role is a job function or job title within the organization associated with authority and responsibility.

Permission is an approval of a particular operation to be performed on one or more objects. As shown in Figure 1, the relationships between users and roles, and between roles and permissions are many-to-many (i.e. a permission can be associated with one or more roles, and a role can be associated with one or more permissions).

Recently, role based access control (RBAC) has been widely used in database system management and operating system products since its management advantages [Wang et al. 2003ADC]. In 1993, the National Institute of Standards and Technology (NIST) developed prototype implementations, sponsored external research [Feinstein
Many organizations prefer to centrally control and maintain access rights, not so much at the system administrator's personal discretion but more in accordance with the organization's protection guidelines [David et al. 1993]. RBAC is being considered as part of the emerging SQL3 standard for database management systems, based on their implementation in Oracle 7 [Sandhu R. 1997]. Many RBAC practical applications have been implemented [Barkley et al. 1999, Ferraiolo et al. 1999, Sandhu R. 1998, Wang et al. ACSC04, Wang et al. TSMC].

However, there is a consistency problem when using RBAC management. For instance, if there are hundreds of permissions and thousands of roles in a system, it is very difficult to maintain consistency because it may change the authorization level, or imply high-level confidential information to be derived when more than one permission is requested and granted.

The permissions assigned to a role by administrators may conflict. For example, the permission for approving a loan in a bank is conflicting with the permission of funding a loan. These two permissions cannot be assigned to a role; however, because of role hierarchies, a role may still have these permissions even if they have been revoked from the role. In the latter case, a user with this role is able to access objects in the permission and has operations on the objects. There are evident problems with the processes of assigning and revocation.

**Authorization granting problem --** How to check whether a permission is in conflict with the permissions of a role?

**Authorization revocation problem --** How to find whether permissions of a role have been revoked from the role or not?

For example, Figure 2 shows a system administrative role (BankSO) in a bank to manage regular roles such as AUDITOR, TELLER, ACCOUNT_REP and MANAGER. Role MANAGER inherits AUDITOR and TELLER. ACCOUNT_REP

![Diagram of RBAC relationship](image)
has a SSD relationship with AUDITOR as well as DSD relationship with TELLER [Wang et al. TKDE].

The administrative role BankSO can assign audit permission or cash operation permission to a role but not both, otherwise it compromises the security of a bank system. Our aim is to provide relational algebra algorithms to solve the problems and then automatically check conflicts when assigning and revoking.

![Administrative role and role Relationships in a bank](image)

Based on the database and its tables such as ROLES, SEN-JUN in the paper [Wang et al. 2003ADC], this paper is going to develop formal approaches to check the conflicts and thereby help allocate the permissions without compromising the security. The formal approaches are based on relational structure and relational algebra operations. To my knowledge, this is the first attempt in this area to develop formal approaches for permission allocation and conflict detection.

<table>
<thead>
<tr>
<th>RoleName</th>
<th>MANAC</th>
<th>AUDC</th>
<th>ACCOUNT_REPC</th>
<th>TELLERC</th>
</tr>
</thead>
<tbody>
<tr>
<td>MANAGER</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>AUDITOR</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>ACCOUNT_REP</td>
<td>-1</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>TELLER</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 1: The relation ROLES in Figure 2

The ROLES relation in Figure 2 is in Table 1. The attribute TELLERC shows whether the role TELLER is conflicting with the RoleName in the relation or not. For
instance, in the third tuple, a user with role TELLER has conflicts with the role AUDITOR.

SEN-JUN - This is a relation of roles in a system. Senior is the senior of the two roles. Table 2 expresses the SEN-JUN relationship in Figure 2.

<table>
<thead>
<tr>
<th>Senior</th>
<th>Junior</th>
<th>PermsName</th>
<th>Oper</th>
<th>Obj</th>
<th>ConfPer</th>
</tr>
</thead>
<tbody>
<tr>
<td>MANAGER</td>
<td>AUDITOR</td>
<td>Approval</td>
<td>approve</td>
<td>Cash/check</td>
<td>Funding</td>
</tr>
<tr>
<td>MANAGER</td>
<td>TELLER</td>
<td>Funding</td>
<td>invest</td>
<td>cash</td>
<td>Approval</td>
</tr>
<tr>
<td>TELLER</td>
<td>BANK</td>
<td>Audit</td>
<td>audit</td>
<td>record</td>
<td>TELLER</td>
</tr>
<tr>
<td>AUDITOR</td>
<td>BANK</td>
<td>Teller</td>
<td>transfer</td>
<td>cash</td>
<td>Audit</td>
</tr>
</tbody>
</table>

Table 2: SEN-JUN in Figure 2

Table 3: An example of the relation PERM

The new tables like PERM and ROLE_PERM are needed. PERM - This is a relation of \(\{\text{PermName}, \text{Oper}, \text{Object}, \text{ConfPer}\}\):

- \text{PermName} is the primary key for the table, and is the name of the permission in the system.
- \text{Oper} is the name of the operation granted. It has information about the object that the operation is granted on.
- \text{Object} is the database item that can be accessed by the operation. It can be a database, a table, a view, an index or a database package.
- \text{ConfPer} is a set of permissions that conflict with the \text{PermName} in the relation.

For example, a staff member in a bank cannot have both permissions of approval and funding as well as both permissions of audit and teller. The relation of PERM can be expressed as Table 3.

ROLE-PERM - is a relationship between the ROLES and the PERM, listing what permissions are granted to what roles. It has two attributes:

- \text{RoleName} is a foreign key RoleName from the table ROLES.
- \text{PermName} is a foreign key PermName from the table PERM which is assigned to the role.

Suppose the permission Approval is assigned to role TELLER and the permission Funding to role MANAGER, Table 4 expresses the permission-role relationship.

<table>
<thead>
<tr>
<th>RoleName</th>
<th>PermName</th>
<th>Admin.role</th>
<th>Role Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>MANAGER</td>
<td>Funding</td>
<td>BankSO</td>
<td>[Bank, MANAGER]</td>
</tr>
<tr>
<td>TELLER</td>
<td>Approval</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4: ROLE-PERM

Table 6: Example of Can-revokep

Based on these relations, we describe the Authorization granting algorithm and revocation algorithms in this paper.

The paper is organized as follows. We recall the relational algebra-based authorization granting and revocation algorithms developed in our previous work. The extensions of the algorithms are described in section 3. Comparisons to related work are discussed in section 4 and the conclusions are in section 5.
2 Authorization granting and revocation algorithms for PRA

We recall granting and revocation algorithms for PRA based on relational algebra in this section. Details can be found from [Wang et al. 2003ADC].

The notion of a Prerequisite conditionp, Can-assignp and Can-revokep mentioned in the paper is a key part in the processes of permission role assignment. The Prerequisite conditionp is used to test whether or not permission can be assigned to roles while the Can-assignp is used to verify what role range's permissions an administrator can assign.

For a given set of roles \( R \) let \( CR \) denote all possible prerequisite conditions that can be formed using the roles in \( R \). Not every administrator can assign permission to a role. The relation of Can-assignp \( \subseteq AR \times CR \times 2^R \) provides what permissions can be assigned by administrators with prerequisite conditions, where \( AR \) is a set of administrative roles.

Permission-role assignment (PA) is authorized by Can-assignp relation. Table 5 shows the Can-assignp relations with the prerequisite conditions in the example.

<table>
<thead>
<tr>
<th>Admin.role</th>
<th>Prereq.ConditionP</th>
<th>Role Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>BankSO</td>
<td>BANK &amp; TELLER &amp; AUDITOR</td>
<td>[ACCOUNT_REP, ACCOUNT_REP]</td>
</tr>
<tr>
<td>BankSO</td>
<td>BANK &amp; TELLER &amp; ACCOUNT_REP</td>
<td>[AUDITOR, AUDITOR]</td>
</tr>
<tr>
<td>BankSO</td>
<td>BANK &amp; AUDITOR &amp; ACCOUNT_REP</td>
<td>[TELLER, TELLER]</td>
</tr>
</tbody>
</table>

Table 5: Can-assignp relation in Figure 2

Supposing an administrator role ADrole wants to assign a permission \( p_j \) to a role \( r \) with a set of permissions \( P \). \( P^* \) is an extension of \( P \), \( P^* = \{ p \mid p \in P \} \cup \{ p \mid \exists r' \in R, r' < r, (p, r') \in PA \} \). There are two major steps in the following permission granting algorithm. The first step is to check whether the ADrole can assign the permission \( p_j \) to \( r \) or not. The set of Prerequisite conditionp associated with ADrole can be obtained from the table Can-assignp while the set of roles associated with permission \( p_j \) is obtained from the table ROLE-PERM. The ADrole can build the membership of permission \( p_j \) and role \( r \) only if there is a role in the both sets. This means permission \( p_j \) satisfies prerequisite condition. The second step is to determine whether the permission \( p_j \) conflicts with the permissions of \( r \) or not, in other words, whether \( p_j \) conflicts with permission set \( P^* \) or not. The set of conflicting permissions of \( p_j \) can be retrieved from table PERM. Permission \( p_j \) is conflicting with role \( r \) if the intersection of the set and \( P^* \) is not empty.

There are related subtleties that arise in RBAC concerning the interaction between granting and revocation of permission-role membership. A relation Can-revokep \( \subseteq AR \times 2^R \) provides which permissions in what role range can be revoked. Table 6 gives an example of the Can-revokep relation. We have two revocation algorithms, one is a weak revocation algorithm that is for explicit member of a role only, the other one is a strong revocation algorithm that is used to delete explicit memberships between permissions and roles as well as implicit memberships.
The meaning of \( \text{Can-revokep(BankSO, } \{\text{Bank, MANAGER}\}) \) in Table 6 is that a member of the administrative role BankSO can revoke the membership of a permission from any role in \{Bank, MANAGER\}.

Two algorithms were developed for revocation of a permission \( p_j \) from a role \( r \) by an administrative role ADrole. In the weak revocation approach, only whether or not permission \( p_j \) is an explicit member of role \( r \) needs to be determined. Operation of the weak revocation has no effect when the permission \( p_j \) is not an explicit member of the role \( r \). The role set associated with \( p_j \) is gained from the relation of ROLE-PERM while the role set of role range with ADrole is obtained from the relation Can-revokep. The permission \( p_j \) can be revoked if the intersection of these two role sets is not empty.

A role still owns a permission of a system, which has been weakly revoked, if the role is senior to another role associated with the permission. To solve the authorization revocation problem, we need strong revocation, which requires revocation of both explicit and implicit membership. Strong revocation of a permission's membership in role \( r \) requires that the permission be removed not only from explicit membership in \( r \), but also from explicit and implicit membership in all roles junior to \( r \). Strong revocation therefore has a cascading effect up-wards in the role hierarchy. The first step in the strong revocation algorithm is to test whether \( p_j \) is in \( P \) or not. If the test is negative, that means \( p_j \) is neither an explicit member nor an implicit member of the role \( r \). When this case occurs, the strong revocation has no effect for the role. Otherwise, \( p_j \) is either an explicit member or an implicit member of \( r \). In this step, the membership of \( p_j \) is revoked from \( r \) if \( p_j \in P \); then the role set of roles that are junior to role \( r \) can be retrieved from the relation SEN-JUN. For all roles in both the set and \( P \), the relationships between these roles and permission \( p_j \) are revoked.

3 Extensions of the algorithms with mobility of permissions

Similar to the mobility of user-role relationship, permissions can also be assigned to roles as mobile and immobile members [Wang et al. ACSC2005]. There are four kinds of permission-role membership for a given role \( x \) [Sandhu and Munnwer 1999, Wang et al. Special].

1: Explicit Mobile Member EMPx: \( \text{EMPx} = \{p, (p, Mx) \in PA\} \)
2: Explicit Immobile Member EIMPx: \( \text{EIMPx} = \{p, (p, IMx) \in PA\} \)
3: Implicit Mobile Member ImMPx: \( \text{ImMPx} = \{p, \exists x' < x, (p, Mx') \in PA\} \)
4: Implicit Immobile Member ImIMPx: \( \text{ImIMPx} = \{p, \exists x' < x, (p, IMx') \in PA\} \)

A prerequisite condition \( PM \) is evaluated for a permission \( p \) by interpreting role \( x \) to be true if

\[
p \in \text{EMx} \lor (p \in \text{ImMx} \land p \notin \text{EIMx})
\]

and \( x \) to be true if

\[
p \notin \text{EMx} \land p \notin \text{EIMx} \land p \notin \text{ImMx} \land p \notin \text{ImIMx}
\]
In other words, \( x \) denotes mobile membership (explicit or implicit) and \( \overline{x} \) denotes absence of any kind of membership.

For a given set of roles \( R \) let \( CR \) denote all possible prerequisite conditions with mobility of permission-role relationship that can be formed using the roles in \( R \). Not every administrator can assign a role to a user. The following relations provide what permissions an administrator can assign as mobile members or immobile members with prerequisite conditions.

*Can-assignp-M* is a relation of \( AR \times CR \times 2^R \), which is used for permission-role assignments with mobile members; where \( AR \) is a set of administrative roles. Permission-role assignments with immobile members are authorized by the relation \( Can-assignp-IM \subseteq AR \times CR \times 2^R \).

Permission-role assignment (PA) is authorized by *Can-assignp-M* and *Can-assignp-IM* relations. Table 7 and Table 8 shows the *Can-assignp-M* and *Can-assignp-IM* relations with the prerequisite conditions in the bank example.

<table>
<thead>
<tr>
<th>Admin.role</th>
<th>Prereq.ConditionPM</th>
<th>Role Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>BankSO</td>
<td>BANK</td>
<td>[BANK, BANK]</td>
</tr>
<tr>
<td>BankSO</td>
<td>BANK \land TELLER</td>
<td>[AUDITOR, AUDITOR]</td>
</tr>
<tr>
<td>BankSO</td>
<td>BANK \land AUDITOR</td>
<td>[TELLER, TELLER]</td>
</tr>
</tbody>
</table>

*Table 7: Can-assignp-M in the example*

<table>
<thead>
<tr>
<th>Admin.role</th>
<th>Prereq.ConditionPM</th>
<th>Role Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>BankSO</td>
<td>BANK</td>
<td>[BANK, BANK]</td>
</tr>
<tr>
<td>BankSO</td>
<td>BANK \land TELLER</td>
<td>[AUDITOR, AUDITOR]</td>
</tr>
<tr>
<td>BankSO</td>
<td>BANK \land AUDITOR</td>
<td>[TELLER, TELLER]</td>
</tr>
</tbody>
</table>

*Table 8: Can-assignp-IM in the example*

The meaning of *Can-assignp-M*(BankSO, BANK, TELLER, AUDITOR) is that a member of the administrative role BankSO can assign a permission whose current membership satisfies the prerequisite condition BANK to be a mobile member of roles TELLER and AUDITOR.

Supposing an administrator role \( ADrole \) wants to assign a permission \( p_j \) to role \( r \) with a set of permissions \( P \) which has mobile and immobile memberships with \( r \). The \( p_j \) has mobile or immobile membership with \( r \) if \( ADrole \) can assign without conflicts. The following algorithm applies to both of mobile and immobile members. \( P^* \) is an extension of \( P \), \( P^* = \{ p | p \in P \} \cup \{ p | \exists r' \in R, r' < r, (p, r') \in PA \} \).

**Authorization granting algorithm**

GrantMP(ADrole, P, p_j)

Input: ADrole, role \( r \) and a permission \( p_j \).
Output: true if ADrole can assign the permission \( p_j \) to \( r \) with no conflicts; false otherwise.
Begin:
Step 1. /* Whether the ADrole can assign the permission $p_j$ to $r$ as mobile or immobile member or not */

Suppose $S_{M1} = S_M \cap R$ and $S_{IM1} = S_{IM} \cap R$ where

$S_M = \Pi \text{Prereq.ConditionPM} (\sigma \text{admin.role} = \text{ADrole (Can-assign-M)})$

$S_{IM} = \Pi \text{Prereq.ConditionPM} (\sigma \text{admin.role} = \text{ADrole (Can-assign-IM)})$

$R = \Pi \text{RoleName(\sigma PermName = \text{pj (ROLE-PERM)})}$

if $p_j$ is a mobile member of $r$ and $S_{M1} \neq \emptyset$,
then there exists role $r_1 \in S_{M1}$, such that

$r_1 \in \Pi \text{Role Range (\sigma (ADrole, r)(Can-assign-M)), and (pj, r \in PA)}$

/* $p_j$ is in the range to be assigned as a mobile member by ADrole in Can-assign-M */

If $p_j$ is an immobile member of $r$ and $S_{IM1} \neq \emptyset$,
then there exists role $r_i \in S_{IM1}$, such that

$r_i \in \Pi \text{Role Range (\sigma (ADrole, r)(Can-assign-IM)), and (pj, r \in PA)}$

go to step 2

/* $p_j$ is in the range to be assigned as an immobile member by ADrole in Can-assign-IM */

else return false and stop.
/* the admini.role has no right to assign the role $r_i$ as a mobile or immobile member to $R$ */

Step 2. /* whether the permission $p_j$ is conflicting with permissions of $r$ or not */

Let

$\text{ConfPermS} = \Pi \text{ConfPerm (\sigma PermName = \text{pj (PERM)})}$

/* It is the conflicting permission set of the permission $p_j$ */

if $\text{ConfPermS} \cap P^* \neq \emptyset$,
then
return false; /* $p_j$ is a conflicting permission with role $r$ */

else
return true. /* $p_j$ is not a conflicting permission with $r$ */

This algorithm provides a way to decide whether a permission can be assigned to a role as mobile or immobile member. For mobile member, $S_{M1}$ cannot be empty, and for immobile member, $S_{IM1}$ cannot be empty.
**Theorem 1**: The authorization granting algorithm can prevent conflicts when assigning a permission to a role with mobile and immobile memberships.

**Proof** Assuming an administrator role ADrole wants to assign a permission $p_j$ as a mobile member to a role which associates with a permission set $P$. Step 1 in the algorithm has checked whether the ADrole can assign $p_j$ as a mobile member to the role or not, and the second step has decided whether the permission $p_j$ conflicts with permissions in $P$ or not. Indeed, $p_j$ can be assigned to the role if for all $p_i \in P'$, $p_i$ is not in the conflicting permission set of $p_j$. Otherwise $p_j$ is a conflicting permission with $P'$.

Similar to the time complexity analysis in the last section, we have the following corollary.

**Corollary 1**: The authorization granting algorithm has time complexity $O(n^2)$ for the case of $n$ roles in a system.

Now we consider revocation of permission-role membership. Similar to Can-assignp-M and Can-assignp-IM relations in granting a permission to a role, there are Can-revokep-M and Can-revokep-IM relations. Relations $\text{Can-revokep-M} \subseteq AR \times CR \times 2^R$ and $\text{Can-revokep-IM} \subseteq AR \times CR \times 2^R$ show which role range of mobile membership and immobile membership administrative roles can revoke respectively, where $AR$ is a set of administrative roles.

The meaning of $\text{Can-revokep-M}$ (ShopSO, SHOP, [SHOP, MANAGER]) in Table 9 is that a member of the administrative role ShopSO can revoke mobile membership of a permission from any role in [SHOP, MANAGER] subject to the prerequisite condition SHOP. $\text{Can-revokep-IM}$ is similar with respect to immobile membership.

<table>
<thead>
<tr>
<th>Admin.role</th>
<th>Prereq.ConditionPRM</th>
<th>Role Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>ShopSO</td>
<td>SHOP</td>
<td>[SHOP, MANAGER]</td>
</tr>
</tbody>
</table>

Table 9: Can-revokep-M

The evaluation of a prerequisite condition for the revoke model is different from the grant model. In the revoke model a prerequisite condition $\text{PRM}$ is evaluated for a permission $p$ by interpreting roles $x$ to be true if

$p \in EM_x \lor p \in EIM_x \lor p \in IM_x \lor p \in ImIM_x$

and $x'$ to be true if

$p \notin EM_x \land p \notin EIM_x \land p \notin IM_x \land p \notin ImIM_x$

Due to role hierarchy, a role $x'$ has all permissions of role $x$ when $x'>x$. A user with two roles $\{x', x\}$ still has the permissions of $x$ if only to revoke $x$ from the user. To solve the authorization revocation problem along with mobility of permission, we need to revoke the explicit member of a permission first if a role is an explicit member, then revoke the implicit member.

Following are two algorithms for revocation of a permission $p_j$ as mobile or immobile members from a set of permission $P$ by an administrative role ADrole, where $P$ is a set of permissions which are assigned to a role $r$. The first one is the
weak revocation algorithm and the second is the strong revocation algorithm. The weak revocation only revokes explicit mobile and immobile memberships from \( r \) and does not revoke implicit mobile and immobile memberships but the strong revocation revokes both explicit and implicit mobile and immobile members. The structure of the weak revocation algorithm is shown in Figure 3.

**Weak revocation Algorithm**

Weak\_revokeMP(ADRrole, r, pj)
Input: ADRrole, a roles \( r \) and a permission \( pj \).
Output: true if ADRrole can weakly revoke role \( pj \) from \( r \), false otherwise.

Begin:
If \( pj \notin \{ p | (p, r) \in PA \} \),
return false; /* there is no effect with the operation of the weak revocation since the permission \( pj \) is not an explicit member of the role \( r \) */
else /* \( pj \) is an explicit member of \( r \), either an mobile member or immobile member */

Case 1: \( pj \) is an mobile member of \( r \),
Roles\( withp_j = \Pi \) RoleName (\( \sigma \) PermName = \( pj \) (ROLE-PERM))
/* Roles with permission \( pj \) */
PreM = \( \Pi \) Prereq.ConditionPRM(\( \sigma \) \{admin.role = ADRrole\}(Can-revokep-M))
/* Prerequisite condition with ADRole */
if \( RP = Roleswithp_j \cup PreM \neq \phi \)
RevokeRangeM = \( \Pi \) Role Range(\( \sigma \) admin.role = ADRrole(Can-revokep-M))
if \( RR = Roleswithp_j \cup RevokeRangeM \neq \phi \),
return, true.
/* the mobile member \( pj \) is revoked */
else
return false;
/* the mobile member \( pj \) cannot be revoked since the role \( r \) is not in the role range to be revoked */
else
return false and stop.
/* The \( pj \) does not satisfy the prerequisite conditions */

Case 2: \( pj \) is an immobile member of \( r \)
PreIM = \( \Pi \) Prereq.ConditionPRM(\( \sigma \) \{admin.role = ADRrole\}(Can-revokep-IM))
/* Prerequisite condition with ADRole */
if \( RPI = Roleswithp_j \cap PreIM \neq \phi \)
RevokeRangeIM = \( \Pi \) Role Range(\( \sigma \) admin.role = ADRrole(Can-revokep-IM))
if \( RRI = Roleswithp_j \cap RevokeRangeIM \neq \phi \),
return true,
The weak revocation algorithm can be used to check whether an administrator can weakly revoke mobile and immobile memberships from roles or not. We have the following result with the weak revocation algorithm.

\[ \text{Weak_revoke}(\text{ADrole}, r, p_j) \]

\[ \text{if the permission is an explicit member of role r and the ADrole has the right to revoke } p_j \text{ from the Can-revoke-M and Can-revoke-IM relations.} \]

\[ \text{A role still owns a permission of a system, which has been weakly revoked, if the role is senior to another role associated with the permission. To solve the} \]

authorization revocation problem, we need strong revocation, which requires
revocation of both explicit-implicit membership and mobile-immobile memberships.
Strong revocation of a permission's membership in role $r$ requires that the permission
be removed not only from explicit mobile and immobile membership in $r$, but also
from explicit, and implicit mobile and immobile membership in all roles junior to $r$.

**Strong revocation algorithm**

```plaintext
Strong_revoke(ADrole, r, pj)
Input: ADrole, a role $r$ and a permission $pj$.
Output: true, if it can strong revoke the permission $pj$ from $r$; false otherwise.
Begin:
If $pj \not\in P^*$,
return false;
/* there is no effect of the strong revocation since the permission is not an explicit
and implicit member of the role $r$ */
else,
Case 1. if $pj \in P$, do Weak_revoke(ADrole, r, pj);
    /* $pj$ is weakly revoked from $r$ as mobile or immobile */;
Case 2. Suppose $\text{Jun} = \prod \text{Junior (} \sigma \{\text{Senior}=r\} \{\text{SEN-JUN}\})$,
for all $y \in \text{Jun} \cap P$ $\smallsetminus R$,
    if $pj \subseteq R$ is a mobile member of the role $y$, do
    Weak_revoke(ADrole, y, pj) as $pj \in EMy$;
    if $pj \subseteq R$ is an immobile member of the role $y$, do
    Weak_revoke(ADrole, y, pj) as $pj \in EIMy$;
    /* the permission $pj$ is weakly revoked from all such $y \in \text{Jun} \cap P$ */.
If all the weak revocation s are successful,
return true;
otherwise,
return false.
/* if one weak revocation cannot finish*/

We have the following consequence.

**Theorem 3:** The explicit mobile and immobile and implicit mobile and immobile
members of role $pj$ are revoked from a role by the Strong revocation algorithm
$\text{Strong_revoke}(ADrole, r, pj)$ if the ADrole has the right to revoke $pj$ from the
$\text{Can-revokep-M}$ and $\text{Can-revokep-IM}$ relations.

**Corollary 2:** The authorization revocation problem is solved by the Weak
revocation algorithm and Strong revocation algorithm.
4 Related work

There are several other related works on the security of relational databases [Chu and Tseng, 2008, Yang et al. 2008, Osborn et al. 1996]. The interaction between RBAC and relational databases are presented in [Osborn et al. 1996]. Two experiments are described. One is a role-based front end to a relational database with discretionary access control. The other is a role graph to show the roles in a standard relational database. Some relational concepts like roles, users and permissions are provided. Our model also supports such concepts even though it has a large variety. However, the main difference between our algorithms and the scheme in [Osborn et al. 1996] is that we focus on the solutions of the conflicts of roles and permissions, and the latter focuses on the correlation of RBAC with discretionary access controls. Their work discusses the relationship between roles and discretionary access controls, they do not address the allocation of permissions to roles without conflicts. In our work, we developed detailed algorithms for allocating roles and permissions and checking their conflicts.

An architecture for the integration of tutoring approaches and process scaffolds into existing collaborative applications have been developed by Harrer et al. [2008]. The architecture allows to combine existing research on explicit representations of collaborative learning processes with the availability of existing and tested collaborative learning environments, but also to control the learning environments and thus enables adaptation of the tools to the current state of the learning process. Their work is different from the work in this paper. We focus on the possible conflicts in the permission assignments with role-based access model while they developed an architecture for adoption existing tools.

An oracle implementation for permission-role assignment has been proposed in [Sandhu and Bhamidipati1998]. In [Sandhu and Bhamidipati1998], the difference between permission-role assignment and Oracle database management system was analyzed. Furthermore, through prerequisite conditions, the paper has demonstrated how to use Oracle stored procedures for implementation. However, the work in this chapter substantially differs from that proposal. Differences are due to the consistency problem that arises in [Sandhu and Bhamidipati1998]:

It is very difficult to keep the consistency by reflecting security requirements between global network objects and local network objects if there are hundreds of roles and thousands of users in a system.

This problem is completely overcome in our algorithms because the algorithms focus on the conflicts between roles and permissions. The authorization granting algorithms are used to find conflicts and prevent some secret information from being derived while the strong revocation algorithms are used to check whether a role still has permissions of another role.

5 Conclusions

This paper has provided new authorization allocation algorithms for mobility of permission-role assignments that are based on relational algebra operations. They are the authorization granting algorithm, weak revocation algorithm, and strong
revocation algorithm. The algorithms can automatically check conflicts when granting more than one permission to a role in a system. They can prevent users associated with roles from accessing unauthorized use of facilities when the permissions of the roles are changed within the organization and demand the modification of security rights. The permissions can be allocated without compromising the security in RBAC and provide secure management for systems. For a better understanding, a bank example is adopted in the paper to explain the ideas of mobile members, immobile members, roles and users. Finally, we have discussed the related work in this area.

The future work will be the development of a system management with XML which involves the role-based group delegation subsystem.

Acknowledgements

The authors would like to thank the reviewers for their constructive suggestions and comments on the paper.

References


