Role-based administration of user-role assignment: The URA97 model and its Oracle implementation

Ravi Sandhu and Venkata Bhamidipati

Laboratory for Information Security Technology, ISSE Department, Mail Stop 4A4, George Mason University, Fairfax, VA 22033, USA
E-mail: sandhu@isse.gmu.edu

In role-based access control (RBAC) permissions are associated with roles, and users are made members of appropriate roles thereby acquiring the roles' permissions. The principal motivation behind RBAC is to simplify administration. An appealing possibility is to use RBAC itself to manage RBAC, to further provide administrative convenience. In this paper we investigate one aspect of RBAC administration concerning assignment of users to roles. We define a role-based administrative model, called URA97 (User-Role Assignment '97), for this purpose and describe its implementation in the Oracle database management system. Although our model is quite different from that built into Oracle, we demonstrate how to use Oracle stored procedures to implement it.

1. Introduction

Role-based access control (RBAC) has recently received considerable attention as a promising alternative to traditional discretionary and mandatory access controls (see, for example, [3,5,7,9,11,14,15,19–21]). In RBAC permissions are associated with roles, and users are made members of appropriate roles thereby acquiring the roles' permissions. This greatly simplifies management of permissions. Roles are created for the various job functions in an organization and users are assigned roles based on their responsibilities and qualifications. Users can be easily reassigned from one role to another. Roles can be granted new permissions as new applications and systems are incorporated, and permissions can be revoked from roles as needed. Role–role relationships can be established to lay out broad policy objectives.

In large enterprise-wide systems the number of roles can be in the hundreds or thousands, and users can be in the tens or hundreds of thousands, maybe even millions. Managing these roles and users, and their interrelationships is a formidable task that often is highly centralized and delegated to a small team of security administrators. Because the main advantage of RBAC is to facilitate administration of permissions, it is natural to ask how RBAC itself can be used to manage RBAC. We believe the use of RBAC for managing RBAC will be an important factor in the long-term success of RBAC. Decentralizing the details of RBAC administration without loosing central control over broad policy is a challenging goal for system designers and architects.

As we will see there are many components to RBAC. RBAC administration is therefore multi-faceted. In particular we can separate the issues of assigning users to roles, assigning permissions to roles, and assigning roles to roles to define a role hierarchy. These activities are all required to bring users and permissions together. However, in many cases, they are best done by different administrators (or administrative roles). Assigning permissions to roles is typically the province of application administrators. Thus a banking application can be implemented so credit and debit operations are assigned to a teller role, whereas approval of a loan is assigned to a managerial role. Assignment of actual individuals to the teller and managerial roles is a personnel management function. Assigning roles to roles has aspects of user-role assignment and role-permission assignment. Role–role relationships establish broad policy. Control of these relationships would typically be relatively centralized in the hands of a few security administrators.

In this paper we have focussed our attention exclusively on user-role assignment. We recognize that a comprehensive administrative model for RBAC must account for all three issues mentioned above, among others. However, user-role assignment is a particularly critical administrative activity. We feel it is the right one to focus on in taking our first step towards what will eventually evolve into a comprehensive administrative model.

In large systems user-role assignment is likely to be the first administrative function that is decentralized and delegated to the hands of users rather than security or system administrators. Assigning people to tasks is a normal managerial function. Assigning users to roles should be a natural part of assigning users to tasks. Empowering managers to do this routinely is one way of making security an enabling user-friendly technology rather than an intrusive and cumbersome nuisance as it all too often turns out to be. A manager who can assign a user to perform certain tasks should not have to ask someone else to enroll this user in appropriate roles. This should happen transparently and conveniently.

A user-role assignment model can also be used for managing user-group assignment and therefore has applicability beyond RBAC. The difference between roles and groups was hotly debated at the First ACM Workshop on RBAC [18]. Workshop attendees arrived at the consensus that a group is a named collection of users (and possibly other groups). Groups serve as a convenient shorthand notation for collections of users and that is the main motivation for introducing them. Roles are similar to groups in that they can serve as a shorthand for collections of users, but they go beyond groups in also serving as a shorthand for a collection of permissions. Assigning users to roles or users to groups are therefore essentially the same function. Assigning permissions to roles and permissions to groups, on the other hand, can have rather different characteristics. We need not get into this latter issue here since our focus is on user-role, or equivalently user-group, assignment.

In this paper we propose a model for the assignment of users to roles by means of administrative roles and permissions. For ease of reference we call this model as

URA97 (user-role assignment 1997). URA97 imposes strict limits on individual administrators regarding which users can be assigned to which roles. We then describe an implementation of URA97 in the Oracle database management system [4,12]. Oracle's administrative model for user-role assignment is very different from URA97. Nevertheless, we show how to use Oracle's stored procedures to implement URA97.

The principal contribution of URA97 is to provide a concrete example of what is meant by role-based administration of user-role assignment. Another central contribution of this paper is to demonstrate that an existing popular product, namely Oracle, provides the necessary base mechanisms and extensibility to program the behavior of URA97. URA97 is defined in context of the family of RBAC96 family of models due to Sandhu et al. [19]. However, it applies to almost any RBAC model, including [3,7,9,11,15], because user-role assignment is a basic administrative feature which will be required in any RBAC model.

The rest of this paper is organized as follows. We begin by reviewing the RBAC96 family of models in Section 2. In Section 3 we define the administrative model called URA97 for user-role assignment which itself is role-based. This is followed by a quick review of relevant RBAC features of Oracle in Section 4. Our implementation of URA97 in Oracle is described in Section 5. Section 6 concludes the paper.

2. The RBAC96 models

A general family of RBAC models called RBAC96 was defined by Sandhu et al. [19]. Figure 1 illustrates the most general model in this family. In Fig. 1 a single headed arrow indicates a one to one relationship and a double headed arrow indicates a many to many relationship. For simplicity we overload the term RBAC96 to refer to the family of models as well as its most general member.

The top half of the figure shows roles and permissions in the system that regulate access to data and resources. The bottom half shows administrative roles and administrative permissions. RBAC96 is based on five sets of entities called users (U), roles (R), and permissions (P), and their administrative counterparts called administrative roles (AR) and administrative permissions (AP). It is required that administrative roles and administrative permissions be respectively disjoint from the regular (i.e., non-administrative) roles and permissions. Moreover regular permissions can only be assigned to regular roles and administrative permissions can only be assigned to administrative roles.

Intuitively, a user is a human being or an autonomous agent, a role is a job function or job title within the organization with some associated semantics regarding the authority and responsibility conferred on a member of the role, and a permission is an approval of a particular mode of access to one or more objects in the system. Administrative permissions control operations which modify the components of RBAC, such as adding new users and roles and modifying the user assignment and permission assignment relations. Regular permissions on the other hand control operations

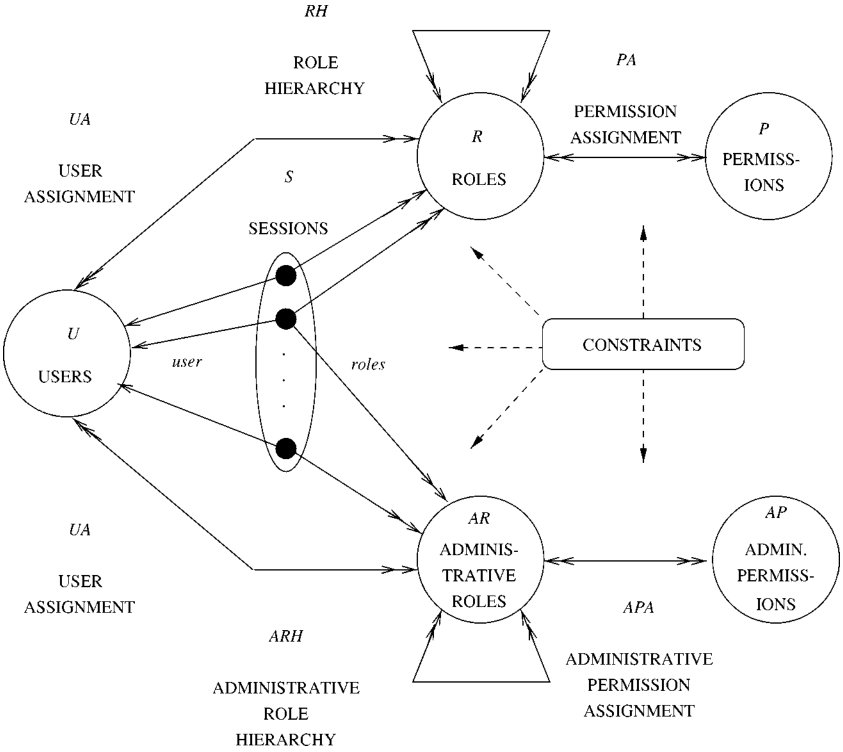

Fig. 1. The RBAC96 model.

on the data and resources and do not permit administrative operations. We loosely use the term role to include both regular and administrative roles, making this distinction precise whenever appropriate. Similarly for the term permission.

The user assignment (UA) and permission assignment (PA and APA) relations of Fig. 1 are many-to-many, as indicated by double-headed arrow. A user can be a member of many roles, and a role can have many users. Similarly, a role can have many permissions, and the same permission can be assigned to many roles. There is a partially ordered role hierarchy RH, also written as $\geqslant$, where $x \geqslant y$ signifies that role x inherits the permissions assigned to role y. Equivalently $x \geqslant y$ signifies a user who is a member of x is also implicitly a member of y. If $x > y$ we say x is senior to y or equivalently y is junior to x. Inheritance along the role hierarchy is transitive and multiple inheritance is allowed in partial orders. There is similarly a partially ordered administrative role hierarchy ARH.

Each session in Fig. 1 relates one user to possibly many roles. Intuitively, a user establishes a session and activates some subset of roles that he or she is a member of (directly or indirectly by means of the role hierarchy). The double-headed arrows from a session to R and AR indicate that multiple roles and administrative roles can be simultaneously activated. The permissions available to the user are the union of permissions from all roles activated in that session. Each session is associated with a single user, as indicated by the single-headed arrow from the session to U. This association remains constant for the life of a session. A user may have multiple sessions open at the same time, each in a different window on the workstation screen for instance. Each session may have a different combination of active roles. The concept of a session equates to the traditional notion of a subject in access control. A subject (or session) is a unit of access control, and a user may have multiple subjects (or sessions) with different permissions active at the same time.

Finally, Fig. 1 shows a collection of constraints. Constraints can apply to any of the preceding components. An example of constraints is mutually disjoint roles, such as purchasing manager and accounts payable manager, where the same user is not permitted to be a member of both roles. Another example is a limit on the maximum number of users that can be members of some role.

The following definition formalizes the above discussion.

Definition 1. The RBAC96 model has the following components:

- U is a set of users,
- R and AR are disjoint sets of roles and administrative roles, respectively,
- P and AP are disjoint sets of permissions and administrative permissions,
- $UA \subseteq U \times (R \cup AR)$, is a many-to-many user to role, and administrative role, assignment relation,
- $PA \subseteq P \times R$ and $APA \subseteq AP \times AR$, are, respectively, many-to-many permission to role assignment and administrative permission to administrative role assignment relations,
- $RH \subseteq R \times R$ and $ARH \subseteq AR \times AR$, are, respectively, partially ordered role and administrative role hierarchies (written as $\geqslant$ in infix notation),
- S is a set of sessions,
- $user : S \to U$, is a function mapping each session s_i to the single user $user(s_i)$ and is constant for the session's lifetime,
- $roles : S \to 2^{R \cup AR}$ is a function mapping each session s_i to a set[1] of roles and administrative roles

  $$roles(s_i) \subseteq \{r \mid (\exists r' \geqslant r)[(user(s_i), r') \in UA]\}$$

  (which can change within a single session) so that session s_i has the permissions

  $$\bigcup_{r \in roles(s_i)} \{p \mid (\exists r'' \leqslant r)[(p, r'') \in PA \cup APA]\},$$

  and

[1] Recall that 2^X is the set of all subsets of X, also called the power set of X.

- there is a collection of constraints stipulating which values of the various components enumerated above are allowed or forbidden.

Motivation and discussion about various design decisions made in developing this family of models is given in [17,19]. It is worth emphasizing that RBAC96 distinguishes roles and permissions from administrative roles and permissions, respectively, where the latter are used to manage the former. How are administrative permissions and roles managed in turn? One could consider a second level of administrative roles and permissions to manage the first level ones and so on. We feel such a progression of administration is unnecessary. Administration of administrative roles and permissions is under control of the chief security officer or delegated in part to administrative roles.

3. The URA97 administrative model

RBAC has many components as described in the previous section. Administration of RBAC involves control over each of these components including creation and deletion of roles, creation and deletion of permissions, assignment of permissions to roles and their removal, creation and deletion of users, assignment of users to roles and their removal, definition and maintenance of the role hierarchy, definition and maintenance of constraints and all of these in turn for administrative roles and permissions. A comprehensive administrative model would be quite complex and difficult to develop in a single step.

Fortunately administration of RBAC can be partitioned into several areas for which administrative models can be separately and independently developed to be later integrated. In particular we can separate the issues of assigning users to roles, assigning permissions to roles and defining the role hierarchy. In many cases, these activities would be best done by different administrators. Assigning permissions to roles is typically the province of application administrators. Thus a banking application can be implemented so credit and debit operations are assigned to a teller role, whereas approval of a loan is assigned to a managerial role. Assignment of actual individuals to the teller and managerial roles is a personnel management function. Design of the role hierarchy relates to design of the organizational structure and is the function of a chief security officer under guidance of a chief information officer.

In this paper our focus is exclusively on user-role assignment. As discussed in Section 1 this is likely to the first and most widely decentralized administrative task in RBAC. In the RBAC96 framework of Fig. 1 control of UA is vested in the administrative roles AR. For simplicity we limit our scope to assignment of users to regular roles. Assignment of users to administrative roles is centralized under the chief security officer. In general the chief security officer has complete control over all aspects of RBAC96.

In the rest of this section we develop a model called URA97 in which RBAC is used to manage user-role assignment. We define URA97 in two steps dealing with granting a user membership in a role and revoking a user's membership. URA97 is deliberately designed to have a very narrow scope. For example creation of users and roles is outside its scope. In spite of its simplicity URA97 is quite powerful and goes much beyond existing administrative models for user-role assignment, such as the one implemented in Oracle. It is also applicable beyond RBAC to user-group assignment.

3.1. URA97 grant model

In the simplest case user-role assignment can be completely centralized in a single chief security officer role. This is readily implemented in existing systems such as Oracle. However, this simple approach does not scale to large systems. Clearly it is desirable to decentralize user-role assignment to some degree.

In several systems, including Oracle, it is possible to designate a role, say, junior security officer (JSO) whose members have administrative control over one or more regular roles, say, A, B and C. Thus limited administrative authority is delegated to the JSO role. Unfortunately these systems typically allow the JSO role to have complete control over roles A, B and C. A member of JSO can not only add users to A, B and C but also delete users from these roles and add and delete permissions. Moreover, there is no control on which users can be added to the A, B and C roles by JSO members. Finally, JSO members are allowed to assign A, B and C as junior to any role in the existing hierarchy (so long as this does not introduce a cycle). All this is consistent with classical discretionary thinking whereby member of JSO are effectively designated as "owners" of the A, B and C roles, and therefore are free to do whatever they want to these roles.

In URA97 our goal is to impose restrictions on which users can be added to a role by whom, as well as to clearly separate the ability to add and remove users from other operations on the role. The notion of a prerequisite condition is a key part of URA97.

Definition 2. A prerequisite condition is a boolean expression using the usual $\wedge$ and $\vee$ operators on terms of the form x and $\overline{x}$ where x is a regular role (i.e., $x \in R$). A prerequisite condition is evaluated for a user u by interpreting x to be true if $(\exists x' \geqslant x)(u, x') \in UA$ and $\overline{x}$ to be true if $(\forall x' \geqslant x)(u, x') \notin UA$. For a given set of roles R let CR denotes all possible prerequisite conditions that can be formed using the roles in R.

In the trivial case a prerequisite condition can be a tautology which is always true. The simplest non-trivial case of a prerequisite condition is test for membership in a single role, in which situation that single role is called a prerequisite role.

User-role assignment is authorized in URA97 by the following relation.

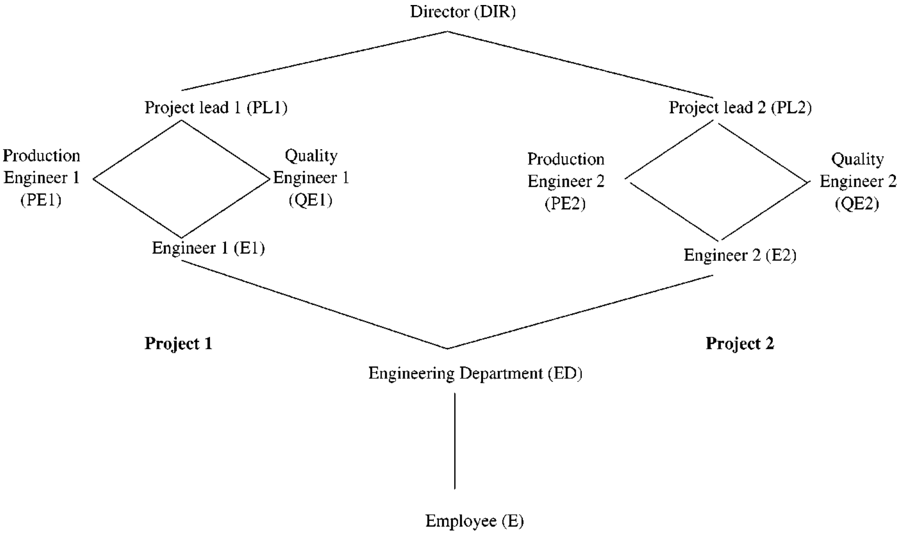

Fig. 2. An example role hierarchy.

Definition 3. The URA97 model controls user-role assignment by means of the relation *can-assign* $\subseteq AR \times CR \times 2^R$.

The meaning of *can-assign*$(x, y, \{a, b, c\})$ is that a member of the administrative role x (or a member of an administrative role that is senior to x) can assign a user whose current membership, or non-membership, in regular roles satisfies the prerequisite condition y to be a member of regular roles a, b or c.[2]

To appreciate the motivation behind the *can-assign* relation consider the role hierarchy of Fig. 2 and the administrative role hierarchy of Fig. 3. Figure 2 shows the regular roles that exist in a engineering department. There is a junior-most role E to which all employees in the organization belong. Within the engineering department there is a junior-most role ED and senior-most role DIR. In between there are roles for two projects within the department, project 1 on the left and project 2 on the right. Each project has a senior-most project lead role (PL1 and PL2) and a junior-most engineer role (E1 and E2). In between each project has two incomparable roles, production engineer (PE1 and PE2) and quality engineer (QE1 and QE2).

Figure 2 suffices for our purpose but this structure can, of course, be extended to dozens and even hundreds of projects within the engineering department. Moreover, each project could have a different structure for its roles. The example can also be extended to multiple departments with different structure and policies applied to each department.

[2] User-role assignment is subject to constraints, such as mutually exclusive roles or maximum cardinality, that may be imposed. The assignment will succeed if and only if it is authorized by *can-assign* and it satisfies all relevant constraints.

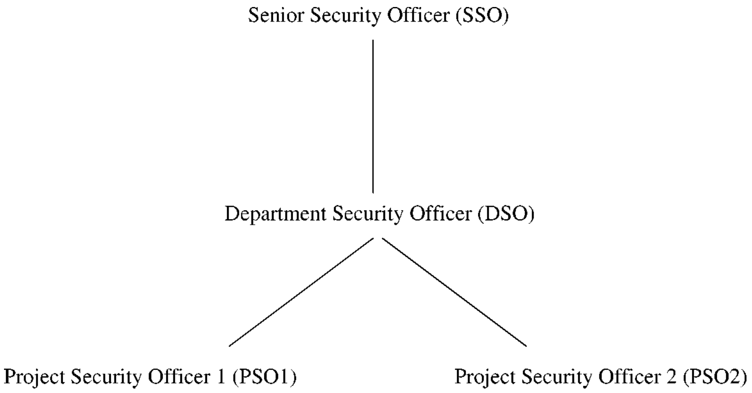
Senior Security Officer (SSO)
Department Security Officer (DSO)
Project Security Officer 1 (PSO1)
Project Security Officer 2 (PSO2)

Fig. 3. An example administrative role hierarchy.

Figure 3 shows the administrative role hierarchy which co-exists with Fig. 2. The senior-most role is the senior security officer (SSO). Our main interest is in the administrative roles junior to SSO. These consist of two project security officer roles (PSO1 and PSO2) and a department security officer (DSO) role with the relationships illustrated in the figure.

The role structure shown in Fig. 2 becomes a project oriented if the users are assigned to roles in a single project. If the users are assigned to roles from multiple projects then the structure is of matrix-form and If the users are assigned to same functional role in different projects then the structure is of functional oriented.

3.1.1. Prerequisite roles

For sake of illustration we define the *can-assign* relation shown in Table 1(a). This example has the simplest prerequisite condition of testing membership in a single role known as the prerequisite role.

The PSO1 role has partial responsibility over project 1 roles. Let Alice be a member of the PSO1 role and Bob a member of the ED role. Alice can assign Bob to any of the E1, PE1 and QE1 roles, but not to the PL1 role. Also if Charlie is not a member of the ED role, then Alice cannot assign him to any project 1 role. Hence, Alice has authority to enroll users in the E1, PE1 and QE1 roles provided these users are already members of ED. Note that if Alice assigns Bob to PE1 he does not need to be explicitly assigned to E1, since E1 permissions will be inherited via the role hierarchy. The PSO2 role is similar to PSO1 but with respect to project 2. The DSO role inherits the authority of PSO1 and PSO2 roles but can further add users who are members of ED to the PL1 and PL2 roles. The SSO role can add users who are in the E role to the ED role, as well as add users who are in the ED role to the DIR role. This ensures that even the SSO must first enroll a user in the ED role before that user is enrolled in a role senior to ED. This is a reasonable specification for *can-assign*. There are, of course, lots of other equally reasonable specifications in this context. This is a matter of policy decision and our model provides the necessary flexibility.

Table 1

Example of *can-assign* with prerequisite roles

(a) Subset notation

Administrative role	Prerequisite role	Role set
PSO1	ED	{E1, PE1, QE1}
PSO2	ED	{E2, PE2, QE2}
DSO	ED	{PL1, PL2}
SSO	E	{ED}
SSO	ED	{DIR}

(b) Range notation

Administrative role	Prerequisite role	Role range
PSO1	ED	[E1, PL1)
PSO2	ED	[E2, PL2)
DSO	ED	(ED, DIR)
SSO	E	[ED, ED]
SSO	ED	(ED, DIR]

In general, one would expect that the role being assigned is senior to the role previously required of the user. That is, if we have *can-assign*(a, b, C) then b is junior to all roles $c \in C$. We believe this will usually be the case, but we do not require it in the model. This allows URA97 to be applicable to situations where there is no role hierarchy or where such a constraint may not be appropriate.

The notation of Table 1(a) has benefited from the administrative role hierarchy. Thus for the DSO we have specified the role set as {PL1, PL2} and the other values are inherited from PSO1 and PSO2. Similarly for the SSO. Nevertheless explicit enumeration of the role set is unwieldy, particularly if we were to scale up to dozens or hundreds of projects in the department. Moreover, explicit enumeration is not resilient with respect to changes in the role hierarchy. Suppose a third project is introduced in the department, with roles E3, PE3, QE3, PL3 and PSO3 analogous to corresponding roles for projects 1 and 2. We can add the following row to Table 1(a).

Administrative role	Prerequisite role	Role set
PSO3	ED	{E3, PE3, QE3}

This is a reasonable change to require when the new project and its roles are introduced into the regular and administrative role hierarchies. However, we also need to modify the row for DSO in Table 1(b) to include PL3.

3.1.2. Range notation

Consider instead the range notation illustrated in Table 1(b). Table 1(b) shows the same role sets as Table 1(a) but defines these sets by identifying a range within the role hierarchy of Fig. 1(a) by means of the familiar closed and open interval notation.

Definition 4. Role sets are specified in the URA97 model by the notation below

$$
\begin{aligned}
&[x, y] = \{r \in R \mid x \geqslant r \wedge r \geqslant y\},\\
&(x, y] = \{r \in R \mid x > r \wedge r \geqslant y\},\\
&[x, y) = \{r \in R \mid x \geqslant r \wedge r > y\},\\
&(x, y) = \{r \in R \mid x > r \wedge r > y\}.
\end{aligned}
$$

This notation is resilient to modifications in the role hierarchy such as addition of a third project which requires addition of the following row to Table 1(b).

Administrative role	Prerequisite role	Role range
PSO3	ED	[E3, PL3)

No other change is required since the [ED, DIR) range specified for the DSO will automatically pick up PL3.

The range notation is, of course, not resilient to all changes in the role hierarchy. Deletion of one of the end points of a range can leave a dangling reference and an invalid range. Standard techniques for ensuring referential integrity would need to be applied when modifying the range hierarchy. Changes to role–role relationships could also cause a range to be drastically different from its original meaning. Nevertheless the range notation is much more convenient than explicit enumeration. There is also no loss of generality in adopting the range notation since every set of roles can be expressed as a union of disjoint ranges.

Strictly speaking the two specifications of Table 1 are not precisely identical. In Table 1(a) the DSO role is explicitly authorized to enroll users in PL1 and PL2, and the inherits the ability to enroll users in other project 1 and 2 roles from PSO1 and PSO2. On the other hand, in Table 1(b) the DSO role is explicitly authorized to enroll users in all project 1 and 2 roles. As it stands the net effect is the same. However, if modifications are made to the role hierarchy or to the PSO1 or PSO2 authorizations the effect can be different. The DSO authorization in Table 1(a) can be replaced by the following row to make Table 1(a) more nearly identical to Table 1(b).

Administrative role	Prerequisite role	Role set
DSO	ED	{E1, PE1, QE1, PL1, E2, PE2, QE2, PL2}

Now even if the PSO1 and PSO2 roles of Table 1(a) are modified respectively to the role sets {E1} and {E2}, the DSO role will still retain administrative authority over all project 1 and project 2 roles. Of course, explicit and implicit specifications will never behave exactly identically under *all* circumstances. For instance, introduction of a new project 3 will exhibit differences as discussed above. Conversely, the DSO authorization in Table 1(b) can be replaced by the following rows to make Table 1(b) more nearly identical to Table 1(a).

Administrative role	Prerequisite role	Role range
DSO	ED	[PL1, PL1]
DSO	ED	[PL2, PL2]

Table 2

Example of *can-assign* with prerequisite conditions

Administrative role	Prerequisite condition	Role range
PSO1	ED	[E1, E1]
PSO1	$\mathrm{ED} \wedge \overline{\mathrm{QE1}}$	[PE1, PE1]
PSO1	$\mathrm{ED} \wedge \overline{\mathrm{PE1}}$	[QE1, QE1]
PSO1	$\mathrm{PE1} \wedge \mathrm{QE1}$	[PL1, PL1]
PSO2	ED	[E2, E2]
PSO2	$\mathrm{ED} \wedge \overline{\mathrm{QE2}}$	[PE2, PE2]
PSO2	$\mathrm{ED} \wedge \overline{\mathrm{PE2}}$	[QE2, QE2]
PSO2	$\mathrm{PE2} \wedge \mathrm{QE2}$	[PL2, PL2]
DSO	ED	(ED, DIR)
SSO	E	[ED, ED]
SSO	ED	(ED, DIR]

There is an analogous situation with the SSO role in Table 1. Clearly, we must anticipate the impact of future changes when we specify the *can-assign* relation.

3.1.3. Prerequisite conditions

An example of *can-assign* which uses prerequisite conditions rather than prerequisite roles is shown in Table 2. The authorizations for PSO1 and PSO2 have been changed relative to Table 1.

Let us consider the PSO1 tuples (analysis for PSO2 is exactly similar). The first tuple authorizes PSO1 to assign users with prerequisite role ED into E1. The second one authorizes PSO1 to assign users with prerequisite condition $\mathrm{ED} \wedge \overline{\mathrm{QE1}}$ to PE1. Similarly, the third tuple authorizes PSO1 to assign users with prerequisite condition $\mathrm{ED} \wedge \overline{\mathrm{PE1}}$ to QE1. Taken together the second and third tuples authorize PSO1 to put a user who is a member of ED into one but not both of PE1 and QE1. This illustrates how mutually exclusive roles can be enforced by URA97. PE1 and QE1 are mutually exclusive with respect to the power of PSO1. However, for the DSO and SSO these are not mutually exclusive. Hence, the notion of mutual exclusion is a relative one in URA97. The fourth tuple authorizes PSO1 to put a user who is a member of both PE1 and QE1 into PL1. Of course, a user could have become a member of both PE1 and QE1 only by actions of a more powerful administrator than PSO1.

In RBAC users are made members of roles because of their job function or task assignment in the interest of the organization. Prerequisite conditions allow us to specify the requirements that need to be met before granting a role to a user. Generally an employee who is an engineer is not given access to HR or Pay roll information, he has access to only his personal information. A person belonging to Pay Roll department has access to information of all the employees or employees of the division to which he has been designated. In real life many large organizations have these kind of specifications and policies in place. Prerequisite conditions provide capabilities to enforce these policies.

3.2. URA97 revoke model

We now turn to consideration of the URA97 revoke model. The objective is to define a revoke model that is consistent with the philosophy of RBAC. This causes us to depart from classical discretionary approaches to revocation.

In the classical discretionary approach to revocation there are at least two issues that introduce complexity and subtlety [2,10]. Suppose Alice grants Bob some permission P. This is done at Alice's discretion because Alice is either the owner of the object to which P pertains or has been granted administrative authority on P by the actual owner. Alice can later revoke P from Bob. Now suppose Bob has received permission P from Alice and from Charlie. If Alice revokes her grant of P to Bob he should still continue to retain P because of Charlie's grant. A related issue is that of cascading revokes. Suppose Charlie's grant was in turn obtained from Alice, perhaps Bob's permission should end up being revoked by Alice's action. Or perhaps it should not, because Alice only revoked her direct grant to Bob but not the indirect one via Charlie which really occurred at Charlie's discretion. A considerable literature has developed examining the subtleties that arise, especially when hierarchical groups and negative permissions or denials are brought into play (see, for example, [1,6,8,13,16]).

The RBAC approach to authorization is quite different from the traditional discretionary one. In RBAC users are made members of roles because of their job function or task assignment in the interest of the organization. Granting of membership in a role is specifically not done at the grantor's whim. Suppose Alice makes Bob a member of a role X. In URA97 this happens because Alice is assigned suitable administrative authority over X via some administrative role Y and Bob is eligible for membership in X due to Bob's existing role memberships (and non-memberships) satisfying the prerequisite condition. Moreover, there are some organizational circumstances which cause Alice to grant Bob this membership. It is not merely being done at Alice's personal fancy. Now if at some later time Alice is removed from the administrative role Y there is clearly no reason to also remove Bob from X. A change in Alice's job function should not necessarily undo her previous grants. Presumably some other administrator, say Dorothy, will take over Alice's responsibility. Similarly, suppose Alice and Charlie both grant membership to Bob in X. At some later time Bob is reassigned to some other project and no longer needs to be a member of role X. It is not material whether Alice or Charlie or both or Dorothy revokes Bob's membership. Bob's membership in X is being revoked due to a change in organizational circumstances.

To summarize, in classical discretionary access control the source (direct or indirect) of a permission and the identity of the revoker is typically taken into account in interpreting the revoke operation.[3] These issues do not arise in the same way for

[3] This is true more in theory than practice, because many commercial products opt for a simpler semantics than implied by a strict owner-based discretionary viewpoint.

revocation of user-role assignment in RBAC. However, there are related subtleties that arise in RBAC concerning the interaction between granting and revocation of user-role membership and the role hierarchy. We will illustrate these in a moment.

We now introduce our notation for authorizing revocation.

Definition 5. The URA97 model controls user-role revocation by means of the relation *can-revoke* $\subseteq AR \times 2^R$.

The meaning of *can-revoke*(x, Y) is that a member of the administrative role x (or a member of an administrative role that is senior to x) can revoke membership of a user from any regular role $y \in Y$. Y is specified using the range notation of Definition 4. We say Y defines the *range of revocation*. The precise semantics of revocation in URA97 needs to be carefully defined to explain its interaction with the role hierarchy.

3.2.1. Weak revocation

In URA97 we define two notions of revocation called *weak* and *strong*. Recall that UA is the user assignment relation.

Definition 6. Let us say a user U is an *explicit member* of role x if $(U, x) \in UA$, and that U is an *implicit member* of role x if for some $x' > x$, $(U, x') \in UA$.

Note that a user can simultaneously be an explicit and implicit member of a role.

Weak revocation has an impact only on explicit membership. It has the straightforward meaning stated below.

Definition 7 (*Weak revocation algorithm*).

1. Let *u* have a session with administrative roles $A = \{a_1, a_2, \ldots, a_k\}$, and let *u* try to weakly revoke *v* from role x.
2. If *v* is not an explicit member of x this operation has no effect, otherwise there are two cases.
   (a) There exists a *can-revoke* tuple (b, Y) such that there exists $a_i \in A$, $a_i \geqslant b$ and $x \in Y$.
   In this case *v*'s explicit membership in x is revoked.
   (b) There does not exist a *can-revoke* tuple as identified above.
   In this case the weak revoke operation has no effect.

Let us consider the example of *can-revoke* shown in Table 3 and interpret it in context of the hierarchies of Figs 2 and 3. Let Alice be a member of PSO1, and let this be the only administrative role she has. Alice is authorized to weakly revoke membership of users from roles E1. Table 4(a) illustrates whether or not Alice can weakly revoke membership of a user from role E1. The effect of Alice's weak revocation of each of these users from E1 is shown in Table 4(b). There is no effect

Table 3

Example of *can-revoke*

Administrative role	Role range
PSO1	[E1, PL1)
PSO2	[E2, PL2)
DSO	(ED, DIR)
SSO	[ED, DIR]

Table 4

Example of strong revocation

(a) Prior to weak revocation

User	E1	PE1	QE1	PL1	DIR	Alice can revoke user from E1
Bob	Yes	No	No	No	No	Yes
Cathy	No	Yes	Yes	No	No	Yes
Dave	Yes	Yes	Yes	Yes	No	Yes
Eve	No	No	No	Yes	Yes	Yes

(b) After weak revocation

User	E1	PE1	QE1	PL1	DIR	Alice revoke user from E1
Bob	No	No	No	No	No	removed from E1
Cathy	No	Yes	Yes	No	No	no effect
Dave	No	Yes	Yes	Yes	Yes	removed from E1
Eve	No	No	No	Yes	Yes	no effect

of weak revocation on Cathy and Eve because they are not explicit members of E1 role. On the other hand Bob and Dave are removed from E1 role. Dave however still holds the E1 permissions because of his membership in senior roles.

3.2.2. *Strong revocation*

Strong revocation in URA97 requires revocation of both explicit and implicit membership. Strong revocation of U's membership in x requires that U be removed not only from explicit membership in x, but also from explicit (or implicit) membership in all roles senior to x. Strong revocation therefore has a cascading effect upwards in the role hierarchy. However, strong revocation in URA97 takes effect only if all implied revocations upward in the role hierarchy are within the revocation range of the administrative roles that are active in a session.

Let us consider the example of *can-revoke* shown in Table 3 and interpret it in context of the hierarchies of Figs 2 and 3. Let Alice be a member of PSO1, and let this be the only administrative role she has. Alice is authorized to strongly revoke membership of users from roles E1, PE1 and QE1. Table 5(a) illustrates whether or not Alice can strongly revoke membership of a user from role E1. The effect of Alice's strong revocation of each of these users from E1 is shown in Table 5(b). Alice is not allowed to strongly revoke Dave and Eve from E1 because they are members

Table 5

Example of strong revocation

(a) Prior to strong revocation

User	E1	PE1	QE1	PL1	DIR	Alice can revoke user from E1
Bob	Yes	Yes	No	No	No	Yes
Cathy	Yes	Yes	Yes	No	No	Yes
Dave	Yes	Yes	Yes	Yes	No	No
Eve	Yes	Yes	Yes	Yes	Yes	No

(b) After strong revocation

User	E1	PE1	QE1	PL1	DIR	Alice revoke user from E1
Bob	No	No	No	No	No	removed from E1, PE1
Cathy	No	No	No	No	No	removed from E1, PE1, QE1
Dave	Yes	Yes	Yes	Yes	Yes	no effect
Eve	Yes	Yes	Yes	Yes	Yes	no effect

of senior roles outside the scope of Alice's revoking authority. If Alice was assigned to the DSO role she could strongly revoke Dave from E1 but still would not be able to strongly revoke Eve's membership in E1. In order to strongly revoke Eve from E1, Alice needs to be in the SSO role.

The general rule is that strong revocation takes effect within the revocation range authorized for an administrative role. The precise statement of the strong revocation algorithm becomes complicated because of the administrative role hierarchy and the possible existence of several tuples in *can-revoke* which determine the outcome. In the example above Alice is allowed to strongly revoke Cathy from E1 because of *can-revoke*(PSO1, [E1, PL1)). We should have the same result if the instead of this single *can-revoke* range for PSO1 we have two ranges *can-revoke*(PSO1, [E1, PE1]) and *can-revoke*(PSO1, [E1, QE1]). Finally, because of the session concept in RBAC96 we must also pay attention to which roles Alice has turned on in the particular session. These considerations lead to the following algorithm for strong revocation.

Definition 8 (*Strong revocation algorithm*).

1. Let *u* have a session with administrative roles $A = \{a_1, a_2, \ldots, a_k\}$, and let *u* try to strongly revoke *v* from role x.
2. Find all *can-revoke* tuples $(b_1, X_1), (b_2, X_2), \ldots, (b_p, X_p)$ such that there exists $a_i \in A,\ a_i \geqslant b_j$ and $x \in X_j$ for $j = 1 \ldots p$.
3. Let $\widehat{X} = X_1 \cup X_2 \cup \cdots \cup X_p$ where the union is over the actual roles identified by the ranges $X_1, X_2, \ldots, X_p$.
4. There are two cases.

   (a) There exists $y \notin \widehat{X}$ such that *v* is a member of y and $y > x$.
   In this case *u*'s strong revocation has no effect.

(b) There does not exist a role y as identified above (therefore all senior roles to x to which *v* belongs are in $\widehat{X}$).
In this case *v*'s membership is revoked from role x and all roles senior to x.

In context of our example this algorithm will treat *can-revoke*(PSO1, [E1, PL1)) as equivalent to *can-revoke*(PSO1, [E1, PE1]) and *can-revoke*(PSO1, [E1, QE1]). It is similarly equivalent to *can-revoke*(PSO1, [E1, E1]), *can-revoke*(PSO1, [PE1, PE1]) and *can-revoke*(PSO1, [QE1, QE1]).

The strong revocation algorithm can also be expressed in terms of weak revoke by the following all-or-nothing transaction.

1. Let *u* have a session with administrative roles $A = \{a_1, a_2, \ldots, a_k\}$, and let *u* try to strongly revoke *v* from role x.
2. Find all roles $y \geqslant x$ and *v* is a member of y.
3. Weak revoke *v* from all such y as if *u* did this weak revoke.
4. If any of the weak revokes fail then *u*'s strong revoke has no effect otherwise all weak revokes succeed.

An alternate approach would be to do only those weak revokes that succeed and ignore the rest. We decided to go with a cleaner all-or-nothing semantics in URA97.

So far we have looked at the cascading of revocation upward in the role hierarchy. There is a downward cascading effect that also occurs. Consider Bob in our example who is a member of E1 and PE1. Suppose further that Bob is an explicit member of PE1 and thereby an implicit member of E1. What happens if Alice revokes Bob from PE1? If we remove (Bob, PE1) from the UA relation, Bob's implicit membership in E1 will also be removed. On the other hand if Bob is an explicit member of PE1 and also an explicit member of E1 then Alice's revocation of Bob from PE1 does not remove him from E1. The revoke operations we have defined in URA97 have the following effect.

Property 1. Implicit membership in a role a is dependent on explicit membership in some senior role $b > a$. Therefore when explicit membership of a user is revoked from b, implicit membership is also automatically revoked on junior role a unless there is some other senior role $c > a$ in which the user continues to be an explicit member. (This will require $b \not\geqslant c$.)

As we have discussed earlier, when a user's administrative roles are revoked that user's assignments and revocations remain in effect because these were done for organizational reasons and not at the user's whim. A related issue is what happens when the prerequisite condition which authorized Alice to assign Bob to a role gets changed. Say that Alice as PSO1 assigns Bob to PE1, as per the second PSO1 tuple of Table 2. Later somehow Bob is made a member of QE1, perhaps by a user in DSO or SSO role. This assignment negates the prerequisite condition which enabled

Alice to do her assignment. Bob's membership in PE1 will nevertheless continue. We feel this is the appropriate action. The prerequisite conditions of URA97 (and at other places in ARBAC97) are not invariants that hold for all time. They are simply enabling conditions at the moment that assignment is made.

As another example of the enabling but not invariant nature of prerequisite conditions consider the following in context of the *can-assign* relation of Table 1. Suppose Alice as PSO1 enrolls Bob into PE1 due to his prerequisite membership in ED. Later Charles as SSO revokes Bob from ED. Should Alice's assignment of Bob to PE1 be negated since the prerequisite condition has been negated? It depends on Charles' intention, which in turn depends on the organizational reason for this revocation. If Charles really needs to clear out Bob from the engineering department the correct course of action is a strong revocation of Bob from ED. If Charles does a weak revoke of Bob's explicit membership in ED he is leaving open the option that Bob will continue to participate in engineering department roles till such time as Bob is revoked from all of them (say by project security officers). This latter option can be useful in allowing Bob to gracefully leave the engineering department without an abrupt termination. In such cases it might be useful for Charles to be able to freeze Bob's membership in engineering department roles so that Bob cannot be assigned to new roles. This can be done using prerequisite conditions. A role called EF (for engineering frozen) can be defined and non-membership in EF required in the prerequisite condition of all *can-assign* tuples that authorize users to be assigned to engineering department roles.

Note that our examples of *can-assign* in Table 1(b) and *can-revoke* in Table 3 are complementary in that each administrative role has the same range for adding users and removing users from roles. Although this would be a common case we do not impose it as a requirement on our model.

We have defined URA97 so that the same revocation range applies for both strong and weak revocation. In principle we could define different ranges for these two operations. We do not feel this added complexity would be justified.

3.3. Summary of URA97

URA97 controls user-role assignment by means of the relation *can-assign* $\subseteq AR \times CR \times 2^R$. Role sets are specified using the range notation of Definition 4. Assignment has a simple behavior whereby *can-assign*(a, b, C) authorizes a session with an administrative role $a' \geqslant a$ to enroll any user who satisfies the prerequisite condition b into any role $c \in C$. The prerequisite condition is a boolean expression using the usual $\wedge$ and $\vee$ operators on terms of the form x and $\overline{x}$, respectively, denoting membership and non-membership regular role x.

Revocation is controlled in URA97 by the relation *can-revoke* $\subseteq AR \times 2^R$. Weak revocation applies only to explicit membership in a single role as per the algorithm of Definition 7. Strong revocation cascades upwards in the role hierarchy as per the algorithm of Definition 8. In both cases revocation cascades downwards as noted in Property 1.

4. Oracle RBAC and related features

The Oracle database management system [4,12] provides support for RBAC including support for hierarchical roles. However, Oracle does not directly support the URA97 model. In particular, Oracle has a strong discretionary flavor to its administrative model for user-role assignment and revocation. Also the Oracle revocation model is similar to our weak revoke and does not cascade revocation upwards in the role hierarchy like our strong revoke does. This is reasonable given Oracle's discretionary orientation. Nevertheless, we will see in the next section how it is possible to use Oracle's stored procedures to implement URA97. In this section we briefly review relevant features of Oracle access control.

4.1. Privileges

Oracle has two kinds of privileges, system privileges and object privileges. System privileges authorize actions on a particular type of object for example create table, create user, etc. There are over 60 distinct system privileges. Object privileges authorize actions on a specific object (table, view, procedure, package, etc.). Typical examples of object privileges are select rows from a table, delete rows, execute procedures, etc.

Who can grant or revoke privileges from users or roles? The answer depends on various issues such as whether it is a system or an object privilege, and whether the object is owned by the user, etc. In order to grant or revoke a system privilege the user should have the admin option on that privilege or the user should have GRANT_ANY_PRIVILEGE system privilege. In order to grant or revoke an object privilege a user should own that particular object or the user should have grant option on the object if it is owned by someone else.

4.2. Roles in Oracle

Oracle provides roles (from Oracle 7.0 onwards) for ease of management of privilege assignment. System and object privileges can be granted to a role. A role can be granted to any other role (circular granting is not allowed). Any role can be granted to any user in the database. A role can either be enabled or disabled during a session. This includes both explicit and implicit roles that a user is a member of. Enabling a role will implicitly enable all the roles granted to it directly or transitively. The system privileges related to role management are CREATE_ROLE, GRANT_ANY_ROLE, DROP_ROLE, and DROP_ANY_ROLE.

Information about privileges assigned to a role can be obtained from Oracle's built-in views ROLE_SYS_PRIVILEGES, ROLE_TAB_PRIVILEGES, and ROLE_ROLE_PRIVS. When a regular user performs query on these views these views only show information pertaining to the roles granted to that user. However, the Oracle internal user SYS will see information about all the roles through these views.

The view SESSION_ROLES provides information about roles that are enabled in a session. The view ROLE_ROLE_PRIVS shows information about which roles are directly assigned to another role. Roles inherited transitively are not shown. For example, if role C was granted to role B and role B to role A the ROLE_ROLE_PRIVS view will show that B has been granted to A and C to B, but will not show the implied transitive C to A grant.

4.3. Procedures, functions and packages

Oracle provides a programmatic approach to manipulate database information using procedural schema objects called PL/SQL (Procedural Language/SQL) program units. Procedures, functions and packages are different types of PL/SQL objects. PL/SQL extends the capabilities of SQL by providing some programming language features such as conditional statements, loops etc. Procedures are also referred to as stored procedures.

A procedure is a collection of instructions which can be grouped together and are performed on database objects to add, modify or delete database information. In order to create a procedure a user should have the CREATE_PROCEDURE system privilege. A procedure can be executed by a user who owns it or by a user who has execute privileges on it.

A stored procedure runs with the privileges of the user who owns it and not the user who is executing it. This feature gives great flexibility in enforcing security. For example suppose we want a user to perform some operations on a database but we do not want to grant privileges explicitly. Then one can write a procedure embedded with necessary operations, and grant execute privileges on the procedure to the user.[4]

Functions are very similar to procedures. The only difference between a function and a procedure is that a procedure call is a PL/SQL statement itself, while functions are called as part of an expression. A function always returns a value when it is called.

Packages are PL/SQL constructs that store related objects together. A package is essentially a named declarative section. It can contain procedures, functions, variables etc. A package consists of two parts, the specification part and body, stored separately in the data dictionary. The package specification, also known as package header, contains the information about the contents of the package. The package body contains code for the subprograms declared in the header.

5. Implementing URA97 in Oracle

To implement URA97 we define Oracle relations which encode the *can-assign* and *can-revoke* relations of URA97. The *can-assign* relation of URA97 is implemented

[4] The privileges that are referenced in a procedure should have been explicitly granted to the user who owns the procedure. Privileges obtained by the owner via a role cannot be referenced in a procedure.

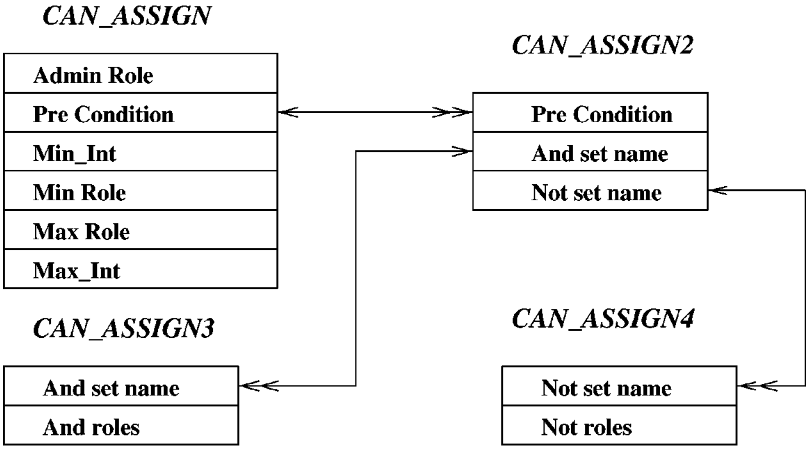

Fig. 4. Entity-relation diagram for *can-assign* relation.

in Oracle as per the entity-relation diagram of Fig. 4. We assume that the prerequisite condition is converted into disjunctive normal form using standard techniques. Disjunctive normal form has the following structure.

$$(\cdots \wedge \cdots \wedge \cdots \wedge \cdots) \vee (\cdots \wedge \cdots \wedge \cdots \wedge \cdots) \vee \cdots \vee (\cdots \wedge \cdots \wedge \cdots \wedge \cdots)$$

Each $\cdots$ is a positive literal x or a negated literal $\overline{x}$. Each group $(\cdots\wedge\cdots\wedge\cdots\wedge\cdots)$ is called a disjunct. For a given prerequisite condition *can-assign2* has a tuple for each disjunct. All positive literals of a single disjunct are in *can-assign3,* while negated literals are in *can-assign4*.

The four PSO1 tuples of Table 2 are represented by this scheme as shown in Table 6. The prerequisite conditions in this case all have a single disjunct. An example with multiple disjuncts is shown in Table 7.

The *can-revoke* relation of URA97 is represented by a single Oracle relation. For example, Table 3 is represented as shown in Table 8.

The *can-assign*, *can-assign2*, *can-assign3*, *can-assign4*, and *can-revoke* relations are owned by the DBA who also decides what their content should be. In addition we have three accompanying procedures and a package to support these. There is one procedure each for assigning a user to a role, doing a weak revoke of membership and doing a strong revoke of membership, respectively as follows.

- ASSIGN
- WEAK_REVOKE
- STRONG_REVOKE

Execute privilege on these procedures is given to all administrative roles. We achieve this by introducing a junior-most administrative role, say GSO (generic security officer), and assigning it the permission to execute these procedures.

Table 6

Oracle *can-assign* relations for PSO1 from Table 2

(a) *can-assign*

AR	PC	Min_Int	Min_Role	Max_Role	Max_Int
PSO1	C1	[	E1	E1	]
PSO1	C2	[	PE1	PE1	]
PSO1	C3	[	QE1	QE1	]
PSO1	C4	[	PL1	PL1	]
...	...	...	...	...	...

(b) *can-assign*2

PC	and_set_name	not_set_name
C1	ASET1	null
C2	ASET2	NSET2
C3	ASET3	NSET3
C4	ASET4	null
...	...	...

(c) *can-assign*3

and_set_name	and_roles
ASET1	ED
ASET2	ED
ASET3	ED
ASET4	PE1
ASET4	QE1
...	...

(d) *can-assign*4

not_set_name	not_roles
NSET2	QE1
NSET3	PE1
...	...

These relations and accompanying procedures and packages are owned by the DBA. Our implementation also maintains an audit relation which keeps a log of all attempted assignment and revoke operations and their outcome. The audit relation is also owned by the DBA.

Oracle does not provide convenient primitives for testing whether or not a user is an implicit member of a particular role. Testing explicit membership is straightforward since explicit membership is encoded as a tuple in Oracle's system relations. To test implicit membership, however, we need to chase the role hierarchy. Oracle also does not provide direct support for enumerating roles in a range set. We built a PL/SQL package to support these requirements and assist in writing our stored procedures, as discussed below.

In our implementation of URA97 a user invokes the stored procedure to grant or revoke a role from or to another user. The procedure calls are then as follows.

- ASSIGN(user, trole, arole)
- WEAK_REVOKE(user, trole, arole)
- STRONG_REVOKE(user, trole, arole)

Table 7

Oracle *can-assign* Relations for Prerequisite Condition $(A \wedge D \wedge \overline{E}) \vee (B \wedge \overline{D} \wedge \overline{F})$

(a) *can-assign*

AR	PC	Min_Int	Min_Role	Max_Role	Max_Int
SO1	C1	…	…	…	…
…	…	…	…	…	…

(b) *can-assign*2

PC	and_set_name	not_set_name
C1	ASET1	NSET1
C1	ASET2	NSET2
…	…	…

(c) *can-assign*3

and_set_name	and_roles
ASET1	A
ASET1	D
ASET2	B
…	…

(d) *can-assign*4

not_set_name	not_roles
NSET1	E
NSET2	F
NSET2	D
…	…

Table 8

Oracle *can-revoke* relation

AR	Min_Int	Min_Role	Max_Role	Max_Int
PSO1	[	E1	PL1	)
PSO2	[	E2	PL2	)
DSO	(	ED	DIR	)
SSO	[	ED	DIR	]

The parameters user and trole (target role) specify which user is to be added to trole, or to be weakly or strongly revoked from trole. The arole parameter specifies which administrative role should be applied (with respect to the user who is invoking the URA97 procedure). The procedure code will check whether or not the user who calls the procedure has turned on the arole.[5]

All the three procedures follow three basic steps.

1. If the user executing the procedure is an explicit or implicit member of arole then proceed to step 2, else stop execution and return an error message indicating this is not an authorized operation.

[5] It is relatively straightforward to specify a set of administrative roles instead of a single arole, and we plan to extend our implementation to do that.

2. The tuple(s) from *can-assign* (for assign procedure) or *can-revoke* (for revocation procedures) are obtained where AR role value equals or is junior to the arole parameter specified in the procedure call.
3. If trole is in the specified range for any one of the tuples selected in step 2, then assign or revoke the trole else return an appropriate error message.
   In case of ASSIGN also check whether the user being assigned to trole satisfies the prerequisite condition specified in the authorizing *can-assign* tuple or not.
   In case of STRONG_REVOKE the operation may still fail due to all-or-nothing semantics.

The implementation of steps 1 and 3 involves complex queries built on Oracle internal tables. These queries are performed dynamically at runtime. In order to check whether the user is a member of arole (in step 1) and whether the role is in the specified range for one of the relevant *can-assign* or *can-revoke* tuples (in step 3), we use Oracle CONNECT BY clause in our queries. By using CONNECT BY clause, one can traverse a tree structure corresponding to the role hierarchy in one direction. One can start from any point within the role hierarchy and traverse it towards junior or senior roles. But there is no control on the end point of the traversal. Specific branches or an individual node of the tree can be excluded by hard coding their values. Such hard coding is not appropriate for a general purpose stored procedure. In our implementation we overcome this problem by performing multiple queries and intersecting them to get the exact range. We specifically do not hard code any parameters in our queries.

In order to modularize our implementation we developed a package which performs the necessary checks involved in steps 1 and 3. All the procedures call this package to do the verification. The package contains several functions. Each one is designed to perform certain tasks, for example, we have a function called *user_has_admin_role*. This function takes the parameters from the procedure which has called it and returns the results to the calling procedure. There are other functions which determine the range for a given arole.

Our implementation is convenient for the DBA since the stored procedures and packages we provide are generic and can be reused by other databases. The DBA only needs to define the roles and administrative roles, and configure the *can-assign* and *can-revoke* relations. Our implementation is available in the public domain for other researchers and practitioners to experiment with.

6. Conclusion

In this paper we have developed the URA97 model for assigning users to roles and revoking users from roles. URA97 is defined in context of the RBAC96 model [19]. However, it should apply to almost any RBAC model, including [3,7,9,11,15], because user-role assignment is a basic administrative feature which will be required in any RBAC model.

Authorization to assign and revoke users to and from roles is controlled by administrative roles. The model requires users must previously satisfy a designated prerequisite condition (stated in terms of membership and non-membership in roles) before they can be enrolled via URA97 into additional roles. URA97 applies only to regular roles. Control of membership in administrative roles remains entirely in hands of the chief security officer. We have identified strong and weak revocation operations in URA97 and have defined their precise meaning.

The paper has also described an implementation of URA97 using Oracle stored procedures. Oracle's built in primitives are cumbersome to use for determining indirect membership in roles. We have implemented suitable functions and packages to enable this conveniently. These should be of use to other researchers and practitioners and are available in the public domain.

In future work we will extend URA97 to develop more comprehensive role-based administrative models encompassing administration of role-permission assignment and role–role relationships. We will also investigate how URA97 can be adapted for user-group assignment on platforms such as Unix and Windows NT (including simulation of group hierarchies which neither product provides). More generally we feel our work will inspire other researchers and developers to investigate administrative models in a systematic, scientific and experimental approach. We feel the security community has much to gain by pursuing such work.

Acknowledgment

This work is partially supported by the National Science Foundation and the National Security Agency.

References

[1] E. Bertino, P. Samarati and S. Jajodia, Authorizations in relational database management systems, in: *Proceedings of 1st ACM Conference on Computer and Communications Security*, Fairfax, VA, November 3–5, 1993, pp. 130–139.

[2] R. Fagin, On an authorization mechanism, *ACM Transactions on Database Systems* **3**(3) (1978), 310–319.

[3] D. Ferraiolo, J. Cugini and R. Kuhn, Role-based access control (RBAC): Features and motivations, in: *Proceedings of 11th Annual Computer Security Application Conference*, New Orleans, LA, December 11–15, 1995, pp. 241–248.

[4] S. Feuerstein, *Oracle PL/SQL Programming*, O'Reilly & Associates, Inc., 1995.

[5] D. Ferraiolo and R. Kuhn, Role-based access controls, in: *Proceedings of 15th NIST-NCSC National Computer Security Conference*, Baltimore, MD, October 13–16, 1992, pp. 554–563.

[6] E.B. Fernandez, J. Wu and M.H. Fernandez, User group structures in object-oriented database authorization, in: *Database Security VIII: Status and Prospects*, J. Biskup, M. Morgernstern and C. Landwehr, eds, North-Holland, Amsterdam, 1995.

[7] L. Guiri and P. Iglio, A formal model for role-based access control with constraints, in: *Proceedings of IEEE Computer Security Foundations Workshop 9*, Kenmare, Ireland, June 1996, pp. 136–145.

[8] E. Gudes, H. Song and E.B. Fernandez, Evaluation of negative, predicate, and instance-based authorization in object-oriented databases, in: *Database Security IV: Status and Prospects*, S. Jajodia and C.E. Landwehr, eds, North-Holland, Amsterdam, 1991, pp. 85–98.

[9] L. Guiri, A new model for role-based access control, in: *Proceedings of 11th Annual Computer Security Application Conference*, New Orleans, LA, December 11–15, 1995, pp. 249–255.

[10] P.P. Griffiths and B.W. Wade, An authorization mechanism for a relational database system, *ACM Transactions on Database Systems* **1**(3) (1976), 242–255.

[11] M.-Y. Hu, S.A. Demurjian and T.C. Ting, User-role based security in the ADAM object-oriented design and analyses environment, in: *Database Security VIII: Status and Prospects*, J. Biskup, M. Morgernstern and C. Landwehr, eds, North-Holland, Amsterdam, 1995.

[12] G. Koch and K. Loney, *Oracle The Complete Reference*, Oracle Press, 1995.

[13] T. Lunt, Access control policies: Some unanswered questions, in: *Proceedings of IEEE Computer Security Foundations Workshop II*, Franconia, NH, June 1988, pp. 227–245.

[14] I. Mohammed and D.M. Dilts, Design for dynamic user-role-based security, *Computers & Security* **13**(8) (1994), 661–671.

[15] M. Nyanchama and S. Osborn, Access rights administration in role-based security systems, in: *Database Security VIII: Status and Prospects*, J. Biskup, M. Morgernstern and C. Landwehr, eds, North-Holland, 1995.

[16] F. Rabitti, E. Bertino, W. Kim and D. Woelk, A model of authorization for next-generation database systems, *ACM Transactions on Database Systems* **16**(1) (1991).

[17] R. Sandhu, Rationale for the RBAC96 family of access control models, in: *Proceedings of the 1st ACM Workshop on Role-Based Access Control*, ACM, 1997.

[18] R. Sandhu, Roles versus groups, in: *Proceedings of the 1st ACM Workshop on Role-Based Access Control*, ACM, 1997.

[19] R.S. Sandhu, E.J. Coyne, H.L. Feinstein and C.E. Youman, Role-based access control models, *IEEE Computer* **29**(2) (1996), 38–47.

[20] S.H. von Solms and I. van der Merwe, The management of computer security profiles using a role-oriented approach, *Computers & Security* **13**(8) (1994), 673–680.

[21] C. Youman, E. Coyne and R. Sandhu, eds, *Proceedings of the 1st ACM Workshop on Role-Based Access Control*, Nov. 31–Dec. 1, 1995, ACM, 1997.