Proposed NIST Standard for Role-Based Access Control

DAVID F. FERRAIOLO
National Institute of Standards and Technology
RAVI SANDHU
SingleSign On. Net and George Mason University, sandhu@gmu.edu or www.list.gmu.edu
SERBAN GAVRILA
VDG Incorporated
and
D. RICHARD KUHN and RAMASWAMY CHANDRAMOULI
National Institute of Standards and Technology

In this article we propose a standard for role-based access control (RBAC). Although RBAC models have received broad support as a generalized approach to access control, and are well recognized for their many advantages in performing large-scale authorization management, no single authoritative definition of RBAC exists today. This lack of a widely accepted model results in uncertainty and confusion about RBAC's utility and meaning. The standard proposed here seeks to resolve this situation by unifying ideas from a base of frequently referenced RBAC models, commercial products, and research prototypes. It is intended to serve as a foundation for product development, evaluation, and procurement specification. Although RBAC continues to evolve as users, researchers, and vendors gain experience with its application, we feel the features and components proposed in this standard represent a fundamental and stable set of mechanisms that may be enhanced by developers in further meeting the needs of their customers. As such, this document does not attempt to standardize RBAC features beyond those that have achieved acceptance in the commercial marketplace and research community, but instead focuses on defining a fundamental and stable set of RBAC components. This standard is organized into the RBAC Reference Model and the RBAC System and Administrative Functional Specification. The reference model defines the scope of features that comprise the standard and provides a consistent vocabulary in support of the specification. The RBAC System and Administrative Functional Specification defines functional requirements for administrative operations and queries for the creation, maintenance, and review of RBAC sets and relations, as well as for specifying system level functionality in support of session attribute management and an access control decision process.

Categories and Subject Descriptors: C.2.0 [**Computer Communication Networks**]: General—*security and protection*

General Terms: Security, Standardization

Author's address: D. F. Ferraiolo, NIST, 100 Bureau Drive, Stop 8930, Gaithersburg, MD 20899-8930; e-mail: david.ferraiolo@nist.gov.
Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
© 2001 ACM 1094-9224/01/0800–0224 $5.00

Additional Key Words and Phrases: Role-based access control, security, access control, authorization management, standards

1. INTRODUCTION

In recent years, vendors have begun implementing role-based access control (RBAC) features in their database management, security management, and network operating system products, without general agreement as to what constitutes an appropriate set of RBAC features. Several RBAC models have been proposed[1] without any attempt at standardizing salient RBAC features. To identify RBAC features that exhibit true enterprise value and are practical to implement, the National Institute of Standards and Technology (NIST) has conducted and sponsored market analysis [Ferraiolo et al. 1993; Smith et al. 1996], developed prototype implementations [Ferraiolo et al. 1999], and sponsored external research [Feinstein 1996]. However NIST is not alone in recognizing the potential benefits of RBAC technology. Significant research has been performed at the university level in developing new RBAC models and applications, and researchers, vendors, and users have gathered on an annual basis to present papers and discuss issues related to RBAC in a formal workshop setting. As a result of these efforts much has been learned about RBAC and its practical implementation. Our overall understanding of RBAC has dramatically advanced, and a nascent consensus has begun.

A first effort at defining a consensus standard for RBAC was proposed at the 2000 ACM Workshop on Role-Based Access Control [Sandhu et al. 2000]. Published comments on this earlier document [Jaeger and Tidswell 2000] assisted in developing the reference model and functional specification proposed in this article. Panel session discussions at the 2000 ACM Workshop also contributed toward evolving the initial model into the following proposals. This standard represents a proposal and a public draft, and should not be considered a final version.

1.1 Background

The concept of roles has been used in software applications for at least 25 years, but it is only within the past decade that role-based access control has emerged as a full-fledged model as mature as traditional mandatory access control (MAC) and discretionary access control (DAC) concepts. The roots of RBAC include the use of groups in UNIX and other operating systems, privilege groupings in database management systems [Baldwin 1990; Thomsen 1991; Ting et al. 1992], and separation of duty concepts described in earlier papers [Clark and Wilson 1987; Sandhu 1988; Brewer and Nash 1989]. The modern concept of RBAC embodies all these notions in a single access control model in terms of roles and role hierarchies, role activation, and constraints on user/role membership and role set activation [Ferraiolo and Kuhn 1992]. These constructs are

[1]Please see Ferraiolo and Kuhn [1992], Nyanchama and Osborn [1994], Ferraiolo et al. [1995], Giuri and Iglio [1996], and Sandhu et al. [1996].

common to the early formal definitions of RBAC proposed by various authors [Ferraiolo et al. 1995; Sandhu et al. 1996; Nyanchama and Osborn 1994]. A comprehensive framework for RBAC models was defined by Sandhu et al. [1996], and expanded in subsequent publications [Sandhu 1998; Sandhu et al. 1999; Ahn and Sandhu 2000].

RBAC models have matured to the point where they are now being prescribed as a generalized approach to access control. For instance, recently RBAC was found to be "the most attractive solution for providing security features in multidomain digital government infrastructure," [Joshi et al. 2001b], and has shown its relevance in meeting the complex needs of Web-based applications [Joshi et al. 2001a]. RBAC models have been shown to be "policy-neutral" in the sense that by using role hierarchies and constraints, a wide range of security policies can be expressed [Osborn et al. 2000]. Security administration is also greatly simplified by the use of roles to organize access privileges. For example, if a user moves to a new function within the organization, the user can simply be assigned to the new role and removed from the old one, whereas in the absence of an RBAC model, the user's old permissions would have to be individually revoked, and new permissions would have to be granted. In addition, administration constraints may need to be enforced to prevent information misuse and prevent fraudulent activities. A typical authorization constraint, broadly relevant and well recognized, is separation of duties (SoD). Reducing the risk of fraud by not allowing any individual to have sufficient authority within the system to single-handedly perpetrate fraud is the intent of SoD. Such constraints can be easily expressed using an RBAC model through SoD constraints on roles, user-role assignments, and role-permission assignments. Furthermore, using constraints on the activation of user assigned roles, users can sign on with the least privilege set required for any access. In case of inadvertent errors, such least privilege assignments can contain damage.

Although existing RBAC models and implementations are relatively similar on fundamental RBAC concepts, they differ in significant details. Points of similarity and differences are not obvious, because many models use different terminology to describe the same concepts. Because RBAC is a relatively new technology and because products and models come from different commercial and academic backgrounds, little consensus exists on what to call the different parts. RBAC is also a rich and open-ended technology, which ranges from very simple at one extreme to fairly complex and sophisticated at the other. Treating RBAC as a single model is therefore unrealistic. A single model would either include or exclude too much, and would only represent one point along a spectrum of technologies and choices.

1.2 RBAC Standardization

To address these issues of scope and terminology, this proposed standard begins with an RBAC *Reference Model* defining a collection of model components. The RBAC reference model defines sets of basic RBAC elements (i.e., users, roles, permissions, operations, and objects) and relations as types and functions that are included in this standard. The RBAC reference model serves two

tools by guaranteeing interoperability and portability over a broad spectrum of products and administrative services.

The rest of this article is organized as follows. Section 2 gives an overview and a rationale for the four RBAC components that are included in this standard. These components are later used as a basis for modeling RBAC features and are used to provide a uniform and consistent set of functional definitions that are applied in the specification and packaging of the RBAC functional specification. Section 3, the RBAC reference model, provides a rigorous definition of these components as a collection of relations on sets of RBAC basic elements. Section 4 provides an overview of the RBAC functional specification in three areas: administrative operations, administrative review capabilities, and RBAC system functions. A complete and detailed requirement specification for each RBAC component is provided as an Appendix. Section 5 describes the method of packaging RBAC functional components into an environment-specific collection of RBAC administrative operations and queries and system functions.

2. COMPONENT OVERVIEW

This RBAC standard is organized into two main parts: the RBAC Reference Model and the RBAC Functional Specification. The RBAC Reference Model provides a rigorous definition of RBAC sets and relations. The reference model has two primary objectives: to define a common vocabulary of terms for use in consistently specifying requirements and to set the scope of the RBAC features included in the standard. The RBAC Functional Specification defines requirements over administrative operations for the creation and maintenance of RBAC element sets and relations; administrative review functions for performing administrative queries; and system functions for creating and managing RBAC attributes on user sessions and making access control decisions.

The RBAC model and functional specification are organized into four RBAC components, as described in the following sections. A rationale for each of these components is also provided. Readers relatively new to RBAC can skim this section and revisit it after reading the descriptions of the four components of the model in the following section.

2.1 Core RBAC

Core RBAC embodies the essential aspects of RBAC. The basic concept of RBAC is that users are assigned to roles, permissions are assigned to roles, and users acquire permissions by being members of roles. Core RBAC includes requirements that user-role and permission-role assignment can be many-to-many. Thus the same user can be assigned to many roles and a single role can have many users. Similarly, for permissions, a single permission can be assigned to many roles and a single role can be assigned to many permissions. Core RBAC includes requirements for user-role review whereby the roles assigned to a specific user can be determined as well as users assigned to a specific role. A similar requirement for permission-role review is imposed as an advanced review function. Core RBAC also includes the concept of user sessions, which allows selective activation and deactivation of roles. Finally, Core RBAC requires

that users be able to simultaneously exercise permissions of multiple roles. This precludes products that restrict users to activation of one role at a time.

Rationale. Core RBAC captures the features of traditional group-based access control as implemented in operating systems through the current generation. As such it is widely deployed and familiar technology. The features required of Core RBAC are essential for any form of RBAC. The main issue in defining Core RBAC is to determine which features to exclude. This proposed standard has deliberately kept a very minimal set of features in Core RBAC. In particular, these features accommodate traditional but robust group-based access control. Not every group-based mechanism qualifies because of the requirements given above. One of the features omitted as mandatory for Core RBAC is permission-role review. Although highly desirable, we recognize that many well-accepted RBAC systems do not provide this feature. The requirement that users can be assigned to and can simultaneously activate multiple roles could arguably be considered too strong for a core model. This requirement seems appropriate when we have a large number of diverse roles (say hundreds or thousands). It may not be appropriate for situations with few roles (say tens). For now we have chosen to define the core model as presented here. But we leave open the possibility of defining a smaller core model as the standard progresses.

2.2 Hierarchical RBAC

Hierarchical RBAC adds requirements for supporting role hierarchies. A hierarchy is mathematically a partial order defining a seniority relation between roles, whereby senior roles acquire the permissions of their juniors, and junior roles acquire the user membership of their seniors. This standard recognizes two types of role hierarchies.

—*General Hierarchical RBAC.* In this case, there is support for an arbitrary partial order to serve as the role hierarchy, to include the concept of multiple inheritance of permissions and user membership among roles.

—*Limited Hierarchical RBAC.* Some systems may impose restrictions on the role hierarchy. Most commonly, hierarchies are limited to simple structures such as trees or inverted trees.

Rationale. Roles can have overlapping capabilities; that is, users belonging to different roles may be assigned common permissions. Furthermore, within many organizations there are a number of general permissions that are performed by a large number of users. As such, it would prove inefficient and administratively cumbersome to specify repeatedly their general permission-role assignments. To improve efficiency and support organizational structure, RBAC models as well as commercial implementations include the concept of role hierarchies. Role hierarchies in the form of an arbitrary partial ordering are arguably the single most desirable feature in addition to core RBAC. This feature has often been mentioned in the literature [Ferraiolo et al. 1995; Sandhu et al. 1996; Moffett 1998] and has precedence in existing RBAC implementations. Justification for requiring the transitive, reflexive, and antisymmetric properties of a partial order has been extensively discussed in the literature

[Ferraiolo et al. 1995; Nyanchama and Osborn 1999; Sandhu et al. 1996]. There is a strong consensus on this issue. Nevertheless there are a number of products that support only restricted hierarchies, which provide substantially improved capabilities beyond Core RBAC.

2.3 Static Separation of Duty Relations

Separation of duty relations are used to enforce conflict of interest policies. Conflict of interest in a role-based system may arise as a result of a user gaining authorization for permissions associated with conflicting roles. One means of preventing this form of conflict of interest is though *static separation of duty* (SSD), that is, to enforce constraints on the assignment of users to roles. An example of such a static constraint is the requirement that two roles be mutually exclusive; for example, if one role requests expenditures and another approves them, the organization may prohibit the same user from being assigned to both roles. The SSD policy can be centrally specified and then uniformly imposed on specific roles. Because of the potential for inconsistencies with respect to static separation of duty relations and inheritance relations of a role hierarchy, we define SSD requirements both in the presence and absence of role hierarchies.

—*Static Separation of Duty*. SSD relations place constraints on the assignments of users to roles. Membership in one role may prevent the user from being a member of one or more other roles, depending on the SSD rules enforced.

—*Static Separation of Duty in the Presence of a Hierarchy*. This type of SSD relation works in the same way as basic SSD except that both inherited roles as well as directly assigned roles are considered when enforcing the constraints.

With respect to the constraints placed on the user-role assignments for defined sets of roles, we define SSD as a pair (*role set*, n) where no user is assigned to n or more roles from the role set. As such, we recognize a variety of SSD policies. For example, a user may not be assignable to *every* role in a specified role set, while a strong deployment of the same feature may restrict a user from being assigned to any combination of *two or more* roles in the role set.

Rationale. From a policy perspective, SSD relations provide a powerful means of enforcing conflict of interest and other separation rules over sets of RBAC elements. Static constraints generally place restrictions on administrative operations that have the potential to undermine higher-level organizational Separation of Duty policies.

Static constraints can take on a wide variety of forms. A common example is that of static separation of duty, which defines mutually disjoint user assignments with respect to sets of roles [Kuhn 1997; Giuri and Iglio 1996]. However, static constraints have been shown to be a powerful means of implementing a number of other important separation of duty policies. For example, Gligor et al. [1998], and Simon and Zurko [1997], [Ahn and Sandhu 1999, 2000; Jaeger 2000] have identified SSD relations to include constraints on users, operations, and objects as well as combinations thereof. Some authors [Ahn and Sandhu

1999, 2000; Jaeger 2000] have studied other forms of constraints recently, but so far consensus has not been developed. The static constraints defined in this standard are limited to those relations that place restrictions on sets of roles and in particular on their ability to form user-role assignment relations. Although formal RBAC models and RBAC policy specifications have grown well beyond these simple relations, we know of no commercial products that implement these advanced static separation of duty relations.

2.4 Dynamic Separation of Duty Relations

Dynamic separation of duty (DSD) relations, like SSD relations, limit the permissions that are available to a user. However DSD relations differ from SSD relations by the context in which these limitations are imposed. DSD requirements limit the availability of the permissions by placing constraints on the roles that can be activated within or across a user's sessions.

Similar to SSD relations, DSD relations define constraints as a pair (*role set*, n) where n is a natural number ≥ 2, with the property that no user session may activate n or more roles from the role set.

Rationale. DSD properties provide extended support for the principle of least privilege in that each user has different levels of permission at different times, depending on the task being performed. This ensures that permissions do not persist beyond the time that they are required for performance of duty. This aspect of least privilege is often referred to as *timely revocation of trust*. Dynamic revocation of permissions can be a complex issue without the facilities of dynamic separation of duty, and as such it has been generally ignored in the past for reasons of expediency.

SSD provides the capability to address potential conflict of interest issues at the time a user is assigned to a role. DSD allows a user to be authorized for roles that do not cause a conflict of interest when acted on independently, but which produce policy concerns when activated simultaneously. Although this separation of duty requirement could be achieved through the establishment of a static separation of duty relationship, DSD relationships generally provide the enterprise with greater efficiency and operational flexibility.

3. THE ROLE-BASED ACCESS CONTROL REFERENCE MODEL

The NIST RBAC model is defined in terms of four *model components*: Core RBAC, Hierarchical RBAC, Static Separation of Duty Relations, and Dynamic Separation of Duty Relations. Core RBAC defines a minimum collection of RBAC elements, element sets, and relations in order to completely achieve a role-based access control system. This includes user-role assignment and permission-role assignment relations, considered fundamental in any RBAC system. In addition, Core RBAC introduces the concept of role activation as part of a user's session within a computer system. Core RBAC is required in any RBAC system, but the other components are independent of each other and may be implemented separately.

The Hierarchical RBAC component adds relations for supporting role hierarchies. A hierarchy is mathematically a partial order defining a seniority

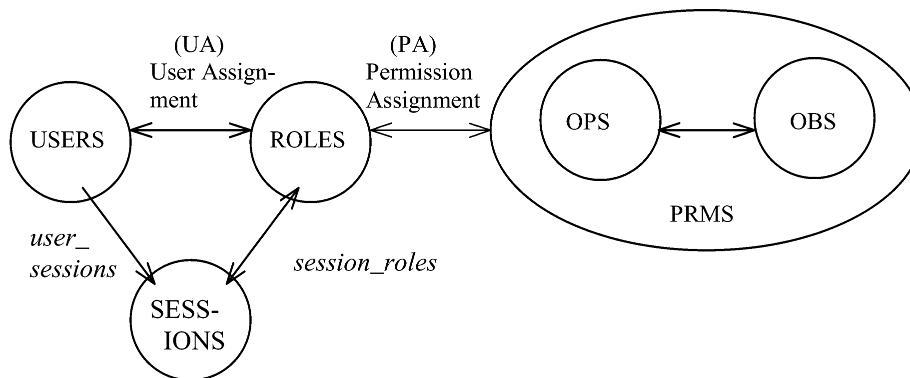

Fig. 1. Core RBAC.

relation between roles, whereby senior roles acquire the permissions of their juniors and junior roles acquire users of their seniors. In addition, Hierarchical RBAC goes beyond simple user- and permission-role assignment by introducing the concept of a role's set of authorized users and authorized permissions. A third model component, Static Separation of Duty Relations, adds exclusivity relations among roles with respect to user assignments. Because of the potential for inconsistencies with respect to Static Separation of Duty Relations and inheritance relations of a role hierarchy, the SSD relations model component defines relations in both the presence and absence of role hierarchies. The fourth model component, Dynamic Separation of Duty Relations, defines exclusivity relations with respect to roles that are activated as part of a user's session.

Each model component is defined by the subcomponents:

—a set of basic element sets;
—a set of RBAC relations involving those element sets (containing subsets of Cartesian products denoting valid assignments); and
—a set of mapping functions that yield instances of members from one element set for a given instance from another element set.

It is important to note that the RBAC reference model defines a taxonomy of RBAC features that can be composed into a number of feature packages. Rather then attempting to define a complete set of RBAC features, this model focuses on providing a standard set of terms for defining the most salient features as represented in existing models and implemented in commercial products.

3.1 Core RBAC

Core RBAC model element sets and relations are defined in Figure 1. Core RBAC includes sets of five basic data elements called users (USERS), roles (ROLES), objects (OBS), operations (OPS), and permissions (PRMS). The RBAC model as a whole is fundamentally defined in terms of individual users being assigned to roles and permissions being assigned to roles. As such, a role is a means for naming many-to-many relationships among individual users

and permissions. In addition, the Core RBAC model includes a set of sessions (SESSIONS) where each session is a mapping between a user and an activated subset of roles that are assigned to the user.

A *user* is defined as a human being. Although the concept of a user can be extended to include machines, networks, or intelligent autonomous agents, for simplicity reasons we limit a user to a person in this article. A *role* is a job function within the context of an organization with some associated semantics regarding the authority and responsibility conferred on the user assigned to the role. *Permission* is an approval to perform an operation on one or more RBAC protected objects.[2] An *operation* is an executable image of a program, which upon invocation executes some function for the user. The types of operations and objects that RBAC controls are dependent on the type of system in which they will be implemented. For example, within a file system, operations might include read, write, and execute; within a database management system, operations might include insert, delete, append, and update.

The purpose of any access control mechanism is to protect system resources. However, in applying RBAC to a computer system, we speak of protected objects. Consistent with earlier models of access control an *object* is an entity that contains or receives information. For a system that implements RBAC, the objects can represent information containers (e.g., files or directories in an operating system, and/or columns, rows, tables, and views within a database management system) or objects can represent exhaustible system resources, such as printers, disk space, and CPU cycles. The set of objects covered by RBAC includes all of the objects listed in the permissions that are assigned to roles.

Central to RBAC is the concept of role relations, around which a role is a semantic construct for formulating policy. Figure 1 illustrates *user assignment* (*UA*) *and permission assignment* (*PA*) relations. The arrows indicate a many-to-many relationship (e.g., a user can be assigned to one or more roles, and a role can be assigned to one or more users). This arrangement provides great flexibility and granularity of assignment of permissions to roles and users to roles. Without these conveniences there is an enhanced danger that a user may be granted more access to resources than is needed because of limited control over the type of access that can be associated with users and resources. Users may need to list directories and modify existing files, for example, without creating new files, or they may need to append records to a file without modifying existing records. Any increase in the flexibility of controlling access to resources also strengthens the application of the principle of least privilege.

Each session is a mapping of one user to possibly many roles, that is, a user establishes a session during which the user activates some subset of roles that he or she is assigned. Each session is associated with a single user and each user is associated with one or more sessions. The function *session_roles* gives

[2]Negative permissions have been discussed in the literature, but are not included in this model. The model leaves open the possibility of incorporating negative permissions in an RBAC implementation.

us the roles activated by the session and the function *user_sessions* gives us the set of sessions that are associated with a user. The permissions available to the user are the permissions assigned to the roles that are activated across all the user's sessions.

We summarize the above in the following definition.

Definition 1. Core RBAC.

—*USERS*, *ROLES*, *OPS*, *and OBS* (users, roles, operations, and objects, respectively).

—$UA \subseteq USERS \times ROLES$, a many-to-many mapping user-to-role assignment relation.

—*assigned_users*: $(r{:}ROLES) \rightarrow 2^{USERS}$, the mapping of role r onto a set of users. Formally: $assigned_users(r) = \{u \in USERS \mid (u, r) \in UA\}$.

—$PRMS = 2^{(OPS \times OBS)}$, the set of permissions.

—$PA \subseteq PRMS \times ROLES$, a many-to-many mapping permission-to-role assignment relation.

—$assigned_permissions(r{:}\ ROLES) \rightarrow 2^{PRMS}$, the mapping of role r onto a set of permissions. Formally: $assigned_permissions(r) = \{p \in PRMS \mid (p, r) \in PA\}$.

—$Ob(p{:}\ PRMS) \rightarrow \{\mathrm{op} \subseteq \mathrm{OPS}\}$, the permission-to-operation mapping, which gives the set of operations associated with permission p.

—$Ob(p{:}\ PRMS) \rightarrow \{\mathrm{ob} \subseteq \mathrm{OBS}\}$, the permission-to-object mapping, which gives the set of objects associated with permission p.

—*SESSIONS*, the set of sessions.

—$user_sessions\ (u{:}\ USERS) \rightarrow 2^{SESSIONS}$, the mapping of user u onto a set of sessions.

—$session_roles\ (s{:}\ SESSIONS) \rightarrow 2^{ROLES}$, the mapping of session s onto a set of roles. Formally: $session_roles\ (s_i) \subseteq \{r \in ROLES \mid (session_users\ (s_i), r) \in UA\}$.

—$avail_session_perms(s{:}SESSIONS) \rightarrow 2^{PRMS}$, the permissions available to a user in a session, $\bigcup_{r \in session_roles(s)} assigned_permissions(r)$.

We now define role hierarchies as inheritance relationships between roles.

3.2 Hierarchal RBAC

This model component introduces role hierarchies (RH) as indicated in Figure 2. Role hierarchies are commonly included as a key aspect of RBAC models[3] and are often included as part of RBAC product offerings [Chandramouli and Sandhu 1998]. Hierarchies are a natural means of structuring roles to reflect an organization's lines of authority and responsibility (see Figure 3).

Role hierarchies define an inheritance relation among roles. Inheritance has been described in terms of permissions [Nyanchama and Osborn 1999]; that

[3]Please see Ferraiolo et al. [1995], Sandhu et al. [1996], Nyanchama and Osborn [1999], and Moffett [1998].

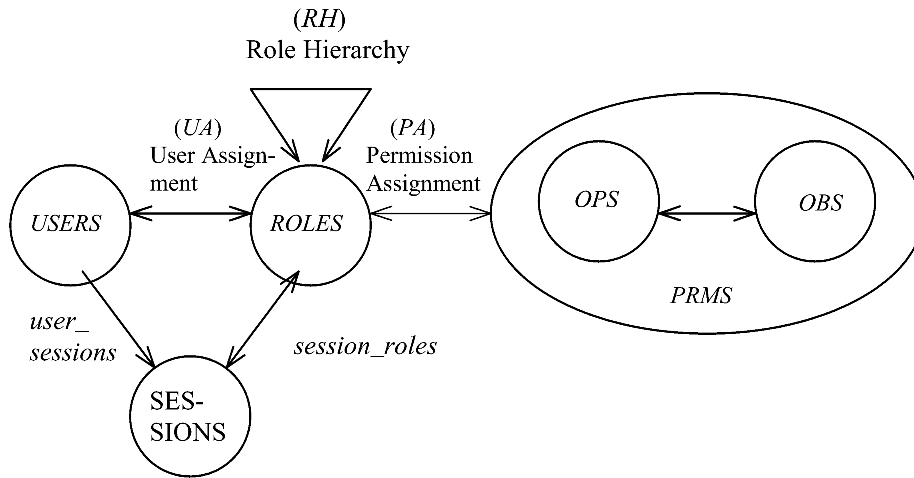

Fig. 2. Hierarchical RBAC.

is, r_1 "inherits" role r_2 if all privileges of r_2 are also privileges of r_1. Other authors have proposed a stronger definition of inheritance [Nyanchama and Osborn 1999] as well as alternate interpretations [Kuhn 1998; Sandhu 1998a]. We have adopted the most widely used definition.[4]

This standard recognizes both general and limited role hierarchies. General role hierarchies provide support for an arbitrary partial order to serve as the role hierarchy, to include the concept of multiple inheritances of permissions and user membership among roles. Limited role hierarchies impose restrictions resulting in a simpler tree structure (i.e., a role may have one or more immediate ascendants, but is restricted to a single immediate descendant). Note that an inverted tree is also possible. Examples of possible hierarchical role structures are shown in Figure 3. Note that user membership is inherited top-down, and role permissions are inherited bottom-up.

We first formally define general role hierarchies.

Definition 2a. General Role Hierarchies.

—$RH \subseteq ROLES \times ROLES$ is a partial order on *ROLES* called the inheritance relation, written as $\succeq$, where $r_1 \succeq r_2$ only if all permissions of r_2 are also permissions of r_1, and all users of r_1 are also users of r_2. Formally: $r_1 \succeq r_2 \Rightarrow authorized_permissions(r_2) \subseteq authorized_permissions(r_1) \wedge authorized_users(r_1) \subseteq authorized_users(r_2)$.

—$authorized_users(r{:}\ ROLES) \rightarrow 2^{USERS}$, the mapping of role r onto a set of users in the presence of a role hierarchy. Formally: $authorized_users(r) = \{u \in USERS \mid r' \succeq r\ (u, r') \in UA\}$.

[4]For some distributed RBAC implementations, role permissions are not managed centrally, while the role hierarchies are. For these systems, role hierarchies are managed in terms of user containment relations: role r_1 "contains" role r_2 if all users authorized for r_1 are also authorized for r_2. Note, however, that user containment implies that a user of r_1 has (at least) all the privileges of r_2, while the permission inheritance for r_1 and r_2 does not imply anything about user assignment.

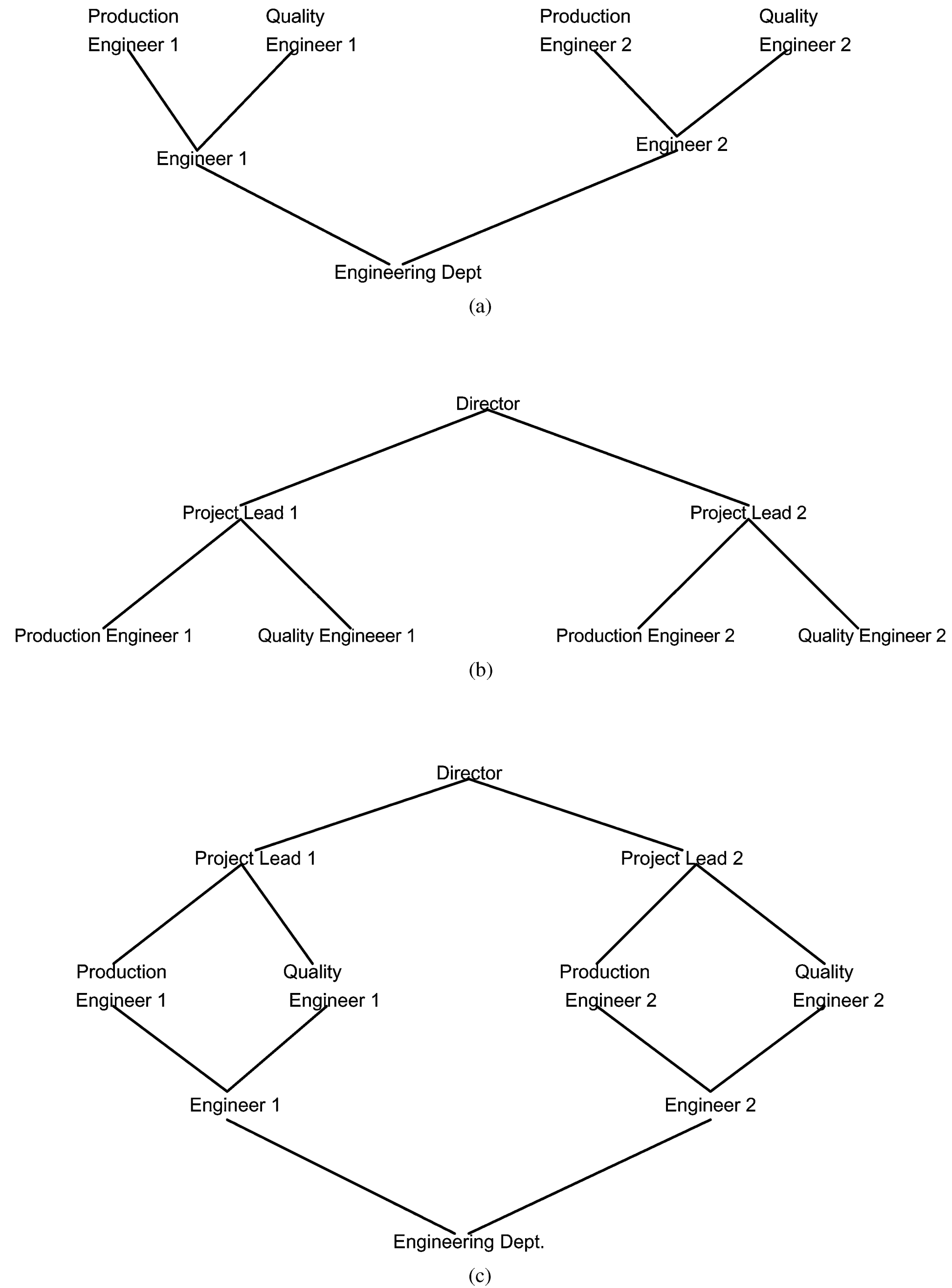

Fig. 3. Example role hierarchies: (a) tree; (b) inverted tree; (c) lattice.

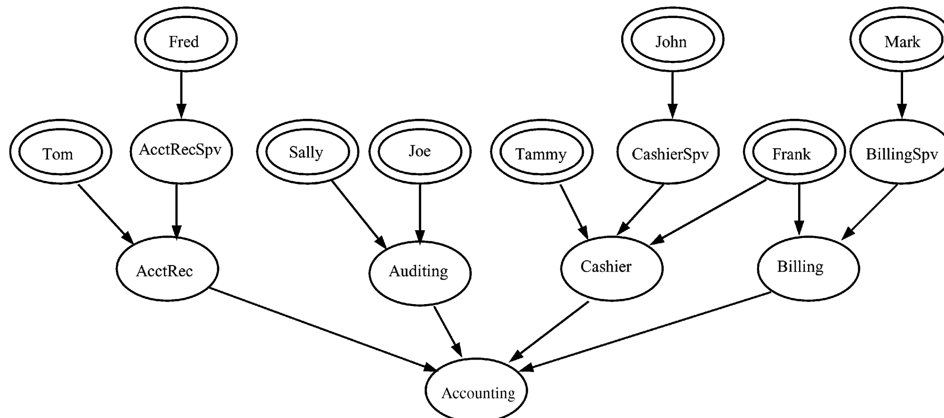

Fig. 4. Accounting roles.

—*authorized_permissions*(r: *ROLES*) $\rightarrow$ 2^{PRMS}, the mapping of role r onto a set of permissions in the presence of a role hierarchy. Formally: $authorized_permissions(r) = \{p \in PRMS \mid r' \succeq r, (p, r') \in PA\}$.

General role hierarchies support the concept of *multiple inheritance*, which provides the ability to inherit permission from two or more role sources and to inherit user membership from two or more role sources. Multiple inheritances provide important hierarchy properties. The first is the ability to compose a role from multiple subordinate roles (with fewer permissions) in defining roles and relations that are characteristic of the organization and business structures, which these roles are intended to represent. Second, multiple inheritances provide uniform treatment of user/role assignment relations and role/role inheritance relations. Users can be included in the role hierarchy, using the same relation $\succeq$ to denote the user assignment to roles, as well as permission inheritance from a role to its assigned users.

Roles in a limited role hierarchy are restricted to a single *immediate descendant*. Although limited role hierarchies do not support multiple inheritances, they nonetheless provide clear administrative advantages over Core RBAC.

We represent r_1 as an immediate descendent of r_2 by $r_1 \succ\succ r_2$, if $r_1 \succeq r_2$, but no role in the role hierarchy lies between r_1 and r_2. That is, there exists no role r_3 in the role hierarchy such that $r \succeq r_3 \succeq r_2$, where $r_1 \neq r_2$ and $r_2 \neq r_3$.

We now define limited role hierarchies as a restriction on the immediate descendants of the general role hierarchy.[5]

Definition 2b. Limited Role Hierarchies.

—Definition 2a with the limitation:
$\forall r, r_1, r_2 \in ROLES, r \succeq r_1 \wedge r \succeq r_2 \Rightarrow r_1 = r_2$.

Figure 4 illustrates properties of a limited role hierarchy. In the role graph of Figure 4, where the users are represented by double ellipses and the

[5]A similar notion of limited role hierarchies for inverted trees can also be defined.

roles by single ellipses, John is assigned to CashierSpv, and is authorized for CashierSpv, Cashier, and Accounting. Also, John's permissions are the union of the permission sets assigned to John, CashierSpv, Cashier, Accounting, and the permissions directly assigned to John. Note that users are permitted to be included in the graph as a result of multiple inheritances. Although the role assignments of Fred, John, and Mark could be represented in a limited role hierarchy, Frank's role assignments could not. Because Core RBAC requires user role assignment to be a many-to-many relation, in the general case users would be precluded from being included as nodes in a limited role hierarchy.

3.3 Constrained RBAC

Constrained RBAC adds separation of duty relations to the RBAC model. Separation of duty relations are used to enforce conflict of interest policies that organizations may employ to prevent users from exceeding a reasonable level of authority for their positions.

As a security principle, SOD has long been recognized for its wide application in business, industry, and government [Brewer and Nash 1989; Clark and Wilson 1987]. Its purpose is to ensure that failures of omission or commission within an organization can be caused only as a result of collusion among individuals. To minimize the likelihood of collusion, individuals of different skills or divergent interests are assigned to separate tasks required in the performance of a business function. The motivation is to ensure that fraud and major errors cannot occur without deliberate collusion of multiple users. This RBAC standard allows for both static and dynamic separation of duty as defined within the next two subsections.

3.3.1 *Static Separation of Duty Relations.* Conflict of interest in a role-based system may arise as a result of a user gaining authorization for permissions associated with conflicting roles. One means of preventing this form of conflict of interest is through *static separation of duty*, that is, to enforce constraints on the assignment of users to roles. Static constraints can take on a wide variety of forms. A common example is that of static separation of duty that defines mutually disjoint user assignments with respect to sets of roles. Static constraints have also been shown to be a powerful means of implementing a number of other important separation of duty policies.[6] For example, Gligor et al. [1998] formally defined four other types of static separation of duty policies. Static constraints beyond separation of duties have been identified in Ahn and Sandhu [2000]. Although formal RBAC models and policy specifications have grown well beyond simple relations, we know of no commercial products that implement these advanced static constraint relations.

The static constraints defined in this model are limited to those relations that place restrictions on sets of roles and in particular on their ability to form *UA* relations. This means that if a user is assigned to one role, the user is prohibited from being a member of a second role. For example, a user who is assigned to the

[6]Please see Ferraiolo et al. [1995], Kuhn [1997], Simon and Zurko [1997], Gligor et al. [1998], and Giuri and Iglio [1996].

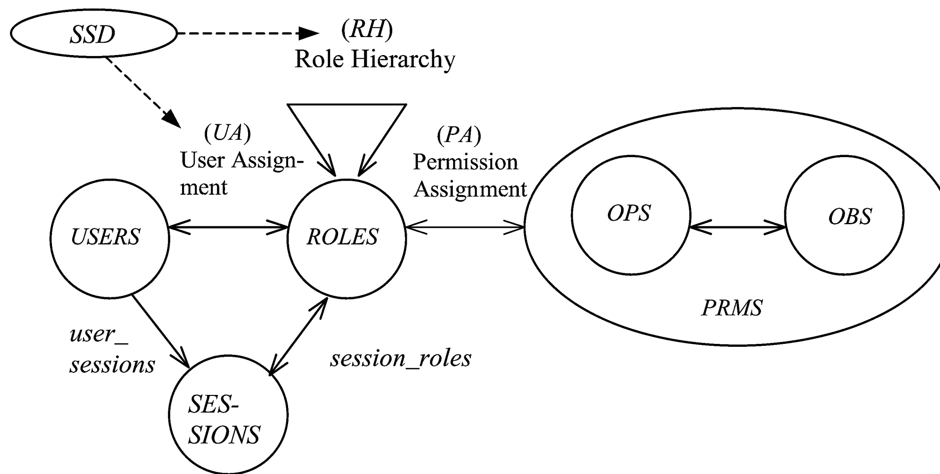

Fig. 5. SSD within Hierarchical RBAC.

role Billing Clerk may not be assigned to the role Accounts Receivable Clerk. That is, the roles Billing Clerk and Accounts Receivable Clerk are mutually exclusive. An SSD policy can be centrally specified and then uniformly imposed on specific roles. From a policy perspective, static constraint relations provides a powerful means of enforcing conflict of interest and other separation rules over sets of RBAC elements. Static constraints generally place restrictions on administrative operations that have the potential to undermine higher-level organizational separation of duty policies.

RBAC models have defined SSD relations with respect to constraints on user-role assignments over pairs of roles (i.e., no user can be simultaneously assigned to both roles in SSD). Although real-world examples of this SSD policy exist, this definition is overly restrictive in two important aspects: the size of the set of roles in the SSD and the combination of roles in the set for which user assignment is restricted. In this model we define SSD with two arguments: a role set that includes two or more roles, and cardinality greater than one indicating a combination of roles that would constitute a violation of the SSD policy. For example, an organization may require that no one user may be assigned to three of the four roles that represent the purchasing function.

As illustrated in Figure 5, SSD relations may exist within Hierarchical RBAC. When applying SSD relations in the presence of a role hierarchy, special care must be applied to ensure that user inheritance does not undermine SSD policies. As such, role hierarchies have been defined to include the inheritance of SSD constraints [Gavrila and Barkley 1998; Ferraiolo et al. 1999]. If, for example, the role Accounts Receivable Supervisor inherits Accounts Receivable Clerk, and Accounts Receivable Clerk has an SSD relationship with Billing Clerk, then Accounts Receivable Supervisor also has an SSD relationship with Billing Clerk. To address this potential inconsistency we define SSD as a constraint on the authorized users of the roles that have an SSD relation.

The formal definition of static separation of duty is given below.

Definition 3a. Static Separation of Duty.

—$SSD \subseteq (2^{ROLES} \times N)$ is a collection of pairs (rs, n) in *Static Separation of Duty*, where each rs is a role set, t a subset of roles in rs, and n is a natural number ≥ 2, with the property that no user is assigned to n or more roles from the set rs in each $(rs, n) \in SSD$. Formally: $\forall (rs, n) \in SSD, \forall t \subseteq rs : |t| \geq n \Rightarrow \bigcap_{r \in t} assigned_users\,(r) = \emptyset$.

Definition 3b. Static Separation of Duty in the Presence of a Hierarchy.

—In the presence of a role hierarchy *static separation of duty* is redefined based on authorized users rather than assigned users as follows.

$$\forall (rs, n) \in SSD, \forall t \subseteq rs : |t| \geq n \Rightarrow \bigcap_{r \in t} authorized_users\,(r) = \emptyset.$$

3.3.2 *Dynamic Separation of Duty Relations.* Static Separation of Duty relations reduce the number of potential permissions that can be made available to a user by placing constraints on the users that can be assigned to a set of roles. Dynamic Separation of Duty relations, like SSD relations, are intended to limit the permissions that are available to a user. However DSD relations differ from SSD relations by the context in which these limitations are imposed. SSD relations define and place constraints on a user's total permission space. This model component defines DSD properties that limit the availability of the permissions over a user's permission space by placing constraints on the roles that can be activated within or across a user's sessions. DSD properties provide extended support for the principle of least privilege in that each user has different levels of permission at different times, depending on the role being performed. These properties ensure that permissions do not persist beyond the time that they are required for performance of duty. This aspect of least privilege is often referred to as *timely revocation of trust*. Dynamic revocation of permissions can be a rather complex issue without the facilities of dynamic separation of duty, and as such it has been generally ignored in the past for reasons of expediency.

This model component provides the capability to enforce an organization-specific policy of dynamic separation of duty. SSD relations provide the capability to address potential conflict-of-interest issues at the time a user is assigned to a role. DSD allows a user to be authorized for two or more roles that do not create a conflict of interest when acted on independently, but produce policy concerns when activated simultaneously. For example, a user may be authorized for both the roles of Cashier and Cashier Supervisor, where the supervisor is allowed to acknowledge corrections to a Cashier's open cash drawer. If the individual acting in the role Cashier attempted to switch to the role Cashier Supervisor, DSD would require the user to drop the Cashier role, and thereby force the closure of the cash drawer before assuming the role Cashier Supervisor. As long as the same user is not allowed to assume both of these roles at the same time, a conflict of interest situation will not arise. Although this effect could be achieved through the establishment of a Static Separation of Duty

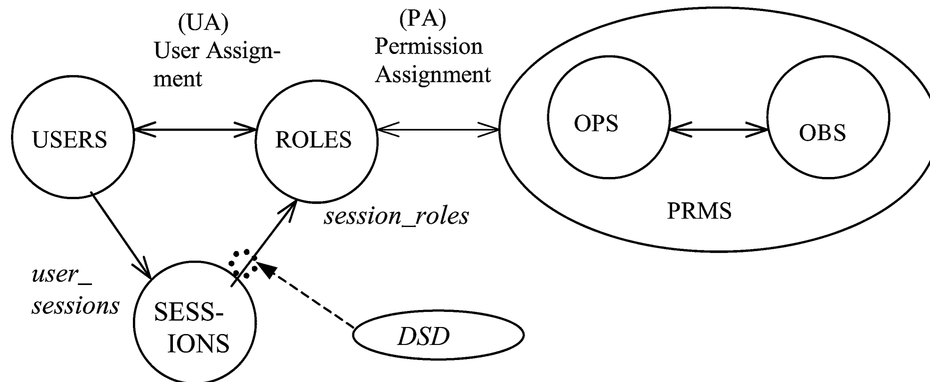

Fig. 6. Dynamic Separation of Duty relations.

relationship, DSD relationships generally provide the enterprise with greater operational flexibility.

We define Dynamic Separation of Duty relations as a constraint on the roles that are activated in a user's session (see Figure 6).

The formal definition of Dynamic Separation of Duty is given below.

Definition 4. Dynamic Separation of Duty.

—$DSD \subseteq (2^{ROLES} \times \mathrm{N})$ is collection of pairs (rs, n) in *Dynamic Separation of Duty*, where each *rs* is a role set and *n* is a natural number ≥ 2, with the property that no subject may activate *n* or more roles from the set *rs* in each $dsd \in DSD$. Formally:

$$\forall rs \in 2^{ROLES}, n \in \mathrm{N}, (rs, n) \in DSD \Rightarrow n \geq 2 \wedge |rs| \geq n, \text{ and}$$

$$\forall s \in SESSIONS, \forall rs \in 2^{ROLES}, \forall role_subset \in 2^{ROLES}, \forall n \in \mathrm{N}, (rs, n) \in DSD,$$
$$role_subset \subseteq rs, role_subset \subseteq session_roles(s) \Rightarrow |role_subset| < n.$$

4. FUNCTIONAL SPECIFICATION OVERVIEW

In this section, we provide an overview of the functionality involved in meeting the requirements for each of the components defined in the previous section. In Section 3, we defined RBAC as four model components in terms of an abstract set of element sets, relations, and administrative queries. In this section, we cast the abstract model concepts into functional requirements for administrative operations, session management, and administrative review. The RBAC functional specification outlines the semantics of the various functions that are required for creation and maintenance of the RBAC Model components (element sets and relations), as well as supporting system functions. These functions can be composed and packaged into higher-level abstract operations in implementations.

The three categories of functions in the RBAC functional specification and their purpose are:

—*Administrative Functions*: creation and maintenance of element sets and relations for building the various component RBAC models;

—*Supporting System Functions*: functions that are required by the RBAC implementation to support the RBAC model constructs (e.g., RBAC session attributes and access decision logic) during user interaction with an IT system; and

—*Review Functions*: review the results of the actions created by administrative functions.

A complete specification of these functions using the Z notation is given in Appendix A. Each subsection in Section 4 provides an overview of the correspondingly numbered subsection in Appendix A (e.g., Section 4.1.2 summarizes A.1.2.) Function descriptions in Appendix A are intended to provide a level of detail sufficient for evaluating RBAC implementations for conformance with the RBAC Reference Model.

4.1 Functional Specification for Core RBAC

4.1.1 *Administrative Functions. Creation and Maintenance of Element Sets*. The basic element sets in Core RBAC are USERS, ROLES, OPS, and OBS. Of these element sets, OPS and OBS are considered predefined by the underlying information system for which RBAC is deployed. For example, a banking system may have predefined transactions (OPS) for savings deposit and others, and predefined data sets (OBS) such as savings files, address files, and other necessary data. Administrators create and delete USERS and ROLES, and establish relationships between roles and existing operations and objects. Required administrative functions for USERS are AddUser and DeleteUser, and for ROLES are AddRole and DeleteRole.

Creation and Maintenance of Relations. The main relations of Core RBAC are (a) user-to-role assignment relation (UA) and (b) permission-to-role assignment relation (PA). Functions to create and delete instances of UA relations are AssignUser and DeassignUser. For PA the required functions are GrantPermission and RevokePermission.

4.1.2 *Supporting System Functions.* Supporting system functions are required for session management and in making access control decisions. An active role is necessary for regulating access control for a user in a session. The function that creates a session establishes a default set of active roles for the user at the start of the session. The composition of this default set can then be altered by the user during the session by adding or deleting roles. Functions relating to the adding and dropping of active roles and other auxiliary functions are:

—CreateSession: creates a User Session and provides the user with a default set of active roles;

—AddActiveRole: adds a role as an active role for the current session;

—DropActiveRole: deletes a role from the active role set for the current session; and

—CheckAccess: determines if the session subject has permission to perform the requested operation on an object.

4.1.3 *Review Functions.* When user-to-role assignment and permission-to-role relation instances have been created, it should be possible to view the contents of those relations from both the user and role perspectives. For example, from the UA relation, the administrator should have the facility to view all the users assigned to a given role as well to view all the roles assigned to a given user. In addition, it should be possible to view the results of the supporting system functions to determine some session attributes such as the active roles in a given session or the total permission domain for a given session. Since not all RBAC implementations provide facilities for viewing role, user, and session permissions or active roles for a session, these functions have been designated as optional/advance functions in our requirement specification. Mandatory (M) and Optional (O) review functions are:

—AssignedUsers (M): returns the set of users assigned to a given role;

—AssignedRoles (M): returns the set of roles assigned to a given user;

—RolePermissions (O): returns the set of permissions granted to a given role;

—UserPermissions (O): returns the set of permissions a given user gets through his or her assigned roles;

—SessionRoles(O): returns the set of active roles associated with a session;

—SessionPermissions (O): returns the set of permissions available in the session (i.e., union of all permissions assigned to sesssion's active roles);

—RoleOperationsOnObject (O): returns the set of operations a given role may perform on a given object; and

—UserOperationsOnObject (O): returns the set of operations a given user may perform on a given object (obtained either directly or through his or her assigned roles).

4.2 Functional Specification for Hierarchical RBAC

4.2.1 *Hierarchical Administrative Functions.* The administrative functions required for hierarchical RBAC include all the administrative functions that were required for Core RBAC. However, the semantics for DeassignUser must be redefined because the presence of role hierarchies gives rise to the concept of authorized roles for a user. In other words, a user may inherit authorization for a role even if he or she is not directly assigned to the role. The hierarchy allows users to inherit permissions from roles that are junior to their assigned roles. An important issue is whether a user can only be deassigned from a role that was *directly* assigned to the user or can be deassigned from one of the (indirectly) authorized roles. The appropriate course of action is left as an implementation issue and is not prescribed in this specification.

The additional administrative functions required for the Hierarchical RBAC model pertain to creation and maintenance of the partial order relation (RH) among roles. The operations for a partial order involve either: (a) creating (or deleting) an inheritance relationship between two *existing roles* in a role set or (b) adding a newly created role at an appropriate position in the hierarchy by making it the ascendant or descendant role of an another role in the *existing hierarchy*. The name and purpose of these functions are summarized below:

—AddInheritance: establish a new immediate inheritance relationship between two existing roles;
—DeleteInheritance: delete an existing immediate inheritance relationship between two roles;
—AddAscendant: create a new role and add it as an immediate ascendant of an existing role; and
—AddDescendant: create a new role and add it as an immediate descendant of an existing role.

The model provides for both general and limited hierarchies. A general hierarchy allows multiple inheritance, while a limited hierarchy is essentially a tree (or inverted tree) structure. For a limited hierarchy, the AddInheritance function is constrained to a single ascendant (or descendant) role.

The outcome of the DeleteInheritance function may result in multiple scenarios. When DeleteInheritance is invoked with two given roles, say Role A and Role B, the implementation system is required to do one of the following. (1) The system may preserve the implicit inheritance relationships that roles A and B have with other roles in the hierarchy. That is, if role A inherits other roles, say C and D, through role B, role A will maintain permissions for C and D after the relationship with role B is deleted. (2) Another option is to break those relationships because an inheritance relationship no longer exists between Role A and Role B.

4.2.2 *Supporting System Functions.* The Supporting System Functions for Hierarchical RBAC are the same as for Core RBAC and provide the same functionality. However because of the presence of a role hierarchy, the functions *CreateSession* and *AddActiveRole* have to be redefined. In a role hierarchy, a given role may inherit one or more of the other roles. When that given role is activated by a user, the question of whether the inherited roles are automatically activated or must be explicitly activated is left as an implementation issue and no one course of action is prescribed as part of this specification. However, when the latter scenario is implemented (i.e., explicit activation) the corresponding supporting functionality shall be provided in the supporting system functions. For example, in the case of the CreateSession function, the active role set created as a result of the new session shall include not only roles directly assigned to a user but also some or all of the roles inherited by those "directly assigned roles" (that were previously included in the default Active Role Set) as well. Similarly, in the AddActiveRole function, a user can activate a directly assigned role or one or more of the roles inherited by the "directly assigned role."

4.2.3 *Review Functions.* All the review functions specified for Core RBAC remain valid for Hierarchical RBAC as well. In addition, the user membership set for a given role includes not only users directly assigned to that *given role* but also those users assigned *to roles that inherit the given role*. Analogously the role membership set for a given user includes not only roles *directly assigned to the given user* but also those *roles inherited by the directly assigned roles*. To

capture this expanded "User Memberships for Roles" and "Role Memberships for a User" the following functions are defined.

—AuthorizedUsers: returns the set of users directly assigned to a given role as well as those who were members of those "roles that inherited the given role."
—AuthorizedRoles: returns the set of roles directly assigned to a given user as well as those "roles that were inherited by the directly assigned roles."

Because of the presence of partial order among the roles, the permission set for a given role includes not only the permissions directly assigned to a given role but also permissions obtained from the roles that the given role inherited. Consequently the permission set for a user who is assigned that given role becomes expanded as well. These "Permissions Review" functions are listed below. As already alluded to, since not all RBAC implementations provide this facility, these are treated as advanced/optional functions:

—RolePermissions: returns the set of all permissions either directly granted to or inherited by a given role;
—UserPermissions: returns the set of permissions of a given user through his or her authorized roles (union of directly assigned roles and roles inherited by those roles);
—RoleOperationsOnObject: returns the set of operations a given role may perform on a given object (obtained either directly or by inheritance); and
—UserOperationsOnObject: returns the set of operations a given user may perform on a given object (obtained directly or through his or her assigned roles or through roles inherited by those roles).

4.3 Functional Specification for SSD Relation

4.3.1 *Administrative Functions.* The administrative functions for an SSD RBAC model without hierarchies shall include all the administrative functions for Core RBAC. However since the SSD property relates to membership of users in conflicting roles, the AssignUser function shall incorporate functionality to verify and ensure that a given user assignment does not violate the constraints associated with any instance of an SSD relation.

As already described under the SSD RBAC reference model, an SSD relation consists of a triplet: (SSD_Set_Name, role_set, SSD_Card). The SSD_Set_Name indicates the transaction or business process in which common user membership must be restricted in order to enforce a conflict of interest policy. The role_set is a set containing the constituent roles for the named SSD relation (and referred to as the Named SSD role set). The SSD_Card designates the cardinality of the subset within the role_set to which common user memberships must be restricted. Hence, administrative functions relating to creation and maintenance of an SSD relation are operations that Create and Delete an instance of an SSD relation, add and delete role members to the role-set parameter of the SSD relation, as well as to change/set the SSD_Card parameter for the SSD relation. These functions are summarized below:

—CreateSSDSet: creates a named instance of an SSD relation;
—DeleteSSDSet: deletes an existing SSD relation;
—AddSSDRoleMember: adds a role to a named SSD role set;
—DeleteSSDRoleMember: deletes a role from a named SSD role set; and
—SetSSDCardinality: sets the cardinality of the subset of roles from the named SSD role set for which common user membership restriction applies.

For the case of SSD RBAC models with role hierarchies (both General Role Hierarchies and Limited Role Hierarchies), the above functions produce the same end result with one exception: constraints governing the combination of role hierarchies and SSD relations shall be enforced when these functions are invoked. For example, roles within a hierarchical chain cannot be made members of a role set in an SSD relation.

4.3.2 *Supporting System Functions.* The Supporting System Functions for an SSD RBAC Model are the same as those for the Core RBAC Model.

4.3.3 *Review Functions.* All the review functions for the Core RBAC model are needed for implementation of the SSD RBAC model. In addition, functions to view the results of administrative functions listed in Section 4.3.1 shall also be provided. These include: (a) a function to reveal the set of named SSD relations created, (b) a function that returns the set of roles associated with a named SSD role set, and (c) a function that gives the cardinality of the subset within the named SSD role set for which common user membership restriction applies.

—SSDRoleSets: returns the set of named SSD relations created for the SSD RBAC model;
—SSDRoleSetRoles: returns the set of roles associated with a named SSD role set; and
—SSDRoleSetCardinality: returns the cardinality of the subset within the named SSD role set for which common user membership restriction applies.

4.4 Functional Specification for DSD Relation

4.4.1 *Administrative Functions.* The semantics of creating an instance of a DSD relation are identical to that of an SSD relation. While constraints associated with an SSD relation are enforced during user assignments (as well as while creating role hierarchies), the constraints associated with DSD are typically enforced only at the time of role activation within a user session. The list of administrative functions that shall be provided for the DSD RBAC model and their purpose are listed below:

—CreateDSDSet: creates a named instance of a DSD relation;
—DeleteDSDSet: deletes an existing DSD relation;
—AddDSDRoleMember: adds a role to a named DSD role set;
—DeleteDSDRoleMember: deletes a role from a named DSD role set; and
—SetDSDCardinality: sets the cardinality of the subset of roles from the named DSD role set for which user activation restriction within the same session applies.

4.4.2 *Supporting System Functions.* Recall from Section 4.1.2 that the Supporting System Functions for Core RBAC are: (a) CreateSession, (b) AddActiveRole, and (c) DeleteActiveRole. These system functions shall be available for a *DSD RBAC model implementation without role hierarchies* as well. However, the additional functionality required of these functions in the DSD RBAC model context is that they should enforce the DSD constraints. For example, during the invocation of the CreateSession function, the default active role set that is made available to the user should not violate any of the DSD constraints. Similarly, the AddActiveRole function shall check and prevent the addition of any active role to the session's active role set that violates any of the DSD constraints.

The semantics of the Supporting System Functions for a DSD RBAC model with role hierarchies (both General Role Hierarchy and Limited Role Hierarchy) are the same as those for corresponding functions for hierarchical RBAC in Section 4.2.2:

—CreateSession: creates a user session and provides the user with a default set of active roles;

—AddActiveRole: adds a role as an active role for the current session; and

—DropActiveRole: deletes a role from the active role set for the current session.

4.4.3 *Review Functions.* All the review functions for the Core RBAC model are needed for implementation of the DSD RBAC model. In addition, functions to view the results of administrative functions listed in Section 4.4.1 shall also be provided. These include: (a) a function to reveal the set of named DSD relations created, (b) a function that returns the set of roles associated with a named DSD role set, and (c) a function that gives the cardinality of the subset within the named DSD role set for which common user membership restriction applies.

—DSDRoleSets: returns the set of named SSD relations created for the DSD RBAC model;

—DSDRoleSetRoles: returns the set of roles associated with a named DSD role set; and

—DSDRoleSetCardinality: returns the cardinality of the subset within the named DSD role set for which user activation restriction within the same session applies.

5. FUNCTIONAL SPECIFICATION PACKAGES

As discussed in Section 1, RBAC is a technology that provides a diverse set of access control management features. In a categorization of these features, Section 4 defined a family of four functional components to include Core RBAC, Hierarchical RBAC, Static Separation of Duty Relations, and Dynamic Separation of Duty Relations. Each functional component includes three sections: administrative operations for the creation and maintenance of RBAC sets and relations, administrative review functions, and system level functions for the binding of roles to a user's session and making access control decisions.

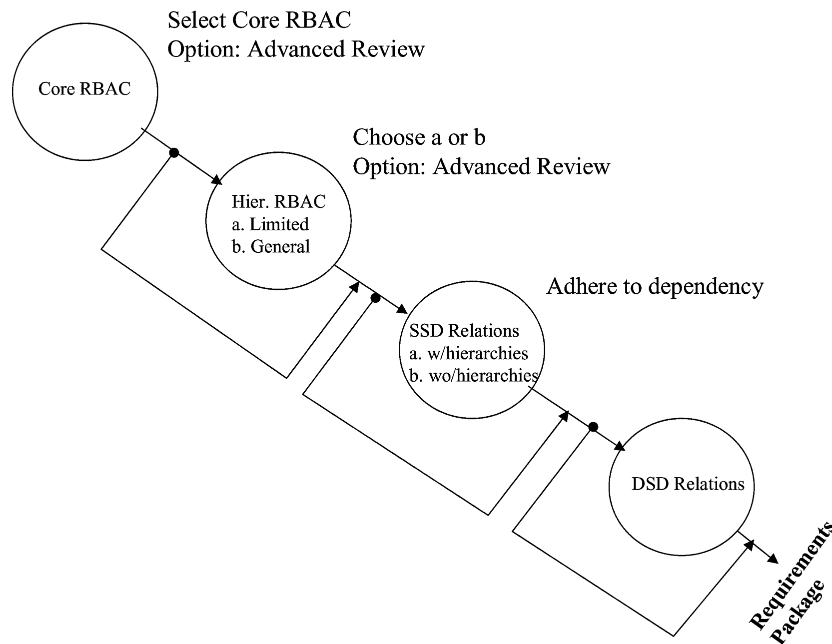

Fig. 7. Methodology for creating functional packages.

In this section we describe a logical approach for defining packages of functional components, where each package may pertain to a different threat environment and/or market segment. The basic concept is that each component can optionally be selected for inclusion into a package with one exception—Core RBAC must be included as a part of all packages. In selecting components, the reader is referred to Section 2 for a rationale of each component. Also, see Figure 7 for an overview of the methodology for composing functional packages.

In defining functional packages Core RBAC is unique in that it is fundamental and must be included in all packages. As such any package must begin with the selection of the Core RBAC. Core RBAC includes an advanced review feature that may be optionally selected. For some environments the selection of the single Core RBAC component may be sufficient.

Hierarchical RBAC includes two subcomponents: General Role Hierarchies and Limited Role Hierarchies. If Hierarchical RBAC is selected to be included in a package then a choice must be made as to which of these subcomponents is to be included. Like Core RBAC, Hierarchical RBAC includes an advanced review feature that may be optionally selected.

The Static Separation of Duty Relations component also includes two subcomponents: Static Separation of Duty Relations and Static Separation of Duty Relations in the Presence of a Hierarchy. If this component is selected for inclusion in a package then a dependency relation must be recognized. That is, if the package includes a Hierarchical RBAC component then Static Separation of Duty Relations in the Presence of a Hierarchy must be included in the package; otherwise the Static Separation of Duty Relations subcomponent must be selected.

The final component is Dynamic Separation of Duty Relations. This component does not include any options or dependency relations other than with Core RBAC.

6. CONCLUSIONS

The driving motivation for RBAC is to simplify security policy administration while facilitating the definition of flexible customized policies. Over the past nine years significant advancements have been made in both the theoretical modeling and practical implementation of RBAC features. Today RBAC is coming to be expected among large organizations and the number of vendors that offer RBAC features is growing rapidly. This development continues without general agreement on RBAC features. This article is a first attempt to develop an authoritative definition of well-accepted RBAC features for use in authorization management systems. Although RBAC continues to be an evolving technology, the RBAC features that were chosen to be included within this proposed standard represent a stable and well-accepted set of features, and are known to be included within a wide breadth of commercial products and reference implementations.

Standardization over a stable set of RBAC features is expected to provide a multitude of benefits, including a common set of benchmarks for vendors, who are already developing RBAC mechanisms, to use in their product specifications. It will give IT consumers, who are the principle beneficiaries of RBAC technology, a basis for the creation of uniform acquisition specifications. In addition, an RBAC standard will allow for the subsequent development of a standard RBAC API that would in turn promote the development of innovative authorization management tools by guaranteeing interoperability and portability.

Although RBAC is often considered a single access control and authorization model, it is in fact composed of a number of models each fit for a specific security management application. RBAC is also an open-ended technology, which ranges from the very simple to fairly sophisticated, as defined in numerous RBAC models and system specifications. Although these models and product specifications seem to agree on a fundamental set of RBAC concepts, they differ significantly in their terminology.

To address these issues, this proposed standard specifies a Reference Model, defined as a collection of four model components. The model components are intended to provide a standard vocabulary of relevant terms for defining a broad range of RBAC features. This proposed standard also includes an RBAC Functional Specification that casts the reference model into a congruent set of functional components, where each component defines specific requirements for administrative operations to create and maintain RBAC sets and relations, review functions, and system features pertaining to the corresponding model component.

RBAC functional model components can be combined into a variety of packages to arrive at a relevant collection of requirements for product development, system evaluation, or system acquisition specification. To facilitate this

packaging of requirements a rationale for the selection of components has been provided.

APPENDIX A: RBAC FUNCTIONAL SPECIFICATION

The RBAC Functional Specification specifies administrative operations for the creation and maintenance of RBAC element sets and relations, administrative review functions for performing administrative queries, and system functions for creating and managing RBAC attributes on user sessions and making access control decisions. Functions are defined with sufficient precision to meet the needs of conformance testing and assurance, while providing developers with design flexibility and the ability to incorporate additional features to meet the needs of users.

The notation used in the formal specification of the RBAC requirements is basically a subset of the Z notation. The only major change is the representation of a schema:

$$\text{Schema-Name (Declaration)} \lhd \text{Predicate}; \ldots; \text{ Predicate} \rhd.$$

Most abstract data types and functions used in the formal specification are defined in Section 3, RBAC Reference Model. New abstract data types and functions are introduced as needed. *NAME* is an abstract data type whose elements represent identifiers of entities that may or may not be included in the RBAC system (roles, users, sessions, etc.).

A.1 Requirements for Core RBAC

A.1.1 *Administrative Commands for Core RBAC*

AddUser: This command creates a new RBAC user. The command is valid only if the new user is not already a member of the *USERS* data set. The *USER* data set is updated. The new user does not own any session at the time of its creation. The following schema formally describes the command AddUser.

$$\begin{aligned}
&AddUser(user : NAME) \lhd \\
&\quad user \notin USERS \\
&\quad USERS' = USERS \cup \{user\} \\
&\quad user_sessions' = user_sessions \cup \{user \mapsto \emptyset\} \rhd
\end{aligned}$$

DeleteUser: This command deletes an existing user from the RBAC database. The command is valid if and only if the user to be deleted is a member of the *USERS* data set. The *USERS* and *UA* data sets and the *assigned_users* function are updated. It is an implementation decision how to proceed with the sessions owned by the user to be deleted. The RBAC system could wait for such a session to terminate normally, or it could force its termination. Our presentation illustrates the case when those sessions are forcefully terminated. The following schema formally describes the command DeleteUser.

$$\begin{aligned}
&DeleteUser(user: NAME) \lhd \\
&\quad user \in USERS
\end{aligned}$$

$[\forall s \in SESSIONS \bullet s \in user_sessions(user) \Rightarrow DeleteSession(s)]$
$UA' = UA \setminus \{r: ROLES \bullet user \mapsto r\}$
$assigned_users' = \{r: ROLES \bullet r \mapsto (assigned_users(r) \setminus \{user\})\}$
$USERS' = USERS \setminus \{user\} \rhd$

AddRole: This command creates a new role. The command is valid if and only if the new role is not already a member of the *ROLES* data set. The ROLES data set and the functions *assigned_users* and *assigned_permissions* are updated. Initially, no user or permission is assigned to the new role. The following schema formally describes the command AddRole.

$AddRole(role : NAME) \lhd$
$role \notin ROLES$
$ROLES' = ROLES \cup \{role\}$
$assigned_users' = assigned_users \cup \{role \mapsto \emptyset\}$
$assigned_permissions' = assigned_permissions \cup \{role \mapsto \emptyset\} \rhd$

DeleteRole: This command deletes an existing role from the RBAC database. The command is valid if and only if the role to be deleted is a member of the *ROLES* data set. It is an implementation decision how to proceed with the sessions in which the role to be deleted is active. The RBAC system could wait for such a session to terminate normally, it could force the termination of that session, or it could delete the role from that session while allowing the session to continue. Our presentation illustrates the case when those sessions are forcefully terminated.

$DeleteRole(role: NAME) \lhd$
$role \in ROLES$
$[\forall s \in SESSIONS \bullet role \in session_roles(s) \Rightarrow DeleteSession(s)]$
$UA' = UA \setminus \{u: USERS \bullet u \mapsto role\}$
$assigned_users' = assigned_users \setminus \{role \mapsto asigned_users(role)\}$
$PA' = PA \setminus \{op: OPS, obj: OBJS \bullet (op, obj) \mapsto role\}$
$assigned_permissions' = assigned_permissions \setminus$
$\{role \mapsto assigned_permissions(role)\}$
$ROLES' = ROLES \setminus \{role\} \rhd$

AssignUser: This command assigns a user to a role. The command is valid if and only if the user is a member of the *USERS* data set, the role is a member of the *ROLES* data set, and the user is not already assigned to the role. The data set UA and the function *assigned_users* are updated to reflect the assignment. The following schema formally describes the command.

$AssignUser(user, role: NAME) \lhd$
$user \in USERS; role \in ROLES; (user \mapsto role) \notin UA$
$UA' = UA \cup \{user \mapsto role\}$

$$assigned_users' = assigned_users \setminus \{role \mapsto assigned_users(role)\} \cup \{role \mapsto (assigned_users(role) \cup \{user\})\} \rhd$$

DeassignUser: This command deletes the assignment of the user *user* to the role *role*. The command is valid if and only if the user is a member of the *USERS* data set, the role is a member of the *ROLES* data set, and the user is assigned to the role. It is an implementation decision how to proceed with the sessions in which the session user is *user* and one of his or her active roles is *role*. The RBAC system could wait for such a session to terminate normally, could force its termination, or could inactivate the role. Our presentation illustrates the case when those sessions are forcefully terminated. The following schema formally describes the command DeassignUser.

$$DeassignUser(user, role{:}\ NAME) \lhd$$
$$user \in USERS;\ role \in ROLES;\ (user \mapsto role) \in UA$$
$$[\forall s{:}\ SESSIONS \bullet s \in user_sessions(user) \wedge role \in session_roles(s) \Rightarrow DeleteSession(s)]$$
$$UA' = UA \setminus \{user \mapsto role\}$$
$$assigned_users' = assigned_users \setminus \{role \mapsto assigned_users(role)\} \cup \{role \mapsto (asigned_users(role) \setminus \{user\})\} \rhd$$

GrantPermission: This command grants a role the permission to perform an operation on an object to a role. The command may be implemented as granting permissions to a group corresponding to that role, that is, setting the access control list of the object involved. The command is valid if and only if the pair (operation, object) represents a permission, and the role is a member of the *ROLES* data set. The following schema formally defines the command.

$$GrantPermission(object, operation, role{:}\ NAME) \lhd$$
$$(operation, object) \in PERMS;\ role \in ROLES$$
$$PA' = PA \cup \{(operation, object) \mapsto role\}$$
$$assigned_permissions' = assigned_permissions \setminus \{role \mapsto assigned_permissions(roles)\} \cup \{role \mapsto (assigned_permissions(role) \cup \{(operation, object)\})\} \rhd$$

RevokePermission: This command revokes the permission to perform an operation on an object from the set of permissions assigned to a role. The command may be implemented as revoking permissions from a group corresponding to that role, that is, setting the access control list of the object involved. The command is valid if and only if the pair (operation, object) represents a permission, the role is a member of the *ROLES* data set, and the permission is assigned to that role. The following schema formally describes the command.

$$RevokePermission(operation, object, role{:}\ NAME) \lhd$$
$$(operation, object) \in PERMS;\ role \in ROLES;\ ((operation, object) \mapsto role) \in PA$$

$$PA' = PA \setminus \{(operation, object) \mapsto role\}$$
$$assigned_permissions' = assigned_permissions \setminus$$
$$\{role \mapsto assigned_permissions(role)\} \cup$$
$$\{role \mapsto (assigned_permissions(role) \setminus \{(operation, object)\})\} \rhd$$

A.1.2 *System Functions for Core RBAC*

CreateSession(*user*, *session*): This function creates a new session with a given user as owner and an active role set. The function is valid if and only if:

—the user is a member of the *USERS* data set, and
—the active role set is a subset of the roles assigned to that user. In an RBAC implementation, the session's active roles might actually be the groups that represent those roles.

The following schema formally describes the function. The *session* parameter, which represents the session identifier, is actually generated by the underlying system.

$$CreateSession(user: NAME; ars{:}2^{NAMES};\ session: NAME\,) \lhd$$
$$user \in USERS; ars \subseteq \{r: ROLES \mid (user \mapsto r) \in UA\}; session \notin SESSIONS$$
$$SESSIONS' = SESSIONS \cup \{session\}$$
$$user_sessions' = user_sessions \setminus \{user \mapsto user_sessions(user)\} \cup$$
$$\{user \mapsto (user_sessions(user) \cup \{session\})\}$$
$$session_roles' = session_roles \cup \{session \mapsto ars\} \rhd$$

DeleteSession(*user*, *session*): This function deletes a given session with a given owner user. The function is valid if and only if the session identifier is a member of the *SESSIONS* data set, the user is a member of the *USERS* data set, and the session is owned by the given user. The following schema formally describes the function.

$$DeleteSession(user, session: NAME) \lhd$$
$$user \in USERS; session \in SESSIONS; sessions \in user_sessions(user)$$
$$user_sessions' = user_sessions \setminus \{user \mapsto user_sessions(user)\} \cup$$
$$\{user \mapsto (user_sessions(user) \setminus \{session\})\}$$
$$session_roles' = session_roles \setminus \{session \mapsto session_roles(session)\}$$
$$SESSIONS' = SESSIONS \setminus \{session\} \rhd$$

AddActiveRole: This function adds a role as an active role of a session whose owner is a given user. The function is valid if and only if:

—the user is a member of the *USERS* data set,
—the role is a member of the *ROLES* data set,
—the session identifier is a member of the *SESSIONS* data set,
—the role is assigned to the user, and
—the session is owned by that user.

In an implementation, the new active role might be a group that corresponds to that role. The following schema formally describes the function.

$$
\begin{aligned}
&AddActiveRole(user, session, role{:}\ NAME) \lhd \\
&\quad user \in USERS;\ session \in SESSIONS;\ role \in ROLES; \\
&\qquad session \in user_sessions(user) \\
&\quad (user \mapsto role) \in UA;\ role \notin session_roles(session) \\
&\quad session_roles' = session_roles \setminus \{session \mapsto session_roles(session)\} \cup \\
&\qquad \{session \mapsto (session_roles(session) \cup \{role\})\} \rhd
\end{aligned}
$$

DropActiveRole: This function deletes a role from the active role set of a session owned by a given user. The function is valid if and only if the user is a member of the *USERS* data set, the session identifier is a member of the *SESSIONS* data set, the session is owned by the user, and the role is an active role of that session. The following schema formally describes this function.

$$
\begin{aligned}
&DropActiveRole(user, session, role{:}\ NAME) \lhd \\
&\quad user \in USERS;\ role \in ROLES;\ session \in SESSIONS \\
&\quad session \in user_sessions(user); role \in session_roles(session) \\
&\quad session_roles' = session_roles \setminus \{session \mapsto session_roles(session)\} \cup \\
&\quad \{session \mapsto (session_roles(session) \setminus \{role\}) \rhd
\end{aligned}
$$

CheckAccess: This function returns a Boolean value meaning the subject of a given session is or is not allowed to perform a given operation on a given object. The function is valid if and only if the session identifier is a member of the *SESSIONS* data set, the object is a member of the *OBJS* data set, and the operation is a member of the *OPS* data set. The session's subject has the permission to perform the operation on that object if and only if that permission is assigned to (at least) one of the session's active roles. An implementation might use the groups that correspond to the subject's active roles and their permissions as registered in the object's access control list. The following schema formally describes the function.

$$
\begin{aligned}
&CheckAccess(session, operation, object{:}\ NAME;\ out\ result{:}\ BOOLEAN) \lhd \\
&\quad session \in SESSIONS;\ operation \in OPS;\ object \in OBJS \\
&\quad result = (\exists r{:}ROLES \bullet r \in session_roles(session) \wedge \\
&\qquad ((operation, object) \mapsto r) \in PA) \rhd
\end{aligned}
$$

A.1.3 *Review Functions for Core RBAC*

AssignedUsers: This function returns the set of users assigned to a given role. The function is valid if and only if the role is a member of the *ROLES* data set. The following schema formally describes the function.

$$
\begin{aligned}
&AssignedUsers(role{:}\ NAME;\ out\ result{:}2^{USERS}) \lhd \\
&\quad role \in ROLES \\
&\quad result = \{u{:}\ USERS \mid (u \mapsto role) \in UA\} \rhd
\end{aligned}
$$

AssignedRoles: This function returns the set of roles assigned to a given user. The function is valid if and only if the user is a member of the *USERS* data set. The following schema formally describes the function.

$$\begin{aligned}&AssignedRoles(user{:}\ NAME;\ result{:}2^{ROLES}) \lhd\\&\quad user \in USERS\\&\quad result = \{r{:}\ ROLES \mid (user \mapsto r) \in UA\} \rhd\end{aligned}$$

A.1.4 *Advanced Review Functions for Core RBAC*

RolePermissions: This function returns the set of permissions (*op*, *obj*) granted to a given role. The function is valid if and only if the role is a member of the *ROLES* data set. The following schema formally describes the function.

$$\begin{aligned}&RolePermissions(role{:}\ NAME;\ result{:}2^{PERMS}) \lhd\\&\quad role \in ROLES\\&\quad result = \{op{:}\ OPS;\ obj{:}\ OBJS \mid ((op, obj) \mapsto role) \in PA\} \rhd\end{aligned}$$

UserPermissions: This function returns the permissions a given user gets through his or her assigned roles. The function is valid if and only if the user is a member of the *USERS* data set. The following schema formally describes this function.

$$\begin{aligned}&UserPermissions(user{:}\ NAME;\ result{:}2^{PERMS}) \lhd\\&\quad user \in USERS\\&\quad result = \{r{:}\ ROLES;\ op{:}\ OPS;\ obj{:}\ OBJS \mid (user \mapsto r) \in UA \land\\&\qquad ((op; obj) \mapsto r) \in PA \bullet (op, obj)\} \rhd\end{aligned}$$

SessionRoles: This function returns the active roles associated with a session. The function is valid if and only if the session identifier is a member of the *SESSIONS* data set. The following schema formally describes this function.

$$\begin{aligned}&SessionRoles(session{:}\ NAME;\ out\ result{:}2^{ROLES}) \lhd\\&\quad session \in SESSIONS\\&\quad result = session_roles(session) \rhd\end{aligned}$$

SessionPermissions: This function returns the permissions of the session *session*, that is, the permissions assigned to its active roles. The function is valid if and only if the session identifier is a member of the *SESSIONS* data set. The following schema formally describes this function.

$$\begin{aligned}&SessionPermissions(session{:}\ NAME;\ out\ result{:}2^{PERMS}) \lhd\\&\quad session \in SESSIONS\\&\quad result = \{r{:}\ ROLES;\ op \in OPS;\ obj \in OBJS \mid r \in session_roles(session) \land\\&\qquad ((op, obj) \mapsto r) \in PA \bullet (op, obj)\} \rhd\end{aligned}$$

RoleOperationsOnObject: This function returns the set of operations a given role is permitted to perform on a given object. The function is valid only if the

role is a member of the *ROLES* data set, and the object is a member of the *OBJS* data set. The following schema formally describes the function.

$$\begin{aligned}
&RoleOperationsOnObject(role\colon NAME;\, obj\colon NAME;\, result{:}2^{OPS}) \lhd \\
&\quad role \in ROLES \\
&\quad obj \in OBJS \\
&\quad result = \{op\colon OPS \mid ((op, obj) \mapsto role \in PA\} \rhd
\end{aligned}$$

UserOperationsOnObject: This function returns the set of operations a given user is permitted to perform on a given object, obtained either directly or through his or her assigned roles. The function is valid if and only if the user is a member of the *USERS* data set and the object is a member of the *OBJS* data set. The following schema formally describes this function.

$$\begin{aligned}
&UserOperationsOnObject(user\colon NAME;\, obj\colon NAME;\, result{:}2^{OPS}) \lhd \\
&\quad user \in USERS \\
&\quad obj \in OBJS \\
&\quad result = \{r\colon ROLES;\, op\colon OPS \mid (user \mapsto r) \\
&\qquad \in UA \wedge ((op, obj) \mapsto r) \in PA \bullet op\} \rhd
\end{aligned}$$

A.2 Requirements for Hierarchical RBAC

A.2a General Role Hierarchies

A.2a.1 *Administrative Commands for General Role Hierarchies*

All functions of Section A.1.1 remain valid. In addition, this section defines the following new specific functions.

AddInheritance: This command establishes a new immediate inheritance relationship $r_asc \succ\succ r_desc$ between existing roles r_asc, r_desc. The command is valid if and only if r_asc and r_desc are members of the *ROLES* data set, r_asc is not an immediate ascendant of r_desc, and r_desc does not properly inherit r_asc (in order to avoid cycle creation). The following schema uses the notations

$$\begin{aligned}
\geq &== \succeq \\
\gg &== \succ\succ
\end{aligned}$$

to formally describe the command

$$\begin{aligned}
&AddInheritance(r_asc, r_desc{:}NAME) \lhd \\
&\quad r_asc \in ROLES; r_desc \in ROLES; \neg(r_asc \gg r_desc); \neg(r_desc \geq r_asc) \\
&\quad \geq' \,=\, \geq \cup \{r, q\colon ROLES \mid r \geq r_asc \wedge r_desc \geq q \bullet r \mapsto q\} \rhd
\end{aligned}$$

DeleteInheritance: This command deletes an existing immediate inheritance relationship $r_asc \succ\succ r_desc$. The command is valid if and only if the roles r_asc and r_desc are members of the *ROLES* data set, and r_asc is an immediate ascendant of r_desc. The new inheritance relation is computed as the

reflexive-transitive closure of the immediate inheritance relation resulting after deleting the relationship $r_asc \succ\succ r_desc$. In this definition, implied relationships are preserved after deletion. The following schema formally describes this command.

$$DeleteInheritance(r_asc, r_desc\colon NAME) \lhd$$
$$r_asc \in ROLES;\ r_desc \in ROLES;\ r_asc >> r_desc$$
$$\succeq' = (\gg \setminus \{r_asc \mapsto r_desc\})^* \rhd$$

AddAscendant: This command creates a new role r_asc, and inserts it in the role hierarchy as an immediate ascendant of the existing role r_desc. The command is valid if and only if r_asc is not a member of the *ROLES* data set, and r_desc is a member of the *ROLES* data set. Note that the validity conditions are verified in the schemas *AddRole* and *AddInheritance*, referred to by *AddAscendant*.

$$AddAscendant(r_asc, r_desc\colon NAME) \lhd$$
$$AddRole(r_asc)$$
$$AddInheritance(r_asc, r_desc) \rhd$$

AddDescendant: This command creates a new role r_desc, and inserts it in the role hierarchy as an immediate descendant of the existing role r_asc. The command is valid if and only if r_desc is not a member of the *ROLES* data set, and r_asc is a member of the *ROLES* data set. Note that the validity conditions are verified in the schemas *AddRole* and *AddInheritance*, referred to by *AddDescendant*.

$$AddDescendant(r_asc, r_desc\colon NAME) \lhd$$
$$AddRole(r_desc)$$
$$AddInheritance(r_asc, r_desc) \rhd$$

A.2a.2 *System Functions for General Role Hierarchies*

This section redefines the functions CreateSession and AddActiveRole of Section A.1.2. The other functions of Section A.1.2 remain valid.

CreateSession(*user*, *session*): This function creates a new session with a given user as owner and a given set of active roles. The function is valid if and only if:

—the user is a member of the *USERS* data set, and

—the active role set is a subset of the roles authorized for that user. Note that if a role is active for a session, its descendants or ascendants are not necessarily active for that session. In an RBAC implementation, the session's active roles might actually be the groups that represent those roles.

The following schema formally describes the function. The parameter *session*, which identifies the session, is actually generated by the underlying system.

$$CreateSession(user{:}\ NAME;\ ars{:}2^{NAME};\ session{:}\ NAME) \lhd$$
$$user \in USERS;\ ars \subseteq \{r, q{:}\ ROLES \mid (user \mapsto q) \in UA \wedge$$
$$q \ \geq r \bullet r\}; session \notin SESSIONS$$
$$SESSIONS' = SESSIONS \cup \{session\}$$
$$user_sessions' = user_sessions \backslash \{user \mapsto user_sessions(user)\} \cup$$
$$\{user \mapsto (user_sessions(user) \cup \{sessions\})\}$$
$$session_roles' = session_roles \cup \{session \mapsto ars\} \rhd$$

AddActiveRole: This function adds a role as an active role of a session whose owner is a given user. The function is valid if and only if:

—the user is a member of the *USERS* data set,
—the role is a member of the *ROLES* data set,
—the session identifier is a member of the *SESSIONS* data set,
—the user is authorized for that role, and
—the session is owned by that user.

The following schema formally describes the function.

$$AddActiveRole(user,\ session,\ role{:}\ NAME) \lhd$$
$$user \in\ USERS;\ session \in SESSIONS;\ role$$
$$\in ROLES;\ session \in user_sessions(user)$$
$$user \in authorized_users(role);\ role \notin session_roles(session)$$
$$session_roles' = session_roles \backslash \{session \mapsto session_roles(sessions)\} \cup$$
$$\{session \mapsto (session_roles(session) \cup \{role\})\} \rhd$$

A.2a.3 *Review Functions for General Role Hierarchies*

All functions of Section A.1.3 remain valid. In addition, this section defines the following review functions.

AuthorizedUsers: This function returns the set of users authorized for a given role, that is, the users that are assigned to a role that inherits the given role. The function is valid if and only if the given role is a member of the *ROLES* data set. The following schema formally describes the function.

$$AuthorizedUsers(role{:}\ NAME;\ out\ result{:}2^{USERS}) \lhd$$
$$role \in ROLES$$
$$result = authorized_users(role) \rhd$$

AuthorizedRoles: This function returns the set of roles authorized for a given user. The function is valid if and only if the user is a member of the *USERS* data set. The following schema formally describes the function.

$$AuthorizedRoles(user{:}\ NAME;\ result{:}2^{ROLES}) \lhd$$
$$user \in USERS$$
$$result = \{r,\ q{:}\ ROLES \mid (user \mapsto q) \in UA \wedge q \geq r\} \rhd$$

A.2a.4 *Advanced Review Functions for General Role Hierarchies*

This section redefines the functions RolePermissions and UserPermissions of Section A.1.4. All other functions of Section A.1.4 remain valid.

RolePermissions: This function returns the set of all permissions (*op, obj*), granted to or inherited by a given role. The function is valid if and only if the role is a member of the *ROLES* data set. The following schema formally describes the function.

$$
\begin{array}{l}
RolePermission(role: NAME;\ result: 2^{PERMS}) \lhd \\
\quad role \in ROLES \\
\quad result = \{q: ROLES;\ op: OPS;\ obj: OBJS \mid (role \geq q) \wedge \\
\quad\quad ((op, obj) \mapsto role) \in PA \bullet (op, obj)\} \rhd
\end{array}
$$

UserPermissions: This function returns the set of permissions a given user gets through his or her authorized roles. The function is valid if and only if the user is a member of the *USERS* data set. The following schema formally describes this function.

$$
\begin{array}{l}
UserPermissions(user: NAME;\ result: 2^{PERMS}) \lhd \\
\quad user \in USERS \\
\quad result = \{r, q: ROLES;\ op: OPS;\ obj: OBJS \mid (user \mapsto q) \in UA \wedge (q \geq r) \wedge \\
\quad\quad ((op, obj) \mapsto r) \in PA \bullet (op, obj)\})\} \rhd
\end{array}
$$

RoleOperationsOnObject: This function returns the set of operations a given role is permitted to perform on a given object. The set contains all operations granted directly to that role or inherited by that role from other roles. The function is valid only if the role is a member of the *ROLES* data set, and the object is a member of the *OBJS* data set. The following schema formally describes the function.

$$
\begin{array}{l}
RoleOperationsOnObject(role: NAME;\ obj: NAME;\ result: 2^{OPS}) \lhd \\
\quad role \in ROLES \\
\quad obj \in OBJS \\
\quad result = \{q: ROLES;\ op: OPS \mid (role \geq q) \wedge ((op, obj) \mapsto role) \in PA \bullet op\} \rhd
\end{array}
$$

UserOperationsOnObject: This function returns the set of operations a given user is permitted to perform on a given object. The set consists of all the operations obtained by the user either directly or through his or her authorized roles. The function is valid if and only if the user is a member of the *USERS* data set and the object is a member of the *OBJS* data set. The following schema formally describes this function.

$$
\begin{array}{l}
UserOperationsOnObject(user: NAME;\ obj: NAME;\ result: 2^{OPS}) \lhd \\
\quad user \in USERS \\
\quad obj \in OBJS \\
\quad result = \{r, q: ROLES;\ op: OPS \mid (user \mapsto q) \in UA \wedge (q \geq r) \wedge \\
\quad\quad ((op, obj) \mapsto r) \in PA \bullet op\} \rhd
\end{array}
$$

A.2b Limited Role Hierarchies

A.2b.1 *Administrative Commands for Limited Role Hierarchies*

This section redefines the function AddInheritance of Section A.2a.1. All other functions of Section A.2a.1 remain valid.

AddInheritance: This command establishes a new immediate inheritance relationship $r_asc \succ\succ r_desc$ between existing roles r_asc, r_desc. The command is valid if and only if r_asc and r_desc are members of the *ROLES* data set, r_asc has no descendants, and r_desc does not properly inherit r_asc (in order to avoid cycle creation). The following schema uses the notations

$$\geq \; == \; \succeq$$
$$\gg \; == \; \succ\succ$$

to formally describe the command

$$\begin{array}{l} AddInheritance(r_asc, r_desc: NAME) \lhd \\ \quad r_asc \in ROLES; r_desc \in ROLES; \forall r \in ROLES \bullet \neg(r_asc >> r); \\ \quad\quad \neg(r_desc \geq r_asc) \\ \quad \geq' \, = \, \geq \cup \{r,\ q: ROLES \mid r \geq r_asc \wedge r_desc \geq q \bullet r \mapsto q\} \rhd \end{array}$$

A.2b.2 *System Functions for Limited Role Hierarchies*

All functions of Section A.2a.2 remain valid.

A.2b.3 *Review Functions for Limited Role Hierarchies*

All functions of Section A.2a.3 remain valid.

A.2b.4 *Advanced Review Functions for Limited Role Hierarchies*

All functions of Section A.2a.4 remain valid.

A.3 Requirements for Static Separation of Duty Relations

The static separation of duty property, as defined in the model, uses a collection SSD of pairs of a role set and an associated cardinality. We define the new data type *SSD*, which in an implementation could be the set of names used to identify the pairs in the collection.

The functions ssd_set and, respectively, ssd_card are used to obtain the role set and the associated cardinality from each SSD pair:

$$\begin{array}{l} ssd_set: SSD \rightarrow 2^{ROLES} \\ ssd_card: SSD \rightarrow \mathsf{N} \\ \forall ssd \in SSD \bullet ssd_card(ssd) \geq 2 \wedge ssd_card(ssd) \leq |ssd_set(ssd)| \end{array}$$

A.3a SSD Relations

A.3a.1 *Administrative Commands for SSD Relations*

This section redefines the function AssignUser of Section A.1.1 and defines a set of new specific functions. The other functions of Section A.1.1 remain valid.

AssignUser: The AssignUser command replaces the command with the same name of Core RBAC. This command assigns a user to a role. The command is valid if and only if:

—the user is a member of the *USERS* data set,
—the role is a member of the *ROLES* data set,
—the user is not already assigned to the role, and
—the SSD constraints are satisfied after assignment.

The data set UA and the function *assigned_users* are updated to reflect the assignment. The following schema formally describes the command.

$$AssignUser(user, role: NAME) \lhd$$
$$user \in USERS;\ role \in ROLES;\ (user \mapsto role) \notin UA$$
$$\forall ssd \in SSD \bullet \bigcap_{\substack{r \in subset \\ subset \subseteq ssd_set(ssd) \\ |subset| = ssd_card(ssd) \\ us = \text{if } r = role \text{ then } \{user\} \text{ else } \emptyset}} (assigned_users(r) \cup us) = \emptyset$$
$$UA' = UA \cup \{user \mapsto role\}$$
$$assigned_users' = assigned_users \setminus \{role \mapsto assigned_users(role)\} \cup$$
$$\{role \mapsto (assigned_users(role) \cup \{user\})\} \rhd$$

CreateSsdSet: This command creates a named SSD set of roles and sets the cardinality n of its subsets that cannot have common users. The command is valid if and only if:

—the name of the SSD set is not already in use,
—all the roles in the SSD set are members of the *ROLES* data set,
—n is a natural number greater than or equal to 2 and less than or equal to the cardinality of the SSD role set, and
—the SSD constraint for the new role set is satisfied.

The following schema formally describes this command:

$$CreateSsdSet(set_name: NAME; role_set: 2^{NAMES}; n: \mathsf{N}) \lhd$$
$$set_name \notin SSD; (n \geq 2) \land (n \leq |role_set|); role_set \subseteq ROLES$$
$$\bigcap_{\substack{r \in subset \\ subset \subseteq role_set \\ |subset| = ii}} assigned_users(r) = \emptyset$$
$$SSD' = SSD \cup \{set_name\}$$
$$ssd_set' = ssd_set \cup \{set_name \mapsto role_set\}$$
$$ssd_card' = ssd_card \cup \{set_name \mapsto n\} \rhd$$

AddSsdRoleMember: This command adds a role to a named SSD set of roles. The cardinality associated with the role set remains unchanged. The command is valid if and only if:

—the SSD role set exists,

—the role to be added is a member of the *ROLES* data set but not of a member of the SSD role set, and

—the SSD constraint is satisfied after the addition of the role to the SSD role set.

The following schema formally describes the command.

$$
\begin{array}{l}
AddSsdRoleMember(set_name{:}\,NAME;\ role{:}\,NAME) \lhd \\
\quad set_name \in SSD;\ role \in ROLE;\ role \notin ssd_set(set_name) \\
\quad \bigcap_{\substack{r \in subset \\ subset \subseteq sss_set(set_name) \cup \{role\} \\ |subset| = n}} assigned_users(r) = \emptyset \\
\quad ssd_set' = ssd_set \setminus \{set_name \mapsto ssd_set(set_name)\} \cup \\
\quad \{set_name \mapsto (ssd_set(set_name) \cup \{role\})\} \rhd
\end{array}
$$

DeleteSsdRoleMember: This command removes a role from a named SSD set of roles. The cardinality associated with the role set remains unchanged. The command is valid if and only if:

—the SSD role set exists,

—the role to be removed is a member of the SSD role set, and

—the cardinality associated with the SSD role set is less than the number of elements of the SSD role set.

Note that the SSD constraint should be satisfied after the removal of the role from the SSD role set. The following schema formally describes the command.

$$
\begin{array}{l}
DeleteSsdRoleMember(set_name{:}\,NAME;\ role{:}\,NAME) \lhd \\
\quad set_name \in SSD;\ role \in ssd_set(set_name); \\
\quad\quad ssd_card\ (set_name) < |ssd_set(set_name)| \\
\quad ssd_set' = ssd_set \setminus \{set_name \mapsto ssd_set(set_name)\} \cup \\
\quad\quad \{set_name \mapsto (ssd_set(set_name) \setminus \{role\})\} \rhd
\end{array}
$$

DeleteSsdSet: This command deletes an SSD role set completely. The command is valid if and only if the SSD role set exists. The following schema formally describes the command.

$$
\begin{array}{l}
DeleteSsdSet(set_name{:}\,NAME) \lhd \\
\quad set_name \in SSD; ssd_card' = ssd_card \setminus \{set_name \mapsto ssd_card(set_name)\} \\
\quad ssd_set' = ssd_set \setminus \{set_name \mapsto ssd_set(set_name)\} \\
\quad SSD' = SSD \setminus \{set_name\} \rhd
\end{array}
$$

SetSsdSetCardinality: This command sets the cardinality associated with a given SSD role set. The command is valid if and only if:

—the SSD role set exists,
—the new cardinality is a natural number greater than or equal to 2 and less than or equal to the number of elements of the SSD role set, and
—the SSD constraint is satisfied after setting the new cardinality.

The following schema formally describes the command.

$$SetSsdSetCardinality(set_name: NAME;\ n{:}\mathsf{N}) \lhd$$
$$set_name \in SSD;\ (n \geq 2) \wedge (n \leq |ssd_set(set_name)|)$$
$$\bigcap_{\substack{r \in subset \\ subset \subseteq ssd_set(set_name) \\ |subset|=n}} assigned_users(r) = \emptyset$$
$$ssd_card' = ssd_card \setminus \{set_name \mapsto ssd_card(set_name)\} \cup \{set_name \mapsto n\} \rhd$$

A.3a.2 *System Functions for SSD*

All functions in Section A.1.2 remain valid.

A.3a.3 *Review Functions for SSD*

All functions in Section A.1.3 remain valid. In addition, this section defines the following functions.

SsdRoleSets: This function returns the list of all SSD role sets. The following schema formally describes the function.

$$SsdRoleSets(out\ result{:}2^{NAME}) \lhd result = SSD \rhd$$

SsdRoleSetRoles: This function returns the set of roles of a SSD role set. The function is valid if and only if the role set exists. The following schema formally describes the function.

$$SsdRoleSetRoles(set_name: NAME;\ out\ result{:}2^{ROLES}) \lhd$$
$$set_name \in SSD$$
$$result = ssd_set(set_name) \rhd$$

SsdRoleSetCardinality: This function returns the cardinality associated with a SSD role set. The function is valid if and only if the role set exists. The following schema formally describes the function.

$$SsdRoleSetCardinality(set_name: NAME; out\ result: \mathsf{N}) \lhd$$
$$set_name \in SSD$$
$$result = ssd_card(set_name) \rhd$$

A.3a.4 *Advanced Review Functions for SSD*

All functions in Section A.1.4 remain valid.

A.3b SSD Relations with General Role Hierarchies

A.3b.1 *Administrative Commands for SSD with General Role Hierarchies*

This section redefines the functions AssignUser and AddInheritance of Section A.2a.1, and the functions CreateSsdSet, AddSsdRoleMember, and SetSsdSet

Cardinality of Section A.3a.1. The other functions of Sections A.2a.1 and A.3a.1 remain valid.

AssignUser: The command AssignUser replaces the command with the same name from Core RBAC with Static Separation of Duties. This command assigns a user to a role. The command is valid if and only if:

—the user is a member of the *USERS* data set,
—the role is a member of the *ROLES* data set,
—the user is not already assigned to the role, and
—the SSD constraints are satisfied after assignment.

The data set UA and the function *assigned_users* are updated to reflect the assignment. The following schema formally describes the command.

$$
\begin{array}{l}
AssignUser(user, role: NAME) \lhd \\
\quad user \in USERS; role \in ROLES; (user \mapsto role) \notin UA \\
\quad \forall ssd \in SSD \bullet \bigcap_{\substack{r \in subset \\ subset \subseteq ssd_set(ssd) \\ |subset| = ssd_card(ssd) \\ au = if\ r = role\ \text{then}\ (user)\ \text{else}\ \emptyset}} (authorized_users(r) \cup au) = \emptyset \\
\quad UA' = UA \cup \{user \mapsto role\} \\
\quad assigned_users' = assigned_users \setminus \{role \mapsto assigned_users(role)\} \cup \\
\qquad \{role \mapsto (assigned_users(role) \cup \{user\})\} \rhd
\end{array}
$$

AddInheritance: This command establishes a new immediate inheritance relationship $r_asc \succ\succ r_desc$ between existing roles r_asc, r_desc. The command is valid if and only if:

—r_asc and r_desc are members of the *ROLES* data set,
—r_asc is not an immediate ascendant of r_desc,
—r_desc does not properly inherit r_asc, and
—the SSD constraints are satisfied after establishing the new inheritance.

The following schema uses the notations

$$
\begin{array}{c}
\geq\ ==\ \succeq \\
\gg\ ==\ \succ\succ
\end{array}
$$

to formally describes the command

$$
\begin{array}{l}
AddInheritance(r_asc, r_desc: NAME) \lhd \\
\quad r_asc \in ROLES; r_desc \in ROLES; \neg(r_asc \gg r_desc); \neg(r_desc \geq r_asc) \\
\quad \forall ssd \in SSD \bullet \bigcap_{\substack{r \in subset \\ subset \subseteq ssd_set(ssd) \\ |subset| = ssd_card(ssd) \\ au = \text{if}\ r = r_desc\ \text{then}\ authorized_user(r_asc)\ \text{else}\ \emptyset}} (authorized_users(r) \cup au) = \emptyset \\
\quad \geq' = \geq \cup \{r,\ q: ROLES \mid r \geq r_asc \wedge r_desc \geq q \bullet r \mapsto q\} \rhd
\end{array}
$$

CreateSsdSet: This command creates a named SSD set of roles and sets the associated cardinality n of its subsets that cannot have common users. The command is valid if and only if:

—the name of the SSD set is not already in use,

—all the roles in the SSD set are members of the *ROLES* data set,

—n is a natural number greater than or equal to 2 and less than or equal to the cardinality of the SSD role set, and

—the SSD constraint for the new role set is satisfied.

The following schema formally describes this command.

$$CreateSsdSet(set_name{:}\,NAME; role_set{:}\,2^{NAMES}; n{:}\,\mathsf{N}) \lhd$$
$$set_name \notin SSD; (n \geq 2) \wedge (n \leq |role_set|); role_set \subseteq ROLES$$
$$\bigcap_{\substack{r \in subset \\ subset \subseteq role_set \\ |subset| = n}} authorized_users(r) = \emptyset$$
$$SSD' = SSD \cup \{set_name\}$$
$$ssd_set' = ssd_set \cup \{set_name \mapsto role_set\}$$
$$ssd_card' = ssd_card \cup \{set_name \mapsto n\} \rhd$$

AddSsdRoleMember: This command adds a role to a named SSD set of roles. The cardinality associated with the role set remains unchanged. The command is valid if and only if:

—the SSD role set exists,

—the role to be added is a member of the *ROLES* data set but not of a member of the SSD role set, and

—the SSD constraint is satisfied after the addition of the role to the SSD role set.

The following schema formally describes the command.

$$AddSsdRoleMember(set_name{:}\,NAME;\ role{:}\,NAME) \lhd$$
$$set_name \in SSD; role \in ROLES; role \notin ssd_set(set_name)$$
$$\bigcap_{\substack{r \in subset \\ subset \subseteq ssd_set(set_name) \cup \{role\} \\ |subset| = n}} authorized_users(r) = \emptyset$$
$$ssd_set' = ssd_set \setminus \{set_name \mapsto ssd_set(set_name)\} \cup$$
$$\{set_name \mapsto (ssd_set(set_name) \cup \{role\})\} \rhd$$

SetSsdSetCardinality: This command sets the cardinality associated with a given SSD role set. The command is valid if and only if:

—the SSD role set exists,

—the new cardinality is a natural number greater than or equal to 2 and less than or equal to the number of elements of the SSD role set, and

—the SSD constraint is satisfied after setting the new cardinality.

The following schema formally describes the command.

$$\begin{array}{l} SetSsdSetCardinality(set_name{:}NAME; n{:}\mathbb{N}) \lhd \\ \quad set_name \in SSD; (n \geq 2) \wedge (n \leq |ssd_set(set_name)|) \\ \quad \bigcap_{\substack{r \in subset \\ subset \subseteq ssd_set(set_name) \\ |subset|=n}} authorized_users(r) = \emptyset \\ \quad ssd_card' = ssd_card \setminus \{set_name \mapsto ssd_card(set_name)\} \cup \{set_name \mapsto n\} \rhd \end{array}$$

A.3b.2 *System Functions for SSD with General Role Hierarchies*

All functions of Section A.2a.2 remain valid.

A.3b.3 *Review Functions for SSD with General Role Hierarchies*

All functions of Sections A.2a.3 and A.3a.3 remain valid.

A.3b.4 *Advanced Review Functions for SSD with General Role Hierarchies*

All functions of Section A.2a.4 remain valid.

A.3c SSD Relations with Limited Role Hierarchies

A.3c.1 *Administrative Commands for SSD with Limited Role Hierarchies*

This section redefines the function AddInheritance of Section A.3b.1. All other functions of Section A.3b.1 remain valid.

AddInheritance: This command establishes a new immediate inheritance relationship $r_asc \succ\succ r_desc$ between existing roles r_asc, r_desc. The command is valid if and only if r_asc and r_desc are members of the *ROLES* data set, r_asc has no descendants, and r_desc does not properly inherit r_asc (in order to avoid cycle creation). The following schema uses the notations

$$\begin{array}{c} \geq\ ==\ \succeq \\ \gg\ ==\ \succ\succ \end{array}$$

to formally describe the command

$$\begin{array}{l} AddInheritance(r_asc, r_desc{:}NAME) \lhd \\ \quad r_asc \in ROLES; r_desc \in ROLES; \forall r \in ROLES \bullet \\ \qquad \neg(r_asc \gg r); \neg(r_desc \geq r_asc) \\ \quad \forall ssd \in SSD \bullet \bigcap_{\substack{r \in subset \\ subset \subseteq ssd_set(ssd) \\ |subset|=ssd_card(ssd) \\ au = \text{if } r = r_desc \text{ then } authorized_users(r_asc) \text{ else } \emptyset}} (authorized_users(r) \cup au) = \emptyset \\ \quad \geq' = \geq \cup \{r, q{:}ROLES | r \geq r_asc \wedge r_desc \geq q \bullet r \mapsto q\} \rhd \end{array}$$

A.3c.2 *System Functions for SSD with Limited Role Hierarchies*

All functions of Section A.2a.2 remain valid.

A.3c.3 *Review Functions for SSD with Limited Role Hierarchies*

All functions of Sections A.2a.3 and A.3a.3 remain valid.

A.3c.4 *Advanced Review Functions for SSD with Limited Role Hierarchies*

All functions of Sections A.2a.4 remain valid.

A.4 Requirements for Dynamic Separation of Duties (DSD) Relations

The Dynamic Separation of Duty property, as defined in the model, uses a collection DSD of pairs of a role set and an associated cardinality. We define the new data type *DSD*, which in an implementation could be the set of names used to identify the pairs in the collection.

The functions *dsd_set* and, respectively, *dsd_card* are used to obtain the role set and the associated cardinality from each DSD pair:

$$dsd_set: DSD \rightarrow 2^{ROLES}$$

$$dsd_card: DSD \rightarrow \mathsf{N}$$

$$\forall dsd \in SSD \bullet dsd_card(dsd) \geq 2 \wedge dsd_card(dsd) \leq |dsd_set(dsd)|$$

4.4a DSD Relations

A.4a.1 *Administrative Commands for DSD Relations*

All functions of Section A.1.1 remain valid. In addition, this section defines the following functions.

CreateDsdSet: This command creates a named DSD set of roles and sets an associated cardinality n. The DSD constraint stipulates that the DSD role set cannot contain n or more roles simultaneously active in the same session. The command is valid if and only if:

—the name of the DSD set is not already in use,
—all the roles in the DSD set are members of the *ROLES* data set,
—n is a natural number greater than or equal to 2 and less than or equal to the cardinality of the DSD role set, and
—the DSD constraint for the new role set is satisfied.

The following schema formally describes this command.

$$CreateDsdSet(set_name: NAME; role_set: 2^{NAMES}; n: \mathsf{N}) \lhd$$
$$set_name \notin DSD; (n \geq 2) \wedge (n \leq |role_set|); role_set \subseteq ROLES$$
$$\forall s: SESSIONS; role_subet: 2^{role_set} \bullet role_subset \subseteq$$
$$session_roles(s) \Rightarrow |role_subset| \leq n$$
$$DSD' = DSD \cup \{set_name\}$$
$$dsd_set' = dsd_set \cup \{set_name \mapsto role_set\}$$
$$dsd_card' = dsd_card \cup \{set_name \mapsto n\} \rhd$$

AddDsdRoleMember: This command adds a role to a named DSD set of roles. The cardinality associated with the role set remains unchanged. The command is valid if and only if:

—the DSD role set exists,

—the role to be added is a member of the *ROLES* data set but not a member of the DSD role set, and

—the DSD constraint is satisfied after the addition of the role to the DSD role set.

The following schema formally describes the command.

$$
\begin{array}{l}
AddDsdRoleMember(set_name: NAME; role: NAME) \lhd \\
\quad set_name \in DSD; role \in ROLES; role \notin dsd_set(set_name) \\
\quad \forall s: SESSIONS; role_subset: 2^{dsd_set(set_name) \cup \{role\}} \bullet \\
\quad\quad role_subset \subseteq session_roles(s) \Rightarrow |role_subset| < dsd_card(set_name) \\
\quad dsd_set' = dsd_set \setminus \{set_name \mapsto dsd_set(set_name)\} \cup \\
\quad\quad \{set_name \mapsto (dsd_set(set_name) \cup \{role\})\} \rhd
\end{array}
$$

DeleteDsdRoleMember: This command removes a role from a named DSD set of roles. The cardinality associated with the role set remains unchanged. The command is valid if and only if:

—the DSD role set exists,

—the role to be removed is a member of the DSD role set, and

—the cardinality associated with the DSD role set is less than the number of elements of the DSD role set.

Note that the DSD constraint should be satisfied after the removal of the role from the DSD role set. The following schema formally describes the command.

$$
\begin{array}{l}
DeleteDsdRoleMember(set_name: NAME; role: NAME) \lhd \\
\quad set_name \in DSD; role \in dsd_set(set_name); \\
\quad\quad dsd_card(set_name) < |dsd_set(set_name)| \\
\quad dsd_set' = dsd_set \setminus \{set_name \mapsto dsd_set(set_name)\} \cup \\
\quad\quad \{set_name \mapsto (dsd_set(set_name) \setminus \{role\})\} \rhd
\end{array}
$$

DeleteDsdSet: This command deletes a DSD role set completely. The command is valid if and only if the DSD role set exists. The following schema formally describes the command.

$$
\begin{array}{l}
DeleteDsdSet(set_name: NAME) \\
\{ \\
\quad set_name \in DSD \\
\quad dsd_card' = dsd_card \setminus \{set_name \mapsto dsd_card(set_name)\} \\
\quad dsd_set' = dsd_set \setminus \{set_name \mapsto dsd_set(set_name)\} \\
\quad DSD' = DSD \setminus \{set_name\} \\
\}
\end{array}
$$

SetDsdSetCardinality: This command sets the cardinality associated with a given DSD role set. The command is valid if and only if:

—the DSD role set exists,

—the new cardinality is a natural number greater than or equal to 2 and less than or equal to the number of elements of the DSD role set, and

—the DSD constraint is satisfied after setting the new cardinality.

The following schema formally describes the command.

$$
\begin{array}{l}
SetDsdSetCardinality(set_name: NAME; n{:}\mathsf{N}) \lhd \\
\quad set_name \in DSD; (n \geq 2) \wedge (n \leq |dsd_set(set_name)|) \\
\quad \forall s: SESSIONS; role_subset{:}2^{dsd_set(set_name)} \bullet \\
\qquad role_subset \subseteq session_roles(s) \Rightarrow |role_subset| < n \\
\quad dsd_card' = dsd_card \backslash \{set_name \mapsto dsd_card(set_name)\} \cup \\
\qquad \{set_name \mapsto n\} \rhd
\end{array}
$$

A.4a.2 *System Functions for DSD Relations*

This section redefines the functions CreateSession and AddActiveRole of Section A.1.2. The other functions of Section A.1.2 remain valid.

CreateSession: This function creates a new session whose owner is the user *user* and a given active role set. The function is valid if and only if:

—the user is a member of the *USERS* data set,

—the session's active role set is a subset of the roles assigned to the session's owner, and

—the session's active role set satisfies the DSD constraints.

The following schema formally describes the function. The *session* parameter, which identifies the new session, is actually generated by the underlying system.

$$
\begin{array}{l}
CreateSession(user: NAME;\ ars{:}2^{NAME}; session: NAME) \lhd \\
\quad user \in USERS; ars \subseteq \{r: ROLES | (user \mapsto r) \in UA\}; session \notin SESSIONS \\
\quad \forall dset: DSD; rset{:}2^{NAME} \bullet \\
\qquad rset \subseteq dsd_set(dset) \wedge rset \subseteq ars \Rightarrow |rset| < dsd_card(dset) \\
\quad SESSIONS' = SESSIONS \cup \{session\} \\
\quad user_sessions' = user_sessions \backslash \{user \mapsto user_sessions(user)\} \cup \\
\qquad \{user \mapsto (user_sessions(user) \cup \{session\})\} \\
\quad session_roles' = session_roles \cup \{session \mapsto ars\} \rhd
\end{array}
$$

AddActiveRole: This function adds a role as an active role of a session whose owner is a given user. The function is valid if and only if:

—the user is a member of the *USERS* data set,

—the role is a member of the *ROLES* data set,

—the session identifier is a member of the *SESSIONS* data set,

—the role is assigned to the user,

—the old active role set completed with the role to be activated satisfies the DSD constraints, and
—the session is owned by that user.

The following schema formally describes the function.

$$
\begin{array}{l}
AddActiveRole(user, session, role{:}NAME) \lhd \\
\quad user \in USERS; session \in SESSIONS; role \in ROLES; \\
\quad\quad session \in user_sessions(user) \\
\quad user \in assigned_user(role); role \notin session_roles(session) \\
\quad \forall dset{:}\ DSD; rset{:}2^{NAME} \bullet \\
\quad\quad rset \subseteq dsd_set(dset) \wedge rset \subseteq session_roles(sessions) \cup \{role\} \Rightarrow \\
\quad\quad\quad |rset| < dsd_card(dset) \\
\quad session_roles' = session_roles \backslash \{session \mapsto session_roles(session)\} \cup \\
\quad\quad \{session \mapsto (session_roles(session) \cup \{role\})\} \rhd
\end{array}
$$

A.4a.3 *Review Functions for DSD Relations*

All functions of Sections A.1.3 remain valid. In addition, this section defines new specific functions.

DsdRoleSets: This function returns the list of all DSD role sets. The following schema formally describes the function.

$$DsdRoleSets(out\ result{:}2^{NAME}) \lhd result = DSD \rhd$$

DsdRoleSetRoles: This function returns the set of roles of a DSD role set. The function is valid if and only if the role set exists. The following schema formally describes the function.

$$
\begin{array}{l}
DsdRoleSetRoles(set_name{:}\ NAME; out\ result{:}2^{ROLES}) \lhd \\
\quad set_name \in DSD \\
\quad result = dsd_set(set_name) \rhd
\end{array}
$$

DsdRoleSetCardinality: This function returns the cardinality associated with a DSD role set. The function is valid if and only if the role set exists. The following schema formally describes the function.

$$
\begin{array}{l}
DsdRoleSetCardinality(set_name{:}NAME; out\ result{:}\mathsf{N}) \lhd \\
\quad set_name \in DSD \\
\quad result = dsd_card(set_name) \rhd
\end{array}
$$

A.4a.4 *Advanced Review Functions for DSD Relations*

All functions of Sections A.1.4 remain valid.

A.4b DSD Relations with Role Hierarchies

A.4b.1 *Administrative Commands for DSD Relations with General Role Hierarchies*

All functions of Sections A.4a.1 and A.2a.1 remain valid.

A.4b.2 *System Functions for DSD Relations with General Role Hierarchies*

This section redefines the functions CreateSession and AddActiveRole of Section A.1.2 (or A.2a.2). All other functions of Section A.1.2 remain valid.

CreateSession: This function creates a new session whose owner is the user *user* and a given active role set. The function is valid if and only if:

—the user is a member of the *USERS* data set,

—the session's active role set is a subset of the roles authorized for the session's owner, and

—the session's active role set satisfies the DSD constraints.

The underlying system generates a new session identifier, which is included in the *SESSIONS* data set. The following schema formally describes the function.

$$
\begin{array}{l}
CreateSession(user{:}\ NAME; ars{:}2^{NAME}; session{:}\ NAME) \lhd \\
\quad user \in USERS; ars \subseteq \{r,\ q{:}\ ROLES | (user \mapsto q) \in UA \wedge q \geq r \bullet r\}; \\
\quad\quad session \notin SESSIONS \\
\quad \forall dset{:}\ DSD; rset{:}2^{NAME} \bullet \\
\quad\quad rset \subseteq dsd_set(dset) \wedge rset \subseteq ars \Rightarrow |rset| < dsd_card(dset) \\
\quad SESSIONS' = SESSIONS \cup \{session\} \\
\quad user_sessions' = user_sessions \backslash \{user \mapsto user_sessions(user)\} \cup \\
\quad\quad \{user \mapsto (user_sessions(user) \cup \{session\})\} \\
\quad session_roles' = session_roles \cup \{session \mapsto ars\} \rhd
\end{array}
$$

AddActiveRole: This function adds a role as an active role of a session whose owner is a given user. The function is valid if and only if:

—the user is a member of the *USERS* data set,

—the role is a member of the *ROLES* data set,

—the session identifier is a member of the *SESSIONS* data set,

—the role is authorized for that user,

—the old active role set completed with the role to be activated satisfies the DSD constraints, and

—the session is owned by that user.

The following schema formally describes the function.

$$
\begin{array}{l}
AddActiveRole(user, session, role{:}\ NAME) \lhd \\
\quad user \in USERS; session \in SESSIONS; role \in ROLES; \\
\quad\quad session \in user_sessions(user) \\
\quad user \in authorized_users(role); role \notin session_roles(session) \\
\quad \forall dset{:}\ DSD; rset{:}2^{NAME} \bullet \\
\quad\quad rset \subseteq dsd_set(dset) \wedge rset \subseteq session_roles(session) \cup \{role\} \Rightarrow \\
\quad\quad\quad |rset| < dsd_card(dset)
\end{array}
$$

$$session_roles' = session_roles \setminus \{session \mapsto session_roles(session)\} \cup \{session \mapsto (session_roles(session) \cup \{role\})\} \triangleright$$

A.4b.3 *Review Functions for DSD Relations with General Role Hierarchies*

All functions of Sections A.4a.3 and A.2a.3 remain valid.

A.4b.3 *Advanced Review Functions for DSD Relations with General Role Hierarchies*

All functions of Section A.2a.4 remain valid.

A.4c DSD Relations with Limited Role Hierarchies

A.4c.1 *Administrative Commands for DSD Relations with Limited Role Hierarchies*

All functions of Sections A.2b.1 and A.4a.1 remain valid.

A.4c.2 *System Functions for DSD Relations with Limited Role Hierarchies*

All functions of Section A.4b.2 remain valid.

A.4c.3 *Review Functions for DSD Relations with Limited Role Hierarchies*

All functions of Section A.4b.3 remain valid.

A.4c.4 *Advanced Review Functions for DSD Relations with Limited Role Hierarchies*

All functions of Section A.2a.4 remain valid.

REFERENCES

AHN, G. AND SANDHU, R. 2000. Role-based authorization constraints specification. *ACM Trans. Inf. Syst. Sec. 3*, 4 (Nov.).

BALDWIN, R. W. 1990. Naming and grouping privileges to simplify security management in large databases. In *Proceedings of the Symposium on Security and Privacy*, IEEE Press, Los Alamitos, Calif., 116–132.

BELL, D. AND LAPADULA. 1976. Secure computer systems: Unified exposition and MULTICS. Tech. Rep. ESD-TR-75-306, The MITRE Corporation, Bedford, Mass., March.

BERTINO, E., BONATTI, P., AND FERRARI, E. 2000. TRBAC: A temporal role-based access control model. In *Proceedings of the Fifth ACM Workshop on Role Based Access Control*, 21–30.

BREWER, D. AND NASH, M. 1989. The Chinese wall security policy. In *Proceedings of the Symposium on Security and Privacy*, IEEE Press, Los Alamitos, Calif., 215–228.

CHANDRAMOULI, R. AND SANDHU, R. 1998. Role-based access control features in commercial database management systems. In *Proceedings of the NIST–NSA National (USA) Computer Security Conference*, 503–511.

CLARK, D. AND WILSON, D. 1987. A comparison of commercial and military computer security policies. In *proceedings of the Symposium on Security and Privacy*, IEEE Press, Los Alamitos, Calif., 184–194.

FADEN, G. 1999. Rbac in Unix administration. In *Proceedings of the Fourth ACM Workshop on Role Based Access Control*, 95–101.

FEINSTEIN, H. 1996. Final report: NIST small business innovative research (SBIR) grant: Role based access control: phase 2. SETA Corp., October.

FERRAIOLO, D. AND KUHN, R. 1992. Role-based access control. In *Proceedings of the NIST–NSA National (USA) Computer Security Conference*, 554–563.

FERRAIOLO, D., BARKLEY, J., AND KUHN, R. 1999. A role-based access control model and reference implementation within a corporate internet. *ACM Trans. Inf. Syst. Sec. 2*, 1.

FERRAIOLO, D., CUGINI, J., AND KUHN, R. 1995. Role-based access control: Features and motivations. In *Proceedings of the Annual Computer Security Applications Conference*, IEEE Press, Los Alamitos, Calif.

FERRAIOLO, D., GILBERT, D., AND LYNCH, N. 1993. An examination of federal and commercial access control policy needs. In *Proceedings of the NIST–NSA National (USA) Computer Security Conference*, 107–116.

GAVRILA, S. AND BARKLEY, J. 1998. Formal specification for RBAC user/role and role relationship management. In *Proceedings of the Third ACM Workshop on Role Based Access Control*, 81–90.

GIURI, L. AND IGLIO, P. 1996. A formal model for role based access control with constraints. In *Proceedings of the Computer Security Foundations Workshop*, IEEE Press, Los Alamitos, Calif., 136–145.

GLIGOR, V. D., GAVRILA, S. I., AND FERRAIOLO, D. F. 1998. On the formal definition of separation-of-duty policies and their composition. In *Proceedings of the Symposium on Security and Privacy*, IEEE Press, Los Alamitos, Calif.

HUANG, W. AND ATLURI, V. 1999. A secure web-based workflow management system. In *Proceedings of the Fourth ACM Workshop on Role Based Access Control*, 83–84.

JAEQER, T. 1999. On the increased importance of constraints. In *Proceedings of the Fourth ACM Workshop on Role-Based Access Control* (Oct.), 33–42.

JAEGER, T. AND TIDSWELL, J. 2000. Rebuttal to the NIST RBAC model proposal. In *proceedings of the Fifth ACM Workshop on Role-Based Access Control* (Berlin, July), 65–66.

JOSHI, J. B. D., AREF, W. G., GHAFOOR, A., AND SPAFFORD, E.H. 2001a. Security models for web-based applications. Commun. *ACM, 44*, 2, Feb. 38–44.

JOSHI, J., GHAFOOR, A., AREF, W. G., AND SPAFFORD, E. H. 2001b. Digital government security infrastructure design challenges. *IEEE Comput. 33*, 2, Feb. 66–72.

KUHN, D. R. 1998. Role based access control on MLS systems without kernel changes. In *Proceedings of the ACM Workshop on Role Based Access Control* (Oct. 22–23), 25–32.

KUHN, R. 1997. Mutual exclusion as a means of implementing separation of duty requirements in role based access control systems. In *Proceedings of the Second ACM Workshop on Role Based Access Control*, 23–30.

LAMPSON, B. 1974. Protection. *ACM Oper. Syst. Rev. 8*, 1, 18–24.

MCCOLLUM, C., MESSING, J., AND NOTARGIACOMO, L. 1990. Beyond the pale of MAC and DAC—Defining new forms of access control. In *Proceedings of the Symposium on Security and Privacy,* IEEE Press, Los Alamitos, Calif., 190–900.

MOFFETT., J. D. 1998. Control principles and role hierarchies. In *Proceedings of the Third ACM Workshop on Role-Based Access Control* (Fairfax, V., Oct. 22–23), 63–69.

NYANCHAMA, M. AND OSBORN, S. 1994. Access rights administration in role-based security systems. In *Database Security, VIII: Status and Prospects,* J. Biskup, M. Morgenstern, and C. E. Landwehr, Eds., North-Holland, 37–56.

NYANCHAMA, M. AND OSBORN, S. 1999. The graph model and conflicts of interest. *ACM Trans. Inf. Syst. Sec. 2*, 1.

OSBORN, S., SANDHU, R., AND MUNAWER, Q. 2000. Configuring role-based access control to enforce mandatory and discretionary access control policies. *ACM Trans. Inf. Syst. Sec. 3*, 2.

SANDHU, R. AND BHAMIDIPATI, V. 1997. Role-based administration of user-role assignment: The URA97 model and its oracle implementation. *J. Compu. Sec. 7*.

SANDHU, R. 1998a. Role activation hierarchies. In *Proceedings of the Third ACM Workshop on Role-Based Access Control* (Fairfax, V., Oct. 22–23), 33–40.

SANDHU, R. 1998b. Role-based access control. In *Advances in Computers*, vol. 46, M. Zelkowitz Eds. Academic, 237–286.

SANDHU, R. 1988. Transaction control expressions for separation of duties. In *Proceedings of the Fourth Aerospace Computer Security Applications Conference* (Orlando, Fla.). IEEE Computer Society Press, Dec. Los Alamitos, Calif., 282–286.

SANDHU, R., BHAMIDIPATI, V., AND MUNAWER, Q. 1999. The ARBAC97 model for role-based administration of roles. *ACM Trans. Inf. Sys. Sec. 2*, 1, (Feb.), 105–135.

Sandhu, R., Coyne, E., Feinstein, H., and Youman, C. 1996. Role-based access control models. *IEEE Comput.*, *29*, (2), (Feb).

Sandhu, R., Ferraiolo, D., and Kuhn, R. 2000. The NIST model for role-based access control: Towards a unified standard. In *Proceedings of the Fifth ACM Workshop on Role-Based Access Control* (Berlin, July), 47–63.

Simon, R. and Zurko, R. 1997. Separation of duty in role based access control environments. In *Proceedings of New Security Paradigms Workshop*, (Sept.).

Smith, C., Coyne, E., Youman, C., and Ganta, S. 1996. Market analysis report: NIST small business innovative research (SBIR) grant: Role based access control: Phase 2. A marketing survey of civil federal government organizations to determine the need for role-based access control security product, SETA Corp., July.

Thomsen, D. J. 1991. Role-based application design and enforcement. In *Database Security, IV*: *Status and Prospects*, S. Jajodia and C. E. Landwehr, Eds., North-Holland, 151–168.

Ting, T. C., Demurjian, S. A., and Hu, M. Y. 1992. Requirements capabilities and functionalities of user-role based security for an object-oriented design model. In *Database Security, IV*: *Status and Prospects*, S. Jajodia and C. E. Landwehr, Eds., North-Holland, 275–296.

Received November 2000; revised July 2000; accepted July 2001