A Proposed Standard for Role-Based Access Control

David F. Ferraiolo
National Institute of Standards and Technology

Ravi Sandhu
George Mason University

Serban Gavrila
VDG Incorporated

D. Richard Kuhn and Ramaswamy Chandramouli
National Institute of Standards and Technology

December 18, 2000

Abstract

This paper describes a proposed standard for role-based access control (RBAC). RBAC is a proven technology for large-scale authorization. However, lack of a widely accepted model results in uncertainty and confusion about its utility and meaning. The standard proposed here seeks to resolve this situation by unifying ideas from prior RBAC models, commercial products, and research prototypes. It is intended to serve as a foundation for product development, evaluation, and procurement specification. RBAC is a rich and open-ended technology, which is evolving as users, researchers and vendors gain experience with its application. Features and components in this standard should be regarded as a core that may be enhanced by developers to meet the needs of customers.

This document does not attempt to extend RBAC features beyond those that have achieved acceptance in the commercial marketplace and research community, but instead focuses on defining fundamental components of RBAC. It is organized into two main parts – the RBAC Reference Model and the RBAC Requirements Specification. The reference model defines the scope of features and provides a consistent vocabulary in support of the requirements specification. The requirements specification defines requirements for administrative operations for the creation and maintenance of RBAC sets and relations, for performing administrative queries and review, as well as for specifying system level functions in support of session attribute assignment and access control decisions.

1 INTRODUCTION

In recent years, vendors have begun implementing role-based access control (RBAC) features in their database management system, 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 [FK92; NO94; FCK95; GI96; SCFY96], 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 has conducted and sponsored market analysis [FGL93; SCYG96], developed prototype implementations [FBK99], and sponsored external research [Fei96]. 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. Although our understanding of RBAC has dramatically advanced, much of the RBAC work has been independently performed without any attempt at standardizing salient RBAC features.

A first effort at defining a consensus standard for RBAC was proposed at the 2000 ACM Workshop on Role Based Access Control [SFK00]. Published comments on this earlier document [JT00] assisted in developing the reference model and requirements specification proposed in this paper. Panel session discussions at the ACM Workshop also contributed toward refining the initial model into the proposals in this paper.

RBAC is a technology that is both new and old. 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 mechanism 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 [Bal90; Tho91; TDH92], and separation of duty concepts described in earlier papers [CW87; San88; BN89]. 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 [FK92]. These constructs are common to the early formal definitions of RBAC proposed by various authors [FCK95; SCFY96; NO94]. A comprehensive framework for RBAC models was defined by Sandhu et al. [SCFY96], and expanded in subsequent publications [San98; SBM99; AS00; OSM00].

Although existing RBAC models and implementations are relatively similar on fundamental RBAC concepts, they differ in significant areas. 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, there is no consensus 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. Recognizing 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.

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 model components provide a standard vocabulary for defining a broad range of RBAC features. In arriving at the scope of RBAC features the authors of this document applied two selection criteria. First, the features must be well understood and well represented within the RBAC literature and well-accepted RBAC models. Second, RBAC features should be known to be viable in that there exist at least one example commercial or reference implementation for each feature.

Both the RBAC Reference Model and the RBAC Requirements Specification are organized into the four components given below:

- Basic RBAC
- Hierarchical RBAC
- Static Separation of Duty Relations
- Dynamic Separation of Duty Relations

Standardization over a stable set of RBAC features is expected to provide a multitude of benefits. These benefits include a common set of benchmarks for vendors, who are already developing RBAC mechanisms, to use in their product specifications and in demonstrating compliance in their marketing efforts. It will give IT consumers, who are the principle beneficiary of RBAC technology, a basis for the creation of uniform acquisition specifications. In addition, an RBAC standard will provide researchers with a basis for devising new and innovative models and techniques, and will allow standardization bodies the ability to develop architecture-specific APIs. These APIs will lead to the development of new and innovative authorization management tools by guaranteeing interoperability and portability over a broad spectrum of products.

The rest of this paper is organized as follows. Section 2 gives an overview of the four RBAC components that provide the basis for modeling features and are used in the specification of a family of RBAC requirements. Section 3, the RBAC reference model provides a rigorous definition of a collection of relations on sets of RBAC basic elements. Section 4, provides an overview of RBAC requirements 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 of RBAC requirement components in defining relevant requirement specification. Section 6 concludes the paper.

2 COMPONENT OVERVIEW

This RBAC standard is organized into two main parts—the RBAC Reference Model and the RBAC Requirements Specifications. The RBAC Reference Model provides a rigorous definition of RBAC sets and relations. The RBAC Requirements Specification define 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 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 model and requirements 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 description 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. Finally, core RBAC requires that users can 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.

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 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 [FCK96; NO99; SCFY96]. 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*, that is, to enforce constraints on the assignment of users to 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 (SSD) that defines mutually disjoint user assignments with respect to sets of roles [Kuh97; GI96]. 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. [GGF98] and Simon and Zurko [SZ97] and Sandhu [San99] have identified SSD relations to include constraints on users, operations and objects as

well as combinations thereof. Some authors [AS00; Jaeger00] 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 μ2, 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 in 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 operational flexibility.

3 THE ROLE-BASED ACCESS CONTROL REFERENCE MODEL

The 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 relation between roles, whereby senor 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 following sub-components:

- a set of basic element sets

- a set of RBAC relations involving those element sets (containing subsets of Cartesian products denoting valid assignments)
- a set of Mapping Functions which 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 person in this paper. 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. 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 it 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, 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.

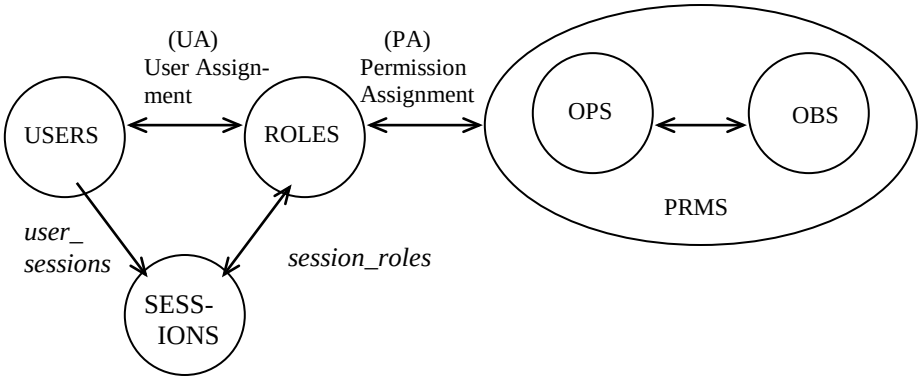

Figure 1: Core RBAC

Each session is a mapping of one user to possibly many roles, i.e., 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 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.
- $PA \subseteq PERMS \times ROLES$, 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:
  $assigned\_permissions(r) = \{p \in PRMS \mid (p, r) \in PA\}$
- $Op(p: PRMS) \rightarrow \{op \subseteq OPS\}$, the permission to operation mapping, which gives the set of operations associated with permission *p*.
- $Ob(p: PRMS) \rightarrow \{ob \subseteq OBS\}$, 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 subjects.

- *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 [FCK95; SCFY96; NO99; Moffett98] and are often included as part of RBAC product offerings [CR98]. 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 [NO99]: 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 [NO99] as well as alternate interpretations [Kuh98; San98]. We have adopted the most widely used definition. For some distributed RBAC implementations, role permissions are not managed centrally, while the role hierarchies are. For these

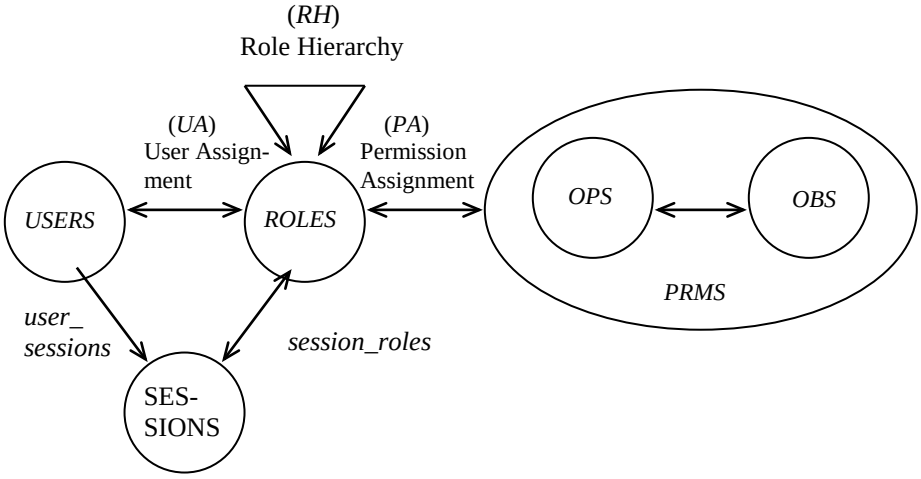

Figure 2: Hierarchical RBAC

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.

This standard recognizes two types of role hierarchies—general role hierarchies 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 descendent). Examples of possible hierarchical role structures are shown in Figure 3.

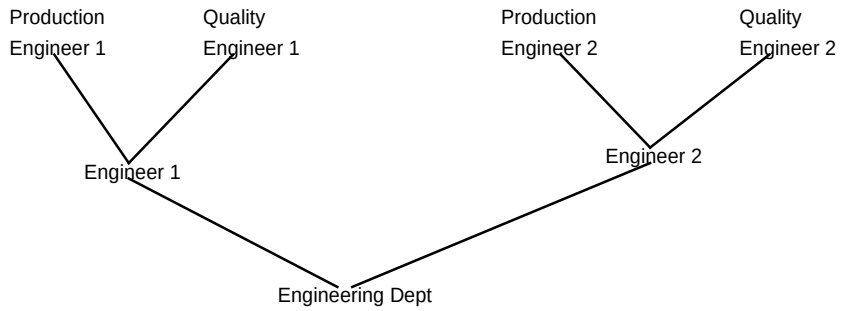

Figure 3a. Tree

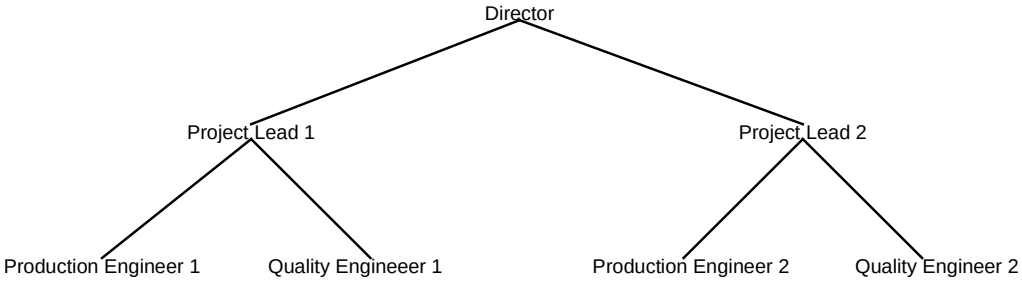

Figure 3b. Inverted Tree

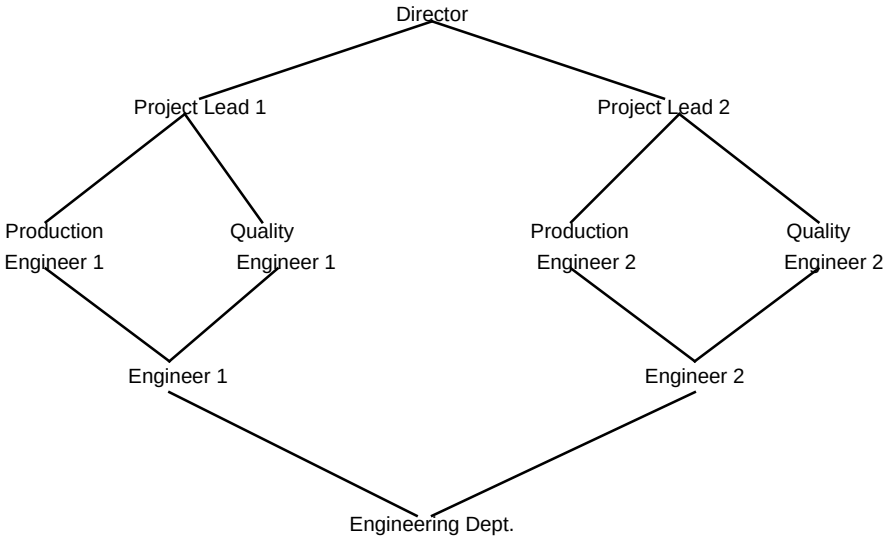

Figure 3c. Lattice

Figure 3. Example Role Hierarchies

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 o, where r_1 o 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, i.e, r_1 o $r_2 \Rightarrow authorized\_permissions(r_2) \subseteq authorized\_permissions(r_1)$.

- $authorized\_users(r: ROLES) \rightarrow 2^{USERS}$, the mapping of role *r* onto a set of users in the presence of a role hierarchy. Formally:
  $authorized\_users(r) = \{u \in USERS \mid r' \text{ o } r, (u, r') \in UA\}$

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

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 two important hierarchy properties. The first is ability to compose a role from multiple subordinate role (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 provides uniform treatment of user/role assignment relations and role/role inheritance relations. Users can be included in the role hierarchy, using the same relation o to denote the user assignment to roles, as well as well as permission inheritance from a role to its assigned users.

Roles in a limited role hierarchy are restricted to a single *immediate descendent.* 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 $r_1 \vee\vee r_2$, if r_1 o 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_1 o r_3 o 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 descendents of the general role hierarchy.

Definition 2b Limited Role Hierarchies:

- Definition 2a with the following limitation:
  $\forall r, r_1, r_2 \in ROLES, r \vee\vee r_1 \wedge r \vee\vee r_2 \Rightarrow r_1 = r_2$

Figure 4 illustrates properties of a general role hierarchy as a Hasse Diagram. Nodes in the graph represent the roles of the hierarchy and there is a directed line segment (arrow) drawn from r_1 to r_2 whenever r_1 is an immediate descendent of r_2. We write $r_1 \rightarrow r_2$ if $r_1 \vee\vee r_2$. In the graph thus created, r_x o r_y if and only if there is a directed path (sequence of arrows) from r_x to r_y. Also, there are no (directed) cycles in the graph of *RH* since the order relation is anti-symmetric and transitive. Usually, we represent the graph with the arcs corresponding to the inheritance relation o oriented top-down. Thus we can say the user membership is inherited top-down, and the role permissions are inherited bottom-up.

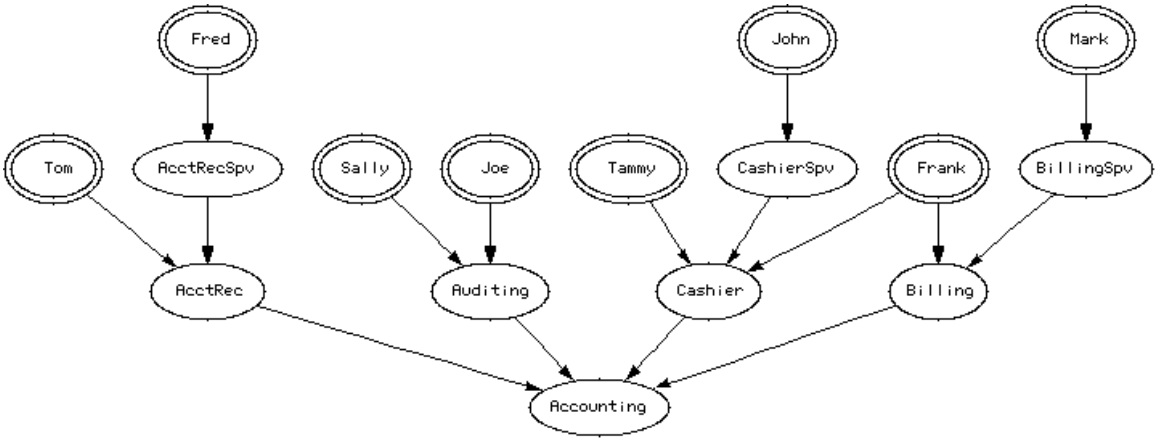

Figure 4. Accounting Roles

In the role graph of figure 4, where the users are represented by double ellipses and the 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, separation of duty has long been recognized for its wide application in business, industry, and government [CW89; CW87]. Its purpose is to ensure that failures of omission or commission within an organization are 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. 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 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.

Static constraints can take on a wide variety of forms. A common example is that of Static Separation of Duty (SSD) 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 [FCK96; Kuh97; SZ97; GGF98; GI96]. For example, Gligor, et al. has formally defined four other types of static separation of duty policies. The static constraints defined in this model are limited to those relations that that place restrictions on sets of roles and in particular on their ability to form *UA* 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.

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 first aspect being the size of the set of roles in the SSD and the second being 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 indication 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 [GB98, FBK99]. 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.

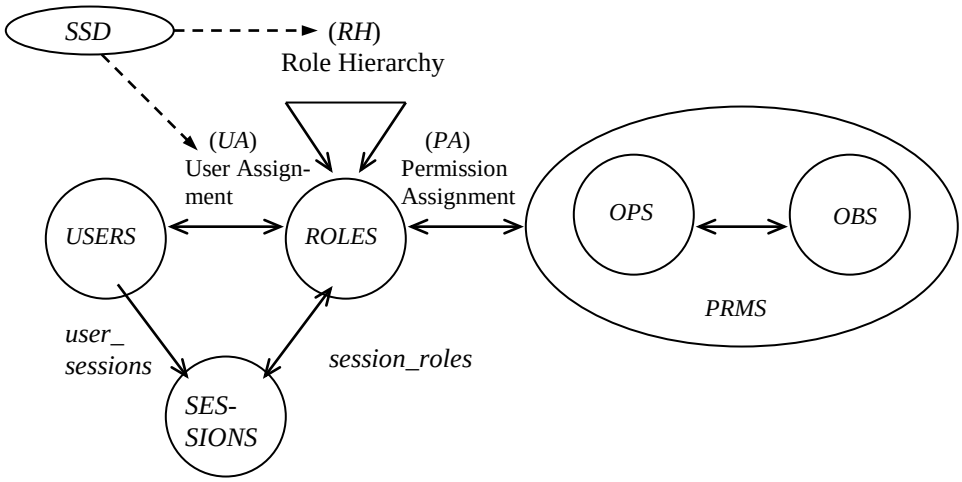

Figure 5: SSD within Hierarchical RBAC

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

Definition 3a Static Separation of Duty:

- $SSD \subseteq (2^{ROLES} \% \mathbb{N})$ is collection of pairs (*rs*, *n*) in *Static Separation of Duty*, where each *rs* is a role set and *n* is a natural number $\mu 2$, with the property that no user is assigned to *n* or more roles from the set *rs* in each $(rs, n) \in SSD$. Formally:

  $$\forall (rs, n) \in SSD, \forall s \subseteq rs : |s| \geq n \Rightarrow \bigcap_{r \in s} assigned\_users(r) = \varnothing.$$

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:

  $$\forall (rs, n) \in SSD, \forall s \subseteq rs : |s| \geq n \Rightarrow \bigcap_{r \in s} authorized\_users(r) = \varnothing.$$

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 (DSD) 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 (DSD). 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 in 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, RBAC 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 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).

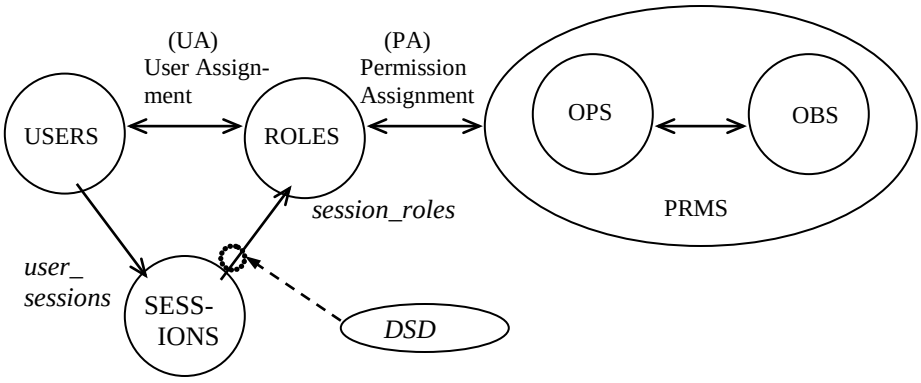

Figure 6: Dynamic Separation of Duty Relations

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

Definition 5 Dynamic Separation of Duty:

- $DSD \subseteq (2^{ROLES} \% \mathbb{N})$ is collection of pairs (*rs*, *n*) in *Dynamic Separation of Duty*, where each *rs* is a role set and *n* is a natural number $\mu 2$, with the property that no subject may activate *n* or more roles from the set *rs* in each $dsd \in DSD$. Formally:
  $\forall rs \in 2^{ROLES}, n \in \mathbb{N}, (rs, n) \in DSD \Rightarrow n \mu 2 . |rs| \mu n$, and
  $\forall s \in SESSIONS, \forall rs \in 2^{ROLES}, \forall role\_subset \in 2^{ROLES}, \forall n \in \mathbb{N}, (rs, n) \in DSD,$
  $role\_subset \subseteq rs, role\_subset \subseteq session\_roles(s) \Rightarrow \xi role\_subset \xi < n.$

4 REQUIREMENTS 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 Requirements 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.

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

- *Administrative Functions* - creation and maintenance of elements 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;
- *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 Requirements 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 two 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 User-to-Role Assignment (UA) relations are AssignUser and DeassignUser. For Permission-to-Role Assignment (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 addition and dropping of active roles and other auxiliary functions are given below:

- 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
- 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 (UA) and Permission-to-Role relation (PA) 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 – like the active roles in a given session, 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/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)

4.2 Requirements 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 among 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
- 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 descendent) role.

The outcome of 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 two things. 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) A second option is to break those relationships because an inheritance relationship no longer exists between Role A and Role B. The question of which semantics the DeleteInheritance should carry is left as an implementation issue and is not prescribed in our specification.

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 re-defined. In a role hierarchy, a given role may inherit one or more of 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 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, since 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 use*r 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 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/her authorized roles (sum of directly assigned roles and roles inherited by those roles)

4.3 Requirements 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 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 - Create 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
- SetSSDCardinality - Sets the cardinality of the subset of roles from 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 Core RBAC model are needed for implementation of SSD RABC 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
- SSDRoleSetCardinality - Returns the cardinality of the subset within the named SSD role set for which common user membership restriction applies

4.4 Requirements Specification for DSD Relation

4.4.1 Administrative Functions

The semantics of creating an instance of 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 enforced only at the time of role activation within a user session. The list of administrative functions that shall be provided for DSD RBAC model and their purpose are listed below:

- CreateDSDSet - Create a named instance of 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
- SetDSDCardinality - Sets the cardinality of the subset of roles from 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
- DropActiveRole - Deletes a role from the active role set for the current session

4.4.3 Review Functions

All the review functions for Core RBAC model are needed for implementation of DSD RABC 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
- DSDRoleSetCardinality - Returns the cardinality of the subset within the named DSD role set for which user activation restriction within the same session applies

5 REQUIREMENT PACKAGES

As eluded 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 requirement components to include Core RBAC, Hierarchical RBAC, Static Separation of Duty Relations, and Dynamic Separation of Duty Relations. Each requirements 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.

In this section we describe a logical approach for defining packages of requirement 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 requirement packages.

In defining requirement 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 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.

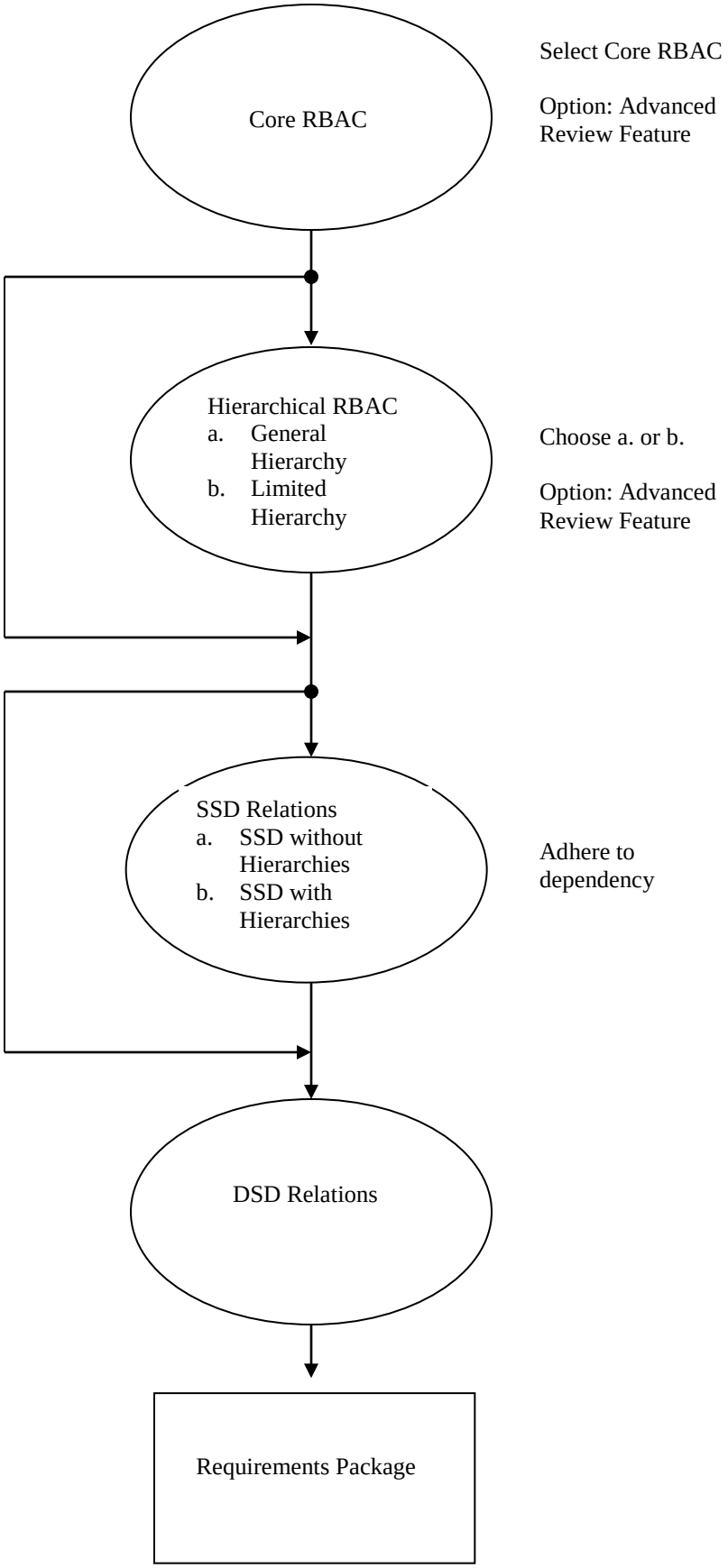

Figure 7: Methodology for Creating Requirements Packages

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 becoming 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 paper 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. These benefits include 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 beneficiary 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 Requirements Specification that casts the reference model into a congruent set of requirement 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 requirement 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.

References

[AS00] Gail Ahn and Ravi Sandhu. "Role-Based Authorization Constraints Specification." *ACM Transactions on Information and System Security*, Volume 3, Number 4, November 2000.

[BL76] D. Bell and La Padula. Secure computer systems: unified exposition and MULTICS. Report ESD-TR-75-306, The MITRE Corporation, Bedford, Massachusetts, March 1976.

[BBF00] E. Bertino, P. Bonatti, and E. Ferrari. TRBAC: a temporal role-based access control model. In *Proc. of fifth ACM Workshop on Role based access control*, pp. 21-30, 2000.

[Bal90] R. W. Baldwin. Naming and grouping privileges to simplify security management in large databases. In *proc. of the Symp. on Security and Privacy*, pp. 116-132. IEEE Press, 1990.

[BN89] D. Brewer and M. Nash. The Chinese wall security policy. In *proc. of the Symp. on Security and Privacy*, pp. 215-228. IEEE Press, 1989.

[CR98] R. Chandramouli and R. Sandhu. Role-based access control features in commercial database management systems. In *Proc. of the NIST-NSA Nat. (USA) Comp. Security Conf.*, pp 503-511, 1998.

[CW87] D. Clark and D. Wilson. A comparison of commercial and military computer security policies. In *proc. of the Symp. on Security and Privacy*, pp. 184-194. IEEE Press, 1987.

[Fei96] H. Feinstein. Final report: NIST small business innovative research (SBIR) grant: role based access control: phase 2. SETA Corp., October 1996.

[Faden99] G. Faden. Rbac in Unix administration. In *Proc. of fourth ACM Workshop on Role based access control*, pp. 95-101, 1999.

[FK92] D. Ferraiolo and R. Kuhn. Role-Based Access Control. In *Proc. of the NIST-NSA Nat. (USA) Comp. Security Conf.*, pp 554-563, 1992

[FCK95] D. Ferraiolo, J. Cugini, and R. Kuhn. Role-based access control: Features and motivations. *In Proc. of the Annual Computer Security Applications Conf.*, IEEE Press, 1995.

[FBK99] D. Ferraiolo, J. Barkley, and R. Kuhn. A role-based access control model and reference implementation within a corporate internet. *ACM Transactions on Information and System Security*, 2(1), 1999.

[FG93] D. Ferraiolo, D. Gilbert, and N. Lynch. An examination of federal and commercial access control policy needs. In *Proc. of the NIST-NSA Nat. (USA) Comp. Security Conf.*, pp 107-116, 1993

[FBK99] D. Ferraiolo, J. Barkley, and R. Kuhn. A role based access control model and reference implementation within a corporate intranet. *ACM Transactions on Information and System Security*, 2(1), 1999.

[GB98] S. Gaverila and J Barkley. Formal specification for rbac user/role and role relationship management. In *Proc. of third ACM Workshop on Role based access control*, pp. 81-90, 1998.

[GI96] L. Giuri and P. Iglio. A formal model for role based access control with constraints. *In proc. of the Computer Security Foundations Workshop*, pp. 136-145. IEEE Press, 1996.

[GGF98] V.D. Gligor, S.I. Gavrila, D.F. Ferraiolo. On the Formal Definition of Separation-of-Duty Policies and their Composition. *Proc. Symp. on Security and Privacy*, IEEE Press,1998.

[HA99] W. Huang and V. Atluri. A secure web-based workflow management system. In *Proc. of fourth ACM Workshop on Role based access control*, pp. 83-84, 1999.

[JT00] T. Jaeger and J. Tidswell. Rebuttal to the NIST rbac model proposal. In *proceedings of 5th ACM Workshop on Role-Based Access Control*, pages 65-66. (Berlin, Germany, July 2000). ACM.

[Kuh97] R. Kuhn. Mutual exclusion as a means of implementing separation of duty requirements in role based access control systems. In *Proc. of Second ACM Workshop on Role based access control*, 1997, pages 23 – 30.

[Kuh98] D.R. Kuhn. Role Based Access Control on MLS Systems Without Kernel Changes. In Proc. *ACM Workshop on Role Based Access Control*, October 22-23,1998 pages 25 - 32.

[Lam74] B. Lampson. Protection. *ACM Operating Sys. Reviews*, 8(1):18-24, 1974.

[MMN90] C. McCollum, J. Messing, L. Notargiacomo. Beyond the pale of MAC and DAC – defining new forms of access control. In *proc. of the Symp. on Security and Privacy*, pp. 190-900. IEEE Press, 1990.

[Moffett98] Jonathan D. Moffett, "Control principles and role hierarchies." *Proc. Third ACM Workshop on Role-Based Access Control*, Fairfax, Virginia, October 22-23, 1998, pages 63 - 69.

[NO94 M. Nyanchama and S. Osborn. Access rights administration in role-based security systems. In J. Biskup, M. Morgenstern, and C. E. Landwehr, editors, *Database Security, VIII*: *Status and Prospects*, pages 37-56. North-Holland, 1994.
[NO99] M. Nyanchama and S. Osborn. The graph model and conflicts of interest. *ACM Transactions on Information and System Security*, 2(1), 1999.
[OSM00] S. Osborn, R. Sandhu and Q. Munawer. Configuring Role-Based Access Control to Enforce Mandatory and Discretionary Access Control Policies. *ACM Transactions on Information and System Security*, 3(2), 2000.
[San88] Ravi Sandhu, "Transaction Control Expressions for Separation of Duties." Proc. Fourth Aerospace Computer Security Applications Conference, Orlando, Florida, IEEE Computer Society Press, December 1988, pages 282-286
[SM97] R. Sandhu and V. Bhamidipoti. Role-Based Administration of User-Role Assignment: The URA97 Model and its Oracle Implementation. *Journal of Computer Security*, Volume 7. 1997.
[SCFY96] R Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role-based access control models. *IEEE Computer*, 29(2), February 1996.
[San98] Ravi Sandhu, "Role Activation Hierarchies." *Proc. Third ACM Workshop on Role-Based Access Control*, Fairfax, Virginia, October 22-23, 1998, pages 33-40.
[San98b] Ravi Sandhu, "Role-Based Access Control." Advances in Computers, Volume 46, (M. Zelkowitz editor), Academic Press, pages 237-286, 1998.
[SBM97] Ravi Sandhu, Venkata Bhamidipati and Qamar Munawer. "The ARBAC97 Model for Role-Based Administration of Roles." *ACM Transactions on Information and System Security*, Volume 2, Number 1, February 1999, pages 105-135.
[SFK00] R. Sandhu, D. Ferraiolo, R. Kuhn. The nist model for role-based access control: Towards a unified standard. In *proceedings of 5th ACM Workshop on Role-Based Access Control*, pages 47-63. (Berlin, Germany, July 2000). ACM.
[SZ97] R. Simon and M. Zurko. Separation of duty in role based access control environments. In *Proc. of, New Security Paradigms Workshop*, September 1997.
[SCYG96] C. Smith, E. Coyne, C. Youman and S. Ganta. 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 1996.
[TDH92] T. C. Ting, S. A. Demurjian, and M. Y. Hu. Requirements capabilities and Functionalities of User-Role Based Security for an Object-Oriented Design Model. In S. Jajodia and C. E. Landwehr, editors, *Database Security, IV*: *Status and Prospects*, pages 275-296. North-Holland, 1992.
[Thomsen91] D. J. Thomsen. Role-based application design and enforcement. In S. Jajodia and C. E. Landwehr, editors, *Database Security, IV*: *Status and Prospects*, pages 151-168. North-Holland, 1991.

Appendix A: RBAC REQUIREMENTS SPECIFICATION

The RBAC Requirements 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 as follows:

Schema-Name (Declaration) ♥ Predicate; …; Predicate ♦.

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:

$$AddUser(user: NAME) \lhd$$
$$user \notin USERS$$
$$USERS' = USERS \cup \{user\}$$
$$user\_sessions' = user\_sessions \cup \{user \mapsto \varnothing\} \rhd$$

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:

$$DeleteUser(user: NAME) \lhd$$
$$user \in USERS$$
$$[\forall s \in SESSIONS \bullet s \in user\_sessions(user) \Rightarrow DeleteSession(s)]$$
$$UA' = UA \setminus \{r: ROLES \bullet user \mapsto r\}$$
$$assigned\_users' = \{r: ROLES \bullet r \mapsto (assigned\_users(r) \setminus \{user\})\}$$
$$USERS' = USERS \setminus \{user\} \rhd$$

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:

$$AddRole(role: NAME) \lhd$$
$$role \notin ROLES$$
$$ROLES' = ROLES \cup \{role\}$$
$$assigned\_users' = assigned\_users \cup \{role \mapsto \varnothing\}$$
$$assigned\_permissions' = assigned\_permissions \cup \{role \mapsto \varnothing\} \rhd$$

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 dset, 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/her active roles is *role*. The RBAC system could wait for such a session to terminate normally, or 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:

$$DeassignUser(user, role: NAME) \lhd$$
$$user \in USERS; role \in ROLES; (user \mapsto role) \in UA$$
$$[\forall s: SESSIONS \bullet s \in user\_sessions(user) \wedge role \in session\_roles(s) \Rightarrow DeleteSession(s)]$$
$$UA' = UA \setminus \{user \mapsto role\}$$
$$assigned\_users' = assigned\_users \setminus \{role \mapsto asigned\_users(role)\} \cup$$
$$\{role \mapsto (asigned\_users(role) \setminus \{user\})\} \rhd$$

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, i.e., 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:

$$GrantPermission(object, operation, role: NAME) \triangleleft$$
$$(operation, object) \in PERMS; role \in ROLES$$
$$PA' = PA \cup \{(operation, object) \mapsto role\}$$
$$assigned\_permissions' = assigned\_permissions \setminus \{role \mapsto assigned\_permissions(roles)\} \cup$$
$$\{role \mapsto (assigned\_permissions(role) \cup \{(operation, object)\})\} \triangleright$$

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, i.e., 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:

$$RevokePermission(operation, object, role: NAME) \triangleleft$$
$$(operation, object) \in PERMS; role \in ROLES; ((operation, object) \mapsto role) \in PA$$
$$PA' = PA \setminus \{(operation, object) \mapsto role\}$$
$$assigned\_permissions' = assigned\_permissions \setminus \{role \mapsto assigned\_permissions(role)\} \cup$$
$$\{role \mapsto (assigned\_permissions(role) \setminus \{(operation, object)\})\} \triangleright$$

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 a 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.

$$CreateSession(user: NAME; ars: 2^{NAMES}; session: NAME) \triangleleft$$
$$user \in USERS; ars \subseteq \{r: ROLES | (user \mapsto r) \in UA\}; session \notin SESSIONS$$
$$SESSIONS' = SESSIONS \cup \{session\}$$
$$user\_sessions' = user\_sessions \setminus \{user \mapsto user\_sessions(user)\} \cup$$
$$\{user \mapsto (user\_sessions(user) \cup \{session\})\}$$
$$session\_roles' = session\_roles \cup \{session \mapsto ars\} \triangleright$$

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:

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

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, and
- the role is a member of the *ROLES* data set, and
- the session identifier is a member of the *SESSIONS* data set, and
- 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:

$$AddActiveRole(user, session, role: NAME) \lhd$$
$$user \in USERS; session \in SESSIONS; role \in ROLES; session \in user\_sessions(user)$$
$$(user \mapsto role) \in UA; role \notin session\_roles(session)$$
$$session\_roles' = session\_roles \setminus \{session \mapsto session\_roles(session)\} \cup$$
$$\{session \mapsto (session\_roles(session) \cup \{role\})\} \rhd$$

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:

$$DropActiveRole(user, session, role: NAME) \lhd$$
$$user \in USERS; role \in ROLES; session \in SESSIONS$$
$$session \in user\_sessions(user); role \in session\_roles(session)$$
$$session\_roles' = session\_roles \setminus \{session \mapsto session\_roles(session)\} \cup$$
$$\{session \mapsto (session\_roles(session) \setminus \{role\}) \rhd$$

CheckAccess
This function returns a Boolean value meaning whether the subject of a given session is allowed or not 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:

$$CheckAccess(session, operation, object: NAME; out\ result: BOOLEAN) \lhd$$
$$session \in SESSIONS; operation \in OPS; object \in OBJS$$
$$result = (\exists r: ROLES \bullet r \in session\_roles(session) \wedge ((operation, object) \mapsto r) \in PA) \rhd$$

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:

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

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:

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

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:

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

UserPermissions
This function returns the permissions a given user gets through his/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:

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

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:

$$SessionRoles(session: NAME; out\ result: 2^{ROLES}) \lhd$$
$$session \in SESSIONS$$
$$result = session\_roles(session) \rhd$$

SessionPermissions
This function returns the permissions of the session *session*, i.e., 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:

$$SessionPermissions(session: NAME; out\ result: 2^{PERMS}) \lhd$$
$$session \in SESSIONS$$
$$result = \{r: ROLES; op \in OPS; obj \in OBJS \mid r \in session\_roles(session) \wedge ((op, obj) \mapsto r) \in PA \bullet (op, obj)\} \rhd$$

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 commands establishes a new immediate inheritance relationship *r_asc* ν *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:

$\mu == /$

$>> == \nu$

to formally describes the command:

$$AddInheritance(r\_asc, r\_desc: NAME) \lhd$$
$$r\_asc \in ROLES; r\_desc \in ROLES; \neg(r\_asc >> r\_desc); \neg(r\_desc \geq r\_asc)$$
$$\geq' = \geq \cup \{r, q: ROLES \mid r \geq r\_asc \wedge r\_desc \geq q \bullet r \mapsto q\} \rhd$$

DeleteInheritance

This command deletes an existing immediate inheritance relationship *r_asc* ν *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 resulted after deleting the relationship *r_asc* ν *r_desc*. The following schema formally describes this command:

$$DeleteInheritance(r\_asc, r\_desc: NAME) \lhd$$
$$r\_asc \in ROLES; r\_desc \in ROLES; r\_asc >> r\_desc$$
$$\geq' = (>> \setminus \{r\_asc \mapsto r\_desc\})^* \rhd$$

AddAscendant

This commands 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*.

$$AddAscendant(r\_asc, r\_desc: NAME) \lhd$$
$$AddRole(r\_asc)$$
$$AddInheritance(r\_asc, r\_desc) \rhd$$

AddDescendant

This commands 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*.

$$AddDescendant(r\_asc, r\_desc: NAME) \lhd$$
$$AddRole(r\_desc)$$
$$AddInheritance(r\_asc, r\_desc) \rhd$$

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 a 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.

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

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

The following schema formally describes the function:

$$AddActiveRole(user, session, role: NAME) \lhd$$
$$user \in USERS; session \in SESSIONS; role \in ROLES; session \in user\_sessions(user)$$
$$user \in authorized\_users(role); role \notin session\_roles(session)$$
$$session\_roles' = session\_roles \setminus \{session \mapsto session\_roles(session)\} \cup$$
$$\{session \mapsto (session\_roles(session) \cup \{role\})\} \rhd$$

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 to a given role, i.e., 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:

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

UserPermissions
This function returns the set of permissions a given user gets through his/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:

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

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 commands establishes a new immediate inheritance relationship *r_asc* ∨∨ *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:

$\mu == /$

$>> == \nu\nu$

to formally describes the command:

$$AddInheritance(r\_asc, r\_desc: NAME) \lhd$$
$$r\_asc \in ROLES; r\_desc \in ROLES; \forall r \in ROLES \bullet \neg(r\_asc >> r); \neg(r\_desc \geq r\_asc)$$
$$\geq' = \geq \cup \{r, q: ROLES | r \geq r\_asc \wedge r\_desc \geq q \bullet r \mapsto q\} \rhd$$

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 (SSD) 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:

$$ssd\_set: SSD \rightarrow 2^{ROLES}$$
$$ssd\_card: SSD \rightarrow \mathbf{N}$$
$$\forall ssd \in SSD \bullet ssd\_card(ssd) \geq 2 \wedge ssd\_card(ssd) \leq |ssd\_set(ssd)|$$

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, and
- the role is a member of the *ROLES* data set, and
- 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:

$$AssignUser(user, role: NAME) \lhd$$

$$user \in USERS; role \in ROLES; (user \mapsto role) \notin UA$$

$$\forall ssd \in SSD \bullet \bigcap_{\substack{r \in subset \\ subset \subseteq ssd\_set(ssd) \\ |subset| = ssd\_card(ssd) \\ us = \text{if } r = role \text{ then } \{user\} \text{ else } \varnothing}} (assigned\_users(r) \cup us) = \varnothing$$

$$UA' = UA \cup \{user \mapsto role\}$$

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

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:

$$CreateSsdSet(set\_name: NAME; role\_set: 2^{NAMES}; n: \mathbb{N}) \lhd$$

$$set\_name \notin SSD; (n \geq 2) \wedge (n \leq |role\_set|); role\_set \subseteq ROLES$$

$$\bigcap_{\substack{r \in subset \\ subset \subseteq role\_set \\ |subset| = n}} assigned\_users(r) = \varnothing$$

$$SSD' = SSD \cup \{set\_name\}$$

$$ssd\_set' = ssd\_set \cup \{set\_name \mapsto role\_set\}$$

$$ssd\_card' = ssd\_card \cup \{set\_name \mapsto n\} \rhd$$

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, and
- 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:

$$AddSsdRoleMember(set\_name: NAME; role: NAME) \lhd$$

$$set\_name \in SSD; role \in ROLES; role \notin ssd\_set(set\_name)$$

$$\bigcap_{\substack{r \in subset \\ subset \subseteq ssd\_set(set\_name) \cup \{role\} \\ |subset| = n}} assigned\_users(r) = \varnothing$$

$$ssd\_set' = ssd\_set \setminus \{set\_name \mapsto ssd\_set(set\_name)\} \cup \{set\_name \mapsto (ssd\_set(set\_name) \cup \{role\})\} \rhd$$

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, and

- 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:

$$DeleteSsdRoleMember(set\_name: NAME; role: NAME) \lhd$$
$$set\_name \in SSD; role \in ssd\_set(set\_name); ssd\_card(set\_name) < |ssd\_set(set\_name)|$$
$$ssd\_set' = ssd\_set \setminus \{set\_name \mapsto ssd\_set(set\_name)\} \cup$$
$$\{set\_name \mapsto (ssd\_set(set\_name) \setminus \{role\})\} \rhd$$

DeleteSsdSet

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

$$DeleteSsdSet(set\_name: NAME) \lhd$$
$$set\_name \in SSD; ssd\_card' = ssd\_card \setminus \{set\_name \mapsto ssd\_card(set\_name)\}$$
$$ssd\_set' = ssd\_set \setminus \{set\_name \mapsto ssd\_set(set\_name)\}$$
$$SSD' = SSD \setminus \{set\_name\} \rhd$$

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, and
- 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:

$$SetSsdSetCardinality(set\_name: NAME; n: \mathbf{N}) \lhd$$
$$set\_name \in SSD; (n \geq 2) \wedge (n \leq |ssd\_set(set\_name)|)$$
$$\bigcap_{\substack{r \in subset \\ subset \subseteq ssd\_set(set\_name) \\ |subset| = n}} assigned\_users(r) = \varnothing$$
$$ssd\_card' = ssd\_card \setminus \{set\_name \mapsto ssd\_card(set\_name)\} \cup \{set\_name \mapsto n\} \rhd$$

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:

$$SsdRoleSetCardinality(set\_name: NAME; out\ result: \mathbf{N}) \lhd$$
$$set\_name \in SSD$$
$$result = ssd\_card(set\_name) \rhd$$

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, SetSsdSetCardinality 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, and
- the role is a member of the *ROLES* data set, and
- 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:

$$AssignUser(user, role: NAME) \lhd$$
$$user \in USERS; role \in ROLES; (user \mapsto role) \notin UA$$
$$\forall ssd \in SSD \bullet \bigcap_{\substack{r \in subset \\ subset \subseteq ssd\_set(ssd) \\ |subset| = ssd\_card(ssd) \\ au = \text{if } r = role \text{ then } \{user\} \text{ else } \varnothing}} (authorized\_users(r) \cup au) = \varnothing$$
$$UA' = UA \cup \{user \mapsto role\}$$
$$assigned\_users' = assigned\_users \setminus \{role \mapsto assigned\_users(role)\} \cup$$
$$\{role \mapsto (assigned\_users(role) \cup \{user\})\} \rhd$$

AddInheritance

This commands establishes a new immediate inheritance relationship *r_asc* ∨∨ *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, and
- *r_asc* is not an immediate ascendant of *r_desc*, and
- *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:

$\mu == /$

$>> == \nu\nu$

to formally describes the command:

$$AddInheritance(r\_asc, r\_desc: NAME) \lhd$$
$$r\_asc \in ROLES; r\_desc \in ROLES; \neg(r\_asc >> r\_desc); \neg(r\_desc \geq r\_asc)$$
$$\forall ssd \in SSD \bullet \bigcap_{\substack{r \in subset \\ subset \subseteq ssd\_set(ssd) \\ |subset| = ssd\_card(ssd) \\ au = \text{if } r = r\_desc \text{ then } authorized\_users(r\_asc) \text{ else } \varnothing}} (authorized\_users(r) \cup au) = \varnothing$$
$$\geq' = \geq \cup \{r, q: ROLES \mid r \geq r\_asc \wedge r\_desc \geq q \bullet r \mapsto q\} \rhd$$

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:

$$CreateSsdSet(set\_name: NAME; role\_set: 2^{NAMES}; n: \mathbf{N}) \lhd$$
$$set\_name \notin SSD; (n \geq 2) \wedge (n \leq |role\_set|); role\_set \subseteq ROLES$$
$$\bigcap_{\substack{r \in subset \\ subset \subseteq role\_set \\ |subset| = n}} authorized\_users(r) = \varnothing$$
$$SSD' = SSD \cup \{set\_name\}$$
$$ssd\_set' = ssd\_set \cup \{set\_name \mapsto role\_set\}$$
$$ssd\_card' = ssd\_card \cup \{set\_name \mapsto n\} \rhd$$

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, and
- 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:

$$AddSsdRoleMember(set\_name: NAME; role: NAME) \lhd$$
$$set\_name \in SSD; role \in ROLES; role \notin ssd\_set(set\_name)$$
$$\bigcap_{\substack{r \in subset \\ subset \subseteq ssd\_set(set\_name) \cup \{role\} \\ |subset| = n}} authorized\_users(r) = \varnothing$$
$$ssd\_set' = ssd\_set \setminus \{set\_name \mapsto ssd\_set(set\_name)\} \cup$$
$$\{set\_name \mapsto (ssd\_set(set\_name) \cup \{role\})\} \rhd$$

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, and

- 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:

$$SetSsdSetCardinality(set\_name: NAME; n: \mathbf{N}) \lhd$$

$$set\_name \in SSD; (n \geq 2) \wedge (n \leq |ssd\_set(set\_name)|)$$

$$\bigcap_{\substack{r \in subset \\ subset \subseteq ssd\_set(set\_name) \\ |subset|=n}} authorized\_users(r) = \varnothing$$

$$ssd\_card' = ssd\_card \setminus \{set\_name \mapsto ssd\_card(set\_name)\} \cup \{set\_name \mapsto n\} \rhd$$

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 commands establishes a new immediate inheritance relationship *r_asc* νν *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:

$$\mu == /$$

$$>> == \nu\nu$$

to formally describes the command:

$$AddInheritance(r\_asc, r\_desc: NAME) \lhd$$

$$r\_asc \in ROLES; r\_desc \in ROLES; \forall r \in ROLES \bullet \neg(r\_asc >> r); \neg(r\_desc \geq r\_asc)$$

$$\forall ssd \in SSD \bullet \bigcap_{\substack{r \in subset \\ subset \subseteq ssd\_set(ssd) \\ |subset|=ssd\_card(ssd) \\ au = \text{if } r=r\_desc \text{ then } authorized\_users(r\_asc) \text{ else } \varnothing}} (authorized\_users(r) \cup au) = \varnothing$$

$$\geq' = \geq \cup \{r, q: ROLES | r \geq r\_asc \wedge r\_desc \geq q \bullet r \mapsto q\} \rhd$$

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:

$$dsd\_set: DSD \rightarrow 2^{ROLES}$$
$$dsd\_card: DSD \rightarrow \mathbf{N}$$
$$\forall dsd \in SSD \bullet dsd\_card(dsd) \geq 2 \wedge dsd\_card(dsd) \leq |dsd\_set(dsd)|$$

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:

$$CreateDsdSet(set\_name: NAME; role\_set: 2^{NAMES}; n: \mathbf{N}) \triangleleft$$
$$set\_name \notin DSD; (n \geq 2) \wedge (n \leq |role\_set|); role\_set \subseteq ROLES$$
$$\forall s: SESSIONS; role\_subset: 2^{role\_set} \bullet role\_subset \subseteq session\_roles(s) \Rightarrow |role\_subset| < n$$
$$DSD' = DSD \cup \{set\_name\}$$
$$dsd\_set' = dsd\_set \cup \{set\_name \mapsto role\_set\}$$
$$dsd\_card' = dsd\_card \cup \{set\_name \mapsto n\} \triangleright$$

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

$$AddDsdRoleMember(set\_name: NAME; role: NAME) \triangleleft$$
$$set\_name \in DSD; role \in ROLES; role \notin dsd\_set(set\_name)$$
$$\forall s: SESSIONS; role\_subset: 2^{dsd\_set(set\_name) \cup \{role\}} \bullet$$
$$role\_subset \subseteq session\_roles(s) \Rightarrow |role\_subset| < dsd\_card(set\_name)$$
$$dsd\_set' = dsd\_set \setminus \{set\_name \mapsto dsd\_set(set\_name)\} \cup$$
$$\{set\_name \mapsto (dsd\_set(set\_name) \cup \{role\})\} \triangleright$$

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, and
- 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:

$$DeleteDsdRoleMember(set\_name: NAME; role: NAME) \triangleleft$$
$$set\_name \in DSD; role \in dsd\_set(set\_name); dsd\_card(set\_name) < |dsd\_set(set\_name)|$$
$$dsd\_set' = dsd\_set \setminus \{set\_name \mapsto dsd\_set(set\_name)\} \cup$$
$$\{set\_name \mapsto (dsd\_set(set\_name) \setminus \{role\})\} \triangleright$$

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:

$$DeleteDsdSet(set\_name: NAME)$$
$$\{$$
$$set\_name \in DSD$$
$$dsd\_card' = dsd\_card \setminus \{set\_name \mapsto dsd\_card(set\_name)\}$$
$$dsd\_set' = dsd\_set \setminus \{set\_name \mapsto dsd\_set(set\_name)\}$$
$$DSD' = DSD \setminus \{set\_name\}$$
$$\}$$

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, and
- 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:

$$SetDsdSetCardinality(set\_name: NAME; n: \mathbf{N}) \triangleleft$$
$$set\_name \in DSD; (n \geq 2) \wedge (n \leq |dsd\_set(set\_name)|)$$
$$\forall s: SESSIONS; role\_subset: 2^{dsd\_set(set\_name)} \bullet$$
$$role\_subset \subseteq session\_roles(s) \Rightarrow |role\_subset| < n$$
$$dsd\_card' = dsd\_card \setminus \{set\_name \mapsto dsd\_card(set\_name)\} \cup \{set\_name \mapsto n\} \triangleright$$

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, and
- 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.

$$
\begin{aligned}
&CreateSession(user: NAME; ars: 2^{NAME}; session: NAME) \lhd \\
&\quad user \in USERS; ars \subseteq \{r: ROLES \mid (user \mapsto r) \in UA\}; session \notin SESSIONS \\
&\quad \forall dset: DSD; rset: 2^{NAME} \bullet \\
&\qquad rset \subseteq dsd\_set(dset) \wedge rset \subseteq ars \Rightarrow |rset| < dsd\_card(dset) \\
&\quad SESSIONS' = SESSIONS \cup \{session\} \\
&\quad user\_sessions' = user\_sessions \setminus \{user \mapsto user\_sessions(user)\} \cup \\
&\qquad \{user \mapsto (user\_sessions(user) \cup \{session\})\} \\
&\quad session\_roles' = session\_roles \cup \{session \mapsto ars\} \rhd
\end{aligned}
$$

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, and
- the role is a member of the *ROLES* data set, and
- the session identifier is a member of the *SESSIONS* data set, and
- the role is assigned to the user, and
- 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:

$$
\begin{aligned}
&AddActiveRole(user, session, role: NAME) \lhd \\
&\quad user \in USERS; session \in SESSIONS; role \in ROLES; session \in user\_sessions(user) \\
&\quad user \in assigned\_users(role); role \notin session\_roles(session) \\
&\quad \forall dset: DSD; rset: 2^{NAME} \bullet \\
&\qquad rset \subseteq dsd\_set(dset) \wedge rset \subseteq session\_roles(session) \cup \{role\} \Rightarrow |rset| < dsd\_card(dset) \\
&\quad session\_roles' = session\_roles \setminus \{session \mapsto session\_roles(session)\} \cup \\
&\qquad \{session \mapsto (session\_roles(session) \cup \{role\})\} \rhd
\end{aligned}
$$

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:

$$DsdRoleSetRoles(set\_name: NAME; out\ result: 2^{ROLES}) \lhd$$
$$set\_name \in DSD$$
$$result = dsd\_set(set\_name) \rhd$$

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:

$$DsdRoleSetCardinality(set\_name: NAME; out\ result: \mathbf{N}) \lhd$$
$$set\_name \in DSD$$
$$result = dsd\_card(set\_name) \rhd$$

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, and
- 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:

$$CreateSession(user: NAME; ars: 2^{NAME}; session: NAME) \lhd$$
$$user \in USERS; ars \subseteq \{r, q: ROLES \mid (user \mapsto q) \in UA \wedge q \geq r \bullet r\}; session \notin SESSIONS$$
$$\forall dset: DSD; rset: 2^{NAME} \bullet$$
$$rset \subseteq dsd\_set(dset) \wedge rset \subseteq ars \Rightarrow |rset| < dsd\_card(dset)$$
$$SESSIONS' = SESSIONS \cup \{session\}$$
$$user\_sessions' = user\_sessions \setminus \{user \mapsto user\_sessions(user)\} \cup$$
$$\{user \mapsto (user\_sessions(user) \cup \{session\})\}$$
$$session\_roles' = session\_roles \cup \{session \mapsto ars\} \rhd$$

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, and
- the role is a member of the *ROLES* data set, and
- the session identifier is a member of the *SESSIONS* data set, and
- the role is authorized for that user, and
- 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:

$$AddActiveRole(user, session, role: NAME) \lhd$$
$$user \in USERS; session \in SESSIONS; role \in ROLES; session \in user\_sessions(user)$$
$$user \in authorized\_users(role); role \notin session\_roles(session)$$
$$\forall dset: DSD; rset: 2^{NAME} \bullet$$
$$rset \subseteq dsd\_set(dset) \wedge rset \subseteq session\_roles(session) \cup \{role\} \Rightarrow |rset| < dsd\_card(dset)$$
$$session\_roles' = session\_roles \setminus \{session \mapsto session\_roles(session)\} \cup$$
$$\{session \mapsto (session\_roles(session) \cup \{role\})\} \rhd$$

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-1and 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.