A FAMILY OF MODELS FOR RULE-BASED USER-ROLE ASSIGNMENT

by

Mohammad Abdullah Al-Kahtani

A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
in Partial Fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Information Technology

Committee:

_____ Dr. Ravi Sandhu, Dissertation Director

_____ Dr. Sushil Jajodia

_____ Dr. Edgar Sibley

_____ Dr. Kris Gaj

_____ Dr. Stephen G. Nash, Associate Dean for Graduate Studies and Research

_____ Dr. Lloyd J. Griffiths, Dean, School of Information Technology and Engineering

Date: _____ Fall Semester 2003
George Mason University
Fairfax, Virginia

A Family of Models for Rule-Based User-Role Assignment

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University

By

Mohammad Abdullah Al-Kahtani
B.S., King Saud University, Riyadh, Saudi Arabia, 1990
M.S., George Mason University, Fairfax, VA, 1995

Director: Dr. Ravi S. Sandhu, Professor
Information and Software Engineering

Fall Semester 2003
George Mason University
Fairfax, VA

ABSTRACT

A FAMILY OF MODELS FOR RULE-BASED USER-ROLE ASSIGNMENT

Mohammad Abdullah Al-Kahtani, Ph.D.

George Mason University, 2003

Dissertation Director: Dr. Ravi Sandhu

Conventional role based access control (RBAC) was designed with closed-enterprise environment in mind where a security officer(s) manually assigns users to roles. However, today, an increasing number of service-providing enterprises make their services available to users via the Internet. Furthermore, many enterprises have users (i.e. workers and/or clients) whose numbers can be in the hundreds of thousands or millions. In addition, RBAC is being supported by software products designed to serve large number of clients such as popular commercial database management systems. All these factors render the manual user-to-role assignment a formidable task which is costly and error-prone. An appealing solution is to automate the assignment process. Besides eliminating the drawbacks of its manual counterpart, automatic assignment, particularly in the case of external user (i.e. clients), extends enterprise-consumers business partnership. In fact some large enterprises have already implemented systems that assign and revoke users automatically, and many of them have achieved 90-95% automation of

administration. Our work lays the theoretical foundation for the implementation of the assignment process. It also serves as a benchmark for software implementations.

In this dissertation, we describe a family of models called RB-RBAC that extends and modifies RBAC96, a well-known RBAC model, to allow the specification of automatic (implicit) user-role assignment. Model A allows specifying a set of authorization rules that can be used to assign users to roles based on users' attributes. Model B extends Model A to allow specifying negative authorization and mutual exclusion among roles. Model C extends Model A to allow constraints specification.

To show the power and usefulness of RB-RBAC, we demonstrate how it can be configured to express Mandatory Access Controls (MAC) and Discretionary Access Controls (DAC).

In addition to RB-RBAC family, we developed an administrative model, ARB-RBAC, which provides the specification needed to administer users' attributes and authorization rules.

Our work demonstrates that it is possible to modify RBAC96 to allow implicit user-role assignment and, at the same time, retain the central features of RBAC96.

DEDICATION

"O my Lord! Grant me that I may be grateful for Thy favour which Thou has bestowed upon me, and upon both my parents, and that I may work righteousness such as Thou mayest approve; and be gracious to me in my issue. Truly have I turned to Thee and truly do I bow (to Thee) in Islam."

The Holy Quran 046.015

ACKNOWLEDGEMENTS

I would like to sincerely express my gratitude and appreciation to my dissertation director, Professor Ravi Sandhu, who has provided valuable guidance and encouragement during my doctoral study.

Also, I would like to thank the members of my dissertation committee, Professor Sushil Jajodia, Professor Edgar Sibley, and Professor Kris Gaj. I am thankful for their valuable comments on my dissertation.

I am particularly thankful to my family, especially my father who flew all the way from Saudi Arabia to attend my public defense.

Also, I would like to thank all of my friends at Mason who made my student life at Mason an unforgettable one.

Table of Contents

Page

List of Tables

List of Figures

Chapter 1: Introduction

1.1 RBAC: The Concept and Features

Role-based access control (RBAC) has emerged as a widely deployed alternative to classical discretionary and mandatory access controls [SCFY1996], [SFK2000], and [FSGK2001]. In order to regulate the access of users to data and system resources, RBAC requires the identification of roles in the system. A role can be viewed as a semantic construct associated with a particular working activity and around which access control policy is formulated. In contrast to traditional access control systems where permissions are assigned to users directly, RBAC associates permissions with roles, and users are made members of appropriate roles, thereby acquiring the roles' permissions. This greatly simplifies management of permissions. Users are assigned to appropriate roles based on factors such as their responsibilities and qualifications. Users can be easily reassigned roles. Since roles in an organization are relatively persistent with respect to user turnover and task re-assignment, RBAC provides a powerful mechanism for reducing the complexity, cost, and potential for error of assigning users permissions within the organization. Roles within an organization typically have overlapping permissions, and as such can be organized in role hierarchies where a senior role includes all of the permissions of its juniors. Role hierarchies reflect the organization’s lines of responsibility and authority. RBAC allows the specification of constraints to control

different aspects such as user-role assignment, permission-role assignment, the structure of the role hierarchy, etc.

The principle motivation behind RBAC is the desire to specify and enforce enterprise security policies in a way that maps naturally to the business practices and organizational structure of the enterprise [Kuhn1997].

In discretionary access control systems, it is assumed that individual users own the objects e.g. files or database tables. However, this assumption is not valid in many enterprises within the private or government sectors. Rather, these objects are owned by the enterprise. This fundamentally affects the way of managing access to these objects. To meet the need of such enterprises, users in RBAC are administratively made member of appropriate roles which, in turn, are administratively associated with permissions. The main features of RBAC, as explained in [FK1995] and [SCFY1996], are:

a. It does not promote any particular security policy. RBAC is policy-neutral which enables it to support different security policies. At the same time, RBAC directly supports three well-known security principles: least privilege, separation of duties, and data abstraction.
b. The administration of authorization data is widely acknowledged as an error-prone process with large and recurring expenses. RBAC provides superb administrative capabilities, as the privileges are gathered in roles to/from which users can be easily assigned/revoked. Roles privileges can be updated without the need to update the privileges assigned to every user.
c. RBAC maps easily to the way enterprises typically conduct business. So, after an RBAC framework is established, the system administrator's prime job will be

assigning and revoking users into and out of roles. This is in contrast to the more conventional and less intuitive process of attempting to administer lower level access control mechanisms directly.

d. RBAC allows centralized and decentralized administration of access control.

1.2 RBAC: The Model

Although several models of RBAC have been published, in this dissertation we adopt RBAC96, a well-known family of models introduced by Sandhu et al. in [SCFY1996]. This model was chosen for several reasons:

a. It captures the important aspects of conventional RBAC.
b. It has become a widely cited authoritative reference.
c. It has been used to develop implementations of various RBAC models and mechanisms using UNIX, Oracle, Windows NT, and so on [Ahn1999].
d. It is closely related to the proposed NIST standard for RBAC [FSGK2001].

The basic components of RBAC96 are:

a. User: A user is typically a human being but can be generalized to include intelligent autonomous agents.
b. Role: A job function or title within an organization with associated semantics regarding the authority and responsibility conferred on a member of the role.
c. Permission: An approval of a particular mode of access to one or more objects in the system.

The above basic components are used to construct more advanced ones:

a. Role hierarchy: Roles are organized in a partial order or hierarchy so that a senior role inherits permissions from junior roles, but not vice-versa. This hierarchy reflects an organization's lines of authority and responsibility.
b. User-role assignment relation UA: A user can be a member of many roles and a role can have many users. Membership is acquired via explicit user-role assignment, which is captured in UA.
c. Permission-role assignment relation PA: A role can have many permissions and the same permission can be assigned to many roles. Permission-role assignment is done manually and is captured in PA.
d. A session: 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 of which he is a member. Multiple roles can be simultaneously activated. Each session is associated with a single user for the life of that session. A user may have multiple sessions open at the same time and each may have a different combination of roles. A subject (or a session) is a unit of access control, and a user may have multiple subjects (or sessions) with different permissions active at the same time.

RBAC96 is comprised of a family of four conceptual models. In brief, these models are:

a. RBAC0: This is the base model and the minimum requirement for any system to support RBAC.
b. RBAC1: This model includes RBAC0 and has the additional feature of role hierarchies, which allows permissions inheritance among roles.
c. RBAC2: This model also includes RBAC0 with the additional feature of constraints, which impose restrictions on configuration of the components of

RBAC. They are shown as dashed lines in Figure 1 that shows the complete RBAC96 model.

d. RBAC3 is the consolidated model, which includes RBAC1 and RBAC2.

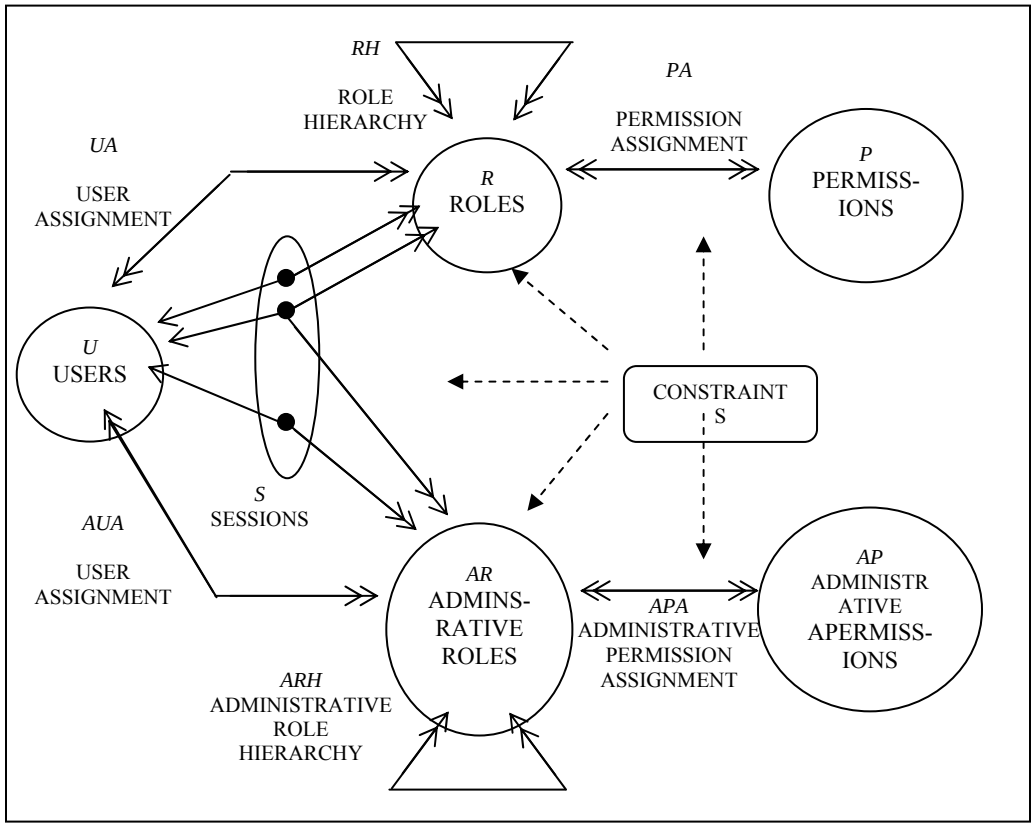

Figure 1: RBAC96 Model

Figures 2 shows the RBAC96 formal model.

-U, a set of users; R and AR, disjoint sets of (regular) roles and administrative roles, P and AP, disjoint set of (regular) permissions and administrative permissions; S, a set of sessions.
-$UA \subseteq U \times R$, user to role assignment relation
$AUA \subseteq U \times AR$, user to administrative role assignment relation
-$PA \subseteq P \times R$, permission to role assignment relation
$APA \subseteq AP \times AR$, permission to administrative role assignment relation
-$RH \subseteq R \times R$, partially order role hierarchy
$ARH \subseteq AR \times AR$, partially order administrative role hierarchy
(both hierarchies are written as $\geq$ in infix notation)
-*user*: $S \rightarrow U$, maps each session to a single user (which does not change)
roles: $S \rightarrow 2^{R \cup AR}$ maps each session s_i to a set of *roles*$(s_i) \subseteq \{r | \exists r' \geq r)[(user(s_i), r') \in UA \cup AUA]\}$ (which can change with time)
session s_i has permissions $\cup_{r \in roles(si)} \{p | (\exists r'' \leq r)[(p, r'') \in PA \cup APA]\}$

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

Figure 2: RBAC96 Formal Model, from [SCFY1996]

1.3 Problem Statement

Conventional RBAC was designed with a closed-enterprise environment in mind where a team of security officers manually assign users to roles. However, the landscape of business and information technologies has changed dramatically in recent years. An increasing number of service-providing enterprises make their services available to their users via the Internet. There has been some work to extend present RBAC models so they can be used to manage users' access to the enterprise services and resources over the Internet [FBK1999], [PSA2001] and [PSG99].

On another front, many enterprises have users (i.e. workers and/or clients) whose numbers can be in the hundreds of thousands or millions [KSM2003]. Typical examples are banks, utility companies, insurance companies and popular Web sites, to name a few.

For such enterprises, although manually assigning internal users to roles may be feasible, this is not always true for external users, i.e. the enterprise customers and business partners, due to their numbers. Also, automated assignment gives the enterprise an edge by extending its user-consumer business partnership.

Moreover, RBAC is being supported by software products designed to serve large number of clients, such as popular commercial database management systems, e.g. Oracle, Informix, and Sybase [RS1998].

All of these factors mentioned above render the manual user-to-role assignment a formidable task for the following reasons:

a. Maintaining up-to-date user role assignment for a large number of users is a costly endeavor because it requires a large team of individuals.
b. Also, as indicated by a study conducted by NIST, membership in roles tend to change relatively fast in comparison to changes in permission-role assignment [SCFY1996]. This compounds the problem and increases the cost even more.
c. Manual assignment is also error-prone, in particular when the number of users is large and their assignment to roles is in flux.

In fact, some enterprises with large customer bases have already implemented systems that assign and revoke users automatically [KSM2003], and many of them have achieved 90-95% automation of administration [Ker2002]. This demonstrates that there is an urgent need for developing a model that provides a sound conceptual basis for the automation process and sets a benchmark for software implementations of the process.

1.4 Thesis Statement

The central thesis of this dissertation is that it is possible to modify and extend RBAC96 to automate user-role assignment (authorization) using rule-based approach based on users' attributes. Consequently, the central features of RBAC96 (such as roles hierarchies, constraints specification) and RBAC96 administration can be specified using users' attributes.

1.5 Suggested Solution

1.5.1 Motivation

An appealing solution is to automatically assign users to roles. As mentioned above, some enterprises with large customer bases have already implemented systems that assign and revoke users automatically and many of them have achieved 90-95% automation of administration. The need for a model that specifies implicit (automatic) user-role assignment stems from the following:

a. The model provides the theoretical basis and specification for the automation process and its implementation.
b. It also sets a benchmark for software implementations of the automation process.

1.5.2 Description

My work in this dissertation has two main components:

1.5.2.1 Rule-Based RBAC (RB-RBAC) Family

In conventional RBAC, the decision to assign /revoke a user to/from a role is done by humans based on some input, typically changes in users' attributes. Hence, it makes sense to base the automation of user-role assignment on these attributes. Also, the automation process should take into account any constraints laid down by the enterprise.

In this dissertation, we describe a family of models which provides the specification needed to automatically assign users to roles based on a finite set of authorization rules defined by the enterprise, hence the name Rule-Based RBAC or RB-RBAC for short. These rules take into consideration the attributes that users possess and any constraints set forth by the enterprise. The location of storing users' attributes is irrelevant as long as the system that implements RB-RBAC can obtain these attributes with high assurance to make an authorization decision. Consequently, attributes may be stored in databases that are local or remote, under or outside the control of the system that implement RB-RBAC, and so on.

At face value, the idea seems very simple and straightforward. Detailed analysis presented in this dissertation, however, reveals the existence of complicated aspects and subtle issues that need to be addressed. A system that implements RB-RBAC takes as an input:

a. A set of user attributes, which is a collection of data relevant to the authorization process.
b. A set of authorization rules that governs user-role assignment.

The main features of RB-RBAC family are:

i. It provides a family of languages to express user-role assignment rules. This is necessary if RB-RBAC is to express the user-role assignment which is manually performed in RBAC96.

ii. It also defines seniority among the rules, which induces a hierarchy among the roles. The need for this feature arises from the fact that RBAC96 recognizes role hierarchies which represent the inheritance of permissions among roles.

iii. It provides the means to specify three important classes of prohibition constraints. This is a central feature of RBAC96. For RB-RBAC to be useful, it should be able to express these classes of constraints.

iv. It can be configured to express Mandatory Access Controls (MAC) and Discretionary Access Controls (DAC), which are widely used. An advantage of RBAC96 is that it can be configured to express both MAC and DAC. Utilizing user-role assignment is a key ingredient to make that possible. The RB-RBAC family fundamentally changes this key ingredient. Consequently, RB-RBAC should be designed such that this feature of RBAC96, that is simulating MAC and DAC, is retained.

The family is composed of three models: Model A, Model B, and Model C which will be analyzed in chapters 3, 4 and 5, respectively.

Some of the results presented in this dissertation were published in a preliminary form in [AS2002] and [AS2003].

1.5.2.2 Administrative RB-RBAC:

This is the second component of my work. Relying on users' attributes to perform user-role assignment has implications on how we administer RB-RBAC model. We propose a

companion administrative model, namely ARB-RBAC, whose specifications are based on users' attributes. It allows the authorized individuals to administer users' attributes and authorization rules.

1.5.3 The Scope of my Work

To put matters in perspective, the formalization of RB-RBAC and ARB-RBAC falls in the model layer of the OM-AM framework suggested by Sandhu [San2000]. This framework is designed to help separate the security objective (*what we want to achieve*) from the underlying architectures and mechanisms used to implement it (*how to achieve what we want*). The Objective and Model layers (Figure 3) state *what* the security objectives and trade-offs are, while the architecture and mechanism layers address *how* to meet these requirements. Since my work falls within the model layer of the OM-AM framework, no architecture or implementation mechanisms are discussed in details.

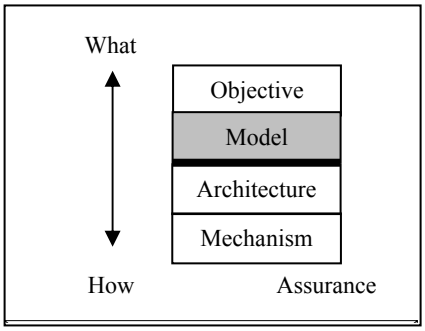

Figure 3: OM-AM Framework

Many architectures and mechanisms exist to support RBAC96. It is our stance that some of these architectures and mechanisms can be utilized for RB-RBAC with appropriate modification.

1.5.4 Assumptions

This work is based on the following assumptions:

1. For simplicity a user in this model is a human being, although the concept of a user can be generalized to include computer processes and the like.
2. Users are properly authenticated before the system that implements RB-RBAC model is triggered to assign them roles. Issues related to identification and authentication are beyond the scope of this dissertation.
3. Users' attributes can be obtained with high assurance via several methods, including providing them along with the authentication information or by fetching from data repositories. Obtaining attributes, though critical for implementing RB-RBAC, does not fall within the model layer in OM-AM framework. As a result, we consider it out of scope and will not discuss it in detail.

It is to be noted that in many real-world enterprises, the number of users is much larger than the number of roles (such as hundreds of thousands or millions of users versus tens or hundreds of roles)[Ker2002] and [KSS2003]. This makes the case for RB-RBAC even stronger.

1.6 Summary of Contributions

The principal contributions of this dissertation are summarized in the following:

a. The formalization of a new family of models called RB-RBAC for the automation of user-role assignment based on a set of authorization rules. These models provide languages to express the authorization rules.
b. The definition of the seniority relation that might hold among authorization rules.

c. The introduction and formalization of the concept of Induced Role Hierarchies (IRH), which are derived from the seniority relation.

d. The analysis of possible discrepancies between the IRH and a Given Role Hierarchy (GRH) that represents the business practices of the enterprise. The IRH represents the formal security policy of the enterprise, while the GRH is the *de facto* security policy. My work helps in identifying any discrepancies between the two policies and provides some insight into their probable reasons and how to reconcile them.

e. Identification and analysis of possible conflict among authorization rules and suggestion of policies to resolve the conflict. Conflict resolution policies are discussed.

f. Analysis of negative authorization in RBAC context. This issue received little attention in the RBAC literature since it is typically discussed in the context of DAC.

g. Providing the syntax and specification that allow the specification of limited scope constraints and invariants in one model.

h. Analysis of three classes of prohibition constraints, their semantics in RB-RBAC context, and their impact on IRH. We introduce different types of constraints within these classes. Also, we suggest three different methods of specifying constraints within RB-RBAC. These methods are compared and contrasted to determine their relative strengths and weaknesses.

i. Introducing a companion administrative model, namely ARB-RBAC, whose specifications are based on users' attributes. This allows the authorized

individuals to perform administrative tasks, such as the administration of the users' attributes, authorization rules, and delegation.

j. Introducing novel concepts to RBAC administration, namely *can_assume* relation which gives security officers a new scope of authority over the user-role assignment process.

k. Specifying *can_delegate* relation that allows users to delegate roles if permitted by the security policy. Our work allows new semantics for delegation based on users' states.

1.7 Organization of the Dissertation

The rest of the dissertation is organized as follows:

- Chapter 2: Gives background about work related to the topic of automating user-role assignment. This includes discussion related to some suggested models, constraints specifications, negative authorization, and RBAC administration.
- Chapter 3: Describes the basic member of the RB-RBAC family, i.e. Model A, which provides implicit user-role assignment in a constraint-free environment. This model provides insight into the main aspects of RB-RBAC, such as sessions, user states, seniority among rules, assembling IRH, IRH/GRH comparison and contrast, etc.
- Chapter 4: Describes Model B, which extends Model A to allow, using the extended syntax, the specification of negative authorization which prohibits users from activating certain roles. As a means to specify separation of duty (SOD) constraints, Model B allows designating certain roles as mutually exclusive roles.

However, it should be noted that only a user-centric SOD can be expressed using Model B.

- Chapter 5: Describes Model C, which also extends Model A to allow the specification of three classes of constraints: Separation of Duty (SOD), cardinality, and prerequisite constraints. Model C provides three different ways to specify these classes of constraints. The chapter concludes with a comparison between Models B and C.
- Chapter 6: Shows how RB-RBAC can be configured to express MAC and DAC. An important advantage of RBAC96 is its ability to simulate MAC and DAC. Since we are proposing RB-RBAC as a replacement for RBAC96 in certain environments, it is crucial to show that RB-RBAC retain that feature of RBAC96. This is not self-evident since RB-RBAC modifies RBAC96 user-role assignment which is central in the simulation of MAC and DAC.
- Chapter 7: Presents the administrative RBAC (ARB-RBAC) model which includes administering the following:
    - Users' attributes
    - The authorization rules
    - SSO direct authorization using *can_assume* relation
    - User-to-user delegation via *can_delegate* relation
- Chapter 8: Concludes the dissertation, reiterates its contributions, and points to future work.

Chapter 2: Background

User-role assignment is an important component of RBAC96 model, or any RBAC model for that matter. However, it is also a labor intensive, error-prone and costly process. Automating this component of RBAC is crucial for utilizing the model, especially for enterprises with large customer bases. Limited attention has been directed to this component in the RBAC literature. There are many aspects involved in the automation that require thorough analysis and formal specification. In this chapter, we review related work in regards to the following areas:

a. The rule-based models
b. Constraint specification
c. Negative Authorization
d. RBAC Administration

2.1 Rule-based Models

Within RBAC literature, there is limited work in this sphere. The first work we cite is that described in [HMM2000] by Herzberg et al. who presents a Trust Establishment (TE) system that defines the mapping of strangers to predefined business roles, based on certificates issued by third parties. Part of the proposed system is an XML-based Trust Policy Language to map users to roles using well-defined logical rules. Each role has one or more rules defining how a client can be assigned to that role. The TE system gathers

certificates related to a specific client and makes a decision regarding the client's eligibility for a specific role. The TE system does not pay attention to relations that might exist among different rules. Another drawback in the TE system is that it is based on bottom-up buildup of the public key infrastructure (PKI), which brings in all the issues related to PKI.

Zhong et al. proposes a scheme to use RBAC on the Web and a procedure for user-role assignment [ZBM2001]. Based on legitimacy of information gathered, assignment policies, and the trustworthiness threshold specified by system administrators, the scheme assigns a client to a role. Users' trustworthiness represents the degree to which the enterprise believes that a user will not do harm to its systems. It is accumulated gradually over time and drops if harmful actions or potentially harmful actions are discovered. There is a major drawback to this approach. A malicious user may use the system for an extended period of time without performing any suspicious acts and, hence, he may acquire highly privileged roles, enabling him to eventually inflict damage on the system. Also, the scheme depends on many security parameters, which must be given initial values. This approach leaves the determination of these values to system administrator(s), but does not provide any guidelines on how to determine them.

Lightweight Directory Access Protocol (LDAP) targets management applications and browser applications that provide read/write interactive access to directories supporting the X.500 models [LDAP1997]. Roles can be stored in directories and retrieved when needed. LDAP has been augmented to support dynamic groups. A dynamic group is an object with a membership list of distinguished names that is dynamically generated using LDAP search criteria. The dynamic membership list may then be interrogated by LDAP

search and *compare* operations, and be used to identify a group's access control subjects [LDAP2001]. This feature could be used to automatically assign users to roles in large enterprises. However, implementing LDAP solely for the sake of dynamically assigning users to roles is an unwieldy solution. Also, LDAP lacks two important features of RBAC: role hierarchies and constraints specification. LDAP returns a simple list of attributes (which represent roles in our case) with no logical structure attached to them. Also, LDAP does not provide a simple way to express constraints.

J. Bacon, K. Moody, and W. Yao suggested Open Architecture for Secure Interworking Services (OASIS) which aims to enable autonomous management domains, called services, to specify their own access control policies and interoperate using service level agreements (SLA's) [YMB2001] and [BMY2002]. OASIS is rule-based in the sense that role activation is linked to satisfying the rules associated with roles. OASIS follows permission-takes-precedence, so it is sufficient for a user to satisfy any of the activation rules associated with a role to activate that role. With each *role activation rule*, there is a companion *role membership rule* of similar structure such that each rule has a list of conditions in the left hand side that are necessary to satisfy in order to activate the role in the right hand side of the rule. These conditions are of three types:

a. Prerequisite roles: OASIS puts a premium on this type of condition since OASIS does not recognize the classical role hierarchies. Instead, OASIS applies prerequisite roles to develop a directed graph structure to represent the run time dependency of each role on its pre-conditions. Stating r_j is a prerequisite role for r_i requires a user to be active in r_j in order for him to activate r_i, in contrast to

RBAC96/ARBAC97 which necessitates the user to be assigned to r_j but does not require that he is active in r_j.

b. Appointment: OASIS introduces the notion whereby being active in certain roles carries the privilege to issue appointment certificates to other users. This is done via credentials. This differs from delegation in the sense that the appointer does not have to be a member of the role being appointed. The appointments have a lifetime and OASIS provides four ways to revoke the appointments.

c. Constraints: These are basically environmental constraints such as time and place of the role activation or service invocation. Each environmental constraint is considered as an atomic proposition. This type of condition is not fully developed in OASIS and no syntax was given to specify them.

The conditions of a *role activation rule* are tested at time of activation. Its companion *role membership rule* indicates which of the role activation conditions must remain true while the role is active. Roles are aggregated in several independently administered security domains which OASIS calls services. When a role is activated in a service an event channel is created and is associated with each membership condition. An event is triggered immediately if any such condition becomes false, causing the role to be deactivated. If any of the related conditions lies within another service, then a corresponding event channel is established with that service.

OASIS recognizes a special class of role activation rules called the initial. They provide the means to allow users to start a session by acquiring initial roles. Explicit user-role assignment is restricted to initial roles.

In contrast to OASIS, which disregards role hierarchies, RB-RBAC recognizes two types of hierarchies. The model helps in analyzing them to determine how compliant the current business practices are with the stated security policy. The analysis provided with the model helps reconcile the practices with the policy in case of discrepancy and gives insight into the appropriate corrective actions needed.

OASIS is rich in terms of expressing rules. We believe, however, that eliminating role hierarchies is a debatable issue, to say the least. Role hierarchies have values not only from the user-assignment perspective of roles but also from the permission-assignment perspective. Also, by making the hierarchies implicit via side effects of role activation rules, the model does not explicitly capture various relations that might exist among roles.

The Enterprise Role-Based Access Control (ERBAC) model had been implemented as a basic concept of a Security Administration Manager (SAM) Jupiter, which is a commercial security administration tool [Ker2002] and [KSM2003]. SAM Jupiter relies on an automation process that uses users' attributes, usually stored in a human resources database, to automatically assign users to roles. However, no formal model is given to describe this process.

There has been some discussion in the literature about the role's life cycle in the context of enterprise security management as the one presented in [KKSM2002]. The discussion highlights the importance of automating user-role assignment but stops short from providing the formal basis for it.

Lastly, we cite the work done by Winsborough et al. in the area of Automated Trust Negotiation (ATN). They aim to serve the needs of collaborative environment by basing

access control decisions on authenticated attributes of entities (i.e. organizations, users, or process in the system) [WL2002]. Authorization decisions are based on the attributes of the requestor, which are established by digitally signed credentials. To protect information of sensitive attribute they introduce the notion of attribute acknowledgment policy. They also introduce a language to express attributes credentials. We believe that this work is complementary to ours since they focus on the storage, retrieval and exchange of credential that contains users' attributes. In RB-RBAC, where we assume that attributes can be obtained with high assurance, the approach suggested by [WL2002] is one way of obtaining the attributes. We focus on what to make of these attributes once they are securely obtained.

2.2 Constraint Specification

In the context of the RBAC96 family and related models such as ARBAC97, constraints are invariants that must hold at all times. Traditionally, the classes of constraints mentioned in context or RBAC96 are SOD constraints, cardinality constraints, and prerequisite role constraints [Ahn1999], [SCFY1996] and [AS2001].

2.2.1 SOD Constraint

SOD constraints are a major class of authorization constraints aimed at preventing fraud and errors, known and practiced since the beginning of commerce. A typical example is that of mutually disjoint organizational roles, such as those of purchasing manager and accounts payable manager. Generally, the same individual is not permitted to belong to both roles because this creates a possibility for committing fraud. There are several types of SOD:

a. Role-Centric SOD: This type enforces mutual exclusion among roles and is further divided into:
    i. Static SOD or strong exclusion: Two roles are strongly exclusive if a single user can never be assigned to both roles [NO1999].
    ii. Dynamic SOD or weak exclusion: Two roles are weakly exclusive if the user cannot activate them simultaneously. [NO1999].
    iii. Session-based Dynamic SOD: This requires that there are no users with two conflicting roles enabled in a session.
b. User-centric SOD [Ahn1999]:
    i. Static: This constraint requires that conflicting users cannot have a common role.
    ii. Dynamic: This requires that there are no two conflicting users active in the same role.

There are other types of SOD (such as permission-centric SOD) but since RB-RBAC is concerned with user-role assignment, these types of constraints will not be discussed.

2.2.2 Cardinality Constraint

This type specifies the maximum number of members in a role [SCFY1996]. This kind of constraint is useful when some roles can only be assigned to a certain number of users, like a manager of a bank branch, a chairman of a department, etc. Another example that highlights the usefulness of cardinality constraint is enforcing licensing agreements [FBK1999]. The discussion of this type of constraint in the two previously quoted works is limited to the static aspect of cardinality. RB-RBAC extends the notion of cardinality constraint into new types that have valuable applicability.

2.2.3 Prerequisite Constraint

The concept of prerequisite role is based on competency and appropriateness, whereby a user can be assigned to role r_i only if the user is already a member of role r_j [SCFY1996, SBM1999]. In RBAC96, prerequisite role constraint is used in a static sense. Assume we have a project that has four roles: *project*, *tester*, *developer*, and *team leader*. The *project* role is the most junior, *tester* and *developer* roles are both senior to *project*, but none of them is senior to the other. To ensure that, for example, only those users who are already members of *project* role can be assigned to *tester* role within that project, we specify a prerequisite role constraint that states that *project* is a prerequisite of *tester*. OASIS extends the notion of prerequisite role into a session-based notion and makes it part of the conditions that a user must satisfy in order to activate another role [YMB2001].

2.2.4 Discussion

Constraints are discussed in the rule-based models proposed in [HMM2000] and [ZBM2001]. RBAC96 allows specifying constraints as invariants without discussion of a specification language. To specify these invariants, *RCL2000*, a constraints specification language, was developed around RBAC96.

OASIS, on the other hand, does not allow specifying constraints as invariants, but rather, all constraints are rule-specific. As such, specifying constraints that have global applicability is cumbersome because they have to be added to each role activation rule and role membership rule.

Also, OASIS does not provide thorough analysis of the handling of Separation of Duty (SOD) constraints. They suggested the use of negative authorization of roles to specify

SOD constraints, but they deferred this for a future work. Moreover, OASIS suffers the following weakness in specifying constraints:

a. The model does not distinguish between different types of dynamic SODs, e.g. simple dynamic and session-based dynamic SOD.
b. The model is unable to specify static SOD, so important access control policies like the Chinese Wall cannot be enforced.
c. Also, it does not provide the means to specify user-centric SOD.

RB-RBAC allows specifying the three kinds of constraints using RB-RBAC defined sets and functions. It also allows the specification of invariants, rule specific constraints, or a mixture of the two.

Bettini et al. proposes a rule-based policy framework that includes provisions and obligations [BJWW2002]. This framework deals with two types of conditions:

a. Provisions, which are conditions that need to be satisfied or actions that must be performed before a decision is rendered, and
b. Obligations, which are conditions or actions that must be fulfilled either by the users or the system after the decision.

A policy consists of a set of policy rules, where each rule is composed of a header and a body. Both the header and the body are composed of predicates. The body of a rule is similar to the attribute expression in RB-RBAC authorization rules. Some policy rules can be derived from others. Each policy rule is associated with a formula that captures the provisions and obligations that apply to that rule. This formula is called *PO*-formula and consists of predicates. The predicates representing provisions and obligations are assigned numerical weights.

A user request may be satisfied using different rules that could be derived differently, i.e. using different derivations, each associated with potentially different sets of provisions and obligations (called PO set). Based on the numerical weights of provisions and obligations and the semantic relationship among them, the framework provides a reasoning mechanism to derive the best derivation. This work could be utilized in RB-RBAC, as will be discussed in Model C.

2.3 Negative Authorization and Conflict Resolution

Negative authorization is rarely mentioned in RBAC literature, mainly because RBAC Models such as RBAC96 and the proposed NIST standard model are based on positive permissions that confer the ability to do something on holders of the permissions [SFK2000]. Bertino and Bonatti mention negative authorization in the context of enabling/disabling roles in temporal RBAC [BB2001]. However, this issue is extensively discussed in other access control models, especially in the context of open policy. The introduction of negative authorization brings with it the possibility of conflict in authorization, an issue that needs to be resolved in order for the access control model to give a conclusive result. The types of conflicts brought about by the negative authorization and conflict resolution polices are discussed in abundance outside RBAC literature. For example, Jajodia et al. suggests a model that is based on logical authorization language that allows users to specify, together with the authorizations, the policy according to which access control decisions are to be made [JSS1997]. The key components of the model are objects, subjects, actions, and rules. Subjects who may be authorized to perform actions on objects include user, roles and groups. The unit of

authorization is an action on an object. The authorization language expresses the policy by means of rules of different types. One type of rule is used to explicitly authorize users, roles or group. Another type of rule is used to derive further authorization based on those provided by the first type of rule. Any conflict that might arise with respect to authorization derivation is resolved using a third type of rule. Several types of conflicts and conflict resolution policies are suggested. RB-RBAC utilizes some of these policies besides those specified in this dissertation for the first time. In another work, Jajodia et al. has given formal definitions for several policies for authorization propagation and conflict resolution [JSSS2001].

2.4 Administration

Sandhu et al. present a model for role-based administration of roles, known as ARBAC97, which describes decentralized administration and has three components [SBM1999]:

a. URA97 which is concerned with user-role assignment
b. PRA97 which is concerned with permission-role assignment
c. RRA97 which deals with role-role assignment

Two central concepts of ARBAC97 are the administrative ranges and the prerequisite conditions, which together regulate and impose restrictions on the administration of roles. The administrative ranges, or role ranges, reflect the limit of the administrator authority, while the prerequisite condition limits the pool of users to whom the administrator can assign roles. We are interested in URA97 model, so the discussion below is cast within that model which is composed of two sub-models:

a. The URA97 Grant Model: A system security officer (SSO) can delegate the authority of assigning users to roles to one or more junior security officers (JSO) who typically are users in administrative roles. The role hierarchy is divided into role ranges such that each JSO is given authority to explicitly assign users to roles that fall in the corresponding role range. The assignment takes effect only if the prerequisite condition is met. This condition determines, based on membership or lack thereof, the pool of users that JSO can assign to the roles in the range.
b. The URA97 Revoke Model: The JSO is authorized to remove users from the roles that are within his assigned range.

The model suffers the following weaknesses, which were identified in a later version of the model, namely, ARBAC02 [OSZ2003]:

a. URA97 requires multi-step user assignments for roles higher in the role hierarchy and it may require the involvement of two or more security officers.
b. Due to the multi-step assignment, the URA relation holds redundant data.
c. To control the actions of JSOs, URA97 stipulates that a prerequisite condition must be met for an action by a JSO to be legitimate. This condition is meant to restrict the pool of users available to the JSO. URA97 uses the role hierarchy to express this condition. However, the role hierarchy is not flexible enough to address the needs of the real world, which sometimes requires a more flexible user pool.

The cause of the above disadvantages of URA97 is the unnecessary coupling between the pool of users and prerequisite roles. Oh et al. recognize the need to remove this coupling by making the pool of users include organizational structure besides the roles, and hence they suggested the ARBAC02 model which is centered on the idea of user pools and permissions pools. [OSZ2003]. They use the organization structure as the basis for defining user and permission pools instead of prerequisite roles in a role hierarchy.

The above models are designed to manage roles in the context of conventional RBAC where user-role assignment is done manually.

Another concept related to user and permission pools is scope, which is presented by Kern et al. to restrict the administrator's authorities in [KSM2003]. A scope is defined as an abstract concept that is used to collect objects over which an administrator has authority. Objects include users, user-role assignment, roles, role-role assignment,

permissions, and role-permission assignment. The objects are gathered according to one or more criteria such as organizational structure, a cost center structure or even combination of several structures. Although it was stated that business roles (regular roles in RBAC terms) are assigned and revoked automatically, no formalization was provided.

Kuhlmann et al. presents a process using data mining techniques to collect and consolidate human resources information, access control information and other security related user information stored in the enterprise databases. The outcome of the process is to discover knowledge about roles or other control access patterns inherent to the business [KSM2003]. The work does not specify a model for administering the attributes.

2.5 Summary

In this chapter we review the literature relevant to RB-RBAC family. Our review has four foci: Rule-Based models, constraints specification, negative authorization, and RBAC administration models. When appropriate, we briefly discussed how these relate to the RB-RBAC family.

Chapter 3: Model A

3.1 Introduction

The Role-Based Access Control (RBAC) model is used to manually assign users to appropriate roles, based on a specific enterprise policy, thereby authorizing them to use the roles' permissions. As mentioned in the introduction, there is a compelling case for automating user-role assignment. We introduce a family of models to dynamically assign users to roles based on a set of rules defined by the enterprise. These rules take into consideration users' attributes and any constraints set forth by the enterprise's security policy. The Rule-Based RBAC (RB-RBAC) models provide a family of languages (*Authorization Specification Languages* or ASL for short) to express these rules. The models also define relations among rules, provide specification for derived induced hierarchies among the roles, and allow constraints specification.

This chapter contains a discussion of Model A, the basic model of the RB-RBAC family. In contrast to RBAC96, where user-role assignment is done explicitly, the RB-RBAC approach is purely implicit user-role assignment.

As pointed out in Chapter 1, my work falls into the model layer of the OM-AM framework suggested by Sandhu in [San2000], thus, unless needed to illustrate a concept related to the model, no architecture or mechanism issues will be discussed.

Figure 4 shows members of the RB-RBAC family. Model A is the most basic among the family. Model B and C extend Model A in different ways. This modular approach facilitates the analysis of each model. We intend to keep Model A as simple as possible to ease the analysis of the model and to use it as a starting point for analyzing more complicated members of the family. This model allows the specification of a set of authorization rules that can be used to assign users to roles based on users' attributes. Model B extends Model A to allow specifying negative authorization and mutual exclusion among roles. Model C extends Model A to allow constraints specification.

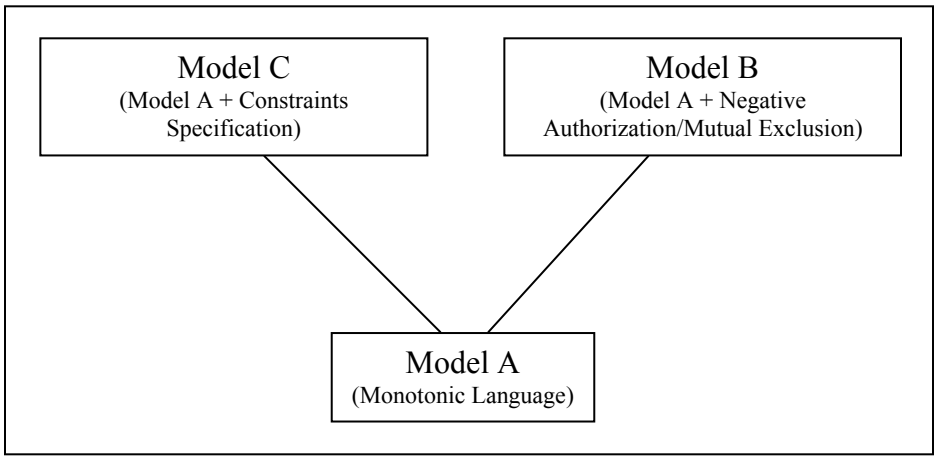

Figure 4: RB-RBAC Family

The main contribution of this chapter is:

a. An informal description of an access control model that allows implicit user-role assignment based on a set of authorization rules.
b. A formal definition of the model.
c. A detailed analysis of the model including how it compares/contrasts to conventional RBAC.

3.2 Analysis of Model A

This is the basic model that provides the foundation for automating user-role assignment. It is also the basic building block to construct more sophisticated RB-RBAC models. I modified RBAC such that user-role assignment becomes rule-based rather than being explicitly assigned. This requires that the enterprise define the set of rules that are triggered to determine which user is authorized to what role(s).

3.2.1 Model A Basic Concepts

Figure 5 shows the main components of the RB-RBAC model[1]:

a) U: The users.

b) AE: The attribute expressions.

c) R: The roles.

d) P: The permissions.

The U, R, and P sets are imported from RBAC96. In RB-RBAC, the security policy of the enterprise is expressed in the form of a set of authorization rules. Each rule takes as an input the attributes expression (a member of AE set) that is satisfied by a user (a member of U set) and produces one or more roles (a member of R set). An attribute expression is a well-formed formula in propositional logic that specifies what combination of attributes values a user must satisfy in order to be authorized to roles specified in the rule. The figure shows that users have many-to-many relation with attribute expressions. One user could satisfy one or more attribute expressions depending on the attributes that he is associated with. Conversely, several users may satisfy identical

[1] The concepts of sessions and role hierarchy are introduced later in this chapter to the RB-RBAC model

attribute expressions. A specific attribute expression corresponds to one or more roles. The figure shows that roles are flat, i.e. no role hierarchy as defined in [SCFY1996] exists among the roles. The figure also shows that a role may correspond to one or more attribute expressions. In conventional RBAC, if a user *u* is deemed by the enterprise to belong to a role, say r_g, the security officer explicitly assigns him to r_g. The system that implements RBAC maintains this information in the UA relation. Assigning *u* to r_g authorizes *u* to activate r_g in a session.

In RB-RBAC, user-role assignment is implicit in the sense that it is done using authorization rules that reflect the security policy of the enterprise. The attributes expressions can be stated using the language provided by the model. Syntactically, a rule has two parts:

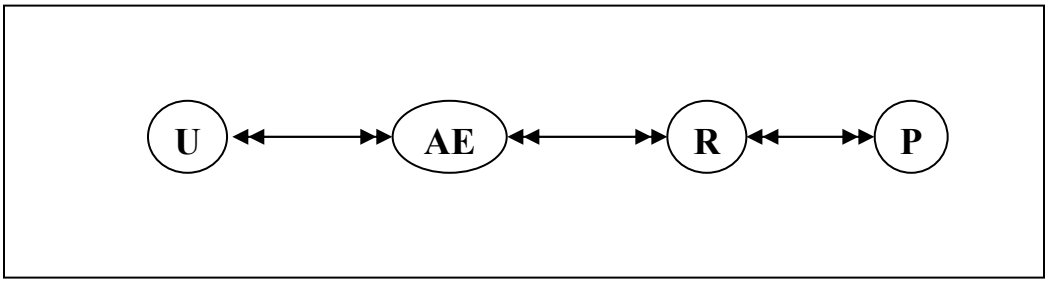

Figure 5: RB-RBAC Model A

a. The left hand side (LHS) of a rule is an attribute expression.

b. One or more role(s) in the right hand side (RHS).

If *u* satisfies the attribute expression, *u* is authorized to the role(s) specified in RHS of the rule. The following is an example of a rule:

$ae_i \Rightarrow r_g$

where ae_i is the attribute expression and r_g is the produced role. If user *u* satisfies ae_i, the system maintains this information in U_AE relation, and this means that *u* is authorized to all the roles in the right hand side of $rule_i$. However, this is not true for all the models and, thus, to keep track of user-role authorization we define the set URAuth as follows:

$$\text{URAuth} = \{(u,r) | (\exists rule_i)[(u, ae_i) \in \text{U\_AE} \wedge r \in RHS(ae_i)\}$$

If $(u,r) \in$ URAuth then this means that *u* is authorized to role *r*. This set is the key component of RB-RBAC since it captures the semantics of the models. Consequently, the definition of this set varies from model to model.

Similar to conventional RBAC, only a user who has authorization on roles that are specified in RHS can activate any combination of these roles. Activating a role enables the user to execute the permissions assigned to that role. As in RBAC96, a user can activate one or more of his authorized roles in a session. Different sessions belonging to the same user can have different roles.

Definition 1

U, R, and P, imported from RBAC96, are the sets of users, roles, and permissions respectively. In addition RB-RBAC Model A has the following components.

1. A set of attribute expressions AE. Elements of AE are denoted as $ae \in$ AE (See the language in section 3.2.5.1).
2. A set of authorization rules where each rule $rule_i$ is written as: $ae_i \Rightarrow$RHS where $\Rightarrow$ is read "generates" or "yields" and RHS $\subseteq$ R.
3. Function $RHS(ae_i)$ = RHS returns the set of roles that user *u* who satisfies ae_i is authorized to activate.

4. U_AE = {(u, ae_i)| (u, ae_i) ∈U×AE ∧ u satisfies ae_i}, (u, ae_i) ∈U_AE means that u is authorized to *RHS*(ae_i).
5. IR is the set of roles produced by all authorization rules:

IR = { r_g | (∃ae_i) [ae_i ∈ AE ∧ r_g∈ *RHS*(ae_i)}

6. URAuth = {(*u*,*r*)| (∃*rule*$_i$)[(*u*, ae_i) ∈U_AE ∧ *r* ∈*RHS*(ae_i)}. For the sake of convenience, we will call the right hand side of this definition as "*A*". We will refer to it in future definitions to simplify the relation of different models to each other.

3.2.2 Rules Invocation

Suppose a user *u* requests a role r_g that is produced by several authorization rules such as the following:

$ae_1 \Rightarrow r_g$

$ae_2 \Rightarrow r_g$

$ae_3 \Rightarrow r_g$

RB-RBAC requires *u* to satisfy at least one of the 3 rules above in order for *u* to be authorized to r_g. This means that there is an implicit “OR” among the rules. If *u* satisfies ae_1 and ae_2, then we say that *rule*$_1$ and *rule*$_2$ are *relevant* rules to distinguish them from *rule*$_3$. The importance of this distinction becomes apparent when discussing constraints in Model C. If *u* satisfies one or more rules that produce different roles, then he is authorized to activate any combination of these roles. Upon receiving a user request of a role, the system that implements RB-RBAC searches the authorization rules set to find a rule which the user satisfies such that the rule yields that requested role.

As a user satisfies more rules, the set of roles that he is authorized to assume does not diminish. Thus Model A is *monotonic*. We will see later in this dissertation that other models do not have this property.

The above discussion is formalized in the following definition.

3.2.3 RB-RBAC User States

In RB-RBAC, a user can be in any of several states *wrt* a specific role. For a given role *r*, we distinguish the following user's states:

a. Assumed (A): user *u* has activated role *r* at least once. Keep in mind that *u* cannot activate *r* unless he is authorized to *r*.
b. Potential (P): user *u* is authorized to role *r* but has not activated it yet.
c. Revoked (R): user *u* has activated role *r* at least once but is not currently authorized to activate it.
d. Not-candidate (N): user *u* has not activated role *r* and is not currently authorized to activate it because he does not have the required attributes for assuming *r*, i.e. *u* is not authorized to *r*.
e. Deleted (Del): user *u* has been deleted from the system by an authorized individual such as the System Security Officer (SSO).

The importance of this distinction among different states of users becomes clear when considering many aspects of RB-RBAC. Take the case of specifying role-based static separation of duty (SOD) constraints. Assume that $\{r_1, r_2\}$ is a set of mutually exclusive roles. User *u* could be a potential user of both roles. If *u* activated r_1 then he can no longer be a potential user of r_2 and his state *wrt* r_2 is changed into N.

Also, when enforcing maximum cardinality constraints pertaining to a given role *r*, potential users should not be taken into consideration since this may create situations where the number of potential users may be greater than the cardinality assigned to *r*. This renders the role inaccessible, which results in depriving users of otherwise legitimate access to *r*.

By the same token, the distinction that we draw between R and N states is useful since it allows the enforcement of policies like the Chinese Wall, where it is critical to keep track of a user's history. Take for example a consultant in a company that provides consultation about competing banks. The company categorizes the information into mutually disjoint conflict of interest (COI) classes where each bank belongs to exactly one COI class. To read information of a bank, one must activate the read role that corresponds to that specific bank. So long as that consultant has not yet accessed any company information about these banks, he has the potential to read information about any bank, i.e. he is in P state *wrt* all roles that allows him to read bank information. Once he activates one of these roles, say r_g, he is to be denied read access to all other banks in the same COI, i.e. his state *wrt* all other roles becomes N. Later, if the consultant changes jobs, his state *wrt* r_g becomes R. If he regains his previous job, the system remembers that he has accessed r_g before. As a result, the system that implements RB-RBAC blocks his access to the rest of the roles.

The state diagram in Figure 6 is with respect to a single role. A user *u* may start as a "potential" user, i.e. in P state, if he has the attributes that satisfy the LHS of the rule that produces role *r*. If *u* does not satisfy any rule that yields *r*, he is considered a "non-candidate" so he will be in N state.

A user state changes according to changes in the following factors:

a. Users' attributes values, denoted by label "ae" in the state diagram.

b. Attribute expressions in the authorization rules, denoted by label "r".

c. Users activating roles which are denoted by label "act".

d. Deletion of users from the system, which is represented by label "d".

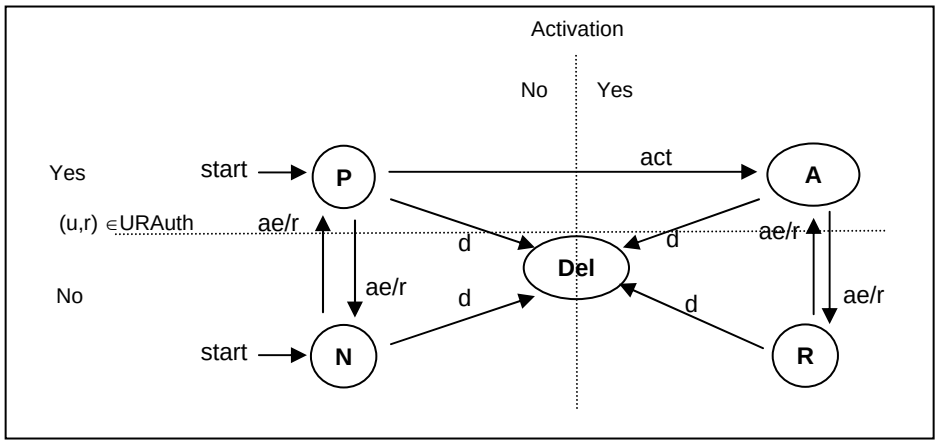

Figure 6: A state diagram of a user with respect to role r

Changes in user's attributes or in attribute expressions of the authorization rules may change the state of the user from P to N and vice-versa. Once a potential user activates a role, say *r*, his state changes into A state. He remains in that state unless one of the following happens:

a. His access right to the role is revoked due to changes in the user's attributes or the authorization rules, i.e. his state becomes R.

b. He is deleted from the system and his state becomes D.

Note that a user can be deleted from the system regardless of his state.

The state diagram can be viewed from different angles. First, if we consider activation, we can vertically break the diagram into two areas. The pre-activation area corresponds to states in which *u* can reside in before activating role *r*. This includes P, N, and Del. The post-activation area includes the states A, R and Del, in which *u* can reside after he activates *r* for the first time. The vertical axes with the label "Activation" represent this distinction. Another way of viewing the diagram is represented by the horizontal axes, which tests whether or not *u* satisfies an authorization rule that produces *r*. The upper part of the diagram that contains states P and A shows that *u* is authorized to *r*, while the lower part contains the states that indicate that *u* is not authorized to *r*.

3.2.4 Sessions

RBAC96 has the concept of a session where 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 of which he is a member. Multiple roles can be simultaneously activated. Each session is associated with a single user for the life of that session. A user may have multiple sessions open at the same time each may have a different combination of roles.

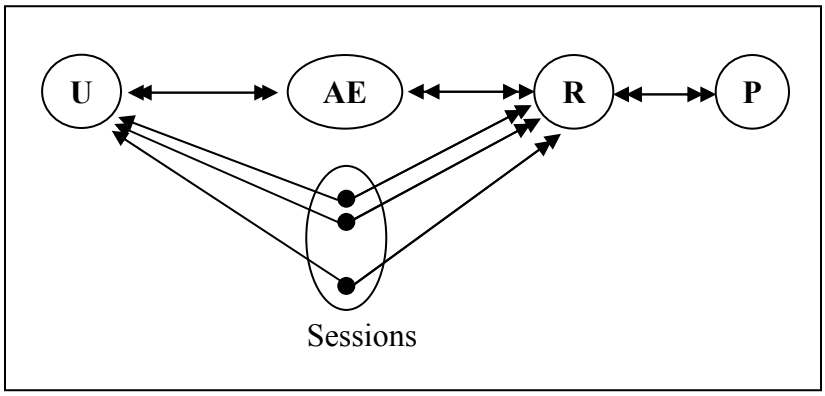

Figure 7: RB-RBAC model A with Sessions

The concept of a session in RB-RBAC is similar to that of RBAC. To represent sessions, "A" user state is further divided into two sub-states: "Act" and "D" which correspond to the "Active" and "Dormant" respectively. "Act" refers to the state where the user is currently active in the role. After deactivating a role, the user becomes dormant with respect to that specific role, i.e. in "D" state. The activation and deactivation of a role demarcates a session. Figure 7 shows RB-RBAC Model A with sessions.

Figure 8 shows a possible scenario of a user using role r_i. The following explains the figure:

t_1: *u* possesses attributes that satisfy a rule which generates *r*. The system that implements RB-RBAC considers *u* a *potential* user of *r* and thus puts *u* in P state *wrt r*.

t_2: User *u* activates *r*, i.e. starts a session. From RB-RBAC perspective, *u* is considered an *active* user, i.e. in Act state.

$t_{2'}$: User *u* ends the session by deactivating *r* and, thus, becomes *dormant* and so the system that implements RB-RBAC updates his state into D.

t_3: User *u* starts another session, i.e. so *u* is considered an *active* user once more, and thus his state becomes Act again.

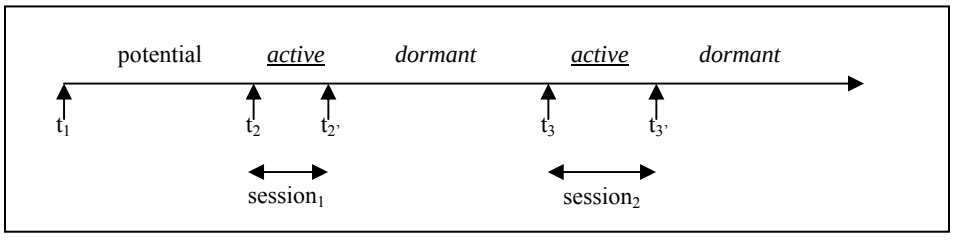

Figure 8: Progression of user's states in RB-RBAC with respect to role r

$t_{3'}$: User *u* ends the session and becomes *dormant* again, i.e. back to state D.

Figure 9 shows the state diagram of a user with respect to a single role with sessions, i.e. after breaking A state into Act and D. It is crucial to keep in mind that when a dormant user *u* in a role *r* wants to reactivate *r*, there must be some rule which *u* satisfies such that the rule yields *r*. This is critical to ensure that, at all times, a user cannot activate a role unless he is authorized to do so.

To motivate the distinction between "Act" and "D" states, consider dynamic SOD, which is enforced among active users only. To specify that constraint, RB-RBAC must be able to differentiate between active and non-active users, which we call dormant. Later in the dissertation, additional justification for this distinction will be given within the context of constraints specification and RB-RBAC administration.

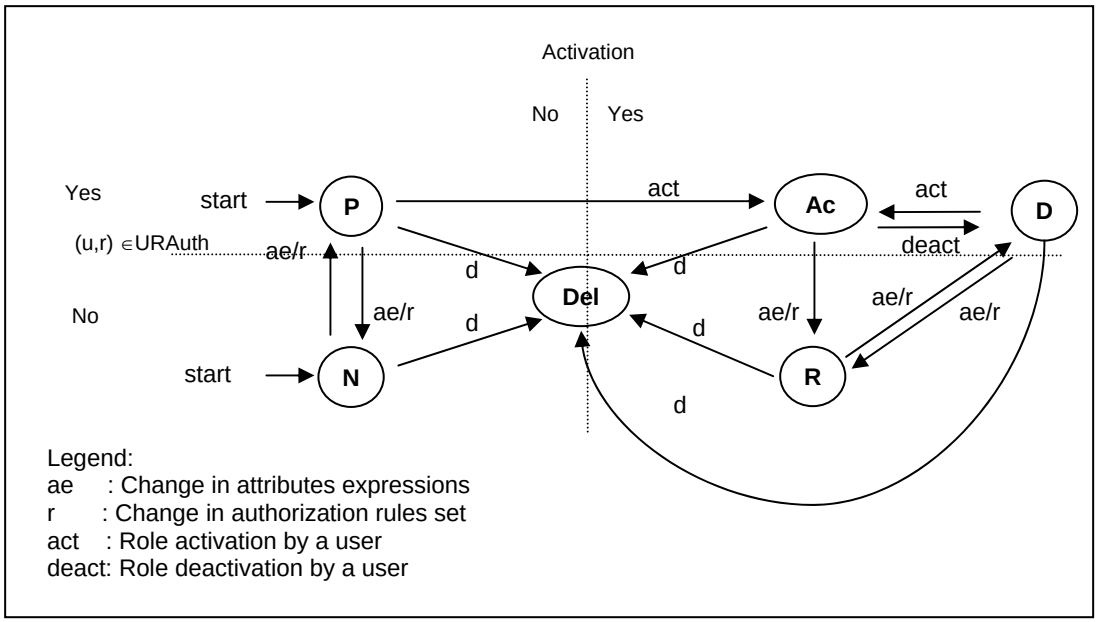

Figure 9: User's State Diagram with Sessions

Because RB-RBAC recognizes the above mentioned user states, the relations URA, URD, URP, URN and URR are defined to hold information about active, dormant, potential, not-candidate, and revoked users respectively.

Definition 2

The concept of session and the functions *sessions* and *user* are imported from RBAC96:

7. *sessions* : $U \rightarrow 2^S$, a function mapping each user u_i to a set of sessions
8. *user* : $S \rightarrow U$, a function mapping each session s_i to the single user *user*(s_i) (constant for the session's lifetime)
9. $URA \subseteq URAuth$, $URA = \{(u,r) | (u, r) \in URAuth \wedge u$ is currently activate *wrt* $r\}$
10. $URD \subseteq URAuth$, $URD = \{(u,r) | (u, r) \in URAuth, \wedge u$ has activated r at least once but is not currently active *wrt* $r\}$
11. $URP \subseteq URAuth$, $URP = \{(u,r) | (u, r) \in URAuth \wedge u$ has never activated $r\}$

    $URAuth = URA \cup URD \cup URP$

    $URA \cap URD = \varnothing$

    $URA \cap URP = \varnothing$

    $URD \cap URP = \varnothing$
12. $URN \subseteq U \times AE$, $URN = \{(u,r) | (u, r) \notin URAuth \wedge u$ has not activated r in the past$\}$
13. $URR \subseteq U \times AE$, $URR = \{(u,r) | (u, r) \notin URAuth \wedge u$ had activated r at least once in the past $\}$
14. *User_State*(u, r) =

    Case:

a. $(u, r) \in$ URP: *User_State*(u, r) = P.

b. $(u, r) \in$ URA: *User_State*(u, r) = Act

c. $(u, r) \in$ URD: *User_State*(u, r) = D.

d. $(u, r) \in$ URR: *User_State*(u, r) = R.

e. $(u, r) \in$ URN: *User_State*(u, r) = N.

f. Del: *u* is deleted by SSO.

These states are mutually exclusive. The state Del is a terminal state.

15. *roles* : $S \rightarrow 2^R$, a function mapping each session s_i to a set of roles *roles*$(s_i) \subseteq \{r \mid (user(s_i), r) \in URAuth\}$ (which can change with time)

3.2.4.1 Revocation

Among the user states that RB-RBAC recognizes is state R, which applies to a user who had activated role *r* in the past but can no longer activate it due to changes in either that user's attributes or the attribute expression of the rule(s) that produces *r*. What if a user was revoked the privilege to access *r* while activating *r*, i.e. while being in a session that involves *r*?

RB-RBAC recognizes 3 modes of action to end the user session:

a. Deferred: In this mode, the system waits until the user finishes the session. Also, it allows him to take full advantage of the role, delegate it to others and receive delegation from others, acquire new roles using this role as a prerequisite, etc. After the user voluntarily deactivates the role, RB-RBAC changes the user's state to R.

b. Graceful: The system limits the user to using the role's permissions and waits until the user voluntarily deactivates the role and then changes the user's state to R. Unlike the deferred case the user can no longer use this role membership for purposes such as delegation or pre-requisite membership.
c. Immediate: In this mode, the system terminates the session immediately and changes the user's state to R.

3.2.5 The Authorization Rules

In this section, we define a language to specify the authorization rules, analyze relations that may exist among the rules, and explain how to derive the role hierarchy they may induce among the generated roles.

3.2.5.1 Specification Language ASL_A

To express authorization rules, RB-RBAC provides ASL_A a language based on a context-free grammar. The production rules of this language and its semantics are discussed below.

3.2.5.1.1 The Production Rules

The terminal symbols: {∧, ¬, <, =, >, ≤, ≠, ≥, IN, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, '{', '}' ,(,)}

The non-terminal symbols: {Attribute_Expression, Term, Relation_Operator, Attribute, Enumerated_Set, Role, Attribute_Value, Roles}

The values of the non-terminal symbols Enumerated_Set, Attribute, Attribute_Value, and Roles are specified by the organization. This makes the language customizable for the organization.

The Start symbol: Rule

The production rules (in BNF notation):

Rule ::= Attribute_Expression||⇒||Roles

The symbol ⇒ means "generates" or "yields" and || is used to denote concatenation. In order for users to be authorized to the role produced by an authorization rule, they have to satisfy the attribute expression mentioned in the left hand side of the rule.

Attribute_Expression ::= Term

| [¬] ||Attribute_Expression

| Attribute_Expression ||∧ ||Attribute_Expression

| (||Attribute_Expression|| ∧ ||Attribute_Expression||)

Attribute expression is a well-formed formula in propositional logic using ¬ and ∧ connectives.

Term::= Attribute||Relation_Operator||Attribute_Value

| Attribute||IN||Enumerated_Set

Examples for terms are:

- Salary > 1500
- Employee IN Salespersons

The enterprise security policy determines the set of valid attributes names. In the above examples, "salary" and "employee" are enterprise-chosen attribute names. The policy also determines the attribute values required in order for users to satisfy the attribute expression of an authorization rule, and, as such, qualify for activating the role produced by that rule. In the first example, users whose salaries

are greater than $1500 satisfy this term. “Salespersons” in the second example is an Enumerated_Set name specified by the enterprise.

Roles ::= Role | {||Role-set||}

Role-set ::= Role | Role||,||Role-set

Accordingly, the set of roles in the RHS of a rule contains one or more roles from which a user who satisfies the rule is authorized to any combination of these roles.

Relation_Operator ::= < | = | > | ≤ | ≠ | ≥

Attribute ::= {*specified by organization*}

Attribute_Value ::= {*specified by organization*}

Enumerated_Set ::= {*specified by organization*}

Role ::= {*specified by organization*}

3.2.5.1.2 The Syntax Diagrams

The syntax diagram of the ASL_A language is shown in Figure 10.

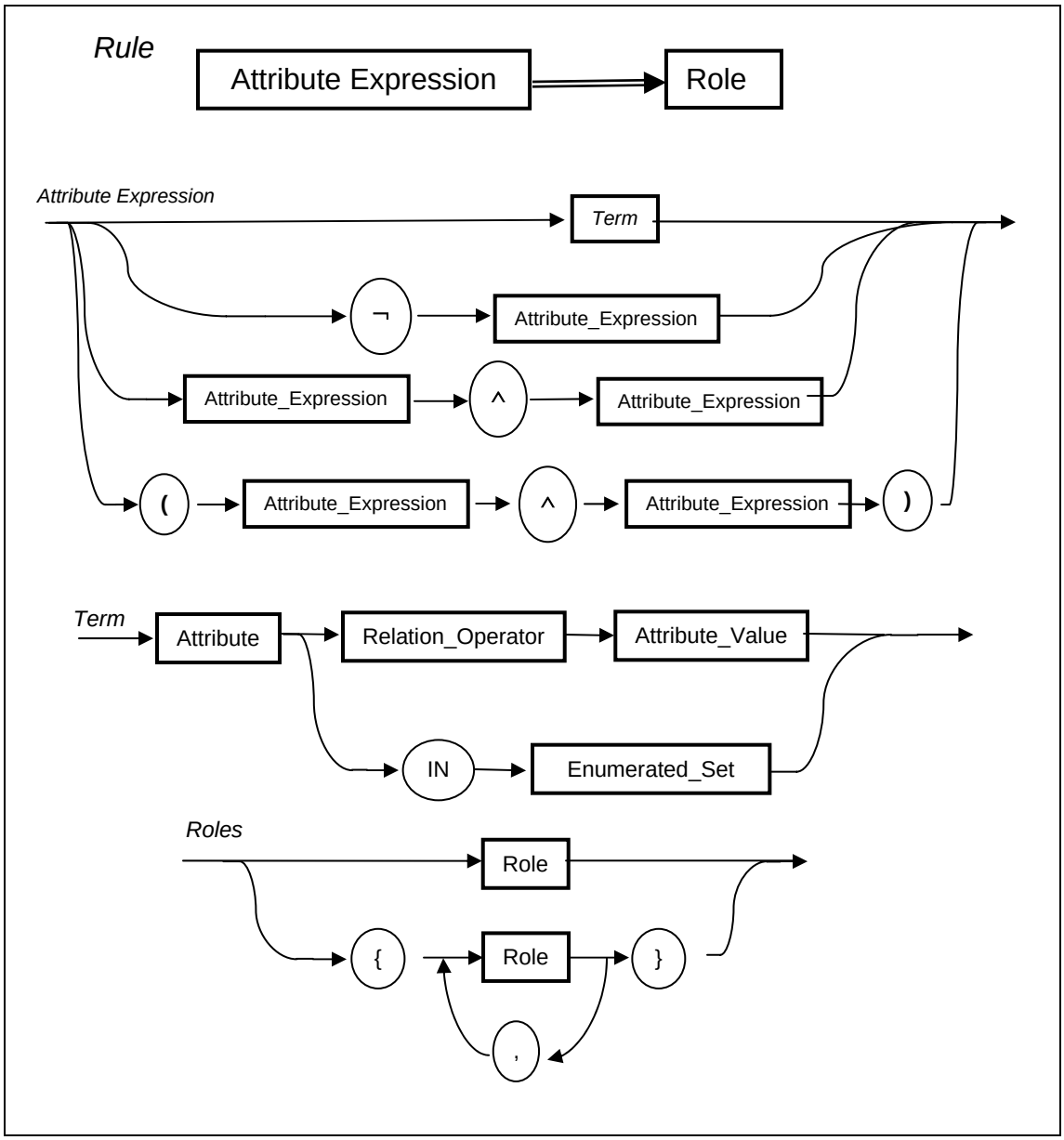

Figure 10: ASL_A language Syntax Diagrams

3.2.5.2 Seniority Among Authorization Rules

Different terms and attribute expressions can have logical relationship with each other. A user who satisfies a term Age ≥ 18 also satisfies a so-called junior term Age ≥ 6. We introduce seniority levels, which are first assigned to the basic building blocks of attributes expressions, namely the terms, and then to the rules. When giving seniority levels to terms of numeric values, the following method is used:

a. For comparative operators {≥, >}, seniority follows the normal order so if x and y are two terms such that x ≥ y, then this implies that x is senior to y.

b. Seniority levels go in reverse order with comparative operators {<, ≤}.

In case of equality operators {=, ≠} and Enumerated_Sets, seniority levels –if they exist- must be explicitly specified.

Clearly, satisfying a term that has high seniority level implies satisfying all the ones that have lower seniority levels (Table 1).

Table 1: Seniority among Terms

	Term	Remarks
1	Age ≥ 18	Term 1 is senior to terms 2 &3 i.e. if 1 is satisfied, then 2 and 3 are also satisfied
2	Age ≥ 13	Term 2 is senior to term 3
3	Age ≥ 6	

$Term_1$ and $term_2$ in Table 1 follow the normal order so $term_1$ is senior to $term_2$, i.e. satisfying $term_1$ implies satisfying $term_2$. Since attribute expressions are well-formed formulas composed of terms, seniority can be determined among these expressions, or in

other words, among the rules whose left hand sides are composed of these expressions. The "≥" symbol, read "is senior to", represents seniority relation among rules:

$$rule_i \geq rule_j \leftrightarrow (ae_i \rightarrow ae_j)$$

where ae_i and ae_j are the LHS of $rule_i$ and $rule_j$ respectively. This implies that users who satisfy $rule_i$ also satisfy $rule_j$ and, hence, are authorized to the roles produced by $rule_j$. In the context of discussing seniority levels, authorization rules and attribute expressions are used interchangeably.

Assume we have the situation shown in Table 2 where the rightmost column in the table shows the relations among attribute expressions of the rules.

Table 2: Relations among Attribute Expressions

Rule	Attribute Expression (LHS)	Roles (RHS)	Relations
$rule_1$	ae_1 = Salary> 1000 **Λ** age > 50	r_1	$ae_1 \rightarrow ae_2$, $ae_1 \rightarrow ae_3$, $ae_1 \rightarrow ae_4$
$rule_2$	ae_2 = Salary> 1000 **Λ** age > 40	r_2	$ae_2 \rightarrow ae_4$ $ae_2 \leftrightarrow ae_3$
$rule_3$	ae_3 = ┐(Salary ≤ 1000 V age ≤ 40)	r_3	$ae_3 \rightarrow ae_4$ $ae_3 \leftrightarrow ae_2$
$rule_4$	ae_4 = Salary> 400	r_4	See the above
$rule_5$	ae_5 = Age > 60	r_5	Not related to any of the attribute expressions

Figure 11 is a graphical representation of the "≥" relation among rules in the table. The twin arrows between ae_2 and ae_3 say that these 2 attribute expressions are logically equivalent.

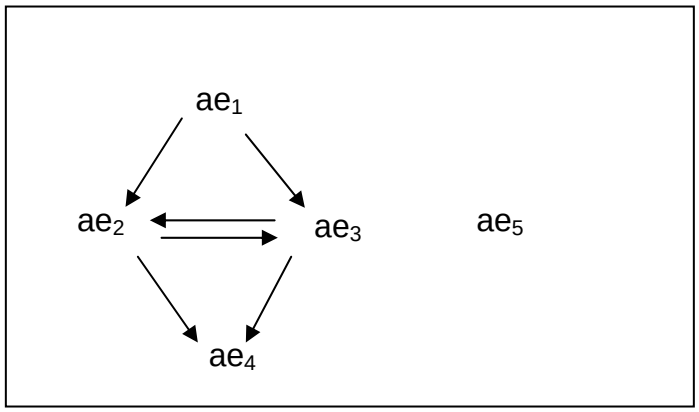

Figure 11: A graph representing seniority relation in Table 2

3.2.5.3 Induced Role Hierarchies

Suppose we have four users: A, B, C and D who satisfy ae_1, ae_2, ae_3, and ae_4 respectively. Since A satisfies ae_1, he is authorized to role r_1. However, since $ae_1 \rightarrow ae_2$, $ae_1 \rightarrow ae_3$ and $ae_1 \rightarrow ae_4$, A is also authorized to r_2, r_3 and r_4. Using similar reasoning, we conclude that B is authorized to r_2, r_3 and r_4; C is authorized to r_2 and r_3 and r_4. In other words, we find that r_1 has one user: A; r_2 has three users: B which it gets from $rule_2$ and A which it inherits because $rule_1 \geq rule_2$, c who is authorized to r_2 because of $ae_2 \leftrightarrow ae_3$. Similarly, r_3 has three users: A, B and C; r_4 has four users: A, B, C and D. This shows a flow of user inheritance among the roles indicating the existence of some role hierarchy. Thus, the "≥"

relation on authorization rules, that is among attributes expressions forming the LHS of the rules, induces a hierarchy among the roles forming the RHS of these rules. This induced role hierarchy (IRH) captures inheritance of user-role assignment. If r_i is senior to r_j then the users who satisfy the LHS of the rule that yields r_i will also satisfy the rules that yields r_j. As a result, the set of r_i users is a subset of r_j users. In other words, user inheritance flows downwards in the IRH graph, that is, a junior role in IRH inherits all the users assigned to its seniors.

IRH captures the enterprise security policy since it is derived from the relations among the authorization rules which represent the policy.

The IRH is formally defined below.

Definition 3

16. $(rule_i \geq rule_j) \leftrightarrow (ae_i \rightarrow ae_j)$.
17. IRH $\subseteq$ IR x IR is a relation such that r_g is senior to r_h ($(r_g, r_h) \in$ IRH is also written as $r_g \geq r_h$):

    $IRH= \{(r_g, r_h) \mid (\forall rule_i)\ [(ae_i \Rightarrow r_g) \rightarrow (\exists rule_j)\ [rule_i \geq rule_j \wedge ae_j \Rightarrow r_h]]\}$

    Intuitively, this means r_g is senior to r_h in IRH if every rule that produces r_g is senior to a rule that produces r_h.

However, this definition is not resilient to changes in the model's semantics. We choose to redefine it in terms of URAuth which reflects users' inheritance as follows:

Definition 4

18. $IRH = \{(r_g, r_h) \mid (u, r_g) \in URAuth \rightarrow (u, r_h) \in URAuth\}$

The elegance of this definition stems from the fact that it confines the changes in the models specification to the definition of URAuth. Intuitively, this definition states that for role r_g to be senior in the IRH to role r_h we require one of the following to be true:

a. $(u, r_g) \in$URAuth via satisfying $rule_i$ such that $r_g \in RHS(ae_i) \wedge (u, ae_i) \in$ U_AE. Here, $(u,r_h) \in$ URAuth either via:
   i. Satisfying $rule_j$ such that $r_h \in RHS(ae_j) \wedge (u, ae_j) \in$ U_AE $\wedge$ $(ae_i \rightarrow ae_j)$.
   ii. Satisfying *can_assume*(r_g, r_h, t, d).

b. $(u, r_g) \in$URAuth via satisfying *can_assume*(r_k, r_g, t, d). In this case, for $(u,r_h) \in$ URAuth to be true, we need *can_assume_with_cascade*(r_g, r_h, t, d), a form of *can_assume* which allows multi-step authorization via *can_assume.*

For Model A only the subcase i of case a applies. The other cases will be relevant after the concept of *can-assume* has been introduced in Section 3.3.

Figure 12 shows different ways of representing IRH that corresponds to the attributes expressions in Table 2. In Figure 12 (a) and (b), we maintain roles r_2 and r_3 as separate entities. However, the authorization rules set that produces Figure 12 (a) will have 2 authorizations rules with logically equivalent attribute expressions such that one of these rules yields r_2 while the other yields r_3. Consequently, any user authorized to r_2 will also be authorized to r_3 and vice-versa.

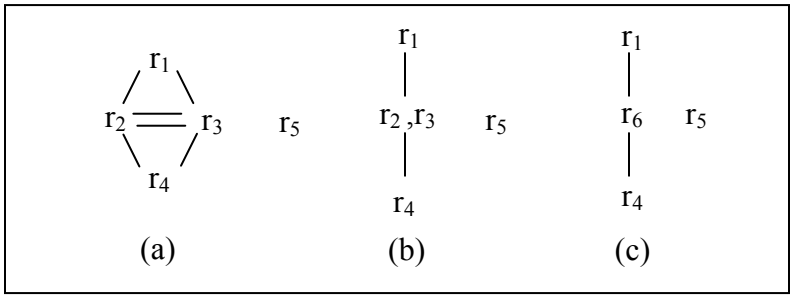

Figure 12: IRH generated by the rules in table 2

The authorization rules can be modified such that one rule produces r_2 and r_3, which corresponds to Figure 12 (b). From a functional standpoint, figures 12 (a) and 12 (b) are equivalent. Figure 12 (c) shows the case in which r_2 and r_3 are collapsed into one role (r_6). From a functional perspective, r_6 is a new role assigned the permissions of r_2 and r_3. From IRH standpoint, the users authorized to r_2 and r_3 are authorized to r_6.

Nonetheless, collapsing roles is not always a prudent course of action for the following reasons:

a) Combining roles of incompatible natures such as a striker and a defender in a video game yields a role that is meaningless in this context.

b) The new role may have a set of permissions that a user does not have to activate at one time in order to perform his task. This violates the principle of *least privilege*, which requires that a user should be given no more privilege than necessary to perform his job.

c) Combining roles may also result in a violation of the principle of *dynamic separation of duties* (SOD). An example of this is the roles of programmer and tester. A user can be assigned to the programmer role, and thus becomes able to perform certain operations on the source code that he develops. Alternatively,

he may choose the tester role where he can test code developed by other programmers but not his. Combining the two roles violates the dynamic SOD.

3.2.5.4 Analysis of the Seniority Relation Among Rules

The seniority among rules is purely a logical implication among attribute expressions that formulate the LHS of the rules. We know that the implication relation is a quasi order because it is reflexive, transitive, but not anti-symmetric since $(ae_i \rightarrow ae_j \wedge ae_j \rightarrow ae_i) \rightarrow (ae_i \leftrightarrow ae_j)$, however, $ae_i \leftrightarrow ae_j$ does not mean $ae_i = ae_j$. In the example above, $(ae_2 \rightarrow ae_3 \wedge ae_3 \rightarrow ae_2) \rightarrow (ae_2 \leftrightarrow ae_3)$ but $ae_2 \neq ae_3$

However, from a functional perspective, ae_2 and ae_3 are equivalent and should be treated as such. Users who satisfy ae_2 also satisfy ae_3, and, thus, are authorized to roles produced by authorization rules that have ae_3 as their LHS. Similarly, users who satisfy ae_3 are authorized to roles produced by authorization rules that have ae_2 as their LHS. Although permissions assigned to r_2 and r_3 could be different, this difference is irrelevant as far as IRH is concerned. IRH does not capture seniority among roles according to the permissions assigned to them. Rather, IRH relation captures user-role assignment inheritance as pointed to above. Accordingly, roles that have identical sets of users can be treated as equivalent. As a result, we can collapse equivalent roles into equivalence classes, a well-known technique that transforms a quasi-order relation into a partial order. Now, we can derive an induced role hierarchy based on equivalence classes, which we refer to as $\mathcal{IRH}$. From user-role assignment standpoint, if a user is authorized to a role that is a member of an equivalence class, then effectively, he is being authorized to every member of that class. Roles have identical sets of users if they are mutually senior to each

other, i.e. belong to the same equivalence class. The above discussion is formalized in the following definition:

Definition 5

19. $\mathcal{IR}$ is the set of equivalence classes that results from defining relation "mutually senior to one another" on IR such that:

    $[r_i] = \{ r_j \mid r_i$ and r_j are mutually senior to one another$\}$

20. $IRH = \{([r_g],[r_h]) \mid \forall u \forall r_g \in [r_g] \forall r_h \in [r_h] [((u,r_g) \in URAuth \rightarrow (u,r_h) \in URAuth) \wedge ((u,r_h) \in URAuth \rightarrow (u,r_g) \in URAuth)]\}$

3.2.5.5 Analysis of the IRH

As discussed above, the definition of IRH yields a quasi-order relation. Being not anti-symmetric is an intrinsic feature since IRH is based on "≥" relation defined on AE, which is also not anti-symmetric.

For the purpose of our discussion, we consider an IRH to be a *well-behaved* relation if it is at least a partial order. To achieve this, we clearly need to impose some restrictions on stating the authorization rules. We will consider two areas that are subject to restrictions:

a. The seniority relations among the authorization rules, or more precisely among the elements of AE. We will analyze the cases in which we require AE to be:
    a) A total order,
    b) A partial order, and
    c) Quasi-order.

b. The number of roles produced by a rule: We will discuss the impact of restricting the cardinality of the roles set produced by authorization rules to one. The impact of cardinality of one or more is also discussed.

These two factors yield six possible scenarios shown in Table 3. We established the consequence of these six cases as follows:

Theorem 1

For each case numbered 1 to 6 in the columns of Table 3 the restriction specified on the "$\rightarrow$" relation on AE and on the number of roles permitted in the right hand side of each rule imply that the IRH and $\mathcal{IRH}$ will be a total (T), partial (P) or quasi (Q) order as indicated in the bottom two rows of Table 3.

Proof:

For all six cases, reflexivity and transitivity of IRH and $\mathcal{IRH}$ follows trivially from the definition. Also, by default, IRH is a quasi-order and $\mathcal{IRH}$ is a partial order. Cases 4, 5 and 6, therefore, follow directly from the definition. Cases 1, 2, and 3 are proven below.

Table 3: Cases Used to Analyze IRH

	CASES					
Restrictions	1	2	3	4	5	6
$\rightarrow$ on AE set:	T	T	P	P	Q	Q
# roles=	1	≥1	1	≥ 1	1	≥1
The resulting induced roles hierarchy based on the restriction on stating authorization rules as in the 6 cases above						
IRH	T	Q	P	Q	Q	Q
$\mathcal{IRH}$	T	T	P	P	P	P

CASE 1:

- Seniority relation on authorization rules (or $\rightarrow$ on AE set) forms a total order.
- Number of roles at the RHS of a rule = 1

IRH is *well-behaved* because it is a total order.

Proof: First, we prove that IRH is a partial order:

To show that IRH is anti-symmetric, consider $r_g \in RHS(ae_i)$, $r_h \in RHS(ae_j)$:

We want to prove that $(r_g \geq r_h \wedge r_h \geq r_g) \rightarrow (r_g = r_h)$ is always true.

By IRH definition,

$$(r_g \geq r_h) \leftrightarrow ((r_g, r_h) \mid (u, r_g) \in \text{URAuth} \rightarrow (u, r_h) \in \text{URAuth})\text{---(1)}$$

Similarly:

$$(r_h \geq r_g) \leftrightarrow ((r_h, r_g) \mid (u, r_h) \in \text{URAuth} \rightarrow (u, r_g) \in \text{URAuth})\text{---(2)}$$

Considering (1) and (2) above, we conclude that $(u, r_h) \in \text{URAuth} \leftrightarrow (u, r_g) \in \text{URAuth}$. The definition of URAuth, there must be at least two attribute expressions ae_x and ae_y such that $ae_x \leftrightarrow ae_y$ and $r_g \in RHS(ae_x) \wedge r_h \in RHS(ae_y)$ and either:

i. $ae_x = ae_y$: But then $(ae_x = ae_y) \rightarrow r_g = r_h$ because there can only be one role in RHS.

ii. $ae_i \neq ae_j$: But since relation "$\rightarrow$" on set AE is a total order, it is anti-symmetric so $(ae_i \leftrightarrow ae_j) \rightarrow (ae_i = ae_j)$ which is a contradiction.

Now, we prove that IRH is a total order: Since "$\rightarrow$" on AE is a total order, it follows $(\forall\, r_i)\ (\forall\, r_j)\ (r_i \geq r_j \vee r_j \geq r_i)$ is true because $(ae_i \rightarrow ae_j) \vee (ae_j \rightarrow ae_i)$ is true.

Since IRH is a total order, it follows that $\mathcal{IRH}$ is identical to IRH. Case 1 is illustrated in Figure 13.

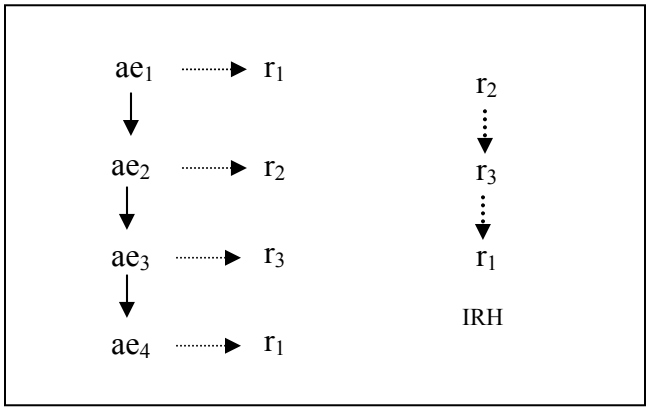

Figure 13: CASE 1 of IRH Analysis

CASE 2:

- Seniority relation on authorization rules (or $\rightarrow$ on AE set) forms a total order.
- Number of roles at the RHS of a rule ≥ 1

IRH is a quasi order by definition. The proof that $\mathcal{IRH}$ is a total order is similar to Case 1 with each role replaced with the corresponding equivalent class. Case 2 is illustrated in Figure 14.

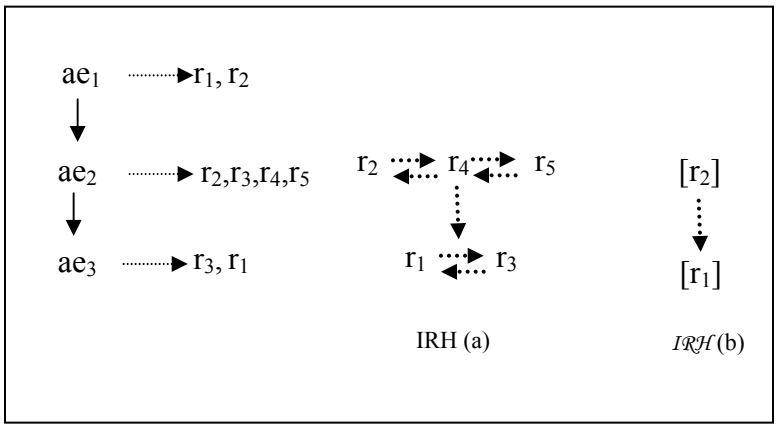

Figure 14: CASE 2 of IRH Analysis

CASE 3:

- Seniority relation on authorization rules (or $\rightarrow$ on AE set) is a partial order.
- Number of roles at the RHS of a rule = 1

The proof that IRH is a partial order follows from the proof for anti-symmetry of IRH in Case 1. However, note that contrary to Case 1, in Case 3 the relation "$\rightarrow$" is a partial order on set AE. Since IRH is a partial order, it follows *IRH* is identical to IRH. Case 3 is illustrated in Figure 15.

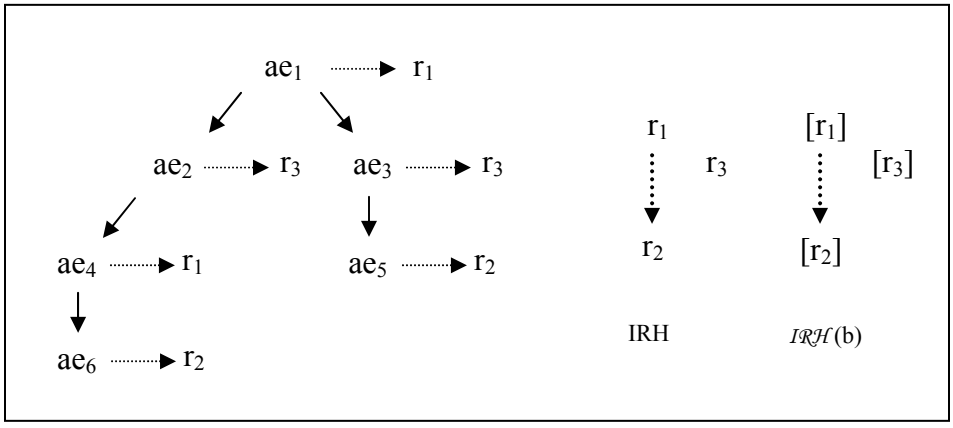

Figure 15: CASE 3 of IRH Analysis

CASE 4:

Cases 4, 5 and 6 follow directly from the definition of IRH and *IRH*. These cases are illustrated in Figures 16, 17 and 18 respectively.

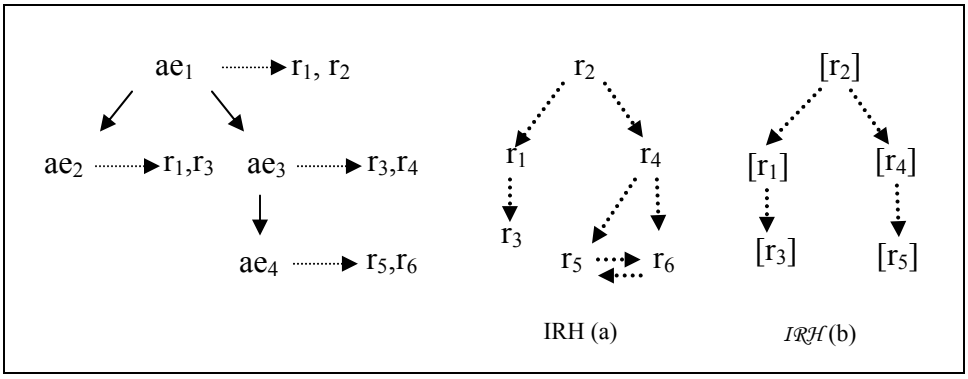

Figure 16: CASE 4 of IRH Analysis

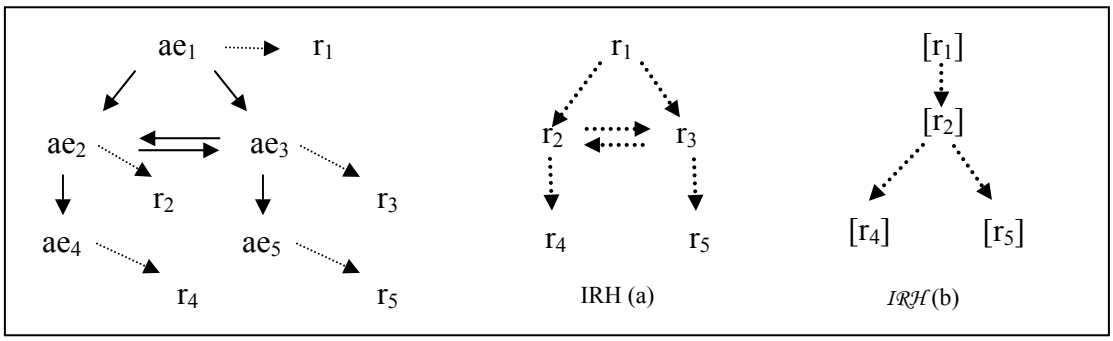

Figure 17: CASE 5 of IRH Analysis

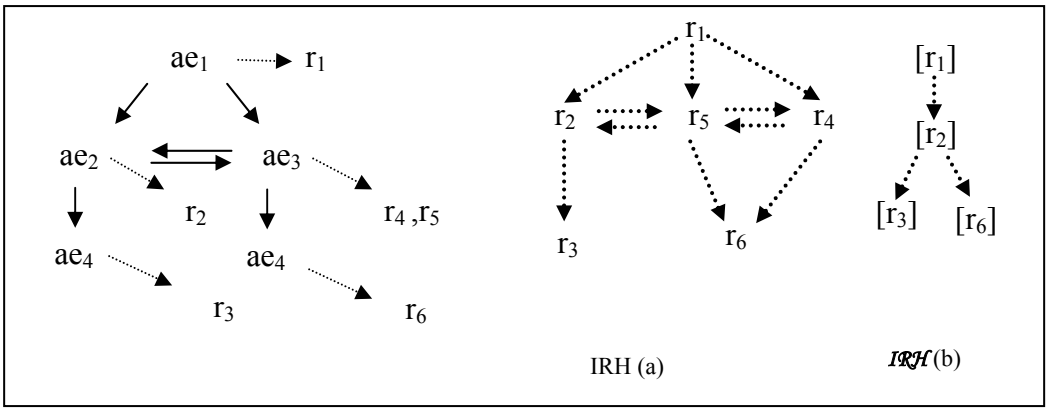

Figure 18: CASE 6 of IRH Analysis

This ends the proof of the theorem.

□

3.2.5.6 Discussion

Cases 1 and 3 of Table 3 indicate that discipline in stating the authorization rules will lead to a *well-behaved* IRH relation. Case 2 further indicates that $\mathcal{IRH}$ will be totally ordered if AE is totally ordered. This case is rather restrictive but could by useful in special circumstances. The remaining cases indicate that we should focus on $\mathcal{IRH}$ rather than IRH in general. The examples presented in the proof also show that we cannot make strong statements about IRH in general. Computing $\mathcal{IRH}$ from IRH is straightforward using well-known algorithms in the literature.

3.2.6 Given Role Hierarchies

So far our discussion of roles is confined to roles derived from the authorization rules, which represent the security policy of an enterprise. IRH is the hierarchy that might exist among these roles. However, roles hierarchies may be derived from other sources. An example of such a hierarchy is the one that is presented by the enterprise to reflect its current business practice. This type of role hierarchy is called given roles hierarchy (GRH), to differentiate it from induced role hierarchies (IRH) [AS2003]. GRH is identical to role hierarchies defined in RBAC96, that is, it is permission-driven:

$$(r_i \geq r_j) \rightarrow r_{j\ \text{permissions}} \subseteq r_{i\ \text{permissions}}$$

where ≥ has the same semantics as in RBAC96.

As such, inheritance of permissions flows upward in the GRH. The introduction of GRH necessitates modifying the definitions of Model A as follows:

3.2.6.1 Possible Discrepancies between IRH and GRH

In an ideal world, IRH[2] and GRH should be mirror images of each other because the business practices of the enterprise (captured in the GRH) is supposed to comply with the security policy (represented by the set of authorization rules) from which IRH is derived. The implicit assumption is that there is correlation between user-to-role assignment and permission-to-role assignment. However, sometimes discrepancies may exist amongst the two hierarchies. This section describes possible discrepancies, their functional implications, and how to resolve them.

Assume we have IRH shown in Figure 19 (a), which was introduced as a by-product of the relations that exist amongst different authorization rules, which are extracted from the

[2] For this discussion, we assume IRH is *well-behaved*, that is, it is a partial order.

security policy. Part (b) of the figure shows a GRH, which represents the actual practice of the enterprise *wrt* permission-role assignment. The two parts of the figure display 2 types of inheritance:

- User-role assignment inheritance, which flows downwards from senior roles to their junior roles (part a of Figure 19).
- Permission-role inheritance, which flows in the opposite direction (part b of Figure 19).

A node in the figure denotes a role, while an edge captures the nature of the relation between the two nodes at its ends. Analyzing the possible discrepancies between the IRH and GRH provides an insight into the meaning of these discrepancies and the ways to reconcile them. Since the roles in IRH and GRH may not be identical we introduce the following notation.

Definition 6

21. *IRH* and *GRH* are the sets of roles in IRH and GRH respectively.

From an IRH perspective, the possible discrepancies between two hierarchies could be classified into the following categories:

3.2.6.1.1 Missing nodes

These are nodes that exist in the GRH but not in the IRH (parts a and b of Figure 19). Depending on the location of the missing node, this category could be divided further into the following categories:

3.2.6.1.1.1 Root Node

Assuming that node r_1 in IRH is missing. Since r_1 is the top-most node in GRH, this could mean one of the following:

a. There is a loss of functionality in the security policy since no user can be assigned to r_1 and use its permissions. In this case, the policy has to be modified by adding an authorization rule that generates r_1.
b. The security policy is designed to split the functions of r_1 among its junior roles. In this case, no changes are needed. This assumes that no permissions are explicitly assigned to r_1 in GRH.

3.2.6.1.1.2 Leaf Node

In the figure, node r_7 is missing in IRH, which means that no authorization rule assigns users to that role. But since the permissions of r_7 are inherited by r_2 and r_3, both are captured in IRH, this scenario neither poses a threat to the system's security, nor does it reduce its functionality. That is:

$(r_i \in GRH \wedge r_i \notin IRH \wedge r_j \in IRH \wedge (r_i, r_k) \notin \text{GRH} \wedge (r_j, r_i) \in \text{GRH}) \rightarrow$ no harm

The term $(r_i, r_k) \notin$ GRH indicates that what is missing (i.e. role r_i) is a leaf node. To reconcile the two hierarchies, r_7 is deleted from GRH and its permissions are added to its immediate ancestor(s). In the real world, r_7 could be the lowest role to which the enterprise does not intend to assign users. Rather, that role is assigned permissions that are common among its senior roles and, thus, it is being used as a building block for constructing these senior roles. If none of r_7 ancestors in GRH belongs to *IRH*, then one of the following is true:

- The security policy, which was used to derive IRH, has overlooked parts of the business practice of the enterprise and, hence, some functionality is missing. In this case, the policy needs to be modified such that an authorization rule will assign users to r_7.
- The business practice followed by the enterprise has created unnecessary roles to which no users are to be assigned. These roles have to be deleted from the GRH and their permissions reassigned.

3.2.6.1.1.3 Internal Node

Role r_3 is missing in the IRH part of the figure. This could result from the enterprise recognizing r_3 as a semantic construct that groups several permissions, but not seeing any need for assigning users to it. From a functionality standpoint, no harm is done so long as at least one of r_3 senior roles is part of IRH, which guarantees that the permissions are accessible. Formally speaking:

$$(r_i \in GRH \wedge r_i \notin IRH \wedge r_j \in IRH \wedge (r_i , r_k) \in \text{GRH} \wedge (r_j , r_i) \in \text{GRH}) \rightarrow \text{no harm}$$

This is similar to the case of leaf node.

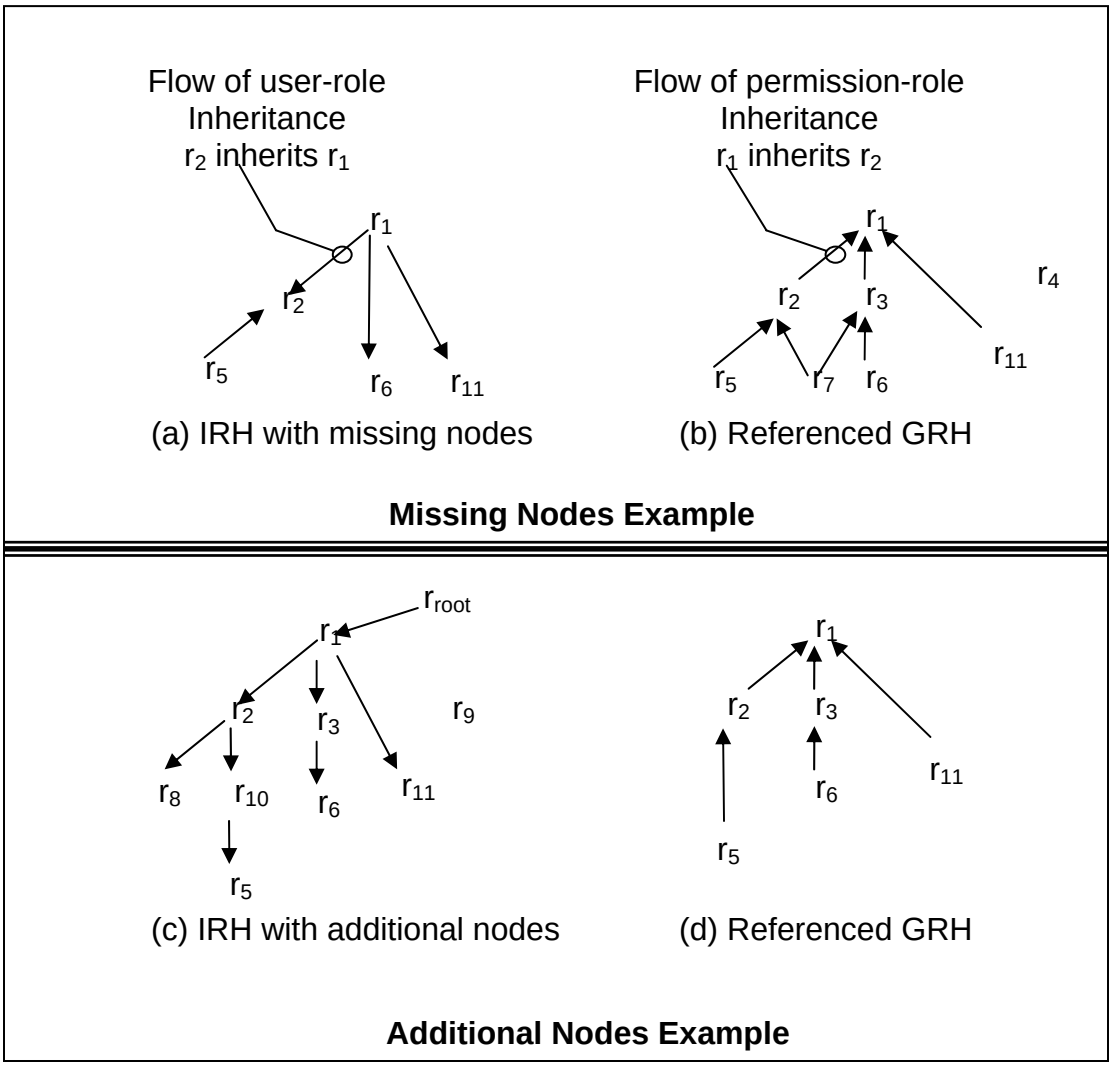

Figure 19: An example of discrepancies between IRH and GRH

3.2.6.1.1.4 Stand-alone Node

r_4 in GRH represents this case. It entails harm only if the following holds:

$$(\forall r_i \in GRH)\ [\ r_{4\ \text{permission set}}\ - \cup(r_{i\ \text{permission set}}) \neq \phi]$$

If the above formula holds, then some permissions of r_4 are not accessible. This indicates a flaw in either the security policy, or the business practice of the enterprise. An example for this case is the security officer who works for a bank but reports to a security company. None of the bank employees are senior to that officer, although the bank's role

hierarchy recognizes his permissions. To remove the discrepancy, we either add an authorization rule that generates r_4, or remove r_4 from the GRH if that does not diminish the functionality.

3.2.6.1.2 Additional Nodes

In the following cases no functionality is ignored by the IRH since all roles in the GRH are present in the IRH. Nonetheless, the security policy has added to IRH empty roles, i.e. they have no permissions associated with them. We use parts c and d of Figure 19 for illustration.

3.2.6.1.2.1 Root Node

Assume there is a role in IRH that is senior to r_1, say r_{root}, then we have 2 possibilities:

- If r_1 is the only child, then $r_{1\ \text{permission set}} = r_{\text{root permission set}}$ which results in functional redundancy. Removing r_{root} from IRH can solve this.
- If r_{root} has more than one child that belong to GRH, then:

$$r_{\text{root permission set}} = \cup\ r_{i\ \text{permission set}}$$

where: r_i is the immediate child of r_{root} in GRH.

GRH may be modified to adopt r_{root}, and thus r_{root} will inherit the permissions of all its juniors.

3.2.6.1.2.2 Leaf Node

In the figure, node r_8 is an example of this case. To reconcile the hierarchies, r_8 must be removed and the security policy must be modified so that the authorization rule(s), which produces r_8, must be altered such that it does not yield this problematic role. Alternatively, the current business practice has to be revised to incorporate r_8 into GRH with the appropriate permissions. IRH provides us with useful insight into the permission set of r_8, that is, it should be a proper subset of the permission set of r_2.

3.2.6.1.2.3 Internal Node

Role r_{10} exists in the IRH but not in GRH. If r_{10} has a single child, which belongs to GRH, then one can assume that r_{10} permissions set is identical to that of its child, however, the set of users assigned to r_{10} is a subset of its child's users set. From a functional standpoint, r_{10} is redundant to its child, r_5 in this case, because its users will be confined to the permissions associated with r_5. Role r_{10} should be removed from IRH and the authorization rules should be modified so they yield r_5 instead of r_{10}. Alternatively, r_{10} can be added to GRH with permission set such that:

$$r_{5\ \text{permission set}} \subset r_{10\ \text{permission set}} \subset r_{2\ \text{permission set}}$$

However, if r_{10} has more than one child, which are nodes in GRH, then r_{10} can be added to GRH such that:

$$r_{10\ \text{permission set}} = \cup\ r_{i\ \text{permission set}}$$

where $r_i \in GRH \wedge r_{10} \geq r_i$

3.2.6.1.2.4 Stand-alone Node

This role has no functional purpose and, thus, has to be discarded. An example for this is r_9. The security policy should be modified by deleting r_9 from the RHS of the authorization rules.

3.2.6.1.3 Missing Edges

The enterprise business practice sees a functional relation, i.e. permissions inheritance between r_1 and r_{11} and captures that in the form of an edge between these two roles in GRH. However, the security policy does not recognize that and, therefore, no user-role inheritance exists between r_1 and r_{11} in Figure 20. In reality, the users assigned to r_1 are capable of utilizing the permissions attached to r_{11} since they are a subset of r_1 permissions even if IRH fails to reveal this relation. This can be eliminated by modifying the policy so that the authorization rule that generates r_1 becomes senior to the one that yields r_{11}.

$(r_i,r_j) \notin \text{IRH} \wedge (r_i \in IRH) \wedge (r_j \in IRH) \wedge (r_i , r_j) \in \text{GRH} \rightarrow$ modify the security policy

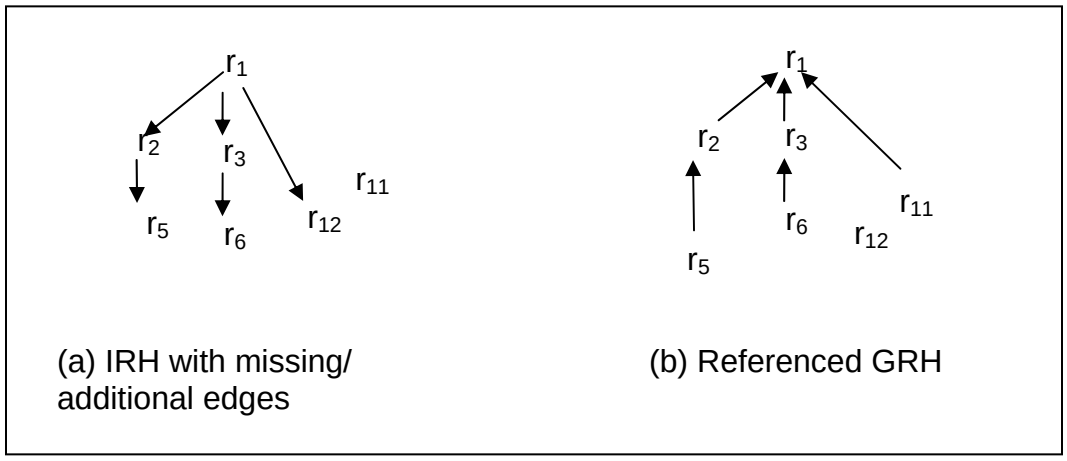

Figure 20: Missing/Additional Edges

3.2.6.1.4 Additional Edges

In Figure 20, IRH has the edge that links r_1 and r_{12}, which GRH does not recognize. From a functional stance, this should not be a problem since it is acceptable to assign a user to roles that are not functionally related. However, if we want to reconcile the two hierarchies, the permissions set of r_1 need to be modified to include that of r_{12}, which results in introducing an edge between the two roles in GRH. Formally:

$(r_i,r_j) \in \text{IRH} \ \wedge (r_i \in GRH) \wedge (r_j \in GRH) \wedge (r_i , r_j) \notin \text{GRH} \rightarrow$ modify r_i permission set

3.2.6.1.5 Inconsistency

Normally, user-role assignment inheritance and permission-role inheritance flow in opposite directions.

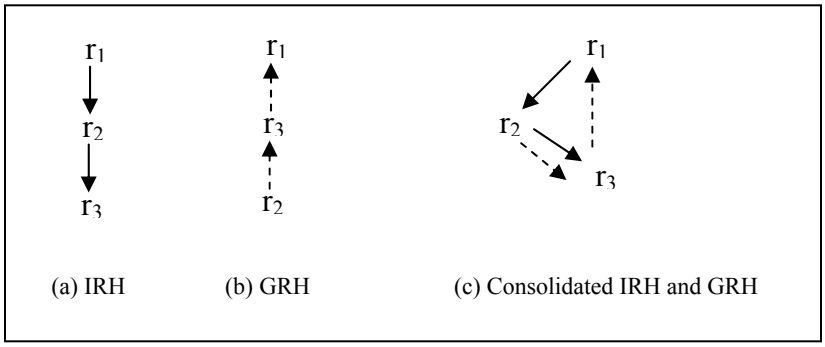

Figure 21: Inconsistency

Figure 21 shows a case in which this normal behavior is violated due to a discrepancy between IRH and GRH. Part (a) of the figure suggests that r_2 is senior to r_3.

Based on this, all users in r_2 will be able to exercise the permissions of r_3 in addition to those of r_2. Accordingly, one can assume that r_3 permissions set is included in r_2. However, part (b) shows that r_2 permissions set is a subset of r_3 permissions set, which indicates that any user assigned to r_3 should be able to use the permissions assigned to r_2.

This result contradicts the one derived from part (a). This contradiction manifests itself graphically as two arrows flowing from r_2 and r_3. Either the policy or the role-permission assignment has to be modified.

3.2.6.2 Discussion

The analysis of IRH/GRH and any discrepancy that might exist between them serves multiple purposes. The discrepancy reveals flaws in the security policy, business practices or both. It also gives insight to about how to fix these flaws and, in some cases, provides guidance regarding permission-role assignment.

3.3 Alternative Ways to Gain Authorization

Thus far, the only way a user *u* may receive authorization to activate a role *r* is to satisfy one or more rule $rule_i$ such that $r \in RHS(rule_i)$. There are situations where this approach is not flexible enough. Thus, to provide flexibility, we do the following:

1. Adopt the concept of delegation.
2. Introduce the concept of *can_assume*.

The semantics, motivation, and formal specifications of these two concepts are discussed in detail in Chapter 7. For now, it suffices us to say that authorized individuals such as the system security officer (SSO) can use *can_delegate* relation to permit regular users to delegate their memberships in specific roles to other users. For example, *can_delegate*(r_g, r_h, *d*, *t*) allows users who are authorized to role r_g to delegate their membership in r_g to users authorized to r_h starting at time *t* for duration of *d*. Also, the SSO may use one form of *can_assume* relation to explicitly authorize users who are authorized to a role, say r_g, to another role, r_h, for a certain duration *d* starting at a specific time *t*. The SSO specifies the duration and the starting time. As a

result, the user(s) in role r_g is authorized to activate role r_h at time *t* for duration of *d*. Introducing these ways of obtaining authorization definitely impacts the definition of URAuth. This impact of introducing *can_assume* relation will be analyzed and formalized when specifying each model. Although similar analysis can be provided for *can_delegate*, due to lack of space such analysis will not be provided. Suppose that an SSO issues *can_delegate*(r_g, r_h, *d*, *t*). Effectively, this means that for each user *u* such that (u, r_g) ∈ URAuth, *u* becomes authorized to r_h until *can_delegate* expires. The modified definition of URAuth is given in the following definition.

Definition 7

22. $\text{URAuth}^{\text{with } can\_assume} = \{(u,r) | (\exists rule_i)[(u, ae_i) \in U\_AE \wedge r \in RHS(ae_i)$

$\vee (\exists rule_j)$ [$(u, ae_j) \in U\_AE \wedge r' \in RHS(ae_j) \wedge can\_assume(r', r, t, d) \wedge$ *can_assume* has not expired])}

Let's call the second term in the right hand side *B*, and hence we say:

$\text{URAuth}^{\text{with } can\_assume} = A \vee B$

Intuitively, this means that there are two ways for users to be authorized to roles: either via satisfying authorization rules, via satisfying a *can_assume* relation or both.

The introduction of *can_assume* calls for reviewing the IRH mathematical properties. The following theorem serves that purpose.

Theorem 2

The IRH that is based on URAuth with *can_assume* is a quasi order.

Proof:

a. For all six cases, reflexivity of IRH follows trivially from the definition.

b. Transitivity: This property means that $(r_g, r_h) \in IRH \wedge (r_h, r_k) \in IRH \rightarrow (r_g, r_k) \in IRH$:

By IRH definition, $(r_g, r_h) \in IRH$ means that $(u, r_g) \in URAuth \rightarrow (u, r_g) \in URAuth$. The way by which a user obtains authorization over the role is irrelevant ------(1).

Also, by IRH definition, $(r_h, r_k) \in IRH$ means that $(u, r_h) \in URAuth \rightarrow (u, r_k) \in URAuth$ ------(2).

From (1) and (2), it follows that $(u, r_g) \in URAuth \rightarrow (u, r_k) \in URAuth$ which means that $(r_g, r_k) \in IRH$. This applies to the six cases.

c. Anti-Symmetry: We will consider cases 1 and 3 only in Theorem 1. This property means that $(r_g, r_h) \in IRH \wedge (r_h, r_g) \in IRH \rightarrow r_g = r_h$. This property does not hold for IRH because of the following:

By the IRH definition, $(r_g, r_h) \in IRH$ could be true because of seniority among rules, i.e. $(\forall rule_i)\ [(ae_i \Rightarrow r_g) \rightarrow (\exists rule_j)\ [rule_i \geq rule_j \wedge ae_j \Rightarrow r_h]]$. On the other hand, $(r_h, r_g) \in IRH$ could be true because of a $can\_assume(r_h, r_g, t, d)$. This does not necessitate that $r_g = r_h$. This proves that IRH is a quasi order.

3.4 Summary

Model A is interesting in and of itself because it lays the formal foundation necessary for the automation of user-role assignment. In this chapter, I discussed the main components of the model, the language it provides to express authorization rules, the relations that might develop among these rules, and how we can use it to derive the IRH and *IRH*. IRH is also analyzed and the possible discrepancies between the IRH and GRH are analyzed,

and used to provide insight into the meaning of these discrepancies and the ways to reconcile them. Also, Model A serves as the basic building block for Models B and C which will be discussed in the following chapters. Figure 22 and 23 summarize the formalization of Model A. For the most part, the discussion will be centered on the URAuth set and how the added semantics affect it.

U, R, and P, imported from RBAC96, are the sets of users, roles, and permissions respectively. In addition RB-RBAC Model A has the following components.

1. A set of attribute expressions AE. Elements of AE are denoted as $ae \in AE$ (See the language in section 3.2.5.1).
2. A set of authorization rules where each rule $rule_i$ is written as: $ae_i \Rightarrow$RHS where $\Rightarrow$ is read "generates" or "yields" and RHS $\subseteq$ R.
3. Function $RHS(ae_i)$ = RHS returns the set of roles that user u who satisfies ae_i is authorized to activate.
4. U_AE = $\{(u, ae_i) | (u, ae_i) \in U \times AE \wedge u$ satisfies $ae_i\}$, $(u, ae_i) \in$U_AE means that u is authorized to $RHS(ae_i)$.
5. IR is the set of roles produced by all authorization rules:
   $IR = \{ r_g | (\exists ae_i) [ae_i \in AE \wedge r_g \in RHS(ae_i)\}$
6. URAuth = $\{(u,r) | (\exists rule_i)[(u, ae_i) \in$U_AE $\wedge r \in RHS(ae_i)\}$. For the sake of convenience, we will call the right hand side of this definition as "*A*". We will refer to it in future definitions to simplify the relation of different models to each other.

The concept of session and the functions *sessions* and *user* are imported from RBAC96:

7. $sessions : U \rightarrow 2^S$, a function mapping each user u_i to a set of sessions
8. $user : S \rightarrow U$, a function mapping each session s_i to the single user $user(s_i)$ (constant for the session's lifetime)
9. URA $\subseteq$ URAuth, URA = $\{(u,r) | (u, r) \in$ URAuth $\wedge u$ is currently activate *wrt r* $\}$
10. URD $\subseteq$ URAuth, URD = $\{(u,r) | (u, r) \in$ URAuth, $\wedge u$ has activated *r* at least once but is not currently active *wrt r* $\}$
11. URP $\subseteq$ URAuth, URP = $\{(u,r) | (u, r) \in$ URAuth $\wedge u$ has never activated *r*$\}$
    URAuth = URA $\cup$ URD $\cup$ URP
    URA $\cap$ URD = $\varnothing$
    URA $\cap$ URP = $\varnothing$
    URD $\cap$ URP = $\varnothing$
12. URN $\subseteq$ U×AE, URN = $\{(u,r) | (u, r) \notin$ URAuth $\wedge$ u has not activated *r* in the past$\}$
13. URR $\subseteq$ U×AE, URR = $\{(u,r) | (u, r) \notin$ URAuth $\wedge u$ had activated *r* at least once in the past $\}$
14. $User\_State(u, r)$ =
    Case:
    a. $(u, r) \in$ URP: $User\_State(u, r)$ = P.
    b. $(u, r) \in$ URA: $User\_State(u, r)$ = Act
    c. $(u, r) \in$ URD: $User\_State(u, r)$ = D.
    d. $(u, r) \in$ URR: $User\_State(u, r)$ = R.
    e. $(u, r) \in$ URN: $User\_State(u, r)$ = N.
    f. Del: u is deleted by SSO.
    These states are mutually exclusive. The state Del is a terminal state.
15. $roles : S \rightarrow 2^R$, a function mapping each session s_i to a set of roles $roles(s_i) \subseteq \{r | (user(s_i), r) \in$ URAuth$\}$ (which can change with time)

16. $URAuth^{with\ can_assume} = \{(u,r) | (\exists rule_i)[(u, ae_i) \in$U_AE $\wedge r \in RHS(ae_i)$
    $\vee (\exists rule_j) [(u, ae_j) \in$ U_AE $\wedge r' \in RHS(ae_j) \wedge can\_assume(r', r, t, d) \wedge$ *can_assume* has not expired])$\}$
    Let's call the second term in the right hand side *B*, and hence we say:
    $URAuth^{with\ can\_assume} = A \vee B$
17. $(rule_i \geq rule_j) \leftrightarrow (ae_i \rightarrow ae_j)$.
18. IRH $\subseteq$ IR x IR is a relation such that r_g is senior to r_h ($(r_g, r_h) \in$ IRH is also written as $r_g \geq r_h$):
    $IRH = \{(r_g, r_h) | (\forall rule_i) [(ae_i \Rightarrow r_g) \rightarrow (\exists rule_j) [rule_i \geq rule_j \wedge ae_j \Rightarrow r_h]]\}$
    Intuitively, this means r_g is senior to r_h in IRH if every rule that produces r_g is senior to a rule that produces r_h.
19. $IRH = \{(r_g, r_h) | (u, r_g) \in$URAuth $\rightarrow (u, r_h) \in$URAuth$\}$
20. $\mathcal{IR}$ is the set of equivalence classes that results from defining relation "mutually senior to one another" on IR such that:
    $[r_i] = \{ r_j | r_i$ and r_j are mutually senior to one another$\}$
21. $IRH = \{([r_g], [r_h]) | \forall u \forall r_g \in [r_g] \forall r_h \in [r_h] [((u,r_g) \in$ URAuth $\rightarrow (u,r_h) \in$ URAuth)
    $\wedge ((u,r_h) \in$ URAuth $\rightarrow (u,r_g) \in$ URAuth)$]\}$

22. *IRH* and *GRH* are the sets of roles in IRH and GRH respectively.

Figure 22: Model A (Part 1)

Theorem:

For each case numbered 1 to 6 in the columns of Table 3 the restriction specified on the "$\rightarrow$" relation on AE and on the number of roles permitted in the right hand side of each rule imply that the IRH and IRH will be a total (T), partial (P) or quasi (Q) order as indicated in the bottom two rows of Table 3.

Proof: Provided in the chapter's text.

Theorem:

The IRH that is based on URAuth with *can_assume* is a quasi order.

Proof: Provided in the chapter's text.

Figure 23: Model A (Part 2)

Chapter 4: Model B

4.1 Introduction

In this chapter we discuss Model B, which extends Model A to allow the specification of negative authorization (Model B_1) and mutual exclusion (Model B_2) by extending the ASL_A language. The extended language is called ASL_{B1} and ASL_{B2}, respectively. This extension has an impact on user authorization, formally represented by URAuth set. Also, it may cause conflict among rules. This conflict is analyzed, and conflict resolution polices are presented; some of them are novel. The definition of URAuth is modified to accommodate the semantics of negative authorization and mutual exclusion. The new definition takes into consideration conflict resolution policies in effect.

Negative authorization is typically discussed in the context of access control systems that adopt open policy. There is an extensive amount of work in this regard, see for example [BSJ1993], [BSJ1997], and [JSMS2001]. However, this issue received little, if any, attention in the RBAC literature. Model B_1 is the first RBAC model that provides detailed analysis of different aspects of negative authorization in an RBAC context. This includes providing semantics, identifying cases of conflict, suggesting conflict resolution policies including novel policies, the impact of negative authorization on URAuth, IRH, GRH and any RB-RBAC enforcement architecture. In Model B_2, conflict among

authorization rules due to mutual exclusion, the impact of mutual exclusion on URAuth, IRH, GRH and any RB-RBAC enforcement architecture are discussed.

4.2 .Analysis of Model B

4.2.1 Negative Authorization (Model B_1)

In the real world of access control, there are two well-known decision policies [JSMS2001]:

a. Closed policy: This policy allows access if there exists a corresponding positive authorization and denies it otherwise.
b. Open policy: This policy denies access if there exists a corresponding negative authorization and allows it otherwise.

Bertino et al. contends that the closed policy approach has a major problem in that the lack of a given authorization for a given user does not prevent this user from receiving this authorization later on. They therefore proposed an explicit *negative* authorization as blocking authorizations. Whenever a user receives a negative authorization, his positive authorizations become blocked [BSJ1997].

In a database context, Bertino et al. defines negative authorization as that if a user has it for a privilege on a table, the user can neither exercise nor administer that privilege. Moreover, a negative authorization for a privilege on a table renders the user incapable of exercising the privilege on the table even though he may have received, or will receive in the future, authorizations giving that privilege. This is particularly important in environments where authorization administration is decentralized and other users, besides the owner of a table, can grant authorization [BSJ1993]. So when Discretionary Access Controls (DAC) is applied, negative authorization is critical.

Bertino and Bonatti mention negative authorization in the context of enabling/disabling roles in temporal RBAC where negative authorization simply means disabling the role [BB2001]. This is different from the semantics given to this concept in RB-RBAC, as will be discussed shortly.

4.2.1.1 The ASL_{B1} Syntax

ASL_{B1} imports the syntactic constructs of ASL_A but it modifies the syntax of Roles as follows:

Roles ::= [¬] Role

role-set ::= Role | Role||,||role-set

4.2.1.2 Semantics

The syntax above allows specifying negative authorization on roles such as the following:

$$ae_k \Rightarrow \neg\, r_i$$

The rule above states that once a user satisfies ae_k the system that implements RB-RBAC prohibits him from assuming r_i. Typically, we assume a closed policy where users are prohibited from activating roles unless explicitly authorized, however, there are situations where negative authorization provides an extra safeguard to prevent users from getting unauthorized access to roles as discussed in the next section.

4.2.1.3 Motivation

The motivations to use negative authorization are not as strong in environments where RBAC is applied. Even though users-role assignment could be decentralized [SBM1999], it is not left to users' discretion to assign other users to roles. Instead a small number of individuals (e.g. SSOs) are entrusted with applying the enterprise security policy regarding user-role assignment. As a result, the possibility of assigning a user to a role that violates the enterprise security policy in RBAC environment is slim. However, since RB-RBAC automates this process, negative authorization provides an extra safeguard, since it is not always easy to foresee all possible combinations of roles a user can assume based on his attributes, which change over time. Negative authorization helps in blocking any user whosoever satisfies certain criteria (expressed as attributes expression) from assuming certain roles. Also, it can be used to block receiving authorization of certain roles via *can_assume* and *can_delegate* relations. The system security officer (SSO) can use *can_assume* relation to explicitly authorize users who are authorized to a role, say r_g, to another role, r_h, for a certain duration *d* starting at a specific time *t*. As a result, the user(s) in role r_g is authorized to activate role r_h at time *t* for duration of *d*. Also, he may use *can_delegate* relation to permit regular users to delegate their memberships in specific roles to other users. These relations are discussed in more details in Chapter 7. To motivate the use of negative authorization in the context of RBAC, consider the example of a military unit that has a *Commander* and 4 staff officers, usually known as *G1* through *G4* as depicted in Figure 24.

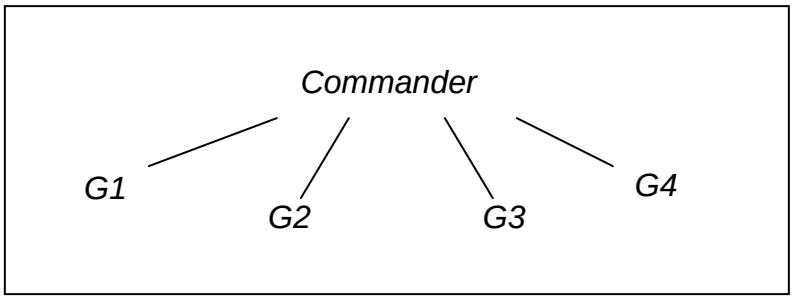

Figure 24 : RBAC Hierarchy for a Battalion

The commander can delegate his authority to any of his staff officers as long as the staff officer meets certain criteria specified by the military regulations. In Table 4, we show a security policy that specifies a possible real world situation. The policy uses negative authorization to prevent a *Commander* from delegating his role to a staff officer whose rank is lower than a Lt. Colonel.

Table 4: Example to Motivate Negative Authorization

Attributes in the System: a_1: rank-type = officer a_2: Staff course = T a_3: Leadership course = T a_4: Rank ≥ Lt. Colonel a_5: Assignment Order = T
Authorization Rules: $a_1 \wedge a_2 \Rightarrow \{G_1, G_2, G_3, G_4\}$ $a_1 \wedge a_2 \wedge a_3 \wedge a_4 \wedge a_5 \Rightarrow$ *Commander* $\neg a_4 \Rightarrow \neg$*Commander*
can_delegate(*Commander*, G_1, *d*, *t*)

4.2.1.4 Conflict Due to Negative Authorization

Introducing "¬" to the RHS may lead to conflict in the state of a single user *wrt* a single role. The conflict is due to simultaneous positive and negative authorizations. Using the

set of authorization rules shown in Figure 25, the following are several variations of this type:

a. Conflict among unrelated rules like the one between *rule*$_2$ and *rule*$_3$. If *u* satisfies *rule*$_2$ and *rule*$_3$ simultaneously then *u* should be authorized to assume r_1 (i.e. *u* is in P state *wrt* r_1) and denied r_1 at the same time (i.e. *u* is in N state *wrt* r_1). This case is represented by the following:

$$(u, ae_i) \in U\_AE \wedge (u, ae_j) \in U\_AE \wedge r \in RHS(ae_i) \wedge \neg r \in RHS(ae_j)$$

b. Conflict among related rules: *rule*$_3$ and *rule*$_5$ are conflicting because if *u* satisfies *rule*$_3$ then he is denied r_1 (i.e. *u* is in N state *wrt* r_1), but at the same time, authorized to assume r_1 (i.e. *u* is in P state *wrt* r_1) because *rule*$_3 \geq$ *rule*$_5$. This case is represented by the following:

$$(u, ae_i) \in U\_AE \wedge (u, ae_j) \in U\_AE \wedge r \in RHS(ae_i) \wedge \neg r \in RHS(ae_j)$$

$$\wedge ((ae_i \rightarrow ae_j) \vee (ae_j \rightarrow ae_i))$$

c. Conflict between an authorization rule and an action taken by authorized individuals, e.g. SSO. Suppose that SSO issued the following:

$$can\_assume(r_4, r_3, t, d)$$

This allows users who are authorized to r_4 to activate r_3. If *u* satisfies ae_1, i.e. *u* is in N state *wrt* r_3, and at the same time is authorized to r_4. Nonetheless, the *can_assume* relation above authorizes *u* to r_3, which leads to a conflict.

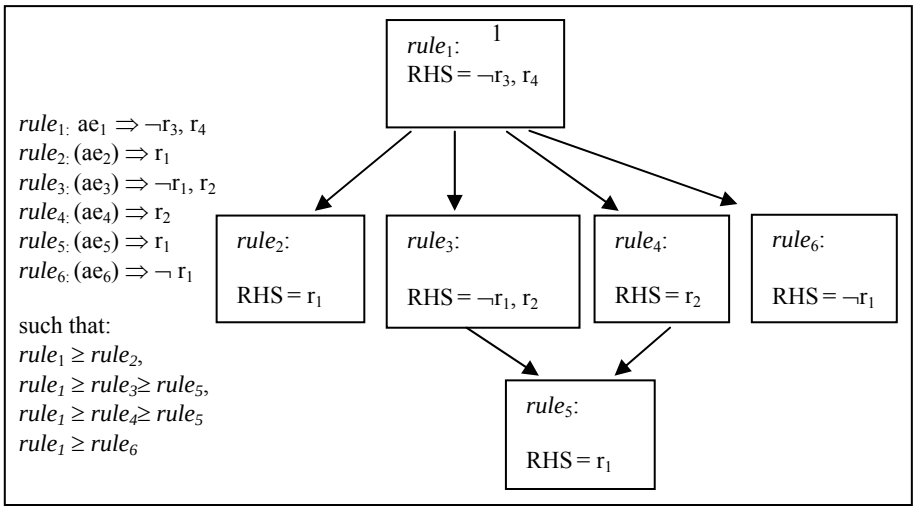

Figure 25: A Set of Conflicting Authorization Rules

4.2.1.5 Conflict Resolution Policies

Conflict resolution policies have been discussed extensively in the literature, see for example, [BSJ1997], [JSS1997] and [JSMS2001]. Most notable among them are:

a. Denial Takes Precedence (DTP): Negative authorizations are always adopted when conflict exists.

b. Permission Takes Precedence (PTP): Positive authorizations are always adopted when conflict exists.

These two policies in their original form suffer the following deficiencies:

a. They are very rigid in the sense that they do not allow the specifying of special cases that violate the policy enforced. Suppose a hospital has a policy that has the following authorization rules:

rule$_1$: No. of years in residency ≤ 1 ⇒ intern

rule$_2$: No. of years in residency ≤ 1 ⇒ ¬ ER_doctor

Naturally, during the holiday seasons large numbers of the medical staff take their yearly vacation. However, this period of the year witnesses a surge in the number of people admitted to the emergency room. Clearly, additional medical staff are needed to handle this surge in demand of medical care. The administration may allow interns to work in the ER, and hence authorizes them to role ER-doctor.

One way to handle this is to change the hospital policy by deleting $rule_2$. This course of action is not preferred because it might lead to unseen side effects. Also, it might lead to a breach in the security policy if the SSO forgets to add it back after the holiday season is over. A better solution is to use *can_assume* relation as follows:

can_assume(intern, ER_doctor, *t*, *d*)

This authorizes interns to activate the role ER_doctor, i.e. to work in the emergency room. *can_assume* conflicts with $rule_2$, so if DTP is enforced, the interns will not be able to work in the ER unless $rule_2$ is deleted. What is needed in this situation is a relaxed version of DTP that allows the stating of this exception in the security policy.

b. DTP with negative authorization is useful in a closed policy environment. However, PTP renders negative authorization meaningless in such environments. This is so because *wrt* any role that is associated with negative authorization, there could be only one of the following possibilities:

   i. Conflict may arise: Since PTP is enforced, the negative authorization is ignored.

ii. No conflict may arise: There is no need for the negative authorization since we are assuming a closed policy.

Based on that, we argue that there is a need for more flexible conflict resolution policies. The following section discusses newly formulated conflict resolution policies; some of them specify DTP policy with varying degrees of flexibility.

4.2.1.5.1 Localized DTP (LDTP)

DTP policy resolves any conflict in favor of denial, even if the conflict occurs among unrelated rules from among the relevant rules. This is rather restrictive since it means the more rules a user satisfies, the higher is the risk that he might be denied access to a role due to a conflict in authorization. This is counter-intuitive and requires us to modify the policy so that the conflict among unrelated rules is resolved in favor of permission. In other words, the denial is localized to conflict among comparable rules. We name the modified policy: the Localized DTP, or LDTP for short.

4.2.1.5.2 Flexible DTP (FDTP)

This policy enforces DTP in cases where conflict occurs among authorization rules. Otherwise, it enforces PTP which allows for exceptions caused by *can_assume* and *can_delegate* relations. Thus, when FDTP is enforced, in the example of the hospital discussed above, an intern can work as an ER doctor via *can_assume* relation without the need to remove $rule_2$ from the authorization rules set. In other words, FDTP policy authorizes *u* to role *r* if there is no conflict *wrt* role *r*, or if there is a *can_assume* relation which authorizes *u* to role *r* even if *u* receives a negative authorization *wrt* to *r*.

In Table 5 we summary how the afore-discussed policies compare and contrast.

Table 5: Comparison of Conflict Resolution Policies

Policy ↓	Conflicting Parties		
	Comparable Rules	Non-comparable Rules	Rules and SSO-initiated authorization (*can_assume* and *can_delegate*)
DTP	Denial	Denial	Denial
PTP	Permission	Permission	Permission
LDTP	Denial	Permission	Denial
FDTP	Denial	Denial	Permission

The entry at the intersection of the fourth row with the third column, for example, means that under LDTP if the conflicting parties are non-comparable rules, then permission prevails.

4.2.1.5.3 Weighted Rules

Authorization rules are assigned weights according to criteria determined by the enterprise such as:

i. The seniority of the rule, so $rule_3$ has higher weight than $rule_5$ in Figure 25 and, thus, the negative authorization is enforced.

ii. The seniority of the rule issuer, e.g. an SSO-issued rule has higher weight than a rule issued by a junior security officer.

4.2.1.5.4 Labeled Roles

Each role is assigned a label, which could be either one of the following values: DTP or PTP. If r_g and r_h are respectively labeled DTP and PTP, then in case of conflict *wrt* r_g, DTP is always enforced, while PTP is enforced in case of r_h. The notion and notation of

role ranges [SBM1999] could be utilized in this context. An *assign_label* relation can be defined as follows:

$$assign\_label \subseteq \{\text{DTP, PTP}\} \times 2^{IR}$$

So, *assign_label*(DTP, [r_g, r_h]) assigns DTP label to all roles in the range [r_g, r_h]. There are some subtle issues that need to be analyzed further. Suppose we have two roles r_g and r_h such that $r_g \geq r_h$. Suppose also that we assign the labels DTP and PTP to r_g and r_h respectively. If *u* satisfies authorization rules such that he has conflict in both roles, then based on the labels assigned to the roles, *u* is authorized to r_h but not to r_g. Although this reduces the privileges available to *u*, this situation is not problematic since senior roles are naturally assigned more permission and, as thus, it is wise to err on the side of denial in case of conflict. However, assume that the labels were reversed and that *u* has conflict in both roles. The resolution will be such that *u* is authorized to r_g but not to r_h, which is very problematic since r_h's permissions are a subset of r_g's permissions. To follow this policy strictly, we need to suspend this subset of permissions, which may render r_g deficient or even meaningless. We have not found any practical example in which this scenario is applicable. So, when assigning labels to roles, we require that the roles higher in the hierarchy receive labels of equal or higher level than their juniors. We assume that DTP label is higher than PTP. We believe this requirement is reasonable since senior roles are naturally assigned more permission, so they need more protection.

4.2.1.6 Users' Authorization in Model B_1

In the previous section we discuss several policies that can be deployed to resolve conflicts that may arise among authorization appointed to a specific user. In this section, we modify the definition of the set "URAuth" under selected policies to reflect the impact

of conflict, if it exists, on user's authorization. While it is possible to do this with respect to all the conflict resolution policies that we have newly introduced, for the sake of brevity, we choose to focus on DTP, PTP, LDTP and FDTP.

Definition 8

1. The model's syntax is shown in section 4.2.1.1.
2. URAuth varies according to the policy enforced:
   a. PTP: URAtuh in PTP with/without *can_assume* is similar to the corresponding URAtuh in Model A.
   b. DTP : $\text{URAuth}^{\text{DTP}} = A \wedge C$, or

$$\text{URAuth}^{\text{DTP}} = \{(u,r) | (\exists rule_i)[(u, ae_i) \in \text{U\_AE} \wedge r \in RHS(ae_i)$$

$$\wedge \neg (\exists rule_j)[(u, ae_j) \in \text{U\_AE} \wedge \neg r \in RHS(ae_j)]\}$$

   c. DTP with *can_assume*: $\text{URAuth}^{\text{DTP with } can\_assume} = (A \vee B) \wedge C$, or

$$\text{URAuth}^{\text{DTP with } can\_assume} = \{(u,r) | ((\exists rule_i)[(u, ae_i) \in \text{U\_AE} \wedge r \in RHS(ae_i)]$$

$$\vee (\exists rule_j) [(u, ae_j) \in \text{U\_AE} \wedge r' \in RHS(ae_j) \wedge$$

$$can\_assume(r', r, t, d) \wedge can\_assume \text{ has not expired }])$$

$$\wedge \neg (\exists rule_j)[(u, ae_j) \in \text{U\_AE} \wedge \neg r \in RHS(ae_j)]\}$$

   d. LDTP: We modify the term C to require the conflicting rules to be comparable. Call the modified term C', thus $\text{URAuth}^{\text{LDTP}} = A \wedge C'$

$$\text{URAuth}^{\text{LDTP}} = \{(u,r) | ((\exists rule_i)[(u, ae_i) \in \text{U\_AE} \wedge r \in RHS(ae_i)]$$

$$\wedge \neg (\exists rule_j)[(u, ae_j) \in \text{U\_AE} \wedge \neg r \in RHS(ae_j)$$

$$\wedge ((ae_j \rightarrow ae_i) \vee (ae_i \rightarrow ae_j)])$$

   e. LDTP with *can_assume*: $\text{URAuth}^{\text{LDTP with } can\_assume} = (A \vee B) \wedge C'$

$$\text{URAuth}^{\text{LDTP with } can\_assume} = \{(u,r) | ((\exists rule_i)[(u, ae_i) \in \text{U\_AE} \wedge r \in RHS(ae_i)]$$

$$\vee (\exists rule_j) [(u, ae_j) \in \text{U\_AE} \wedge r' \in RHS(ae_j) \wedge$$

$$can\_assume(r', r, t, d) \wedge can\_assume \text{ has not expired }])$$

$$\wedge \neg (\exists rule_j)[(u, ae_j) \in \text{U\_AE} \wedge \neg r \in RHS(ae_j) \wedge ((ae_j \rightarrow ae_i) \vee (ae_i$$

$$\rightarrow ae_j)] \}$$

f. FDTP: $\text{URAuth}^{\text{FDTP}} = \text{URAuth}^{\text{DTP}}$

g. FDTP with *can_assume*: $\text{URAuth}^{\text{FDTP with } can\_assume} = (A \wedge C) \vee B$

$$\text{URAuth}^{\text{FDTP with } can\_assume} = \{(u,r) | ((\exists rule_i)[(u, ae_i) \in \text{U\_AE} \wedge r \in RHS(ae_i)]$$

$$\wedge \neg (\exists rule_j)[(u, ae_j) \in \text{U\_AE} \wedge \neg r \in RHS(ae_j)])$$

$$\vee (\exists rule_j) [(u, ae_j) \in \text{U\_AE} \wedge r' \in RHS(ae_j) \wedge$$

$$can\_assume(r', r, t, d) \wedge can\_assume \text{ has not expired }]\}$$

Table 6: Authorizatioin Under Different Conflict Resolution Policies in Model B_1

Policy	URAuth Without *can_assume*	URAuth With *can_assume*
PTP	A	$(A \vee B)$
DTP	$A \wedge C$	$(A \vee B) \wedge C$
LDTP	$A \wedge C'$	$(A \vee B) \wedge C'$
FDTP	$A \wedge C$	$(A \wedge C) \vee B$

Table 6 summaries the definition of URAuth under different policies.

4.2.1.7 IRH in Model B_1

The modifications we introduced on the definition of URAuth do not affect the properties of IRH. Consequently, the IRH that is based on URAuth without *can_assume* will have the properties stated in Theorem 1. On the other hand, the IRH is a quasi order when it is based on URAuth with *can_assume* as stated in Theorem 2.

4.2.1.1 GRH in Model B_1

We have stated in Model A analysis that we may be provided with a given role hierarchy GRH that represents the current business practice of the enterprise. The GRH is identical to role hierarchies defined in RBAC96, that is, it is permission-driven:

$$(r_i \geq_{GRH} r_j) \rightarrow r_{j\ permissions} \subseteq r_{j\ permissions}$$

where $\geq_{GRH}$ has the same semantics as in RBAC96. As such, inheritance of permissions flows upward in the GRH. When a GRH is present, the rule $ae_i \Rightarrow \neg r_g$ may have one of the following two possible semantics:

a. Propagation prohibited: Users who satisfy ae_i should be prohibited from assuming r_g. This is the interpretation given previously.

b. Propagation allowed: Negative authorization propagates upward in GRH such that users who satisfy ae_i should be prohibited not only from assuming r_g, but also from assuming any role r_k such that $r_k \geq_{GRH} r_g$. This ensures that the user cannot circumvent the system by assuming r_k, whose permissions are a superset of r_g's. From a functional perspective, this may not be desirable since it is usually the case that the prohibition is targeting users who merely satisfy $rule_i$, but not those who can assume roles higher in the hierarchy by virtue of satisfying rules senior to $rule_i$, which usually means that they meet higher security requirement. Allowing the negative authorization to propagate upward requires modification of the definition of URAuth. For a user to be authorized to a role *r*, not only do we require that *u* has positive authorization *wrt r* and does not have negative authorization *wrt r*, but we also require that *u* does not have negative authorization wrt any role *r'* such that $r \geq_{GRH} r'$ i.e. *r* is senior to *r'* in GRH.

Definition 9

URAuth definition is modified to take propagation of negative authorization into account. We need to modify term *C* as follows:

Term *C* becomes: $C_{modified} = \neg\, (\exists rule_j)[(u, \text{ae}_j) \in \text{U\_AE} \wedge \neg r' \in RHS(\text{ae}_j) \wedge r \geq_{GRH} r']$

Notice that we can replace the term C' in Definition 8 with $C_{modified}$ since $r \geq_{GRH} r'$ implies that the rules that generate r and r' are comparable.

4.2.1.2 Related Issues

4.2.1.2.1 User State Diagram

Suppose that the system that implements RB-RBAC has the following set of rules only:

$rule_i$: $ae_i \Rightarrow r_g$

$rule_j$: $ae_j \Rightarrow r_h$

$rule_k$: $ae_k \Rightarrow \neg r_h$

Let's consider the following scenarios assuming DTP is in effect and using Figure 26:

Scenario 1: Assume that *u* satisfies $rule_j$ only. Since *u* does not satisfy $rule_k$ i.e. there is no negative authorization associated with r_h, $(u, r_h) \in$URAuth. In other words, *u* could be in any of the following states *wrt* r_h: P, D, or Act. A change in *u*'s attributes or in the authorization rules may cause the system that implements RB-RBAC to invoke $rule_k$ assigning negative authorization to *u wrt* r_h. Accordingly, $(u, r_h) \notin$URAuth and *u*'s state will be changed from P to N or from D or Act to R. The arrows labeled ae/r represent this.

Scenario 2: Assume that *u* satisfies $rule_i$ only. Since *u* does not satisfy $rule_k$ i.e. there is no negative authorization associated with r_g, $(u, r_g) \in$URAuth. As a result, *u* could be in any of the following states *wrt* r_g: P, D, or Act. A change in *u*'s attributes or in the authorization rules that cause the system that implements RB-RBAC to invoke $rule_k$ assigning negative authorization to *u wrt* r_h. If $r_g \geq r_h$ and propagation is allowed, *u*'s state will be changed as in scenario 1.

A change in *u*'s attributes or in the authorization rules may make *u* no more able to satisfy $rule_k$, and thus, *u* is no more authorized to $\neg r_h$. Also, *u* could become unable to

satisfy $rule_k$ either because it was modified or deleted. This results in changing his state from N back to P, or from R to D.

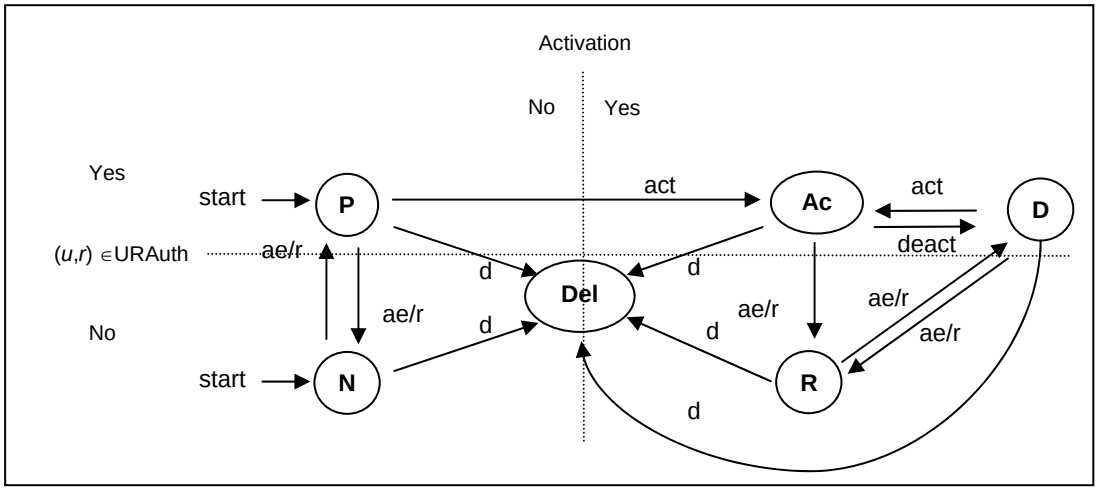

Figure 26: State diagram of a user with respect to role r

4.2.1.2.2 Enforcement Requirements

Enforcing the negative authorization requires that the system which implements RB-RBAC has access to all relevant attributes. This requirement affects the architectural options that can be used to enforce Model B since the system must either have these attributes under its control or be granted access to them when needed. If this is not the case, then users may evade the model. Consider the following unrelated rules:

$rule_i$: $ae_i \Rightarrow \neg\ r_h$

$rule_j$: $ae_j \Rightarrow r_h$

If these rules were in public domain or were somehow unconcealed, then users whose attributes satisfy both ae_i and ae_j can avoid $rule_i$ simply by not providing the attributes necessary to satisfy ae_i. Though this may not be a problem under PTP policy, it amounts

to a security breach under DTP policy. If RB-RBAC has access to users' attributes, DTP policy can be enforced.

4.2.2 Mutual Exclusion (Model B_2)

The specification, motivation, usage, and implications of mutual exclusion in RBAC have received a fair amount of attention in the literature [Kuhn1997], [FK1995] and [SCFY1996]. Mutual exclusive roles, which are used to express separation of duty policies, are a powerful feature of RBAC96 that is supported by commercial software, such as Informix database management systems, to enforce static and dynamic separation of duty policies. The classical example for two mutually exclusive roles comes from organizations that prohibit a single individual from requesting and approving of a major expenditure. Expressing separation of duty policies is crucial since these policies are often closely tied to RBAC, which is a natural tool for implementing separation of duty. Several variations of mutual exclusion have been discussed in the RBAC literature, the most important of which are:

a. Static: Any two roles that have been specified as mutually exclusive cannot both be included in the set of roles *assigned* to a user.
b. Dynamic: Any two roles that have been specified as mutually exclusive cannot both be included in the set of roles *activated* by a user.
c. Complete: Mutually exclusive roles do not share any permission.
d. Partial: Mutually exclusive roles may share some, but not all, permissions.

4.2.2.1 The ASL_{B2} Language

4.2.2.1.1 ASL_{B2} Syntax

To allow the specification of mutual exclusion, the syntax of ASL_{B2} is as follows:

Roles ::= [Dynamic | Session dynamic] ME

ME ::= {||role-set||} || ⊕ || {||role-set||}

| {||role-set||} || ⊕ || ME

role-set ::= Role

| Role||,||role-set

So, mutually exclusive roles can be specified in a rule as follows:

$ae_k \Rightarrow$ [Dynamic | Session dynamic] $\{role\_set_a\} \oplus \{role\_set_b\}$

Each of $role\_set_a$ and $role\_set_b$ can be composed of one role or more. This allows expressing several variations of mutual exclusion among roles:

a. role-vs-role:

$ae_k \Rightarrow \{r_i\} \oplus \{r_j\}$

Roles r_i and r_j are identified to be mutually exclusive.

b. role-vs-role_set: In the following example the mutual exclusion is specified between r_i and any role $r_j \in role\_set_x$:

$ae_k \Rightarrow \{r_i\} \oplus \{role\_set_x\}$

We assume that $r_i \notin role\_set_x$.

c. role_set-vs-role_set: In the following example the mutual exclusion is specified between any two roles r_i and r_j such that $r_i \in role\_set_x$ and $r_j \in role\_set_y$:

$ae_k \Rightarrow \{role\_set_x\} \oplus \{role\_set_y\}$

Here, we assume that $role\_set_x \cap role\_set_y = \varnothing$.

Also, ASL_{B2} allows specifying mutual exclusion among multiple roles or role sets. For example, $ae_k \Rightarrow \{r_i\}\oplus,..,\oplus\{r_m\}$ means that if *u* assumes r_i where $1 \leq i \leq$, n, then *u* cannot assume any role r_k where $1 \leq k \leq$, n and $k \neq i$. Thus far, we used the language to specify static mutual exclusion. The key word "Dynamic" could be inserted to specify simple dynamic mutual exclusive roles as follows:

$$ae_k \Rightarrow \text{Dynamic } \{r_i\}\oplus \{r_j\}$$

Similarly, session-based dynamic mutual exclusion could be specified by inserting the key word "Session dynamic" before the RHS of the rule. This feature of ASL_{B2} enables us to specify static mutual exclusion among certain roles and, at the same time, specify dynamic mutual exclusion among other roles.

4.2.2.1.2 Semantics

The following is the meaning of the 3 types of mutual exclusion allowed by RB-RBAC:

a. Static: If r_i and r_j are specified as mutually exclusive roles, then if user *u* activates any one of them, say role r_i, then *u* is permanently banned from activating r_j. This is useful in specifying role-centric SOD constraints, which means that no user should assume r_i and r_j.

b. Simple Dynamic: If r_i and r_j are specified as mutually exclusive roles, then *u* cannot activate both roles at the same time. However, if *u* becomes dormant in one of them, he can activate the other. This provides a role-centric dynamic SOD. The reserved word "Dynamic" is used to identify this type.

c. Session Dynamic: If r_i and r_j are specified as mutually exclusive roles of this type, then *u* cannot activate them simultaneously in one session. We use the reserved word "Session dynamic" to identify this type.

Note that the roles which are declared mutually exclusive are treated as such *wrt* the users who satisfy the rule(s) in which the roles are specified mutually exclusion.

4.2.2.2 Conflict Due to Mutual Exclusion

Mutual exclusion may introduce conflict among authorization rules. There are two types of conflict:

4.2.2.2.1 Type A conflict

This type occurs due to conflict among authorization rules or between some authorization rules and authorization granted by the SSO via *can_assume* or *can_delegate*. We have identified the following cases of this conflict where r_g and r_h are two roles that are mutually exclusive:

a. Conflict among rules: We recognize conflict only if it takes place among authorization rules. We specify this case as follows:

$$\{(u,r_g), (u,r_h)\} \subseteq \text{URAuth}^{\text{Model A}} \wedge \{\{r_g\} \oplus \{r_h\}\} \subseteq \textit{RHS}(ae_j) \wedge (u, ae_j) \in \text{U\_AE}$$

$\text{URAuth}^{\text{Model A}}$ contains user's authorization according to Model A, i.e. without considering the conflict caused by the mutual exclusion. We use $\{\{r_g \oplus r_h\}\} \subseteq \textit{RHS}(ae_j)$ to mean that $rule_j$ generates r_g and r_h which are mutually exclusive. Note that Model B_2 allows us to identify two roles to be mutually exclusive only *wrt* to users who satisfy the rule that specify the mutual exclusion. In other words, it is possible

for a user *u* to satisfy rules that authorize him to the roles that are identified to be mutually exclusive via other rules. Consider the following scenario:

$$r_g \in RHS(ae_i)$$

$$r_h \in RHS(ae_j)$$

$$\{\{r_g\} \oplus \{r_h\}\} \subseteq RHS(ae_k)$$

Suppose that the rules are not related and that *u* satisfies $rule_i$ and $rule_j$. Form Model B_2 perspective, *u* has not met the criteria needed to invoke the mutual exclusion, and thus, the system does not prevent *u* from activating both roles. There are two forms of this conflict:

1. Conflict among related rules: The case is formalized as follows:

$$\{r_g, r_h\} \subseteq RHS(ae_i) \wedge (u, ae_i) \in U\_AE$$

$$\wedge \{\{role\_set_x\} \oplus \{role\_set_y\}\} \subseteq RHS(ae_j)$$

$$\wedge r_g \in role\_set_x \wedge r_h \in role\_set_y \wedge (u, ae_j) \in U\_AE$$

$$\wedge ((ae_i \rightarrow ae_j) \vee (ae_j \rightarrow ae_i))$$

   Suppose that $rule_i$ is senior to $rule_j$ and that *u* satisfies $rule_i$. As a result, *u* is authorized to either r_g, r_h, or both. However, *u* also satisfies $rule_j$, hence *u* is authorized to either r_g or r_h but not both.

2. Conflict among unrelated rules: This case is a generalization of the previous one by removing the term: $((ae_i \rightarrow ae_j) \vee (ae_j \rightarrow ae_i))$.

b. Conflict between an authorization rule and authorization granted by the SSO. This case is represented by the following:

$$\{\{role\_set_x\} \oplus \{role\_set_y\}\} \subseteq RHS(ae_j) \wedge r_g \in role\_set_x \wedge r_h \in role\_set_y$$

$$\wedge (u, ae_j) \in U\_AE$$

$$\wedge \textit{can\_assume}(r_g, r_h, t, d) \wedge \textit{can\_assume} \text{ has not expired})$$

4.2.2.2.2 Type B conflict

This type of conflict is identified for the first time in RBAC literature. The conflict is due to contrast in mutual exclusion types. To illustrate, suppose we have the following two rules:

$$rule_i : ae_i \Rightarrow \{\{r_g\} \oplus \{r_h\}\}$$

$$rule_j : ae_j \Rightarrow \text{Dynamic } \{\{r_g\} \oplus \{r_h\}\}$$

The first rule specifies a static mutual exclusion between r_g and r_h while the second specifies a dynamic mutual exclusion between the same roles. So if user *u* activates role r_g, then, according to $rule_i$, the system should permanently prohibit *u* from becoming authorized to r_h. But according to $rule_j$ this ban should be lifted once *u* deactivates r_g.

4.2.2.3 Conflict Resolution Policies

4.2.2.3.1 Type A conflict:

In the following discussion, we present policies to resolve conflict due to mutual exclusion. The novelty of these policies stems from the fact that either they have never been discussed in the RBAC literature, or that they were modified so that they have added semantics. Let's define URInvoked set such:

URInvoked =

{(u,r) | u has activated r at sometime in the past or is currently activating r}

Suppose also:

- u is authorized to r_g and r_h.
- r_g and r_h be two mutually exclusive roles,
- (u, r_g) $\in$URInvoked
- u wants to activate r_h.

Now, on the one hand, u is authorized to r_h. However, on the other hand, he should be prevented from activating r_h since r_g and r_h are mutually exclusive roles. Clearly, there is a conflict. To resolve it, we suggest the following policies:

4.2.2.3.1.1 DTP with modes

This policy has three modes:

i. Static: User u is permanently denied authorization to r_h. This is the extreme case of DTP. Specifying this policy requires a change in our definition of URAuth since it is dependent on the first choice made by the user and the impact of that choice is permanent. In the example above, the fact that u activates r_g makes activating r_h out of reach from u's standpoint.

ii. Simple Dynamic: u is authorized to both roles, however he is denied authorization to r_h as long as he is active *wrt* r_g. The definition of URAuth for case is similar to Model A.

iii. Session Dynamic: u can activate r_h but not in the same session with r_g. In this case also, the definition of URAuth is similar to Model A.

4.2.2.3.1.2 FDTP with modes

This policy enforces DTP in cases where conflict occurs among authorization rules. Also, similar to DTP, it has three modes. However, when a rule conflicts with an authorization issued by the SSO such as *can_assume*, permission always takes precedence. In this case, the three modes become irrelevant since the mutual exclusion is not enforced.

4.2.2.3.1.3 Weighted Rules

Authorization rules are assigned weights according to criteria determined by the enterprise similar to Model B_1.

4.2.2.3.1.4 Labeled roles with modes

This policy enables modes of enforcement of DTP policy. The *assign_label* relation is modified as follows:

$$\text{Modes (M)} = \{DTP^S, DTP^D, DTP^{SD}\}$$

$$assign\_label \subseteq M \times 2^{IR}$$

DTP^S, DTP^D and DTP^{SD} mean DTP in static, simple dynamic and session dynamic modes respectively.

4.2.2.3.2 Type B conflict

Although the policies we presented above could be applied, we will limit the discussion to the basic versions of DTP and PTP. Before stating the policies we need the following theorem.

Theorem 3

If static mutual exclusion holds → Simple dynamic mutual exclusion holds → Session based mutual exclusion holds.

Proof:

Let r_g and r_h be two roles. Then, based on the definition of static, simple dynamic, and session-based mutual exclusion, we can state the following:

If r_g and r_h are identified to be mutually exclusive in static mode, then static mutual exclusion holds between r_g and r_h *wrt* user *u* if

$$\{(u,r_g) , (u,r_h)\} \subseteq \text{URAuth}^{\text{Model A}} \wedge \{(u,r_g) , (u,r_h)\} \not\subset \text{URA} \cup \text{URD} \quad \text{.....(i)}$$

If r_g and r_h are identified to be mutually exclusive in simple dynamic mode, then simple dynamic mutual exclusion holds between r_g and r_h *wrt* user *u* if

$$\{(u,r_g) , (u,r_h)\} \subseteq \text{URAuth}^{\text{Model A}} \wedge \{(u,r_g) , (u,r_h)\} \not\subset \text{URA} \quad \text{.....(ii)}$$

If r_g and r_h are identified to be mutually exclusive in session-based mode, then session-based mutual exclusion holds between r_g and r_h *wrt* user *u* if

$$\{(u,r_g) , (u,r_h)\} \subseteq \text{URAuth}^{\text{Model A}} \wedge \{r_g,r_h\} \not\subset \{(u,r_i) \mid r_i \in \textit{roles}(\text{OE}(\text{Sessions}))$$

$$\wedge u = \textit{user}(\text{OE}(\text{Sessions}))\} \quad \text{.....(iii)}$$

such that OE(Sessions) returns the roles activated in one session. OE is a non-deterministic function, pronounced *oneelement* and first introduced in [CS1996], non-deterministically returns one element of a given set. (Further discussion of this function is presented in the following definition)

Based on (i) and (ii) above, the following holds:

If static mutual exclusion holds → Simple dynamic mutual exclusion holds

Based on (ii) and (iii) above, the following holds:

Simple dynamic mutual exclusion holds → Session based mutual exclusion holds

This proves the theorem.

□

4.2.2.3.2.1 DTP

When a conflict of type B arises among rules, the more restrictive form of mutual exclusion is enforced. Between two conflicting rules of type B, the more restrictive is the one that is an antecedent to the other according to the previous theorem.

4.2.2.3.2.2 PTP

The less restrictive form of mutual exclusion is enforced, which is a consequent to the other conflicting type.

4.2.2.4 Users' Authorization in Model B_2

In the previous section we discussed several policies to resolve conflict in case it arises. In this section, we modify the definition of the set "URAuth" under selected policies to specify what roles users are authorized to. We present the formal definition of URAuth for DTP and FDTP with the three types of mutual exclusion. Specification can be provided for URAuth under the remainder of the policies.

Definition 10

1. Syntax: See section 4.2.2.1.1 for details.

   $rule_i$: $ae_i \Rightarrow \{role\_set_1\} \oplus \ldots \oplus \{role\_set_n\}$ such that any role-set, $role\_set_i = \{r_x,..,r_y\} \wedge (role\_set_i \cap role\_set_j = \varnothing$ for any i and j both in [1,n] such that $i \neq j)$

2. $ME\_set(rule_i) = \{ role\_set_j \mid role\_set_j$ is a mutually exclusive role-set in the right hand side of $rule_i\}$

3. URInvoked = $\{(u,r) \mid u$ has activated r at sometime in the past or is currently activating $r\}$

   URInvoked = URA $\cup$ URD $\cup$ URR

4. Non-deterministic functions, Oneelement and Allother, first introduced in [CS1996]:

   - Oneelement (OE): set$\rightarrow$ element.
   - Allother (AO): set$\rightarrow$set, i.e. get set by taking out one element.

   These two functions are related by context since for any set X,

   $$AO(X) = X - OE(X)$$

   and at the same time, neither is a deterministic function. Also, multiple occurrences of OE in a sentence all return the same element x_i form X.

5. For Dynamic or Session Dynamic mutual exclusion: URAuth is identical to its counterpart in Model A.

6. For Static mutual exclusion:

   a. $URAuth^{Static\ DTP} = \{(u,r_g) | (\exists rule_i)[(u, ae_i) \in U\_AE \wedge (r_g \in OE(ME\_set(rule_i)) \wedge r_h \in OE(AO(ME\_set(rule_i)))) \rightarrow \neg \exists (u,r_h)[(u, r_h) \in URInvoked]]\}$ which changes according to the user's initial choice. For the sake of naming convenience, let's call this $D \wedge E$.

   b. $URAuth^{Static\ DTP\ with\ can\_assume} = (D \wedge E) \vee (F \wedge G)$

      $\{(u,r_g) | ((\exists rule_i)[(u, ae_i) \in U\_AE \wedge (r_g \in OE(ME\_set(rule_i)) \wedge r_h \in OE(AO(ME\_set(rule_i)))) \rightarrow \neg \exists (u,r_h)[(u, r_h) \in URInvoked]])$

      $\vee$ (can_assume (r_k, r_g, t, d) $\wedge$ can_assume has not expired $\wedge$ $\neg(\exists rule_j)$ $[r_g \in OE(ME\_set(rule_j)) \wedge r_h \in OE(AO(ME\_set(rule_j))) \wedge$

$(u, ae_j) \in U\_AE \wedge (u, r_h) \in URInvoked])\}$, which changes according to the user's initial choice.

c. $URAuth^{Static\ FDTP} = URAuth^{Static\ DTP} = D \wedge E$

d. $URAuth^{Static\ FDTP\ with\ can\_assume} = (D \wedge E) \vee B$

$\{(u,r_g) | ((\exists rule_i)[(u, ae_i) \in U\_AE \wedge (r_g \in OE(ME\_set(rule_i)) \wedge r_h \in OE(AO(ME\_set(rule_i)))) \rightarrow \neg \exists (u,r_h)[(u, r_h) \in URInvoked]]) \vee (can\_assume\ (r_k, r_g, t, d) \wedge can\_assume$ has not expired)$\}$, which changes according to the user's initial choice.

4.2.2.5 Mutual Exclusion and GRH

Suppose we have the rule: $ae_i \Rightarrow r_g \oplus r_h$. Suppose also that $r_g \geq_{GRH} r_h$, i.e. r_g is senior to r_h in GRH. This scenario is problematic because a user *u* who satisfies the rule can activate r_g. And since $r_g \geq_{GRH} r_h$, *u* can execute the privileges of r_h, rendering the mutual exclusion ineffective. Some researchers argue that mutually exclusive roles should not have a common parent role [Kuhn1997]. Formally, we state this as follows:

$$(r_g \in OE(ME\_set(rule_i)) \wedge r_h \in OE(AO(ME\_set(rule_i)))) \rightarrow$$

$$\neg (\exists\ r_k)\ [r_k \geq_{GRH} r_g \wedge r_k \geq_{GRH} r_h]$$

It could be argued that this is not always desirable. Consider the case of a software development team where we have two roles that are mutually exclusive: *developer* and *tester*. Traditional business practice calls for a *team leader* role which is usually superior to both roles.

A user's choice of a mutually exclusive role may affect the choices available to him by other rules. Suppose, for example, that r_g and r_h are mutually exclusive, and r_h and r_k are mutually exclusive i.e.:

$$\{\{role\_set_x\} \oplus \{role\_set_y\}\} \subseteq RHS(ae_i) \wedge r_g \in role\_set_x \wedge r_h \in role\_set_y \wedge (u, ae_i) \in U\_AE$$

$$\wedge \{\{role\_set_y\} \oplus \{role\_set_z\}\} \subseteq RHS(ae_j) \wedge r_h \in role\_set_y \wedge r_k \in role\_set_z \wedge (u, ae_j) \in U\_AE$$

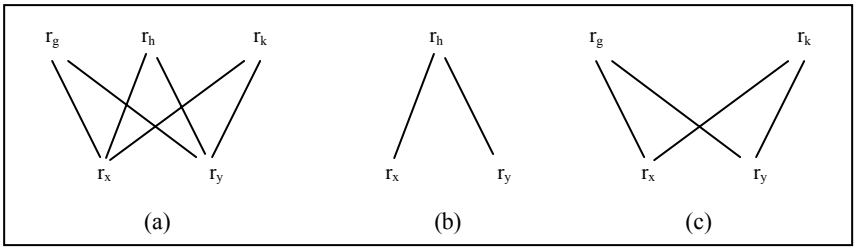

Figure 27: GRH Variations Due to Mutual Exclusion

Suppose that we have the GRH shown in part (a) of Figure 27. If *u* assumes r_h via $rule_i$, then he rules out the possibility of assuming r_k via $rule_j$ which reduces $rule_j$ to: $ae_j \Rightarrow r_h$. As a result, effectively *u* is authorized to the partial GRH shown in part (b). On the other hand, if *u* assumes r_g through $rule_i$, then r_h will be out of reach for him reducing $rule_j$ into: $ae_j \Rightarrow r_k$. This leads to the GRH in part (c). The availability of the GRH to the user is dependent on what type of mutual exclusion is enforced:

- If static mutual exclusion is specified then *u* will be *permanently* associated with only one of these hierarchies based on the rule invoked and user's choice of roles.
- In case of simple dynamic mutual exclusion, *u* will alternate between the two hierarchies at the time of role activation.

- If session-based dynamic mutual exclusion is specified, then *u* can have the above two GRH in effect simultaneously but not in the same session.

4.2.2.6 Related Issues

4.2.2.6.1 User State Diagram

For the sake of discussion, we will focus our attention on the user state diagram when the conflict resolution policy in effect is DTP in static and dynamic modes. Other conflict resolution policies mentioned above can be discussed in a similar way. Suppose we have two rules:

$ae_j \Rightarrow \{r_g, r_h\}$

$ae_i \Rightarrow \{r_g\} \oplus \{r_h\}$

Assume that *u* initially satisfied $ae_j \Rightarrow \{r_g, r_h\}$ only. Accordingly, *u* could be in "P", "Act", or "D" state *wrt* to r_g and r_h. Let us assume that *u* activated r_g. A later change in *u*'s attributes or authorization rules may cause $rule_i$ to become relevant to *u*. As a result, *u* becomes a potential user of r_h. The system behavior varies according to the mode enforced.

a. **Static**: The state diagram of this case is shown in Figure 28. The system implementing RB-RBAC may take one of the following courses of action:
   i. Immediately deactivates *u wrt* all active mutually exclusive roles, in this case, r_g. Later, if *u* wants to activate any mutually exclusive role, say r_h, his state in r_g changes from P to N, or from D to R (label 1).

ii. Waits until *u* willfully deactivates the current role but does not allow *u* to activate any role that is mutually exclusive with r_g. The remainder is like case (i).

Note that once a user is assigned N or R state due to mutual exclusion, that user can no longer go back to P, Act or D states even if the attributes or the rules change. The ae/r label in the arrows leaving N or R states indicates possible changes before a mutual exculsion occurs. This shows that the prohibition imposed on that user is permanent.

b. **Dynamic or Session Dynamic**: The state diagram is similar to Model A.

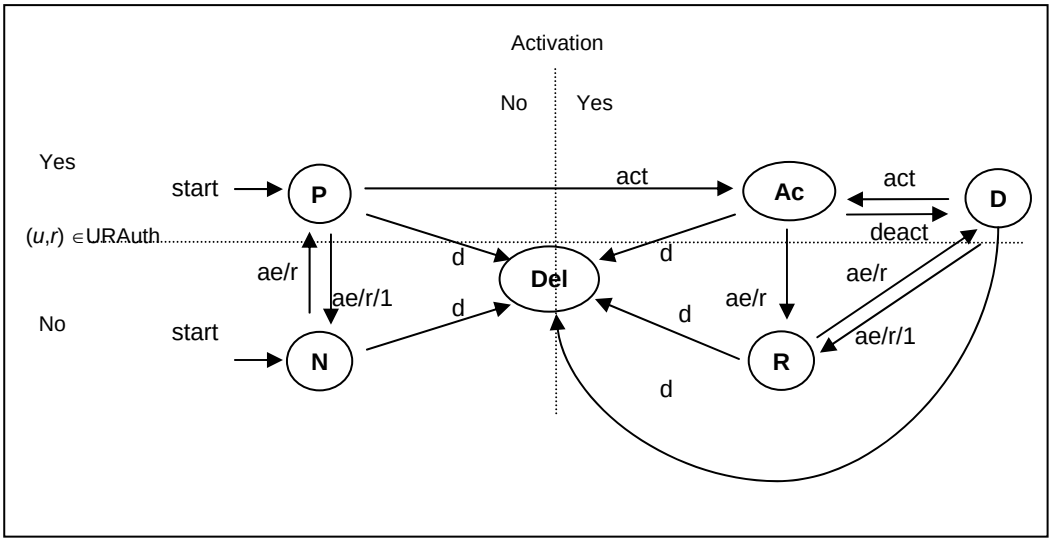

Figure 28: User State Diagram in Static Mutual Exclusion

4.2.2.6.2 Enforcement Requirements

Similar to the case of negative authorization, enforcing the mutual exclusion necessitates that all relevant attributes should be readily available to the system, which affects the enforcement architectures that can be used with Model B. Assume that the following unrelated rules are publicly known:

$rule_i : ae_i \Rightarrow \{r_g, r_h\}$

$rule_j : ae_j \Rightarrow \{r_g\} \oplus \{r_h\}$

Users, whose attributes satisfy both $rule_i$ and $rule_j$, can avoid the separation of duty simply by not providing the attributes necessary to satisfy $rule_j$.

4.2.3 Comparison of Models B_1 and B_2

The semantics of models B_1 and B_2 bear some resemblance, but at the same time include some differences. In B_1, a user *u* who receives negative authorization *wrt* a specific role is not authorized to that role until:

- The authorization rules are modified, and/or
- The user's attributes are changed.

In B_2, when *u* activates a role from among a set of mutually exclusive roles, *u* is prohibited from activating the rest of the roles. Effectively, this can be seen as if *u* has received negative authorization *wrt* the rest of the roles. However, if the mutual exclusion is static, *u* is permanently prohibited from activating these roles. This amounts to a permanent negative authorization which can not be specified using Model B_1.

4.2.4 Discussion

4.2.4.1 Monotonicity

In contrast to Model A which is monotonic, the syntax of ASL_{B1} language permits specifying the rules such that the set of roles that a user is authorized to decreases as the number of rules he satisfies increases. Suppose that we have $(\neg\, r_g) \in RHS(ae_i)$ and $\{r_g, r_h\} \subseteq RHS(ae_j)$. If a user u satisfies $rule_j$, then he is authorized to r_g and r_h. In case of DTP, if u satisfies both rules, he is authorized to r_h only. The above shows that Model B_1 is non-monotonic. We can show that Model B_2 is also non-monotonic if we modify the case above such $\{r_g \oplus r_h\} \subseteq RHS(ae_i)$.

4.2.4.2 Other RBAC Models

In OASIS model, negative authorization of roles was suggested as a means of specifying SOD constraints. No details were given but, rather, it was referred to as a future work [BMY2002]. In addition, OASIS has the following limitations:

a. It does not distinguish between simple dynamic and session-based dynamic SOD.
b. It is unable to handle static SOD, and as a result, important access control policies like the Chinese Wall cannot be enforced.

Model B overcomes these limitations. However, it is my position that negative authorization in RBAC context is less useful than it is in a DAC environment. The limited benefit it brings may not justify the substantial amount of complexity it introduces to the model.

4.3 Summary

Model B extends the language of the base model so that the language allows specifying negative authorization and mutual exclusion. Negative authorization in the context of RBAC is a novel concept. RB-RBAC Model B_1 is the first RBAC model that provides detailed analysis of different aspects of negative authorization in an RBAC context. This includes providing semantics for the negative authorization in this new territory, identifying cases of conflict, suggesting several conflict resolution policies (many of them are novel) and analyzing the impact of negative authorization on IRH, GRH and any RB-RBAC enforcement architecture.

Model B_2 provides the syntax needed to express mutual exclusion among roles in authorization rules. Mutual exclusion is a tool to specify the time-honored Separation of Duty (SOD) constraints. Allowing mutually exclusive roles in the RHS of the authorization rules introduces conflict among rules. Different types of conflict were identified and analyzed and suitable conflict resolution polices were presented and discussed. The impact of mutual exclusion on IRH, GRH and RB-RBAC enforcement architecture is discussed. The formalization of Models B_1 and B_2 is provided in Figures 29 and 30.

In the following chapter, we introduce Model C which allows specifying a set of constraints that includes SODs.

Model B_1

1. The model's syntax is shown in section 4.2.1.1.
2. URAuth varies according to the policy enforced:
   a. PTP: URAtuh in PTP with/without *can_assume* is similar to the corresponding URAtuh in Model A.
   b. DTP : $URAuth^{DTP} = A \wedge C$, or

$$URAuth^{DTP} = \{(u,r) | (\exists rule_i)[(u, ae_i) \in U\_AE \wedge r \in RHS(ae_i) \wedge \neg (\exists rule_j)[(u, ae_j) \in U\_AE \wedge \neg r \in RHS(ae_j)]\}$$

   c. DTP with *can_assume*: $URAuth^{DTP \text{ with } can\_assume} = (A \vee B) \wedge C$, or

$$URAuth^{DTP \text{ with } can\_assume} = \{(u,r) | ((\exists rule_i)[(u, ae_i) \in U\_AE \wedge r \in RHS(ae_i)] \vee (\exists rule_j)[(u, ae_j) \in U\_AE \wedge r' \in RHS(ae_j) \wedge can\_assume(r', r, t, d) \wedge can\_assume \text{ has not expired }]) \wedge \neg (\exists rule_j)[(u, ae_j) \in U\_AE \wedge \neg r \in RHS(ae_j)]\}$$

   d. LDTP: We modify the term C to require the conflicting rules to be comparable. Call the modified term C', thus $URAuth^{LDTP} = A \wedge C'$

$$URAuth^{LDTP} = \{(u,r) | ((\exists rule_i)[(u, ae_i) \in U\_AE \wedge r \in RHS(ae_i)] \wedge \neg (\exists rule_j)[(u, ae_j) \in U\_AE \wedge \neg r \in RHS(ae_j) \wedge ((ae_j \rightarrow ae_i) \vee (ae_i \rightarrow ae_j)]\}$$

   e. LDTP with *can_assume*: $URAuth^{LDTP \text{ with } can\_assume} = (A \vee B) \wedge C'$

$$URAuth^{LDTP \text{ with } can\_assume} = \{(u,r) | ((\exists rule_i)[(u, ae_i) \in U\_AE \wedge r \in RHS(ae_i)] \vee (\exists rule_j)[(u, ae_j) \in U\_AE \wedge r' \in RHS(ae_j) \wedge can\_assume(r', r, t, d) \wedge can\_assume \text{ has not expired }]) \wedge \neg (\exists rule_j)[(u, ae_j) \in U\_AE \wedge \neg r \in RHS(ae_j) \wedge ((ae_j \rightarrow ae_i) \vee (ae_i \rightarrow ae_j)]\}$$

   f. FDTP: $URAuth^{FDTP} = URAuth^{DTP}$
   g. FDTP with *can_assume*: $URAuth^{FDTP \text{ with } can\_assume} = (A \wedge C) \vee B$

$$URAuth^{FDTP \text{ with } can\_assume} = \{(u,r) | ((\exists rule_i)[(u, ae_i) \in U\_AE \wedge r \in RHS(ae_i)] \wedge \neg (\exists rule_j)[(u, ae_j) \in U\_AE \wedge \neg r \in RHS(ae_j)]) \vee (\exists rule_j)[(u, ae_j) \in U\_AE \wedge r' \in RHS(ae_j) \wedge can\_assume(r', r, t, d) \wedge can\_assume \text{ has not expired }]\}$$

3. URAuth definition is modified to take propagation of negative authorization into account. We need to modify term C as follows:

   Term C becomes: $C_{modified} = \neg (\exists rule_j)[(u, ae_j) \in U\_AE \wedge \neg r' \in RHS(ae_j) \wedge r \geq_{GRH} r']$

Figure 29: The Formalization of Model B_1

Model B_2:

1. Syntax: See section 4.2.2.1.1 for details.
   $rule_i$: $ae_i \Rightarrow \{role\_set_1\} \oplus \ldots \oplus \{role\_set_n\}$ such that any role-set, $role\_set_i = \{r_x,..,r_y\} \wedge (role\_set_i \cap role\_set_j = \varnothing$ for any i and j both in [1,n] such that $i \neq j$)
2. $ME\_set(rule_i) = \{ role\_set_j \mid role\_set_j$ is a mutually exclusive role-set in the right hand side of $rule_i\}$
3. $URInvoked = \{(u,r) \mid u$ has activated r at sometime in the past or is currently activating $r\}$
   $URInvoked = URA \cup URD \cup URR$
4. Non-deterministic functions, Oneelement and Allother, first introduced in [CS1996]:
   - Oneelement (OE): set→ element.
   - Allother (AO): set→set, i.e. get set by taking out one element.

   These two functions are related by context since for any set X,
   $$AO(X) = X - OE(X)$$
   and at the same time, neither is a deterministic function. Also, multiple occurrences of OE in a sentence all return the same element x_i form X.
5. For Dynamic or Session Dynamic mutual exclusion: URAuth is identical to its counterpart in Model A.
6. For Static mutual exclusion:
   a. $URAuth^{Static\ DTP} = \{(u,r_g) \mid (\exists rule_i)[(u, ae_i) \in U\_AE \wedge (r_g \in OE(ME\_set(rule_i)) \wedge r_h \in OE(AO(ME\_set(rule_i)))) \rightarrow \neg \exists (u,r_h)[(u, r_h) \in URInvoked]]\}$ which changes according to the user's initial choice. For the sake of naming convenience, let's call this $D \wedge E$.
   b. $URAuth^{Static\ DTP\ with\ can\_assume} = (D \wedge E) \vee (F \wedge G)$
      $\{(u,r_g) \mid ((\exists rule_i)[(u, ae_i) \in U\_AE \wedge (r_g \in OE(ME\_set(rule_i)) \wedge r_h \in OE(AO(ME\_set(rule_i)))) \rightarrow \neg \exists (u,r_h)[(u, r_h) \in URInvoked]])$
      $\vee$ (can_assume $(r_k, r_g, t, d) \wedge can\_assume$ has not expired $\wedge \neg(\exists rule_j)$ $[r_g \in OE(ME\_set(rule_j)) \wedge r_h \in OE(AO(ME\_set(rule_j))) \wedge (u, ae_j) \in U\_AE \wedge (u,r_h) \in URInvoked])\}$, which changes according to the user's initial choice.
   c. $URAuth^{Static\ FDTP} = URAuth^{Static\ DTP} = D \wedge E$
   d. $URAuth^{Static\ FDTP\ with\ can\_assume} = (D \wedge E) \vee B$
      $\{(u,r_g) \mid ((\exists rule_i)[(u, ae_i) \in U\_AE \wedge (r_g \in OE(ME\_set(rule_i)) \wedge r_h \in OE(AO(ME\_set(rule_i)))) \rightarrow \neg \exists (u,r_h)[(u, r_h) \in URInvoked]])$
      $\vee$ (can_assume $(r_k, r_g, t, d) \wedge can\_assume$ has not expired)}, which changes according to the user's initial choice.

Figure 30: The Formalization of Model B_2

Chapter 5: Model C

5.1 Introduction

No discussion of an access control model is complete unless constraints are discussed and ways to specify them are meticulously described. Constraints specification is an extensively discussed issue both in general [CW1986], and within the context of RBAC in specific, see for example [CS1995], [SCFY1996], [FBK1999] and [BMY2002]. In this chapter, we introduce Model C, which extends Model A by allowing the specification of constraints in the authorization rules using different methods. My goal is not to provide an exhaustive discussion of all possible methods of constraints specification, but rather, to discuss selected alternatives, compare them and highlight their pros and cons. These selected methods for specifying constraints are as follows:

a. Rule-Specific Constraints: A constraint is specified as a stand-alone component within the rule to which it applies which limits its scope to users who satisfy that rule.
b. System Attributes: This method extends Model A with attributes that hold values related to system information such as conflicting users, conflicting roles, etc. These are called system attributes and are needed to specify the constraints. Similar to the previous method, the constraints specified are local for each rule.

c. Invariants: As its name indicates, this method specifies constraints that must hold at all times and are applicable to all rules simultaneously.

These three methods were chosen because they are very intuitive. In addition, they have been discussed in the literature, though not necessarily in RBAC context. In fact, RB-RBAC is the first model that allows specifying constraints using these three methods together. RBAC96 recognizes invariants only, while OASIS, at the other extreme, allows rule-specific constraints only. We will focus the analysis of these methods on three classes of constraints which we believe are of importance, especially in context of the RBAC model.

1. Separation of Duty (SOD) constraints: These are a major class of role-based authorization constraints aimed at preventing fraud and errors. A typical example is that of mutually disjoint organizational roles, such as those of purchasing manager and accounts payable manager. Generally, the same individual is not permitted to belong to both roles because this creates a possibility for committing fraud [SCFY1996] , [FBK1999].
2. Cardinality constraints: This type specifies the maximum number of members in a role [SCFY1996]. It is useful when some roles can only be assigned to a certain number of users like a manager of a branch in a bank, a chairman of a department, etc. Also, this type is useful in enforcing licensing agreements, for example [FBK1999].
3. Prerequisite role constraints: The concept of prerequisite role is based on competency and appropriateness, whereby a user can be assigned to role r_i only if the user is already a member of role r_j [SCFY1996].

Because RB-RBAC recognizes different states of users, to specify constraints there are several possible policies that go from one extreme, where the model is constraint-free, to another extreme where constraints are specified at the session level. To illustrate this, consider specifying role-centric SOD constraints, where we have the following options:

a. Constraints-free policy: In structuring the authorization rules, this policy requires that no user can belong to a set of conflicting roles. This policy puts all the responsibilities on the rules and it is what Model A provides.

b. Potential-vs.-assumed policy: The model allows identifying conflicting roles and gathering them into conflicting roles sets. The constraints are specified among potential and assumed users. Assume r_i and r_j belongs to a conflicting role set cr_x, and a user, say *u*, is a potential user of both roles. According to this policy, once *u* assumed r_i, he is permanently banned from assuming r_j. This policy corresponds to static SOD.

c. Active-vs.-dormant policy: This potential-vs.-assumed policy is not fine enough to differentiate between active and dormant users. If *u* in the above example becomes dormant with respect to r_i, he is still unable to activate r_j. The active-vs.-dormant policy specifies constraints that differentiate between active and dormant users. If *u* becomes dormant with respect to r_i, he can activate r_j. While active in r_j he cannot reactivate r_i until he deactivates r_j. This policy corresponds to simple dynamic SOD. The advantage of this policy over the previous one is that when there are large numbers of users, many users may be potential users *wrt* specific roles but they may never assume these roles. In this case, it is more useful to specify constraints using this policy rather than potential vs. assumed policy.

d. Session-based policy: This policy specifies the constraints among roles accessed by a single active user within a single session. As its name indicates, this policy corresponds to session-based dynamic SOD.

5.2 Analysis of Model C

In the following sections we discuss the three methods provided by Model C to allow constraints specification. We analyze each method and highlight its pros and cons. Model C comes with three languages ASL_{C1}, ASL_{C2}, and ASL_{C3}, which correspond to methods 1, 2, and 3 respectively.

5.2.1 Method 1: Rule-Specific Constraints

5.2.1.1 Introduction

The scope of constraints of this kind is limited to the authorization rule in which the constraint is specified. The constraints are specified using the syntactic extension described later. Here is an example:

$$ae_i \Rightarrow r_i \text{ ST } c_x$$

This rule states that users who satisfy attribute expression ae_i are authorized to role r_i only if constraint c_x holds.

5.2.1.1.1 Relations Among Constraints

Since constraints are logical predicates, we can define relations among them based on logical implication. We may start by defining them *wrt* simple predicates and then move to identifying relations among composite constraints that are built using simple

predicates. Relations among constraints have been discussed in the literature; see, for example, [BJWW2002]. Below we identify some relationships between similar constraints consisting of simple predicates.

Theorem 4

With respect to a specific role in Role-centric SOD, the following holds:

a. For any set of conflicting roles:

   Static constraint holds $\rightarrow$ Simple dynamic constraint holds $\rightarrow$ Session-based dynamic constraint holds

b. For two conflicting roles sets, cr_x and cr_y, with c_x and c_y being SOD constraints of the same type on cr_x and cr_y, respectively, the following holds:

$$(cr_x \subseteq cr_y) \rightarrow (c_y \text{ holds} \rightarrow c_x \text{ holds})$$

Proof:

a. The proof for part (a) of the theorem follows from the definition.

b. The proof for part (b): Let cr_x and cr_y be the two sets of role, $\{r_1,..,r_m\}$ and $\{r_1,..,r_n\}$, respectively, such that $cr_x \subseteq cr_y$. Assume also that c_y is a static role-centric SOD constraint that holds on cr_y. This means that a static role-centric SOD holds on any subset of cr_y. If c_x is the constraint that enforces static role-centric SOD on cr_x, then c_y holds $\rightarrow$ c_x holds. The same thing can be said about simple dynamic and session-based dynamic constraints.

□

Theorem 5

With respect to a specific role in User-centric SOD, the following holds:

i. For any set of conflicting roles:

Static holds $\rightarrow$ Dynamic holds

ii. For the 2 conflicting users sets, cu_x and cu_y with c_x and c_y being user-centric SOD constraints on is on cu_x and cu_y respectively, the following holds:

$(cu_x \subseteq cu_y) \rightarrow (c_y \text{ holds} \rightarrow c_x \text{ holds})$

Proof:

a. The proof for part (a) of the theorem follows from the definition.

b. To prove part (b), let cu_x and cu_y be two sets of users, $\{u_1,..,u_m\}$ and $\{u_1,..,u_n\}$, respectively, such that $cu_x \subseteq cu_y$. The rest of the proof is similar to part b of the previous theorem.

□

Theorem 6

With respect to role *r*, the following holds among cardinality constraints:

a. Assume we have two constraints: c_x is specifying the static cardinality, while c_y is specifying the dynamic cardinality. If the static and dynamic cardinality values of *r* are equal, then c_x holds $\rightarrow$ c_y holds.

b. Assume we have two constraints: c_x and c_y are specifying a cardinality value *m* and *n* over of *r* where $n \geq m$, then, c_x holds $\rightarrow$ c_y holds.

Proof: The proof follows from the definition.

□

Theorem 7

With respect to role r_k, the following holds among prerequisite roles constraints:

a. Assume we have two constraints that specify a single role as a prerequisite role for r_k: c_x specifies r_g, while c_y specifies r_h. If $r_g \geq r_h$, then c_x holds$\rightarrow$ c_y holds.

b. Assume we have these two constraints both specify a set of roles as prerequisite roles for r_k: c_x specifies role-set$_g$, while c_y specifies role-set$_h$. If role-set$_g \subseteq$ role-set$_h$, then c_y holds $\rightarrow$ c_x holds.

Proof:

a. Assume that c_x and c_y state that r_g and r_h are prerequisites for r_k, respectively. If $r_g \geq r_h$ then any user *u* who is authorized to r_g is also authorized to r_h. In other words, if c_x holds *wrt u*, then c_y holds too.

b. Suppose that c_x and c_y specify role-set$_g$ and role-set$_h$ as prerequisite for r_k, respectively. If c_y holds for a user *u*, this means *u* is authorized to role-set$_h$. Since role-set$_g \subseteq$ role-set$_h$, then *u* is authorized to role-set$_g$, and hence, c_x holds.

This proves the theorem.

□

5.2.1.1.2 Authorization Policies

In Model A we consider a rule, say *rule*$_i$, *relevant wrt* a user, say *u*, when activate *r* such that $r \in RHS(ae_i)$, if *u* satisfies *rule*$_i$, i.e. $(u,ae_i) \in$U_AE. We require that a user must satisfy all constraints in *one* relevant rule. This concept is important for determining authorization in Model C. There are two possible approaches:

a. PTP: We require that a user must satisfy all constraints in *one* relevant rule.

b. DTP: We require that a user must satisfy all constraints in *all* relevant rules.

The PTP approach is less restrictive, and keeps the model monotonic. On the other hand, it can be argued that DTP approach is more secure since it requires the user to satisfy all the constraints in all the relevant rules. Obviously, the approach selected affects users' authorization i.e. URAuth definition. The following definition reflects this.

Definition 11

Model A definition with the following modifications:

1. An authorization rule $rule_i$: $ae_i \Rightarrow$ RHS ST c_i. The syntax is given in by ASL_{C1} language.
2. PTP approach:
   a. $URAuth^{PTP} = \{(u,r) | (\exists rule_i)[(u, ae_i) \in U\_AE \wedge r \in RHS(ae_i) \wedge c_i$ is true for $u\}$

      Call this $A \wedge \alpha$

   b. URAuth with *can_assume* = $(A \wedge \alpha) \vee B$

      $URAuth^{PTP \text{ with } can\_assume} = \{(u,r) | ((\exists rule_i)[(u, ae_i) \in U\_AE \wedge r \in RHS(ae_i)] \wedge c_i$ is true for u)

      $\vee (\exists rule_j)$ [$(u, ae_j) \in U\_AE \wedge r' \in RHS(ae_j) \wedge can\_assume(r', r, t, d) \wedge$ *can_assume* has not expired]}

3. DTP approach::
   a. $URAuth^{DTP} = \{(u,r) | (\exists rule_i)[(u, ae_i) \in U\_AE \wedge r \in RHS(ae_i)]$

      $\wedge (\forall rule_i)[(u, ae_i) \in U\_AE] \wedge r \in RHS(ae_i) \rightarrow c_i$ is true for $u]\}$

      Call this $A \wedge \chi$

   b. With *can_assume*: $(A \wedge \chi) \vee B$

$$\text{URAuth}^{\text{DTP with } can\_assume} = \{(u,r) \mid ((\exists rule_i)[(u, ae_i) \in U\_AE \wedge r \in RHS(ae_i)]$$

$$\wedge (\forall rule_i)[(u, ae_i) \in U\_AE] \wedge r \in RHS(ae_i) \rightarrow c_i \text{ is true for } u])$$

$$\vee (\exists rule_j) [\ (u, ae_j) \in U\_AE \wedge r' \in RHS(ae_j) \wedge can\_assume(r', r, t,$$

$$d\ ) \wedge can\_assume \text{ has not expired }]\}$$

5.2.1.2 The ASL_{C1} Language

Model C extends ASL_A language of Model A by allowing the specification of constraints in the authorization rules. The resulting language ASL_{C1} (shown in Figure 31) modifies the syntax ASL_A as follows:

Rule ::= Attribute_Expression ⇒ Roles **[SUBJECT TO** Constraints]

Constraints ::= [¬] ||Constraints

| Constraints ||∧ ||Constraints

| (||Constraints ||∧ ||Constraints||)

| Constraint

Constraint ::= {*specified by the organization*}

In the discussion below, we will shorten the reserved word “SUBJECT TO” to “ST”.

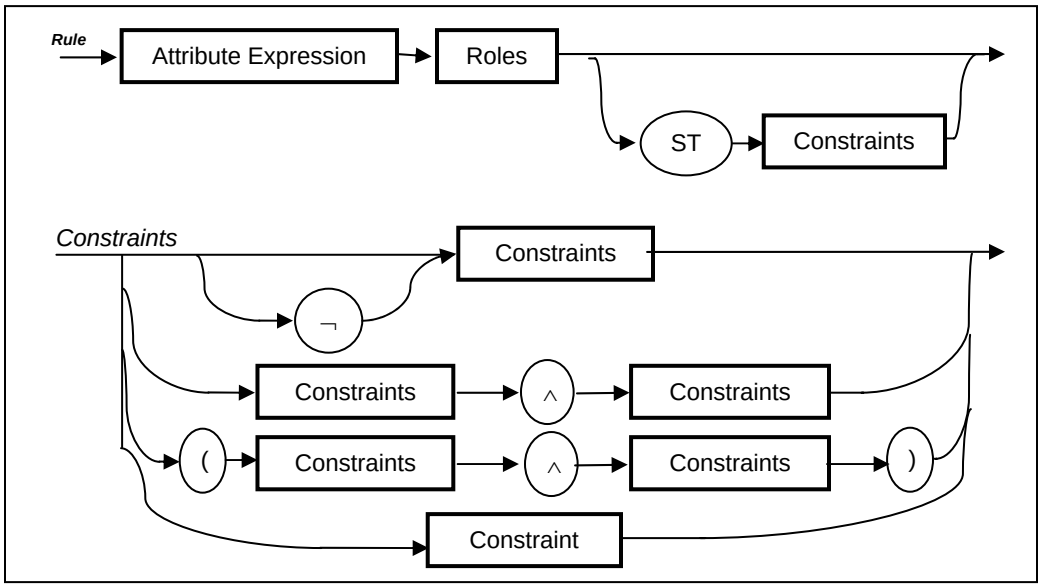

Figure 31: Additional Syntax for ASL_{C1} Language used for Rule-Specific Constraints Method

5.2.1.3 Constraints Specification

To specify constraints, we will use functions and sets that we defined for RB-RBAC model. Note that constraints specified in this section are enforced only when the corresponding authorization rule is invoked. Before we start specifying the constraints, we need the following definitions.

Definition 12

1. The following sets are modified from [Ahn1999]:
   - Conflicting Roles Set (CR) = all conflicting roles sets $\{cr_1,\ldots,cr_n\}$ where $cr_i = \{r_i \ldots r_t\} \subseteq$ IR. CR is a collection of sets of roles that have been defined to conflict.

Conflicting Users Set CU = all conflicting user sets $\{cu_1,\ldots,cu_w\}$ where $cu_i = \{u_i,\ldots,u_x\} \subseteq U$

2. Function *d_roles* (u_i): $U \rightarrow 2^{IR}$ returns roles in which user u_i is dormant

$$d\_roles(u_i) = \{r \in IR | ((u_i,r) \in URD\}$$

3. Current_user: Holds the user who satisfies the authorization rule that the system invokes so the role(s) in its RHS can be activated by the user.
4. Other_Conflicting_Users(u), for short, $OCU(u)$: $U \rightarrow 2^U$, a function that returns a set that holds all users in conflict with u, $= \cup cu_s - \{u\}$ where cu_s is any conflicting user set such that $u \in cu_s$ and $cu_s \in CU$.

5.2.1.3.1 SOD Constraints

RB-RBAC enables specifying the following types of SOD constraints:

5.2.1.3.1.1 Role-Centric SOD

Assume that r_g and r_h are 2 conflicting roles, i.e. both r_g and $r_h \in cr_x$. Model C allows expressing these types of constraints:

i. Static: This means that no user should assume r_g and r_h. In RB-RBAC, this includes active and dormant users. To express this constraint in the production rule that yields r_g, we write

$$ae_i \Rightarrow r_g \text{ ST } |(roles(sessions(\text{Current\_user})) \cup d\_roles\ (\text{Current\_user}) \cup r_g) \cap OE(CR)| \leq 1$$

Current_user is the user who satisfies ae_i and the system that implements RB-RBAC wants to determine if he is authorized to r_g. *roles*(*sessions*(Current_user)) returns all the roles in which Current_user is active while *d_roles* (Current_user) returns all the

roles in which Current_user is dormant. The union of the two gives all assumed roles of Current_user. OE(CR) returns one set of conflicting roles, so the constraint is saying that the set that represents all Current_user's roles does not have any 2 roles that are in conflict.

ii. Simple dynamic SOD: This requires that no user with two conflicting roles be activated simultaneously. In the production rule that yields r_g, we express this constraint:

$$ae_i \Rightarrow r_g \quad ST \mid (roles(sessions(\text{Current\_user})) \cup r_g) \cap OE(CR)| \leq 1$$

Since we only care about roles in which Current_user is active, we use *roles*(*sessions*(Current_user)).

iii. Session-based Dynamic SOD: This requires that there are no users with two conflicting roles enabled in a session.

$$ae_i \Rightarrow r_g \; ST \mid (roles(OE(sessions(\text{Current\_user}))) \cup r_g) \cap OE(CR) \mid \leq 1$$

Function *roles*(OE(*sessions*(Current_user))) returns the roles which Current_user activates in one session.

5.2.1.3.1.2 User-centric SOD

In this section, we specify SOD properties with the notion of conflicting users.

a. Static:

This constraint requires that conflicting users cannot have a common role.

$$ae_i \Rightarrow r_g \quad ST$$

$$((roles(sessions(\text{Current\_user})) \cap d\_roles(OCU(\text{Current\_user}))) = \varnothing$$

$$\wedge\, (roles(sessions(\text{Current\_user})) \cap roles(sessions(OCU(\text{Current\_user}))) = \varnothing)$$

(*roles*(*sessions*(Current_user)) means all roles which are activated by Current_user. *roles*(*sessions*(*OCU*(Current_user))) and *d_roles*(*OCU*(Current_user)) return the roles that other users who conflict with Current_user are activating or are dormant in, respectively. The above expression ensures that no two conflicting users will be in a role even if one of them is dormant. Once a user activates the role, the role is locked up permanently so no other user who belongs to the conflicting set is allowed to assume that role.

b. Dynamic:

This requires that there are no two conflicting users active in the same role. This requirement is expressed as follows:

$ae_i \Rightarrow r_g$ ST

$$roles(sessions(\text{Current\_user})) \cap roles(sessions(OCU(\text{Current\_user}))) = \varnothing$$

If a conflicting user becomes dormant *wrt* r_g, this constraint allows another conflicting user, i.e. belongs to the same conflicting set, to activate r_g.

5.2.1.3.2 Cardinality Constraints

To specify this, we introduce the following definition.

Definition 13

5. Function *a_users* (r_g): IR $\rightarrow 2^U$ returns users active *wrt* role r_g

a_users$(r_g) = \{u \in U | ((u,r_g) \in URA\}$

6. Function *d_users* (r_g): IR $\rightarrow 2^U$ returns users dormant *wrt* role r_g

d_users$(r_g) = \{u \in U | ((u,r_g) \in URD\}$

The static cardinality value of a role r_g is the maximum number of assumed users (i.e. dormant or active) allowed in r_g at any point in time. Dynamic cardinality value specifies the maximum number of users who can simultaneously activate r_g.

Using the definition above, we specify the two types of cardinality constraints:

a. Static cardinality constraint: This specifies that the number of assumed users in role r_g is less than or equal to the static cardinality value we assign to r_g. Thus we write:

$$ae_i \Rightarrow r_g \ \ ST \ \ |a\_users\ (r_g) \cup d\_users\ (r_g)| \leq n$$

So, the maximum number of users, whether active or dormant *wrt r*, should not exceed n.

b. Dynamic cardinality constraint: This specifies that the number of active users in r_g does not exceed the dynamic cardinality value we desire for r_g. To express this we write:

$$ae_i \Rightarrow r_g \ \ ST \mid a\_users\ (r_g)| \leq n$$

So, the maximum number of active users *wrt* r_g should not exceed n.

5.2.1.3.3 Prerequisite Role Constraints

RB-RBAC differentiates between two types of constraints:

a. Static prerequisite constraint: This states that a user who satisfies the attribute expression required by the authorization rule cannot assume roles in its RHS, say r_g,

unless he has already assumed another role r_h regardless of whether he is active or dormant *wrt* r_h. To express this, we write

$$ae_i \Rightarrow r_g \quad ST \ (Current\_user, r_h) \in URA \vee (Current\_user, r_h) \in URD$$

b. Dynamic prerequisite constraint: Assume role r_h is a dynamic prerequisite for role r_g. This means that a user cannot assume role r_g unless he is an active user in role r_h. Thus we write:

$$ae_i \Rightarrow r_g \quad ST \ (Current\_user, r_h) \in URA$$

5.2.1.4 Constraints Specification in the Presence of a GRH

In the following sections we discuss specification of different constraints in the presence of a given role hierarchy. Seniority among roles in the GRH has to be taken into consideration and, as a result, some functions have been modified as in the following definition.

Definition 14

7. *roles**: $S \rightarrow 2^{IR}$ is modified from *roles* to require *roles**$(s_i) = \{r \in IR \mid (\exists r' \geq r)[\ r' \in role(s_i)]\}$ (which can change with time)
8. *a_users**$(r_g) = \{u \in U \mid (\exists r')\ [r' \geq_{GRH} r_g \wedge (u, r') \in URA]\}$ which returns the active users in role r_g and all roles senior to it in a GRH.

5.2.1.4.1 SOD Constraints

5.2.1.4.1.1 Role-Centric SOD

Assume that r_g and r_h are two conflicting roles, i.e. both r_g and $r_h \in cr_x$. Using the definition above, Model C allows us to express three types of constraints:

a. Static:

$ae_i \Rightarrow r_g$ ST

$|(roles^*(sessions(\text{Current\_user})) \cup d\_roles\ (\text{Current\_user}) \cup r_g) \cap OE(CR)| \leq 1$

$roles^*(sessions(\text{Current\_user}))$ returns all the roles in which Current_user is active and all roles junior to them.

b. Simple dynamic SOD:

$ae_i \Rightarrow r_g$ ST | $(roles^*(sessions(\text{Current\_user})) \cup r_g) \cap OE(CR)| \leq 1$

c. Session-based Dynamic SOD:

$ae_i \Rightarrow r_g$ ST $|(roles^*(OE(sessions(\text{Current\_user}))) \cup r_g) \cap OE(CR)| \leq 1$

Function $roles^*(OE(sessions(\text{Current\_user})))$ returns the roles which Current_user activates in one session and their juniors.

5.2.1.4.1.2 User-centric SOD

a. Static:

$ae_i \Rightarrow r_g$ ST

$((roles^*(sessions(\text{Current\_user})) \cap d\_roles(OCU(\text{Current\_user}))) = \varnothing$

$\wedge\ (roles^*(sessions(\text{Current\_user})) \cap roles^*(sessions(OCU(\text{Current\_user}))) = \varnothing)$

($roles^*(sessions(\text{Current\_user}))$ means all roles which are activated by Current_user and their juniors. $roles^*(sessions(OCU(\text{Current\_user})))$ returns roles which other conflicting users in the same conflicting user set cu_i are activating and their junior roles. The above expression ensures that no two conflicting users will be in a role even if one of them is dormant. Once a user assumed the role, the role is locked up permanently so no other user who belongs to the conflicting set is allowed to assume that role.

b. Dynamic:

$ae_i \Rightarrow r_g$ ST

*roles**(*sessions*(Current_user)) ∩ *roles**(*sessions*(*OCU*(Current_user))) = ∅

*roles**(*sessions*(Current_user)) returns all roles which Current_user, who is a conflicting user, activates in sessions and their junior roles. *roles**(*sessions*(*OCU*(Current_user))) returns all roles which are activated by other conflicting users in the same conflicting users set cu_i and their junior roles.

5.2.1.4.2 Cardinality Constraints

This section presents new semantics for the cardinality semantics and scope. The Indirect/Static and Indirect/Dynamic cardinalities, discussed below, are newly added interpretations of the cardinality constraint. Also, the limited scope vs. extended scope is a novel concept in this regard.

So far, we considered two types of cardinality counts: static and dynamic. By introducing GRH, when interpreting the cardinality of a role, one may count only users who assume a role via satisfying an authorization rule, which yields that role; call this direct method. Alternatively, in addition to the users who assume a role directly, cardinality counts those who activate it via activating one of its seniors in GRH. For the sake of argument, we call this the indirect method. This yields four possible interpretations of cardinality:

a. Direct/Static cardinality: Similar to the case of flat roles discussed above, i.e. no GRH.
b. Direct/Dynamic cardinality: Similar to the case of flat roles discussed above, i.e. no GRH.

c. Indirect/Static cardinality: To require that the maximum number of users in r_g ≤ n, we write

$$ae_i \Rightarrow r_g \ \text{ST} \ |a\_user^*(r_g) \cup d\_users(r_g)| \leq n$$

This counts those who are active in r_g or any of its seniors and those who are dormant *wrt* r_g.

d. Indirect/Dynamic cardinality: This states the maximum number of active users allowed to a specific role. Thus, to express that the maximum number of active users in *r* must be less than or greater to m, we write

$$ae_i \Rightarrow r_g \quad \text{ST} \ |a\_users^*(r_g)| \leq m$$

This counts only those who are active in *r* or any of its seniors.

Also, there are two possible scopes for enforcing cardinality constraints:

a. Limited scope: Enforcement of r_g cardinality is limited to the rule in which the constraint is specified.

b. Extended scope: Enforcement of r_g cardinality includes the rule in which the constraint is specified and any rule that authorizes users to r_g.

Enforcing constraints belongs to the mechanism layer in the OM-AM framework discussed in Chapter 1 of the dissertation. As such, it will be discussed briefly to show its relation to the specification of the model. The issue of enforcement scope should not be confused with direct and indirect interpretations of cardinality which determine which users to count. To illustrate, consider the following set of rules:

rule$_1$: $ae_1 \Rightarrow r_k$ ST c_1

rule$_2$: $ae_2 \Rightarrow r_k$

rule$_3$: $ae_3 \Rightarrow r_g$ ST c_3

rule$_4$: $ae_4 \Rightarrow r_g$

where c_1 and c_3 are *cardinality constraints* on r_k and r_g respectively. Suppose that *rule*$_1 \geq$ *rule*$_3$, *rule*$_2 \geq$ *rule*$_4$ and the liberal approach is used to construct IRH. Accordingly, $r_k \geq r_g$ in IRH which we assume is in conformance with GRH. The limited scope calls for enforcing the cardinality constraints when *rule*$_1$ or *rule*$_3$ is invoked. The extended scope *wrt* r_g entails enforcing the cardinality constraint on *rule*$_1$ and *rule*$_2$ since they authorize users to r_k which is senior to r_g. It also entails enforcing the constraint on *rule*$_3$ and *rule*$_4$ because it authorizes users to r_g. If a user *u* satisfies ae_2 and wants to activate r_k, constraint c_3 will be enforced. *u* will not be counted if direct cardinality is enforced, but if indirect cardinality is in effect, *u* will be counted.

Moreover, if a user *u* satisfies *rule*$_1$ which has a cardinality constraint, c_1, the two constraints, i.e. c_1 and c_3, might conflict. Different policies to resolve this conflict can be developed such as:

a. The constraint in the rule being invoked takes precedence, so if a user satisfies *rule*$_1$, c_1 is enforced even if this leads to exceeding the maximum number of users authorized to r_g according to c_3.
b. The constraint with the lower value takes precedence so in the above example, if $c_3 \leq c_1$, a user who satisfies *rule*$_1$ may not be authorized to r_k even if c_1 holds because c_3 will be violated.

In all cases, the scope of enforcement does not say anything about which users to count when enforcing the constraint. This is determined by direct/indirect interpretation. Assume that u_1, u_2, u_3, and u_4, satisfy *rule*$_1$, *rule*$_2$, *rule*$_3$, and *rule*$_4$ respectively. Suppose also that u_1 and u_4 are dormant *wrt* r_k and r_g, while u_2 and u_3 are active *wrt* r_k and r_g

respectively. Table 7 shows the interaction among the four cardinality interpretations and the two scopes of enforcement *wrt* r_g.

Table 7: Possible Interpretation of Cardinality in the Presence of a GRH

	Users Counted			
Scopes of enforcement & Cardinality Interpretation ↓	Active users *wrt* $r_k \geq r_g$	Dormant users *wrt* r_g	Enforcement scope on rules:	Users to count
Direct/Static/Limited	N	Y	3	u_3
Direct/Static/ Extended	N	Y	1→4	u_3, u_4
Direct/Dynamic/Limited	N	N	3	u_3
Direct/Dynamic/Extended	N	N	1→4	u_3
Indirect/Static/Limited	Y	Y	3	u_1, u_2,u_3, u_4
Indirect/Static/ Extended	Y	Y	1→4	u_1, u_2,u_3, u_4
Indirect/Dynamic/ Limited	Y	N	3	u_2,u_3
Indirect/Dynamic/Extended	Y	N	1→4	u_2,u_3

5.2.1.4.3 Prerequisite Role Constraints

When specifying this constraint, the presence of a GRH necessitates taking roles senior to the prerequisite role into consideration. Here is the RB-RBAC specification:

a. Static prerequisite constraint: This states that a user who satisfies the attribute expression required by the authorization rule is not authorized to a role(s) in its RHS, say r_g, unless the user is authorized to another role r_h, or the user is authorized to r_k such that $r_k \geq_{GRH} r_h$.

To express the above requirements, we write

$$ae_i \Rightarrow r_g \quad ST \ ((Current\_user, r_h) \in URA \cup URD)$$

$$\vee (\exists r_k) [r_k \geq_{GRH} r_h \ \wedge (Current\_user, r_k) \in URA \cup URD]$$

b. Dynamic prerequisite constraint: If role r_h is a dynamic prerequisite for role r_g, then this means that a user cannot assume role r_g unless he is an active user in role r_h or any role senior to r_h. Thus we write:

$$ae_i \Rightarrow r_g \quad ST \ ((Current\_user, r_h) \in URA)$$
$$\vee (\exists r_k)[r_k \geq_{GRH} r_h \wedge (Current\_user, r_k) \in URA]$$

5.2.1.5 User States Diagram

Introducing constraints via this method requires that user state diagram of Model A be modified as shown in Figure 32. The modification is represented by the label 'c' which indicates a change in a user's compliance to a constraint. The diagram is equally valid for PTP or DTP approaches. Users' states change according to constraints evaluation. We require performing constraints evaluation as follows:

a. Periodical evaluation: This happens as often as dictated by security policy and should be supported by RB-RBAC enforcement architecture. The user's states *wrt* roles yielded by the rule that specifies constraints are updated every time these constraints are evaluated. Introducing a new constraint or loosing compliance to an existing one may result in changing the user's state from P to N, and from Act or D to R. Similarly, dropping or modifying a constraint may result in changing a user in N or R states to P or D respectively. Note that no user's state can change from R to Act because that requires a user's active measure, which is not possible to invoke while being in R state. Instead, the system that implements RB-RBAC must change the user's state into D first. All these changes are represented by label "c" in the diagram.

b. To maintain consistency of the system, the relevant constraint is evaluated if a potential or dormant user tries to activate a role. This ensures that users who happen to be in these two states cannot exploit the system between periodical evaluations of constraints.

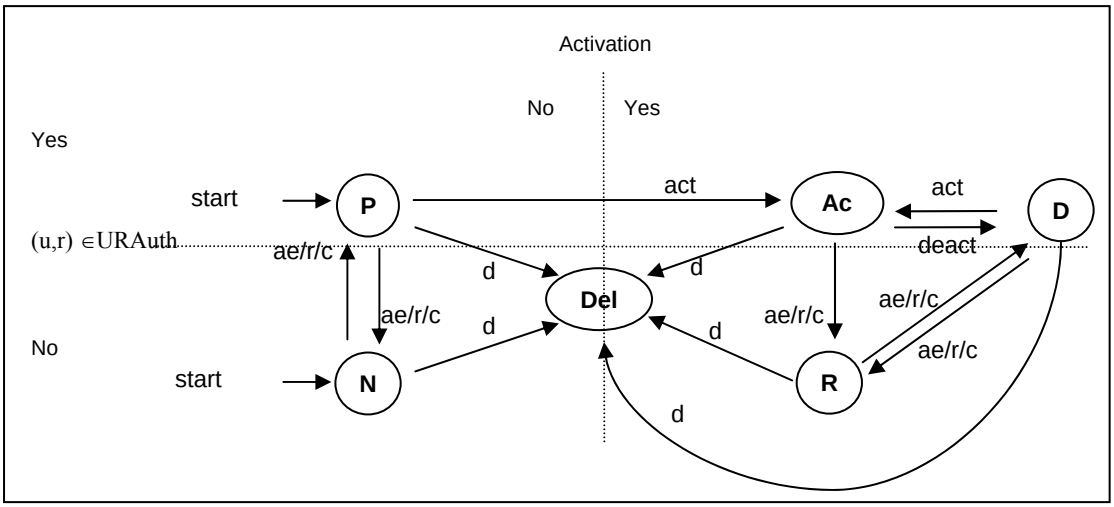

Figure 32: User's State Diagram of Method 1

5.2.1.6 Discussion

This method has the following merits:

a. It provides fine granularity specification of constraints, which is needed when we desire to control the assignment of users who meet certain criteria (represented by the attributes expression in the authorization rule) to roles while unrestricting the assignment of these very roles via meeting other criteria. In contrast, an invariant (discussed later) applies to all rules, and subsequently, to all users rendering it less flexible.

b. From an implementation standpoint, the processing of rules is more efficient since only constraints associated with the invoked rules are evaluated. This is in contrast to invariants where every invariant has to be checked for compliance every time a rule is invoked.

c. From a functional outlook, stating the constraints as a separate part of the rule vis-à-vis integrating them with attributes expressions (as is the case of system attribute method) provides neatness and visibility. It is easy to tell what constraints must hold for a user to assume a specific role.

d. Also, it facilitates RB-RBAC implementation since it simplifies plugging in other constraints specification language. An example for this is the one proposed by Bettini et al. [BJWW2002]. In that work, the authors formalize a rule-based policy framework that includes provisions and obligations. They provide a mechanism that takes provisions into account when it tries to make the *best* derivation among possibly several that support a policy decision. It is our belief that a similar mechanism could be applied regarding constraints to find the rule(s) that authorize the user to assume the requested roles while satisfying the smallest set of constraints. An RB-RBAC constraint corresponds to a *provision-obligation* PO-formula defined in [BJWW2002]. Using the algorithms provided in that work, we can derive the global provisions and obligation (PO) set for the constraints of each authorization rule. To illustrate, assume that r_s, r_t, r_u are three roles such that r_t and r_u are prerequisites of r_s but not simultaneously, i.e. a user *u* is required to activate either r_t or r_u in order to activate r_s. This requires identifying seniority among constraints and generating consumption hierarchy based on the semantic

relations among them as well as on the numerical weights given to them. Then we can use the technique discussed in [BJWW2002] to derive the best valid global PO set (or BPOS). Intuitively, BPOS is the valid set of constraints that has the minimum weight among all valid alternative sets of constraints required to grant user *u* the role r_s. We can use BPOS to determine if it is easier for *u* to satisfy r_t or r_u and, therefore, determine which authorization rule the system that implements RB-RBAC should invoke.

However, this method suffers from the following drawbacks:

a. The usage of this method to specify invariants, i.e. that apply to all rules, is verbose since they must be specified repeatedly in every rule.

b. As a result of the first point, this method is error-prone especially when adding, modifying and removing invariants, and/or adding new rules. It is easy to overlook making the necessary changes across the board, or adding this kind of constraints to newly added rules.

c. The fine granularity the method provides could give undesirable results. Consider the following set of rules:

   $ae_i \Rightarrow r_g$

   $ae_j \Rightarrow r_g$ ST $|a\_user^*(r_g) \cup d\_users\ (r_g)| \leq 7$

   The second rule, $rule_j$, ensures that no more than seven users are authorized to r_g at one time. However, the system theoretically allows an infinite number of users to be authorized to r_g so long as they satisfy $rule_i$. This kind of arrangement is acceptable if we aim to limit the number of users authorized to r_g because they own certain attributes, e.g. they satisfy ae_j, and at the same time we want to set no

restriction on the number of users who are authorized to r_g by virtue of being associated with another set of attributes, ae_i. Nonetheless, there is a chance to overlook the impact $rule_i$ has on the constraint. Every time a user satisfies $rule_i$ and assumes r_g, the number of vacancies available for users satisfying $rule_j$ is decremented.

5.2.2 Method 2: System Attributes

5.2.2.1 Introduction

Similar to the previous method, the scope of constraints in this method is limited to the authorization rules in which they are specified. However, constraints in this method are not treated as a separate entity. Rather, they are merged with the attributes expression in the specifying rule.

5.2.2.2 The ASL_{C2} Language

In Model A, the focus was on users' attributes, which are tested to see if they meet the attribute expressions specified in the authorization rules. To specify the above mentioned three classes of constraints within the LHS of the authorization rules, new types of attributes need to be introduced. These attributes, we call them system attributes, hold values related to system information such as conflicting users, conflicting roles, prerequisite roles, etc. The language ASL_{C2} extends ASL_A by introducing the constructs shown in Figure 33.

The following discussion explains the meaning of the newly introduced system defined attributes:

a. Current_session: Holds the session in which Current_user wants to activate a specific role, say r_g. This attribute is maintained by the system that implements RB-RBAC.

b. Other Conflicting Roles(r_g), *OCR* for short: A function, IR$\rightarrow 2^{IR}$, that returns all roles in conflict with r_g, $= \cup$ cr $-\{r_g\}$ where cr is any conflicting

role set such that $r_g \in cr$. A conflicting roles set is a set that holds roles that are identified to be in conflict with each other.

Attribute ::= System_defined_a | User_defined_a
System_defined_a ::= current_user | current_session | System_defined_s
User_defined_a::= {*specified by the organization*}
Enumerated_Set ::= System_defined_s | User_defined_s
System_defined_s::= Complex | Function
Complex::= Function [(||Parameter ||)]
| Complex || Set _op || Complex | (||Complex || Set _op || Complex ||)
Function::= *roles* |*roles** | *a_users* | *d_users* | *a_users** | *d_users** |*OCU*|*OCR* | *prerequisite*
Parameter::= System_defined_s| Role | Number
Attribute_Value::= S*tat_card* |*Dyn_card* | User_defined_s
Set_op ::= ∪|∩
User_defined_s::= {*specified by the organization*}

Figure 33: Syntax for *ASL$_{C2}$* Language used for System Attribute Method

c. *Stat_card*: Holds the static cardinality of r_g.

d. *Dyn_card*: Holds the dynamic role cardinality of r_g.

e. *Prerequisite*: A function IR$\rightarrow 2^{IR}$, which takes a role and returns its prerequisite role(s).

Figures 34 and 35 depict the syntax diagram of *ASL$_{C2}$* Language.

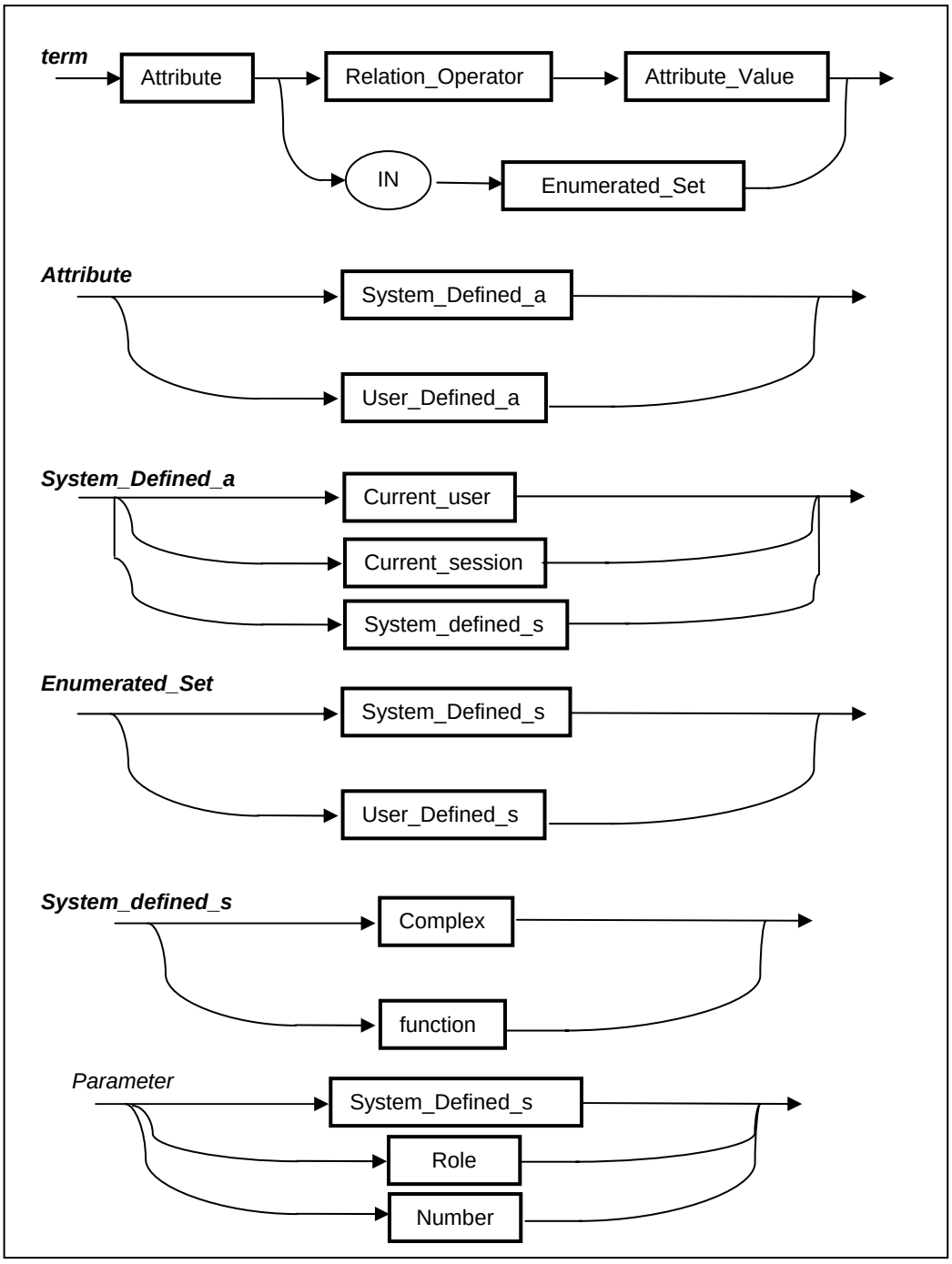

Figure 34: Syntax Diagrams of ASL_{C2} Language Used by the System Attribute Method (part A)

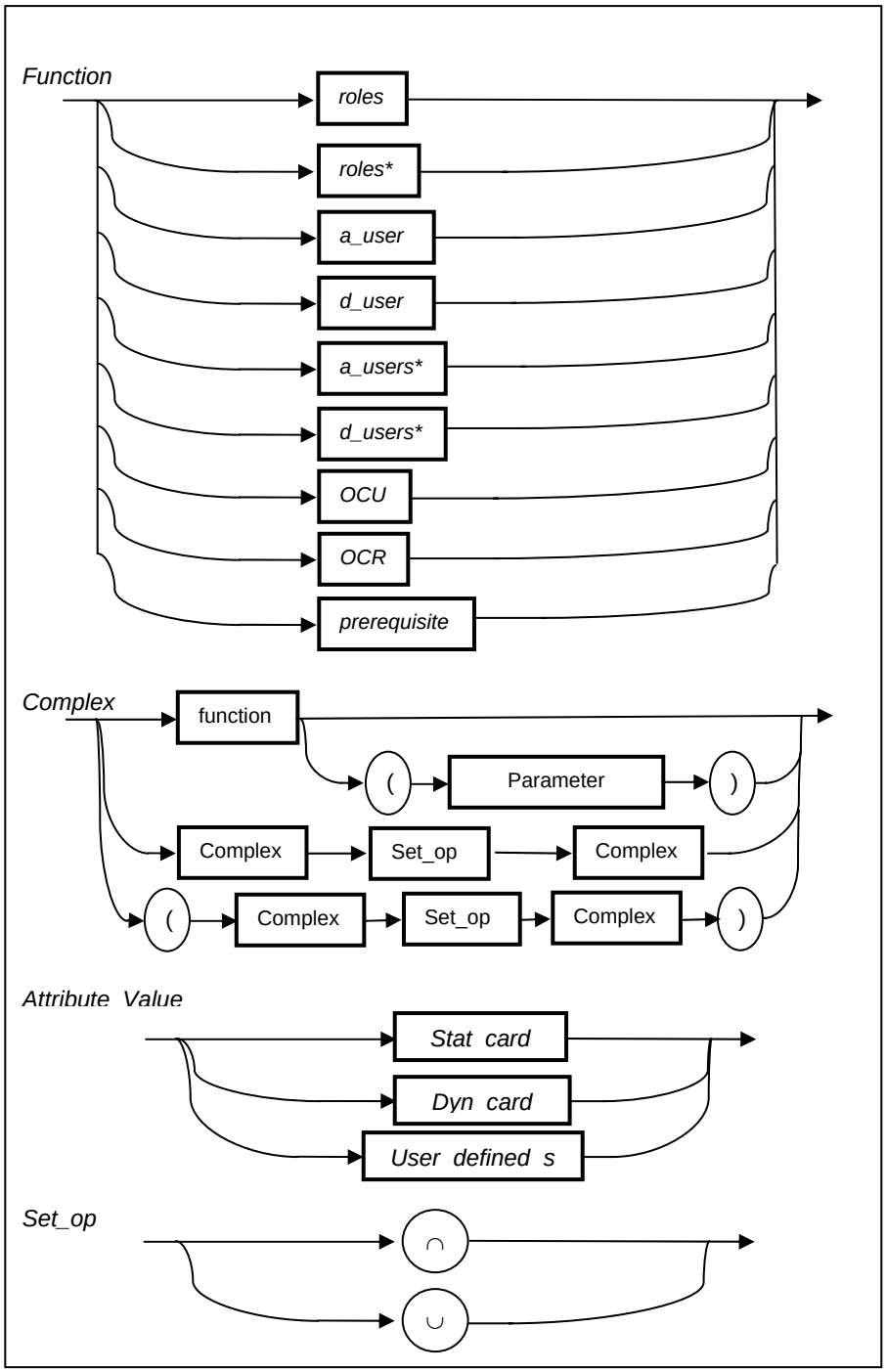

Figure 35: Syntax Diagrams of ASL_{C2} Language Used by the System Attribute Method (part B)

5.2.2.3 IRH Derivation

Since the terms that represent system attributes are blended with other attributes expression terms, the definition given in Model A to derive IRH stands. However, establishing seniority among attributes expressions requires determining seniority among terms representing system attributes.

5.2.2.4 Constraints Specification

In the following examples, let us assume that we desire to specify a constraint for $rule_i$: $ae_i \Rightarrow r_g$ such that the constraint specified is a term that becomes part of ae_i using the logical operator "$\wedge$".

5.2.2.4.1 SOD Constraints

5.2.2.4.1.1 Role-Centric SOD

a. Static: We add the following to ae_i:

$$\neg \text{ (Current\_user IN } (a\_users(OCR(r_g)) \cup d\_users(OCR(r_g))))$$

$OCR(r_g)$ returns all roles in conflict with r_g, thus $a\_users(OCR(r_g))$ and $d\_users(OCR(r_g))$, respectively, return active and dormant users in these roles.

b. Simple dynamic SOD: Add the following:

$$\neg \text{ (Current\_user IN } a\_users(OCR(r_g)))$$

c. Session-based Dynamic SOD:

$$\neg(OCR(r_g) \text{ IN } roles(\text{Current\_session}))$$

The above constraint ensures that no role that is in conflict with the requested role, r_g, is being activated by the current user in his current session. The function

roles(Current_session) returns the roles activated by the current user in the current session.

5.2.2.4.1.2 User-centric SOD

a. Static: ¬ (*OCU*(Current_user) IN {*a_users*(r_i) ∪ *d_users*(r_i)})

b. Dynamic: ¬ (*OCU*(Current_user) IN {*a_users*(r_i)})

5.2.2.4.2 Cardinality Constraints

a. Static: |*a_users*(r_g) ∪ *d_users*(r_g)| ≤ *Stat_card*(r_g)

b. Dynamic: |*a_users*(r_g)| ≤ *Dyn_card*(r_g)

5.2.2.4.3 Prerequisite Roles Constraints

a. Static: Current_user IN (*a_users*(*prerequisite*(r_g)) ∪ *d_users*(*prerequisite*(r_g)))

b. Dynamic: Current_user IN (*a_users*(*prerequisite*(r_g))

5.2.2.5 Constraints Specification in the Presence of a GRH

To take seniority among roles represented by GRH into consideration, we modify the specifications given in the previous section by replacing the functions *a_user* and *roles* with *a_user** and *roles** respectively. Again, we assume that we are to specify a constraint for $rule_i$: $ae_i \Rightarrow r_g$ such that constraints specified below are terms that are connected to ae_i using the "∧" operator.

5.2.2.5.1 SOD Constraints

5.2.2.5.1.1 Role-Centric SOD

a. Static: We add the following to ae_i:

   $\neg$ (Current_user IN ($a\_users^*(OCR(r_g)) \cup d\_users(OCR(r_g))$)

b. Simple dynamic SOD: Add the following:

   $\neg$ (Current_user IN $a\_users^*(OCR(r_g))$)

c. Session-based Dynamic SOD:

   $\neg(OCR(r_g)$ IN $role^*s$(Current_Session))

5.2.2.5.1.2 User-centric SOD

a. Static: $\neg$ (OCU(Current_user) IN $\{a\_users^*(r_i) \cup d\_users(r_i)\}$)

b. Dynamic: $\neg$ (OCU(Current_user) IN $\{a\_users^*(r_i)\}$)

5.2.2.5.2 Cardinality Constraints

Specifying cardinality constraint in the presence of a GRH can be done as follows:

a. Direct/Static cardinality: Similar to the case of flat roles discussed above.

b. Direct/Dynamic cardinality: Similar to the case of flat roles discussed above.

c. Indirect/Static cardinality: To specify this we write:

   $$|a\_users^*(r_g) \cup d\_users(r_g)| \leq Stat\_card(r_g)$$

   This counts users who are active in r_g or any of its seniors and those who are dormant *wrt* r_g.

d. Indirect/Dynamic cardinality: To specify this we write:

   $$|a\_users^*(r_g)| \leq Dyn\_card(r_g)$$

   This counts only those who are active in r_g or any of its seniors.

5.2.2.5.3 Prerequisite Roles Constraints

We need the following definition:

Definition 15

9. *d_users**(r_g) {u ∈U| (∃r'≥ r_g)[(u, r') ∈ URD]} which returns the dormant users in role r_g and all roles senior to it in a GRH.

To specify the constraint, we write:

a. Static:

   Current_user IN (*a_users**(*prerequisite* (r_g)) ∪ *d_users**(*prerequisite* (r_g)))

   Function *d_users** is used here to return the dormant user in r_g and roles senior to it.

b. Dynamic:

   Current_user IN (*a_users**(*prerequisite* (r_g))

5.2.2.6 User State Diagram

The user state diagram for this method is similar to its counterpart in Model A.

5.2.2.7 Discussion

This method has the following advantages:

a. It provides fine granularity specification of constraints and efficiency in implementation as in the rule-specific constraints method.
b. Model A definition of IRH and user's state diagram are used unmodified, which simplifies the analysis.

Nonetheless, this method has the following demerits:

a. If invariants are to be specified using this method, the method is verbose and thus error-prone for the same reason discussed in Method 1.

b. Since the system attributes that are used to specify constraints are mingled with users' attributes, constraints are not as visible and, thus, the possibility of making errors when specifying constraints increases.

c. Lack of clarity also leads to difficulty in mapping the security policy into authorization rules.

d. Also, subtle impact of different rules on constraints specified by other rules may cause undesirable results as discussed in Method 1.

This method bears some resemblance to the scheme suggested by Bacon et al. in OASIS [BMY2002]. OASIS aims to enable autonomous management domains to specify their own access control policies and interoperate using service level agreements (SLA's). OASIS is rule-based in the sense that role activation is linked to satisfying the rules associated with roles. With each *role activation rule*, there is a companion *role membership rule*. The structure of both types of rules is the same such that each rule has a list of conditions in the LHS that are necessary to satisfy in order to activate the role in the RHS of the rule. These conditions are of three types:

a. Prerequisite roles,

b. Appointment, and

c. Constraints.

OASIS regards constraints as atomic propositions but does not provide any syntax to specify them. As opposed to OASIS, RB-RBAC allows the specification of a wider range of constraints and also provides several methods for doing so.

5.2.3 Method 3: Invariants

5.2.3.1 Introduction

An invariant is a constraint that holds all of the time. In RB-RBAC, this means that it applies to all authorization rules. Traditionally, these are the type of constraints that are discussed in the context of RBAC [CS1995], [FK1995], [SCFY1996], [Kuhn1997], [Ahn1999]. In the latter reference, Ahn presented *RCL2000*, a language for specifying constraints in RBAC96.

5.2.3.2 The ASL_{C3} Language

The syntax of ASL_A is slightly modified to allow writing authorization rules with no attributes expression on LHS to indicate that the constraints must hold all of the time. Figure 36 shows the modified syntax, which we call ASL_{C3} language.

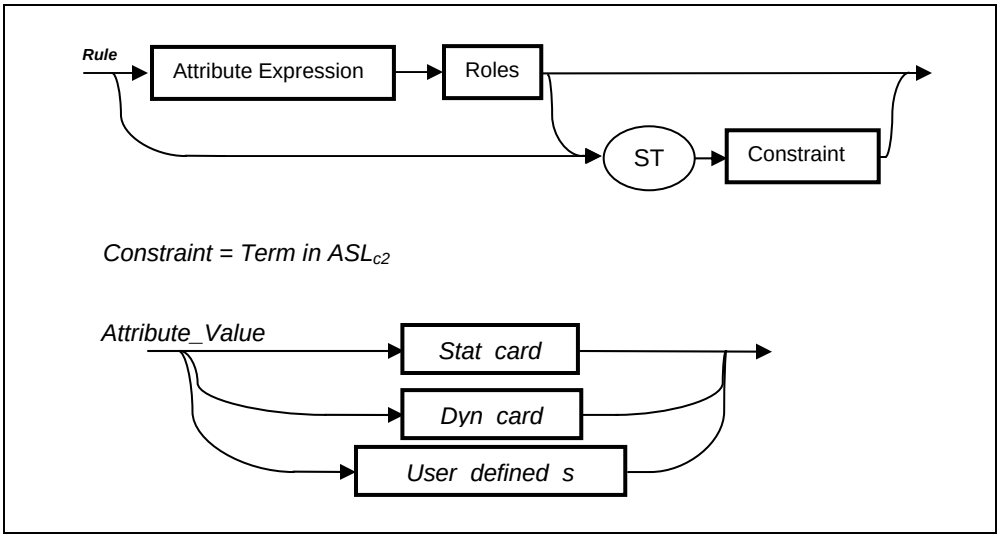

Figure 36: ASL_{C3} Language for Invariants Method

5.2.3.3 IRH Derivation

Because invariants apply equally to all rules, they have no effect on the derivation of IRH. Thus, the definition given in Model A to derive IRH holds.

5.2.3.4 Constraints Specification

The reader should notice that this method carries great resemblance to Method 1. An invariant has the appearance of an authorization rule that has no LHS.

5.2.3.4.1 SOD Constraints

In this section, we build on the work of Ahn proposed in [Ahn1999] with the following modifications:

a. We use RB-RBAC definition of function *roles* and *roles**.
b. We use RB-RBAC function *d_roles* to determine roles in which a user is dormant. This concept is not in RBAC96 which *RCL2000* supports.

We can specify all types of SOD constraints specified by Methods 1 and 2:

5.2.3.4.1.1 Role-Centric SOD

a. Static:

$$\Rightarrow \text{ST} \mid (\textit{roles}(\textit{sessions}(\text{OE(U)})) \cup \textit{d\_roles}\ (\text{OE(U)})) \cap \text{OE(CR)} \mid \leq 1$$

b. Simple dynamic SOD:

$$\Rightarrow \text{ST} \mid \textit{roles}(\textit{sessions}(\text{OE(U)})) \cap \text{OE(CR)} \mid \leq 1$$

c. Session-based Dynamic SOD:

$$\Rightarrow \text{ST} \mid \textit{roles}(\text{OE}(\textit{sessions}(\text{OE(U)}))) \cap \text{OE(CR)} \mid \leq 1$$

5.2.3.4.1.2 User-centric SOD

a. Static:

$$\Rightarrow \text{ST } ((roles(sessions(\text{OE}(\text{OE}(\text{CU})))) \cap d\_roles(\text{AO}(\text{OE}(\text{CU})))) = \varnothing$$

$$\wedge (roles(sessions(\text{OE}(\text{OE}(\text{CU})))) \cap roles(sessions(\text{AO}(\text{OE}(\text{CU}))))) = \varnothing)$$

b. Dynamic:

$$\Rightarrow \text{ST } roles(sessions(\text{OE}(\text{OE}(\text{CU})))) \cap roles(sessions(\text{AO}(\text{OE}(\text{CU})))) = \varnothing$$

5.2.3.4.2 Cardinality Constraints

In order to specify cardinality invariants, we use the functions *Stat_card and Dyn_card* we defined in Method 2. With each role r_g, we associate two cardinality values, static and dynamic, so to specify cardinality invariants for role r_g, we state:

a. Static cardinality:

$$\Rightarrow \text{ST } |a\_users\ (r_g) \cup d\_users\ (r_g)| \leq Stat\_card(r_g)$$

b. Dynamic cardinality:

$$\Rightarrow \text{ST } |a\_users\ (r_g)| \leq Dyn\_card(r_g)$$

These constraints apply to every rule that produces r_g.

5.2.3.4.3 Prerequisite Role Constraints

$\forall r_g, r_h \in \text{IR}, \forall u \in \text{U}$, we can specify the following:

a. Static prerequisite:

$$\Rightarrow \text{ST } (r_g \in roles(sessions(u)) \rightarrow (r_h \in (roles(sessions(u)) \cup d\_roles(u))))$$

b. Dynamic prerequisite:

$$\Rightarrow \text{ST } (r_g \in roles(sessions(u)) \rightarrow r_h \in roles(sessions(u)))$$

5.2.3.5 Constraints Specification in the Presence of a GRH

This is similar to what we have shown above, but we replace *a_users* and *roles* with *a_users** and *roles** respectively.

5.2.3.6 User States Diagram

The user state diagram in this method is similar to its counter part in Method 1.

5.2.3.7 Discussion

Using this method has the following advantages:

a. If no local constraints are to be specified, this method is much less verbose compared to the other two methods discussed earlier, which makes it less error-prone.
b. It is more convenient from an administrative point of view since all applicable constraints can be gathered in one distinguished set.
c. Since constraints apply to all rules, there is little chance to overlook a rule that requires specifying a constraint. This makes the method less error-prone.

So, in general, this method is safer compared to Methods 1 and 2.

Nonetheless, there are some disadvantages of this method such as:

a. All constraints have to be evaluated every time a rule is invoked, irrespective of whether they apply to that rule or not. This renders the method less efficient than the previous two.
b. The method can become extremely verbose if we try to express constraints that apply to specific rules. As an example, assume we want to specify a constraint that applies to users who satisfy an attribute expression of a rule. In Method 1, we

will put the constraint in the rule after the reserved word "ST". In Method 3, we have to specify an invariant that incorporates the attributes expression of that rule and, consequently, applies to the users who satisfy that rule. This can easily become an administrative nightmare. Worse yet, this invariant ought to be checked every time a rule is invoked.

So, from implementation and administrative outlooks, this method is inferior to the previous two methods.

5.2.4 Discussion

We have used the three methods to express the three classes of constraints. This leads to the following theorem:

Theorem 8

The 3 methods are equivalent in expressing the 3 classes of constraints.

Proof: We have demonstrated that each of the 3 methods can express all types of constraints in the presence/absence of the GRH.

□

5.2.4.1 Using a Hybrid Method

The three methods discussed above are not mutually exclusive, but rather complementary. In fact, using the invariants method along with one of the first two methods gives a hybrid method with the following advantages:

a. Functionality: Methods 1 or 2 provide fine granularity constraints while Method 3 offers constraints of global applicability.

b. Implementation: Evaluating the constraints becomes more efficient since constraints pertaining to specific rules will be checked only if the relevant rules are invoked.

c. Security: A prudent design of the mix makes it more secure than merely using Method 1 or 2. This is so because the invariants could be specified in such a way to close any loopholes that might be overlooked when specifying local constraints, i.e. via the first two methods.

5.2.4.2 Conflict among Constraints in a Hybrid Method

If a hybrid method is used, conflict among invariants and local constraints may arise. For example, we might have the following dynamic cardinality constraint specified using Method 1 concerning role r_g and r_h:

$$ae_i \Rightarrow r_g \quad ST \ | a\_users\ (r_g)| \leq 20$$

$$ae_j \Rightarrow r_h \quad ST \ | a\_users\ (r_h)| \leq 290$$

And, at the same time, we have the following invariant:

$$\Rightarrow ST \quad |a\_users\ (r_g)| \leq Dyn\_card(r_g)$$

$$\Rightarrow ST \quad |a\_users\ (r_h)| \leq Dyn\_card(r_h)$$

where $Dyn\_card(r_g)= 13$ and $Dyn\_card(r_h) = 300$. To resolve this conflict, the system that implements RB-RBAC may enforce any of the following policies:

a. Invariant overrides: In case of conflict, an invariant overrides local constraints. So in the example above, at any given time, r_g and r_h will have a maximum of 13 and 300 active users respectively. Since invariants are heaped in one set and are expected to be fewer in number than local constraints, mistakes in specifying

them are less likely to happen than in specifying local constraints. Thus, one can reasonably argue that this policy could yield a more secure set of constraints.

b. Local overrides: This policy is the opposite of the one described above. Based on this policy, r_g and r_h can, respectively, have up to 20 and 290 concurrent active users. The virtue of this policy is that it allows special handling of users who have certain attributes if deemed appropriate by the enterprise. However, the major drawback is that this may introduce security loopholes because it is always easy for security officers to overlook or fail to foresee all possible ways of exploiting such special handling.

c. Denial takes precedence: To provide maximum security, this policy errs on the side of denial, and, therefore, calls for enforcing the more restrictive constraints. Accordingly, at any given time, r_g and r_h will not have more than 13 and 290 active users respectively

d. Permission takes precedence: This policy is the opposite of the previous one, and hence, it leans towards authorizing roles provided that the weakest constraints hold. As a result, r_g and r_h can, respectively, hold up to 20 and 300 active users concurrently.

5.2.5 Summary of Model C

Model C is equivalent to Model A with constraints which can be specified in three equivalent methods. We have analyzed each method, provided the formalization necessary to specify constrains, discussed the impact of constraints on the derivation of IRH, compared it to existing RBAC models (RBAC96 and OASIS), and detailed its pros and cons. New semantics were introduced pertaining to cardinality and prerequisite constraints. At the end of the chapter, we discussed a hybrid method, analyzed its value and the conflict it might introduce, identified conflict cases and provided conflict resolution policies.

5.3 Model B vs. Model C

Model B and C provide the means to specify some SOD constraints. Model B has the advantage of specifying negative authorization while Model C can specify two types of constraints in addition to SOD constraints.

Theorem 9

Model C subsumes Model B.

Proof:

First: Model C can express Model B

a. Model B_1 (Negative Authorization):

In Model B_1 we write $ae_k \Rightarrow \neg\ r_i$ which means that any user *u* that satisfies ae_k is prohibited from being authorized role r_i. To express this in Model C, we specify the following rule such that constraint c_1 never holds:

$$ae_k \Rightarrow r_i \ \text{ST} \ c_1$$

b. Model B_2 (Mutual Exclusion)

Model B_2 syntax allows specifying role-centric SOD constraints only, which is a subset of the constraints that Model C allows us to specify. However, of particular interest, Model B_2 can specify sets of mutually exclusive roles, which represent conflicting groups of roles such as:

$ae_k \Rightarrow \{r_2, r_3\} \oplus \{ r_4, r_5\}$

To represent this in Model C, we break the above mutually exclusive set of roles into the following sets of conflicting roles:

$cr_1 = \{ r_2, r_4\}, cr_2 = \{ r_2, r_5\}, cr_3 = \{ r_3, r_4\}, cr_4 = \{ r_3, r_5\}$

Then, any of the 3 methods of Model C can be used to specify SOD constraint among these conflicting sets.

Second: Model B cannot express some features of Model C

Model B cannot specify user-centric SODs, cardinality or prerequisite constraints. This ends the proof.

□

5.4 Summary

Model B and C extend Model A in different ways. Model B supports negative authorization and mutual exclusion, which can be used to specify certain types of SOD constrains. Model C allows the specification of three types of constraints including SODs. We have shown that Model C subsumes Model B. Figure 37 shows how we view the syntactic and semantic relations among the members of RB-RBAC family. Choosing the model to be implemented depends on the enterprise needs. For example, if only SOD constraints and/or negative authorization are to be enforced, an implementation of Model B does the job. However, if cardinality and/or prerequisite constraints are important for the business practice, then Model C should be used. Figures 37-39 capture the formalization of Model C.

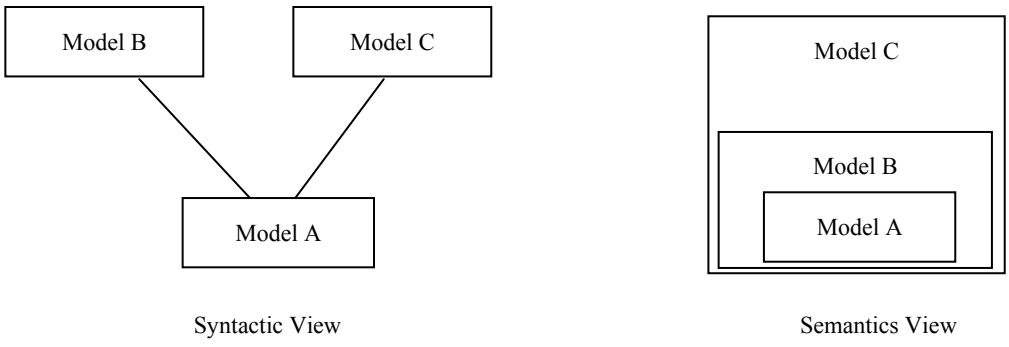

Figure 37: Syntactic and Semantic View of RB-RBAC Family

Model C:

1. Model A definition with the following modifications:
   - An authorization rule $rule_i$: $ae_i \Rightarrow$ RHS ST Constraints. The syntax is given in by ASL_{CI} language.
   - U_AE = {(u, ae_i)| (u, ae_i)$\in$U$\times$AE $\wedge$ u satisfies ae_i $\wedge$ applicable constraints hold according to the policy enforced}
2. URAuth in PTP approach:
   a. $URAuth^{PTP}$= {(u,r)| ($\exists rule_i$)[(u, ae_i) $\in$U_AE $\wedge$ $r \in RHS(ae_i)$ $\wedge$ c_i is true for u}
      Call this $A \wedge \alpha$
   b. URAuth with *can_assume* = $(A \wedge \alpha) \vee B$
      $URAuth^{PTP \text{ with } can\_assume}$ = {(u,r)| (($\exists rule_i$)[(u, ae_i) $\in$U_AE $\wedge$ $r \in RHS(ae_i)$] $\wedge$ c_i is true for u)
      $\vee$ ($\exists rule_j$) [(u, ae_j)$\in$ U_AE $\wedge$ $r' \in RHS(ae_j)$ $\wedge$ *can_assume*(r', r, t, d) $\wedge$ *can_assume* has not expired]}

3. URAuth in DTP approach::
   a. $URAuth^{DTP}$ = {(u,r)| ($\exists rule_i$)[(u, ae_i) $\in$U_AE $\wedge$ $r \in RHS(ae_i)$]
      $\wedge$ ($\forall rule_i$)[(u, ae_i) $\in$U_AE] $\wedge$ $r \in RHS(ae_i) \rightarrow c_i$ is true for u]}
      Call this $A \wedge \chi$
   b. With *can_assume*: $(A \wedge \chi) \vee B$
      $URAuth^{DTP \text{ with } can\_assume}$= {$(u,r)$| (($\exists rule_i$)[($u$, ae_i) $\in$U_AE $\wedge$ $r \in RHS(ae_i)$]
      $\wedge$ ($\forall rule_i$)[(u, ae_i) $\in$U_AE] $\wedge$ $r \in RHS(ae_i) \rightarrow c_i$ is true for u])
      $\vee$ ($\exists rule_j$) [(u, ae_j)$\in$ U_AE $\wedge$ $r' \in RHS(ae_j)$ $\wedge$ *can_assume*(r', r, t, d) $\wedge$ *can_assume* has not expired]}
4. Function *d_roles* (u_i): U $\rightarrow 2^{IR}$ returns roles in which user u_i is dormant
   d_roles(u_i) = {r $\in$IR| ((u_i,*r*) $\in$ URD}
5. Current_user: Holds the user who satisfies the authorization rule which the system invokes so the role(s) in its RHS can be activated by the user.
6. Other_Conflicting_Users(u), for short, *OCU*(u): U $\rightarrow 2^{U}$, a function that returns a set that holds all users in conflict with u, = $\cup$ cu_s– {u} where cu_s is any conflicting user set such that $u \in cu_s$ and $cu_s \in$ CU.
7. Function *a_users* (r_g): IR $\rightarrow 2^{U}$ returns users active *wrt* role r_g
   a_users(r_g) = {u $\in$U| ((u,r_g) $\in$ URA}
8. Function *d_users* (r_g): IR $\rightarrow 2^{U}$ returns users dormant *wrt* role r_g
   d_users(r_g) = {u $\in$U| ((u,r_g) $\in$ URD}
9. *a_users**(r_g) = {u $\in$U| ($\exists r' \geq_{GRH} r_g$)[(u, r') $\in$ URA]} which returns the active users in role r_g and all roles senior to it in a GRH.
10. *d_users**(r_g) {u $\in$U| ($\exists r' \geq_{GRH} r_g$)[(u, r') $\in$ URD]} which returns the dormant users in role r_g and all roles senior to it in a GRH.
11. *roles**: S $\rightarrow$ 2R is modified from *roles* to require *roles**(s_i) = {r $\in$ IR | ($\exists r' \geq$ r)[r' $\in$ *role*(s_i)]} (which can change with time)

Figure 38: Model C / Part A

Model C:

Theorem

With respect to a specific role in Role-centric SOD, the following holds:

a. For any set of conflicting roles:
   Static constraint holds → Simple dynamic constraint holds → Session-based dynamic constraint holds
b. For two conflicting roles sets, cr_x and cr_y, with c_x and c_y being SOD constraints of the same type on cr_x and cr_y, respectively, the following holds:
   $(cr_x \subseteq cr_y) \rightarrow (c_y \text{ holds} \rightarrow c_x \text{ holds})$

Proof: Please see the chapter.

Theorem

With respect to a specific role in User-centric SOD, the following holds:

i. For any set of conflicting roles:
   Static holds → Dynamic holds
ii. For the 2 conflicting users sets, cu_x and cu_y with c_x and c_y being user-centric SOD constraints on cu_x and cu_y respectively, the following holds:
   $(cu_x \subseteq cu_y) \rightarrow (c_y \text{ holds} \rightarrow c_x \text{ holds})$

Proof: Please see the chapter.

Theorem

With respect to role *r*, the following holds among cardinality constraints:

a. Assume we have two constraints: c_x is specifying the static cardinality, while c_y is specifying the dynamic cardinality. If the static and dynamic cardinality values of *r* are equal, then c_x holds → c_y holds.
b. Assume we have two constraints: c_x and c_y are specifying a cardinality value *m* and *n* over of *r* where $n \geq m$, then, c_x holds → c_y holds.

Proof: Please see the chapter.

Theorem

With respect to role r_k, the following holds among prerequisite roles constraints:

a. Assume we have these two constraints both specify a single role as a prerequisite role for r_k: c_x specifies r_g, while c_y specifies r_h. If $r_g \geq_{GRH} r_h$, then c_x holds→ c_y holds.
b. Assume we have these two constraints both specify a set of roles as prerequisite roles for r_k: c_x specifies role-set$_g$, while c_y specifies role-set$_h$. If role-set$_g \subseteq$ role-set$_h$, then c_y holds → c_x holds.

Proof: Please see the chapter.

Theorem
Methods 1, 2, and 3 are equivalent in expressing SOD, Cardinality and Prerequisite constraints.
Proof: Please see the chapter.

Theorem
Model C subsumes Model B.
Proof: Please see the chapter.

Figure 39: Model C / Part B

Chapter 6: Configuring RB-RBAC for other Access Control Models

6.1 Introduction

Access control models have traditionally included mandatory access control (MAC) , also known as lattice-based access control (LBAC), and discretionary access control (DAC). Later, RBAC was introduced, along with claims that its mechanisms are general enough to simulate the traditional methods. Osborn et al. has shown how to configure RBAC96 to enforce MAC and DAC [OSM2000]. Since RB-RBAC modifies and extends RBAC96, it makes sense to ask if RB-RBAC retains this feature of RBAC96, i.e. provides what is required to enforce MAC and DAC. This is desirable since both models are widely implemented in the private and public sectors. This is what we will demonstrate in the following sections.

6.2 Configuring RB-RBAC for MAC

The primary concept of MAC is that the information should flow in one direction in a lattice of security labels. MAC can be configured to provide different security services, e.g. confidentiality, integrity, or confidentiality and integrity together, depending on the way MAC enforces the one directional flow of information [OSM2000].

MAC recognizes two types of components: objects and subjects, although subjects could be treated as objects in certain contexts. A security label, denoted by "λ", is attached to

every object or subject. The label of an object is called a security classification, while a label on a subject is called security clearance. A subject could be a process running on behalf of a user and it is possible for several subjects with different security clearances to run on behalf of the same user. For the purpose of this discussion, we assume that security labels, once assigned, cannot be changed.

Although MAC comes in several variations, the discussion below will be focused on MAC with Liberal *-Property, which uses one direction information flow to enforce confidentiality. We will first start with these definitions [San1993]:

Definition: (Simple Security Property)

Subject s can read object o only if $\lambda(s) \geq \lambda(o)$. This is also known as the no-read up property.

Definition: (Liberal *-Property)

Subject *s* can write object *o* only if $\lambda(s) \leq \lambda(o)$. This is also known as the write up property.

6.2.1 RB-RBAC Construction to simulate LBAC with Liberal *-Property

Consider the lattice in Figure 40. This lattice demonstrates a dual character because the subjects with labels higher up in the lattice have more power with respect to read operation but have less power with respect to write operation. In RBAC terms, each lattice label x is modeled as two roles xR and xW for read and write at label x respectively. Accordingly, two dual role hierarchies are needed. The hierarchy for the "read" roles has the same partial order as dominance relation ($\geq_{LBAC}$), while the hierarchy for the "write" roles has a partial order that is the inverse of dominances ($\geq_{LBAC}$). Since

each user in LBAC has a unique security clearance, we have to represent this in RB-RBAC, where there is no explicit assignment. To do this, we require that each authorization rule authorizes the user to mutually exclusive pairs of roles such that each pair contains two roles: xR and xW. A user's clearance is an attribute that he is associated with. This attribute, along with other attributes, is used to determine what authorization rules the user satisfies and, as a result, what roles he is authorized to assume. The user's clearance $\lambda(u)$ dominates the roles in the right hand side of the rule, i.e. $\lambda(u) \geq_{LBAC} \lambda(x)$. Since an LBAC user can login at any label dominated by the user's clearance, this requirement is captured in RB-RBAC by requiring that each session have exactly two matching roles yR and yW such that the user clearance dominates the session's level.

Regarding the operations on LBAC objects, LBAC is enforced via read and write operation on objects, each of which has a single sensitivity level. In RBAC96, we express this in terms of read and write permissions on individual objects denoted by (o,r) and (o,w) respectively such that each permissions pair ((o,r) and (o,w)) is assigned to exactly one matching pair of xR and xW roles respectively. This does not apply to RB-RBAC as we will discuss shortly.

In the following, we provide two constructions to prove that RB-RBAC can be configured to simulate MAC using Model B_2 and Model C. The proofs provided are valid for any lattice.

Construction 1: Simulating MAC using RB-RBAC model B_2.

Suppose we have a security lattice shown in Figure 40 with labels {H, M_1, M_2, L} and partial order $\geq_{LBAC}$. An equivalent RB-RBAC system using Model B_2 is given by:

a. The set of RB-RBAC roles, IR = {hR, m_1R, m_2R, lR, hW, m_1W, m_2W, lW}.

Where xR and xW are RB-RBAC read and write roles that correspond to LBAC security label x.

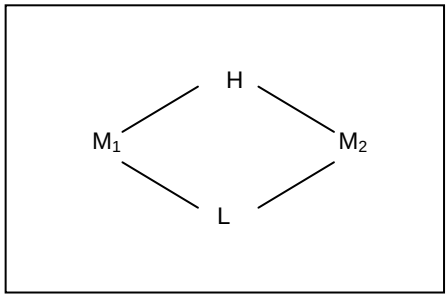

Figure 40: Security Lattice

b. The set of permissions, P = {(*o*,r), (*o*,w) | *o* is an object in the system, and r and w stand for read and write operations respectively}

c. Constraints on user authorization: Since no explicit user-role assignment is done in RB-RBAC, we require that each authorization rule authorizes the user to mutually exclusive pairs of roles such that each pair contains two roles: xR and xW. A user's clearance is an attribute that he is associated with. This attribute, along with other attributes, is used to determine what authorization rules the user satisfies and, as a result, what roles he is authorized to assume. The user's clearance $\lambda(u)$ dominates the roles in the right hand side of the rule, i.e. $\lambda(u) \geq_{LBAC} \lambda(x)$. The authorization rules set for this case is as follows:

- User_clearance = H $\Rightarrow$ Session Dynamic (hR, hW) $\oplus$ (m_1R, m_1W) $\oplus$ (m_2R, m_2W) $\oplus$ (lR, lW)
- User_ clearance = M_1 $\Rightarrow$ Session Dynamic (m_1R, m_1W) $\oplus$ (lR, lW)
- User_ clearance = M_2 $\Rightarrow$ Session Dynamic (m_2R, m_2W) $\oplus$ (lR, lW)
- User_ clearance = L $\Rightarrow$ (lR, lW)

d. Constraints on PA:

- Permission (o,r) allows us to read object "o". Since we do not have a role hierarchy for RB-RBAC roles, we explicitly assign permission (o,r) to every role yR such that y $\geq_{LBAC}$ $\lambda(o)$. So, for example, the permission (o_{m1}, r) allows us to read object o which has $\lambda(o) = M_1$. This permission is assigned to roles hR and m_1R.
- Similarly, permission (o,w) allows us to write object "o". Since no role hierarchy exists for RB-RBAC roles, we explicitly assign permission (o,r) to every role yW such that $\lambda(o) \geq_{LBAC}$ y. To continue with our example, the permission (o_{m1}, w) -which allows us to write object o which has $\lambda(o) = M_1$- is assigned to roles lW and m_1W.

Table 8 shows the permission-role assignment for our example.

Table 8: The permission-role assignment

Roles	Permissions	
	Read	Write
(hR, hW)	(o_h,r), (o_{m1},r), (o_{m2},r), (o_l,r)	(o_h,w)
(m_1R, m_1W)	(o_{m1},r), (o_l,r)	(o_h,w), (o_{m1},w)
(m_2R, m_2W)	(o_{m2},r), (o_1,r)	(o_h,w), (o_{m2},w)
(lR, lW)	(o_l,r)	(o_h,w), (o_{m1},w), (o_{m2},w), (o_l,w)

Theorem 10

An RB-RBAC system defined by Construction 1 satisfies the Simple Security Property and the Liberal *-Property.

Proof:

1. Simple Security Property: According to Construction 1:

We start by stating that subjects in the LBAC terminology correspond to RB-RBAC sessions. For subject *s* to read *o*, (*o*,r) must be in the permissions assigned to a role, which is among the roles available to session *s*, which corresponds to exactly one user *u*. For *u* to be involved in this session, this role must be in the right hand side of a rule that *u* satisfies. A user *u* satisfies the rules based on his clearance. The roles in the right hand side of a rule are dominated by the user's clearance. If *u* satisfies a rule that generates more than one pair, *u* is authorized to activate exactly one pair.

By the constraints on PA given in Construction 1, (*o*,r) is assigned directly to role xR, where x = λ (*o*), and to all roles yR such that $y \geq_{LBAC} x$. For *s* to be able to read *o*, it must have one of these yR in its session. By the definition of roles in an RB-RBAC session, any role junior to zR can be in a session for *u* where z = λ(*u*), i.e., $z \geq_{LBAC} y$ such that y= λ(*s*). This means that a session for *u* can involve one reading role yR such that $z \geq_{LBAC} y$. Consequently, the RB-RBAC system defined above allows subject *s* to read object *o* if $\lambda(u) \geq_{LBAC} \lambda(s)$ and $\lambda(s) \geq_{LBAC} \lambda(o)$. This is the Simple Security Property.

2. Liberal *-Property: According to Construction 1:

The authorization rules are such that if a user who is cleared to level x wishes to run a session at level y, such that $x \geq_{LBAC} y$, the session will have the two active roles yR and yW. If the yW role is available to a user in a session, the user can write

objects for which the permission (o,w) is in yW. Due to the explicit assignment of write permissions to roles, yW will also have the permission (o,w) which allow it to write to objects at its levels or higher. The write permissions are assigned to write roles such that each role in a session *s* can write to objects of clearance that dominates the session's clearance, i.e. $\lambda(o) \geq_{LBAC} \lambda(s)$. This is the Liberal *-Property.

This completes the proof.

□

Construction 2: Simulating MAC using RB-RBAC Model C_1.

Using the previous example, an equivalent RB-RBAC system using Model C_1 is given by:

a. The sets of RB-RBAC roles, permissions, and PA assignment are as in Construction 1.

b. Constraints on user authorization and sessions: We define a collection of disjoint sets of companion roles (CRS) = $\{crs_1,.., crs_n\}$ such that:

   $\forall\ crs_x \in CRS$, $crs_x = \{xR, xW\}$ such that for any two sets crs_i and crs_j, $crs_i \cap crs_j = \varnothing$ for any i and j both in [1,n] such that $i \neq j$. Set crs_x corresponds to label x and has 2 companions RB-RBAC roles: a read role xR and a write role xW. In our example, we have the following:

   $crs_h = \{hR, hW\}$

   $crs_{m1} = \{m_1R, m_1W\}$

   $crs_{m2} = \{ m_2R, m_2W\}$

   $crs_l = \{lR, lW\}$

   We restrict the sessions such that a user cannot activate more than one set of companion roles in a session. To express this, we use Method 1 described in Chapter 5 to write the following constraint on the authorization rules:

   $$c_1 = |roles(\text{Current\_session}) \cap OE(CRS)\ | = 0$$

   Based on the discussion above, we specify the authorization rules set for this case as follows:

   - User_ clearance = H $\Rightarrow \{crs_h, crs_{m1}, crs_{m2}, crs_l\}$ ST c_1

- User_ clearance = $M_1 \Rightarrow \{crs_{m1}, crs_l\}$ ST c_1
- User_ clearance = $M_2 \Rightarrow \{crs_{m2}, crs_l\}$ ST c_1
- User_ clearance = $L \Rightarrow \{crs_l\}$ ST c_1

The rules are specified such that the clearance of any session *s* is dominated by the user's clearance.

Theorem 11

An RB-RBAC system defined by Construction 2 satisfies the Simple Security Property and the Liberal *-Property.

Proof:

1. Simple Security Property: According to Construction 2:

For subject *s* to read *o*, (*o*,r) must be in the permissions assigned to a role, which are among the roles available to session *s*, which corresponds to exactly one user *u*. For *u* to be involved in this session, this role must be in the right hand side of a rule that *u* satisfies. Suppose that a user, say *u*, satisfies some rule by virtue of being associated with a specific user clearance. As a result, *u* is authorized to activate sets of companion roles in the right hand sides (side) of the rule. However, the constraint c_1 limits the user to activating exactly one set.

By the constraints on PA given in Construction 1, (*o*,r) is assigned directly to role xR, where $x = \lambda(o)$, and to all roles yR such that $y \geq_{LBAC} x$. For *s* to be able to read *o*, it must have one of these yR in its session. Remember that *u* can have only one read role at a session. By the definition of roles in an RB-RBAC session, any role junior to zR can be in a session for *u*, i.e., $z \geq_{LBAC} y$ where $z = \lambda(u)$ and $y = \lambda(s)$. This means that a session for *u* can involve one reading role yR such that z

$\geq_{LBAC}$ y. Consequently, the RB-RBAC system defined above allows subject *s* to read object *o* if $\lambda(u) \geq_{LBAC} \lambda(s)$ and $\lambda(s) \geq_{LBAC} \lambda(o)$. This is the Simple Security Property.

2. Liberal *-Property: According to Construction 2:

Similar to the proof provided for Construction 1 after taking into consideration the syntax of Model C_1 as we did in the first part of proving this theorem.

□

6.2.2 Discussion

We have shown that RB-RBAC can be configured to simulate LBAC. The way RB-RBAC is configured to achieve that differs from the way we configure RBAC96 in many aspects. In RBAC96 we require that each authorization rule authorizes the user to mutually exclusive pairs of roles such that each pair contains two roles: xR and xW. A user's clearance is an attribute that he is associated with. This attribute, along with other attributes, is used to determine what authorization rules the user satisfies and, as a result, what roles he is authorized to activate. The user's clearance $\lambda(u)$ dominates the roles in the right hand side of the rule, i.e. $\lambda(u) \geq_{LBAC} \lambda(x)$. We demonstrated how to express LBAC using Model B_2 and Model C_1.

6.3 Configuring RB-RBAC for DAC

DAC enforces a policy of owner-based administration of access rights. An owner of an object is usually, but not necessarily, the creator of the object. The owner discretionarily determines who else can access the object [OSM2000].

Similar to MAC, DAC comes in several variations, mostly regarding the delegation and revocation of access rights of objects to subjects [SM1998]. In all DAC policies discussed, the underlying assumption is that the creator of an object becomes its owner. The owner of an object is the only subject capable of destroying that object.

With respect to granting access to an object, we will simulate the following DAC variations:

a. Strict DAC: This variation mandates the existence of a unique owner for each object with absolute authority to grant access to the object.
b. Liberal DAC: The owner can delegate to other users the discretionary authority of granting access to an object. This delegation may apply to different levels resulting in several variations of Liberal DAC:
    i. One Level Grant: Delegation is limited to the first level recipients, i.e. a delegated user cannot delegate authority of granting access to an object to other users.
    ii. Two Level Grant: The first level recipients of delegated authority can further delegate it to a second level recipients, who, in turn, can delegate it to a third level recipients. Those cannot delegate it to others.

c. DAC with Change of Ownership: The owner of an object can transfer ownership to another user. This variation can be combined with strict or liberal DAC.

Although DAC allows grant-dependent and grant-independent revocation, we will confine the discussion to the latter type for the sake of brevity.

Since DAC is an owner-centric model, we will show how to simulate it using roles that are associated with each object. The main idea in configuring RB-RBAC to simulate DAC is that, with the creation of an object, we specify authorization rules which create administrative and regular roles such that users in the administrative roles have the permissions to grant/revoke attributes to/from users in regular roles. The regular roles are the ones that allow access to the object. The following discussion demonstrates how RB-RBAC can be configured to simulate DAC for object OBJ.

6.3.1 Strict DAC

With the creation of HP object, simulating strict DAC requires:

a. Roles: The creation of two roles:
   i. Administrative role, OWN_OBJ, which has a cardinality of one and is assumed by the owner of OBJ object. This role has the following permissions:
      a) addReadUser_OBJ: Used by the user in this role to give other users the attributes required to satisfy the rule that authorizes them to the regular role READ_OBJ. This user's attribute is OBJ_Owner_Approval.

b) deleteReadUser_OBJ: This permission allows the user of this role to take back the attribute above. Effectively, this means revoking the user in READ_OBJ role.

c) destroyObject_OBJ: Used to delete OBJ object from the system.

ii. Regular role, READ_OBJ, with single permission:

- canRead_OBJ: Authorizes read operation on OBJ object.

This yields two primitive disjoint roles hierarchies; the first one contains the administrative role OWN_OBJ while the other contains READ_OBJ.

b. Attributes names: We need two attributes of Boolean values:

- OBJ_Owner
- OBJ_Owner_Approval.

The owner of OBJ object, which assumes role OWN_OBJ has "OBJ_Owner" attribute. He can assign "OBJ_Owner_Approval" attribute to any user.

c. Authorization Rules: We need two authorization rules:

a) $rule_1$: (OBJ_Owner = True) $\Rightarrow$ OWN_OBJ

b) $rule_2$: (OBJ_Owner_Approval = True) $\Rightarrow$ READ_OBJ

Assume that u_i is the creator of OBJ object. By virtue of that, he has the attribute OBJ_Owner = true so according to $rule_1$ he is authorized to role OWN_OBJ. Once u_i activates the role, he can execute the roles permissions. To authorize user u_j to role READ_OBJ, u_i executes permission addReadUser_OBJ with respect to user u_j, thus u_j obtains attribute OWN_OBJ_Approval. As a result, u_j satisfies $rule_2$ and, subsequently, is authorized to role READ_OBJ. To revoke user u_j authorization to the role, u_i executes

permission deleteReadUser_OBJ with respect to u_j. This takes away the attribute OWN_OBJ_Approval from u_j.

6.3.2 Liberal DAC

6.3.2.1 Liberal DAC with One-Level Grant

The creation of OBJ object requires:

a. Roles: The creation of three roles:
    i. Administrative role, OWN_OBJ, which has a cardinality of one, and is assumed by the owner of OBJ object. This role has the following permissions:
        a) addParent_OBJ: Used by the user in this role to give other users the attributes required to satisfy the rule that authorizes them to PARENT_OBJ, an administrative role junior to OWN_OBJ. This attribute is OBJ_Owner_Apporval.
        b) deleteParent_OBJ: Used to revoke the attribute above.
        c) destroyObject_OBJ: As in static DAC.
    ii. Administrative role, PARENT_OBJ: The user who is authorized to this role has the power to authorize other users to READ_OBJ role. PARENT_OBJ role has the following permissions:
        a. addReadUser_OBJ: Used to give other users the attributes required to satisfy the rule that authorizes them to READ_OBJ role. This attribute is OBJ_Parent_Apporal.
        b. deleteReadUser_OBJ: Used to revoke from other users the approval to use READ_OBJ role.

iii. Regular role, READ_OBJ as in strict DAC

b. Attributes names: Beside the two attributes in strict DAC, we need one more attribute: OBJ_Parent_Approval. We consider attribute OBJ_Owner senior to attribute OBJ_Owner_Approval.

c. Authorization Rules: Three authorization rules are required:

a) $rule_1$: (OBJ_Owner = True) $\Rightarrow$ OWN_OBJ

b) $rule_2$: (OBJ_Owner_Approval = True) $\Rightarrow$ PARENT_OBJ

c) $rule_3$: (OBJ_Parent_Approval = True) $\Rightarrow$ READ_OBJ

Note that $rule_1 \rightarrow rule_2$ because OBJ_Owner ≥ OBJ_Owner_Approval. This produces two disjoint role hierarchies shown in Figure 41. Since u_i is the creator of OBJ object, he can assume the role OWN_OBJ, and thus, he can use permission addParent_OBJ to issue attribute "OBJ_Owner_Approval" to other users, say u_j. Consequently, u_j is authorized to PARENT _OBJ role and as such, can execute permission addReadUser_OBJ with respect to user u_k to authorize the latter to role READ_OBJ by granting him attribute "OBJ_Parent_Approval". Revocation works in a way similar to the method described in strict DAC with the additional role taken into consideration.

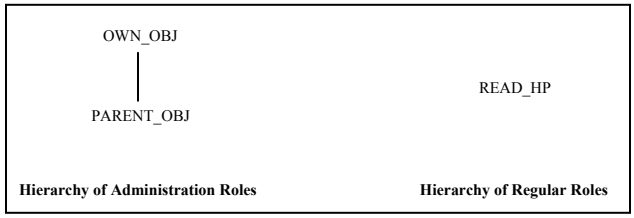

Figure 41: Role Hierarchies for One-Level Grant Liberal DAC

6.3.2.2 Liberal DAC with Two-Level Grant

The creation of OBJ object requires:

a. Roles: In addition to the above three roles, we create PARENTwithGRANT_OBJ as an intermediate role between OWN_OBJ and PARENT_OBJ. Similar to what we did in the One-Level DAC, we assign permissions to OWN_OBJ and PARENTwithGRANT_OBJ roles so that each role can grant/revoke access to its junior.

b. Attributes names: We need an extra attribute: OBJ_ParentWithGrant_Approval. The attributes have the following seniority:

OBJ_Owner ≥ OBJ_ParentWithGrant_Approval ≥ OBJ_Owner_Approval.

c. Authorization Rules: Four authorization rules are required:

a) $rule_1$: (OBJ_Owner = True) ⇒ OWN_OBJ

b) $rule_2$: (OBJ_Owner_Approval = True) ⇒ PARENTwithGRANT_OBJ

c) $rule_3$: (OBJ_ParentWithGrant_Approval = True) ⇒ PARENT_OBJ

d) $rule_4$: (OBJ_Parent_Approval = True) ⇒ READ_OBJ

Note that $rule_1 \geq rule_2 \geq rule_3$. The relations among authorization rules give us the hierarchies shown in Figure 42. To show the usage of the administrative roles hierarchy, assume user *u* has attribute "OBJ_Owner_Approval" so *u* satisfies $rule_2$ and is authorized to PARENTwithGRANT_OBJ role. If *u* activates this role, he is capable of granting/revoking attribute "OBJ_ParentWithGrant_Approval", i.e. effectively granting/revoking membership in role PARENT_OBJ. If *u* wants to grant/revoke membership of READ_OBJ, and since $rule_2 \geq rule_3$, he can activate role PARENT_OBJ. Alternatively, *u* as a member of PARENTwithGRANT_OBJ role, can activate any role junior to its current role, which allows *u* to execute the permissions of PARENT_OBJ.

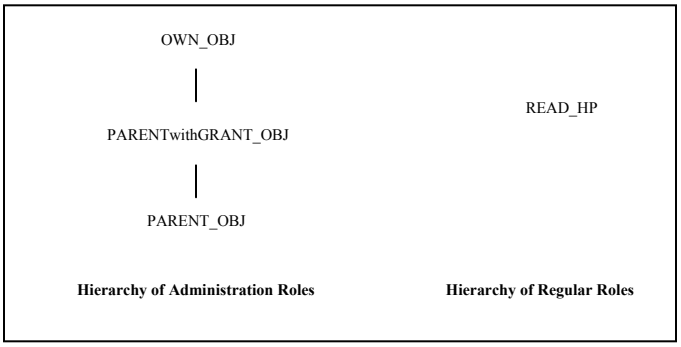

Figure 42: Role Hierarchies for Two-Level Grant Liberal DAC

6.3.3 Change in Ownership

Some DAC variations allow transfer of ownership. Role OWN_OBJ is assigned a new permission to allow its user to transfer ownership, call this permission add_Owner_OBJ. The current owner, say *u*, of the object can execute this new permission to give special attribute, say Legitimate_Owner, to another user, say *v*. RB-RBAC supports two variations of change of ownership:

a. Temporary: The OWN_OBJ role is assigned a dynamic cardinality value of one. Since *v* satisfies the rule below, he is authorized to role OWN_OBJ. If *u* is not active *wrt* that role, *v* can activate it and, as a result, *u* loses his membership to role OWN_OBJ, which effectively transfers ownership to *v*. The authorization rule $rule_1$ is modified as follows:

$$rule_1\text{: (OBJ\_Owner = True} \vee \text{Legitimate \_Owner = True)} \Rightarrow \text{OWN\_OBJ}$$

If *v* becomes dormant, *u* can activate role OWN_OBJ and resume ownership.

b. Permanent: To achieve this, static cardinality is used instead of dynamic cardinality along with attribute Legitimate_Owner. The authorization rule above is modified as follows:

$rule_1$: Legitimate_Owner = True $\Rightarrow$ OWN_OBJ

Initially, *u* has OBJ_Owner = True and so he can activate role OWN-OBJ and grant attribute Legitimate_Owner to another user *v*. Then we modify the authorization rules as discussed above. Now, *v* satisfies rule1 and thus becomes the owner.

6.3.4 Multiple Ownership

We can use the same procedure applied in ownership change to apply multiple ownership. The only modification we make is to remove the cardinality constraint on role OWN_OBJ.

6.3.5 Discussion

We demonstrated how to configure RB-RBAC to simulate DAC. However, we must admit that the number of roles (both administrative and regular) in the system could grow rapidly since each of the objects created requires generating several roles.

6.4 Summary

We have shown that RB-RBAC can be configured to express MAC and several variations of DAC. This is in harmony with the nature of RBAC models, which are policy-neutral.

Chapter 7: RB-RBAC Administration

7.1 Introduction

The administration of systems that implement conventional RBAC is extensively discussed in the literature; see for example [Ker2002], [KKSM2002], [KKSM2002], [SB1999], [SBM1999] and [OSZ2003]. However, administering systems that implement RB-RBAC requires taking into consideration that administrators do not directly assign users to roles. Instead, this process is automated based on authorization rules which authorize users to roles according to the attributes with which they are associated. The issue now becomes how to determine who has the power to administer these attributes and on what basis this power should be distributed if a decentralized approach is sought. From a user perspective, this means that changes made to the attributes of a user, say u, may change the set of rules relevant to u by causing u to satisfy new rules. By the same token, these changes could cause him to fail to satisfy some rules that used to be relevant, hence, revoke him from roles to which he used to be authorized.

Similarly, authorized individuals need to modify authorization rules to reflect changes in the security policy or business practices of the enterprise. Changes made to authorization rules have results similar to the ones caused by changes to users' attributes but on a wider scope. The changes made to rules affect all users who satisfy the set of rules that was subject to the change.

On another front, there are some real world situations that require authorizing certain users to specific roles, which they are not authorized to activate under the current policy. This can be done by changing the authorization rules or users' attributes in order to authorize certain types of users to these roles. This is not always the most prudent course of action as will be discussed later. Instead, we introduced a new concept, *can_assume* relation, which permits attaining the same goal in a controlled manner without the need to make changes to the attributes or the rules.

The model should also allow users to delegate their roles to other users in a controlled way. This concept has been discussed in the context of RBAC, see for example [Barka2001] and [ZAC2003], however, we have modified it to fit RB-RBAC.

In this chapter, we introduce the RB-RBAC administrative model, which we call ARB-RBAC. The model specification includes administering the following:

a. Users' attributes,
b. Authorization rules,
c. *can_assume* relation, and
d. *can_delegate* relation.

In all of the above, we will take a decentralized approach to RB-RBAC administration. The justification for this includes:

a. As has been stated in Chapter 1, we assume that RB-RBAC will be used to provide access control to enterprises with a huge customer base which requires a decentralized administration of their attributes [KSM2003].

b. The new Internet landscape witnesses increasing numbers of enterprises that provide their services to highly diversified customer bases which are geographically scattered, another reason for decentralizing the administration.

We assume that administrators are explicitly assigned to administrative roles, which is a reasonable assumption since the numbers of users who are assigned to these roles are usually low.

7.2 Administering Users' Attributes

Administering users' attributes determines what roles they are authorized to activate. Changes made to a user's attributes affect the set of rules relevant to that user. The new set of relevant rules may:

- Authorize him to activate new roles,
- Revoke him from roles that he used to be authorized to activate,
- Give/take away from him negative authorization,
- Authorize/revoke him to/from mutually exclusive roles.
- Subject him to new constraints.

In this section we introduce several methods of administering users' attributes. The fundamental issue is how the attributes are divided into sets that are independently administered.

7.2.1 Type-centric Administration

The set of attributes of all users is divided into smaller subsets according to attribute types, e.g. personal info, academic, financial, health, etc. The authority of administering them is assigned to: the human resources, trusted academic institutions, banks, and

HMOs, in that order. In brief, each attribute is administered by the entity specialized in that type of attribute. The advantages of this approach are:

a. This type of administration has been actually implemented in the context of conventional RBAC [KSM2003].
b. It authorizes the specialists to administer the type of data with which they are familiar.
c. It fits the natural workflow in organizations.

7.2.2 Organization-centric Administration

The set of attributes of all users is divided into smaller subsets according to the organization units to which users belong. For example, in a manufacturing company, the manufacturing department administers the attributes of its employees; the marketing department does the same with respect to its employees, and so on. For a bank, each branch administers the attributes of its employees and clients. This approach has been implemented in the context of conventional RBAC [KSM2003]. One disadvantage of this method is the complex effect of transferring employees among different departments.

7.2.3 Location-centric Administration

The assumption that users' attributes should be stored under RB-RBAC control is not practical in many circumstances. Instead they may be stored in secure databases that are independently administered. RB-RBAC can retrieve with high assurance the users' attributes it needs to make authorization decisions. The set of attributes of all users is divided into subsets according to the data repository in which they reside. As such, the

authority to administer a set of attributes in a participating database is delegated to individuals in charge of that database.

7.2.4 Security-label Administration

We assign security labels to the attributes used in the system and then divide the attributes into groups according to their security labels. Individuals with appropriate clearance administer the attributes with labels that correspond to the individuals' clearance. The downside of this method is that changes in attributes' labels cause transferring the reasonability of administering these attributes to other administrators.

7.2.5 Role-centric Administration

The concept of role hierarchy is utilized to divide up the attributes into sets, each of which is assigned to an authorized administrator. The sets are not necessarily mutually disjoint. In the next section, we present a thorough analysis of this method. This method has the following advantages:

a. Formal models have been fully developed to support this kind of administration in the context of conventional RBAC. Examples for these models are ARBAC97 and ARBAC02 which are formally specified in [SB1999], [SBM1999] and [OSZ2003].
b. Practical examples to show how to use some of these models have been presented for database management systems, URA97 as discussed in [SB1999], and for a legacy access control system, ERBAC as presented in [M2002].

c. This method is consistent with the nature of RBAC because it is based on using RBAC to administer RBAC.

For these reasons, our attention will be focused on this method although similar analysis could be developed for the rest of the methods.

7.3 Role-centric Administration

7.3.1 Introduction

Among the models suggested to administer RBAC, two are of interest to us: ARBAC97 and its successor ARBAC02. Although a powerful model, ARBAC97 introduces unnecessary coupling between user pools and prerequisite roles, which lead to some undesirable consequences. ARBAC02 presented by OH et al. recognizes the need to remove this coupling by making the pool of users include organizational structure besides the roles [OSZ2003]. The discussion presented here is focused on the parts of these two models that deal with user-assignment. Both ARBAC97 and ARBAC02 models assume that there is a System Security Officer (SSO) who has supreme authority over user-role assignment. However, to decentralize that task, the SSO delegates some authority to one or more junior security officer(s) (JSO) which allows them to assign or revoke users within some designated role range. For this purpose, both models have two sub-models, one for assigning users and the other for revoking them:

a. A grant model: In their respective grant models, both ARBAC97 and ARBAC02 impose restrictions on which of the users can be added to a role by whom. Both models control user-role assignment by means of the relation

   $$can\_assign \subseteq AR \times CR \times 2^{R}$$

   AR, CR, and 2^{R} are the set of administrative roles, the set of prerequisite conditions, and the superset of regular roles, in that order. The meaning of *can_assign* (x, c, role-range) is that a member of the administrative role x (or a member of an administrative role that is senior to x) can assign a user who

satisfies the prerequisite condition c to be a member of regular roles in the role-range. They differ however, in defining the prerequisite condition.

b. A revoke model: This is built on the relation

$$can\_revoke \subseteq AR \times 2^{R}$$

This relation gives the JSO the authority to revoke users who are assigned to roles that fall in his role range.

7.3.2 How RB-RBAC is different

In RB-RBAC, no explicit assignment is performed, but instead, the assignment is rule-based such that the rules are attributes-driven. Obviously, this makes the question of who administers the attributes central. This is what semantically sets ARB-RBAC apart from ARBAC97 and ARBAC02. The difference boils down to the following:

a. The explicit assignment does not generate any side effects other than the inheritance of roles, which is made possible by the role hierarchy. Also, revoking a user from a role is done explicitly. The decision to assign/revoke a user to/from a role is done by humans based on some input. The SSO or JSO role is to execute the actual assignment/revocation. This makes it easy to specify the administrative model in order to impose proper restrictions on the SSO and JSOs.

b. In RB-RBAC, the assignment/revocation is done automatically based on changes that users' attributes witness. These changes could be done by individuals who may not be aware of the impact of the changes they make on users' authorizations. ARB-RBAC is specified such that the authorized individuals are able to make the proper changes provided that the impact of these changes does not go past their designated areas of responsibility. This area of responsibility is

defined using the concept of role ranges adopted from [SB1999]. Changes that go beyond JSO designated areas of responsibility have to be made by a higher security officer whose role range encompasses the desired attributes changes.

So, instead of imposing restrictions on a delegated JSO using role hierarchy or user pools as in the two models above, ARB-RBAC uses a mixture of users' attributes and prerequisite roles. This mixture constitutes a prerequisite condition that allows the SSO to confine the JSO to a designated set of users' attributes that the JSO is authorized to administer.

7.3.3 ARB-RBAC X Model

An SSO can use *can_administer_attributes* relation to delegate to a JSO the authority of administering users' attributes. The relation is as follows:

$$can\_administer\_attributes \subseteq AR \times 2^{Att} \times CR \times 2^{R}$$

The semantic of the relation *can_administer_attributes* (x, y, c, *rr*) is that a member of administrative role x (or a member of an administrative role that is senior to x) is authorized to administer a user's attributes such that:

a. The modified user's attributes $\subseteq$ attribute set y specified in the relation.

b. The user must be a member/not a member of the role specified in the prerequisite condition c.

c. $\forall ae_i$ such that ae_i is satisfied by the resulting attributes the following holds:

- The roles the user becomes authorized to because of the changes in the attributes must be within the role range *rr*. The concept of role range is adopted from [SB1999].

- The changes that the JSO makes have no side effects that may cause a change in the user's state *wrt* roles outside the role range *rr*.

Using the above relation, the SSO may delegate the authority of administrating users' attributes to one or more JSO, each of whom is given the authority to administer the attributes of users who are authorized to roles that fall in that JSO role range. Note that when a JSO changes a user's attributes this may result in authorizing the user to more roles, which means promoting him within the allowed role range. Conversely, the changes could revoke a previously given authorization over some roles in the range which results in demoting the user within the range. Also, notice that the specification allows a JSO to change multiple attributes in a single atomic operation.

Definition 16

1. The following is imported from RBAC96: AR and ARH, which are the set of administrative roles and the administrative role hierarchy. We assume that user-role assignment *wrt* administrative roles is explicitly performed.
2. The notion of role range, *rr* for short, is imported from ARBAC97.
3. 2^{Att} is the power set of all possible attributes.
4. A prerequisite condition is a Boolean expression using the usual $\wedge$ and $\vee$ operators on terms of the form x and x′ where x is a regular role (i.e., $x \in R$). A prerequisite condition is evaluated for a user *u* by interpreting *x* to be true if:
   - $x \in R: (\exists x' \geq_{GRH} x) . (u, x') \in URAuth$

   and x′ to be true if:
   - $x \in R: (\forall x' \geq_{GRH} x) . (u, x') \notin URAuth$
5. CR is the set of all possible prerequisite conditions

6. $r \in rr$ as defined in ARBAC97. This means that the roles that represent the end point of the range may or may not be within the range.
7. AuthorizationOutsideRoleRange or $AORR(u,rr) = \{(u,r)| (u, r) \in \text{URAuth} \wedge r \notin rr\}$. *AORR* returns the sets of roles outside *rr* to which *u* is authorized to activate.
8. Administrating an attribute means the following:
    a. Adding a new attribute to the user's attributes
    b. Modifying the values of an existing attribute
    c. Deleting an attribute
9. ARB-RBAC X model authorizes administering users' attributes via the relation

$$can\_administer\_attributes \subseteq \text{AR} \times 2^{\text{Att}} \times \text{CR} \times 2^{\text{R}}$$

The semantic of the relation *can_administer_attributes* (x, y, c, *rr*) is that a member of administrative role x (or a member of an administrative role that is senior to x) can administer the attributes of a user *u* provided that:

a. The modified *u* 's attributes $\subseteq$ y,
b. *u* satisfies prerequisite condition c, and
c. $\forall ae_i$ that is satisfied by the resulting attributes the following holds:

$$RHS(ae_i) \in rr \wedge AORR(u,rr) = AORR'(u,rr).$$

AORR (*u*,*rr*) and *AORR* ′(*u*,*rr*) are the sets of roles outside *rr* to which *u* is authorized to activate before and after the changes made by the JSO, respectively. This specification is required to ensure that the changes that a security officer makes may not cause a change in the user's authorization *wrt* roles outside the designated role range.

7.3.3.1 Example for X Model

Suppose we have the authorization rules shown in Table 9. To illustrate how to read the table, take the fourth entry, which is read as:

(Has id = T $\wedge$ Deg= Eng $\wedge$ Proj=1 $\wedge$ Specialty=Prod) forms attribute expression ae_4, which constitutes the LHS of the following rule: $ae_4 \Rightarrow$ PE1 where PE1 is a role in the role hierarchy given in Figure 43, extracted from [SBM1999] .

Table 9: Example to Show How ARB-RBAC Works

	Attributes				Corresponding ae	Authorization Rules
Entry no.	Has id	Degree	Project	Specialty		
1	Y	-	-	-	ae_1	$ae_1 \Rightarrow$ E
2	Y	Eng	-	-	ae_2	$ae_2 \Rightarrow$ ED
3	Y	Eng	1	-	ae_3	$ae_3 \Rightarrow$ E1
4	Y	Eng	1	Production	ae_4	$ae_4 \Rightarrow$ PE1
5	Y	Eng	1	Quality	ae_5	$ae_5 \Rightarrow$ QE1
6	Y	Eng	1	All	ae_6	$ae_6 \Rightarrow$ PL1

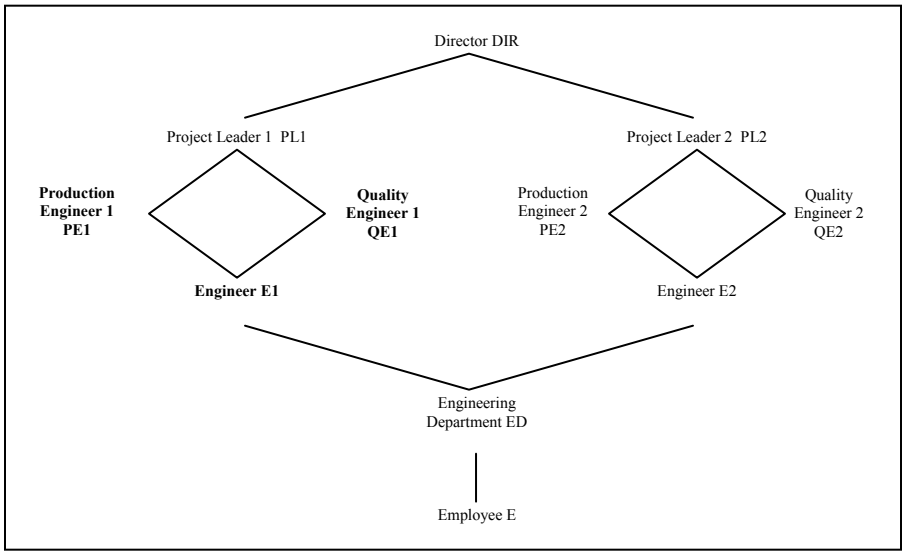

Figure 43: Example Hierarchy

We want the JSO who is in the administrative role PSO1, to be able to administer the attributes of the users whose attributes initially make them authorized to the roles in the role range [E1,PL1) which are shown in boldface in the figure. The *can_administer_attributes* relation is represented in Table 10.

Table 10: *can_administer* Relation

	Elements of *can_administer_attributes* relation			
	Admin. Role:x	Attribute set: y	Prerequisite role: c	Role range:*rr*
1	PSO1	{project, specialty}	-	[E1,E1]
2	PSO1	{project, specialty}	QE1′	[PE1,PE1]
3	PSO1	{project, specialty}	PE1′	[QE1,QE1]

So suppose the PSO1 modified *u*'s attributes as in the following scenarios:

a. Attributes set before: {had id=y, Degree= Eng}

   Attributes set after : {had id=y, Degree= Eng, project =1}

   Based on the relation in entry 1 of Table 10, and since user *u* attributes satisfies ae_3, *u* will be authorized to E1∈[E1,E1] which is in the designated role range. Neither entry 2 nor 3 can be used since E1 ∉ *rr*.

b. Attributes set before: {had id=y, Degree= Eng}

   Attributes set after : {had id=y, Degree= Eng, *project =1, Specialty= Quality*}

   Based on the relation in entry 3 of Table 10, and since user *u* attributes satisfies ae_5, and PE1′= True holds, *u* will be authorized to QE1∈[QE1,QE1] which is in the designated role range.

c. Attributes set before: {had id=y, Degree= Eng, project =1, Specialty= Prod.}

   Attributes set after : {had id=y, Degree= Eng, project =1, *Specialty*= -}

   Based on the relation in entry 1 of Table 10, since user *u* attributes satisfies ae_3, *u* will be authorized to E1$\in$[E1,E1]. In this case, the user is being revoked from PE1.

d. Attributes set before: {had id=y, Degree= Eng, project =1, Specialty= Prod.}

   Attributes set after : {had id=y, Degree= Eng, *project* = -, *Specialty*= -}

   The changes are rejected because the resulting attributes satisfies ae_2 which authorizes *u* ED $\notin$[E1,PL1). To this, the JSO must use Y model.

In all above scenarios, we assume that *AORR* (*u*,*rr*)= *AORR ′*(*u*,*rr*).

7.3.4 ARB-RBAC Y Model

The Y model authorizes a JSO to change the user's attributes such that he is no longer authorized to any role in the range designated to that JSO. Since there is no prerequisite condition, the specification must prevent the JSO from using Y model to illegally authorize a user to roles junior to the role range. To illustrate, assume *u* is authorized to role E in Figure 43. We want to prevent the JSO from changing the attributes of *u* so that *u* becomes authorized to ED. However, the JSO should be capable of modifying the attributes such that user *v* - who is authorized to a role within the role range designated to the JSO - is revoked from all his roles in the range. This is achieved via *can_revoke* relation defined below.

Definition 17

10. ARB-RBAC Y model authorizes administering users' attributes via the relation

$$can\_revoke \subseteq \mathrm{AR} \times 2^{\mathrm{Att}} \times 2^{\mathrm{R}}$$

The semantic of *can_revoke* (x, y, *rr*) is that a member of administrative role x (or a member of an administrative role that is senior to x) can administer the attributes set of a user *u* if the following holds:

1. $\alpha \subseteq$ y, where α is *u* 's attributes before the modification such that α satisfies $ae_i \wedge RHS(ae_i) \in rr$.
2. If β is the set of resulting attributes, then $\forall ae_j$ that is satisfied by the β, the following holds:

$$\beta \text{ satisfies } ae_j \rightarrow RHS(ae_j) \notin rr$$

3. $AORR(u,rr) = AORR'(u,rr)$

Part (a) ensures that the user affected is within the users in the designated role range. Parts (b) and (c) together guarantee that the JSO can revoke users from all the roles within the designated role range but cannot make changes that affect user's authorization *wrt* roles outside the role range designated to that JSO.

7.3.4.1 Example for Y Model

Consider the *can_revoke* relation in Table 11, it permits a user in the administrative role PSO1 to make modification to users attributes that result in revoking users from roles in the range [E1,PL1).

Table 11: *can_revoke* Relation

	Elements of *can_revoke* relation		
	Admin. Role:x	Attribute set: y	Role range:*rr*
1	PSO1	{project, specialty}	[E1,PL1)

Suppose the attributes set before PSO1 action was: {had id=y, Degree= Eng, project =1, Specialty= Prod.}. After the action, it became: {had id=y, Degree= Eng, *project* = -, *Specialty*= -}. This is actually case (d) in the example given for the X model. The changes are accepted because the resulting attributes set satisfies ae_2 which authorizes *u* to role ED to which he is already authorized.

7.4 Administering Authorization Rules

7.4.1 Introduction

Authorization rules are the formal expression of the security policy, which is supposed to govern the business practices of the enterprise. Both the security policy and business practices of the enterprise are subject to changes due to organizational restructuring, the introduction of new business practices or new technologies, etc. The authorization rules must be modified to reflect these changes in the policy or practices. These changes are made through administering the rules, which include:

- Adding new rules
- Modifying existing rules via:
    - Modifying LHS and/or RHS of existing rules by adding/deleting attributes/roles, and changing the stated values required for attributes
    - Adding new constrains
    - Modifying/ Deleting existing constrains
- Deleting existing rules

Changes made to the authorization rules affect all users who satisfy the modified rules. The need to decentralize the administration of the rules is not as pressing as it is in the case of users' attributes. If the number of rules is small, they can be administered centrally. However, in situations where the number of attributes is large, and/or the ways of authorizing roles are very diverse, the number of rules could be very large. In such case, decentralization is appropriate, and hence, each delegated JSO will be authorized to administer a subset of the authorization rules set. organization-centric

method can be applied to group the rules into independently administered subsets where roles are mapped to organizational units such that each unit administers the rules that produce the roles that belong to it. A down side of this method is that it is not resilient to changes in role hierarchy due to organizational up-scaling, down-scaling, restructuring, etc, which might necessitate regrouping the rules.

Another alternative is the role-centric method, where the role hierarchy is divided into role ranges. Subsequently, all the rules whose RHS roles fall in the same role range are grouped in one set. The method is preferred since it facilitates dividing the authority of administering the rules in a way that is compatible with the distribution of power within the organization. Also, this method is resilient to changes in role hierarchy as long as the range is intact. As such, this method is discussed in more detail below.

7.4.2 Specification

Specifying the set of rules over which a JSO is allowed to operate involves specifying these activities:

a. Adding new rules. The roles at the RHS of the new rules must be in the specified role range.
b. Modifying existing rules which involves:
    i. Adding roles to the RHS such that the new roles must be in the specified role range.
    ii. Modifying the expression at the LHS or a rule.
c. Deleting rules.

Definition 18

11. RB-RBAC model authorizes administering users' attributes via the relation

$$\textit{can\_administer\_rule} \subseteq \mathrm{AR} \times 2^{\mathrm{R}}$$

The relation *can_administer_rule*(x, *rr*) authorizes a member of administrative role x (or a member of an administrative role that is senior to x) to administer a set of rules SR provided the following holds:

a. $(\forall ae_i)\ [((ae_i \Rightarrow r_g) \in SR) \rightarrow (r_g \in rr)]$ i.e. The rule set that the JSO works on yields roles that are within his designated role range.

b. The resulting set of rules SR′ yields roles that are within his designated role range i.e. $(\forall ae_j)\ [((ae_j \Rightarrow r_h) \in SR') \rightarrow (r_h \in rr)]$.

7.5 Administering *can_assume* relation

7.5.1 Introduction

In RBAC literature there is some discussion regarding role delegation, see for example [BS2000a], [BS2000b], and [ZAC2003]. One way of doing this is detailed in [Bar2001] where delegation is enabled by the SSO adding the following relation to the RBAC model:

can_delegate (r_g, r_h)

This means that user *u* who is assigned to role r_g is allowed to delegate his membership in r_g to user *v* who is assigned to role r_h. However, *v* is not assigned to the role, and thus cannot activate the role, until *u* actively delegates his membership in r_g to *v*. So, merely adding a pair of roles to *can_delegate* relation does not guarantee that the potential recipient of the delegation will actually be able to assume the delegated role. Instead, the actual delegation is left to the discretion of the delegating party. This notion of delegation might be beneficial in certain circumstances but, in a sense, it is not in compliance with the general theme of RBAC where actions are not left to the discretion of individual users.

We suggest that authorized individuals such as the SSO should be given the power to authorized users who meet certain criteria, specified by the security policy, to roles. This explicit authorization allows the enterprise to respond to situations where it might not be desirable to allow this authorization via modifying the authorization rules. This is achieved via *can_assume* relation, which represents a novel concept in RBAC world. In a sense, this is a departure from the monotheistic approach of RB-RBAC where user-role

assignment is totally implicit, i.e. no human intervention allowed, however it is a justifiable one, as we will argue below.

7.5.2 Motivation

To motivate the concept of *can_assume* relation, consider the following two examples:

7.5.2.1 Company C

Company C wants to promote certain merchandise or services by providing its clients with a limited-time offer. It gives the clients who have been trading with the enterprise for two or more years and have purchased at least $2000 worth of goods all of the benefits provided to the clients who have been trading with the enterprise for three or more years and have purchased at least $2500 worth of goods. This change in the marketing policy is planned to last for two weeks. The change must be reflected in the security policy. From an RB-RBAC standpoint, the benefits that both sets of clients are authorized to are roles. The current security policy includes the following two rules:

$$rule_i: (\text{trading period} \geq 2) \wedge (\text{purchased} \geq \$2000) \Rightarrow \text{Silver\_client}$$

$$rule_j: (\text{trading period} \geq 3) \wedge (\text{purchased} \geq \$2500) \Rightarrow \text{Golden\_client}$$

There is more than one way to modify the security policy to allow this limited-time offer:

i. Modify the current attributes of clients of the first group so they can satisfy the $rule_j$.

ii. Modify the authorization rules such that $rule_i$ yields role Golden_client.

In both cases, this change in the security policy is temporary and after the 2-week period, the attributes or the rules, depending on the way used, should be modified back to what they were before the promotional period. These two solutions work, but with disadvantages. Both require two-step human intervention: one to modify the security

policy to reflect the required changes, and a second step to change the policy back to its initial state. Overlooking step two is always possible. Also, modifying clients attributes may make them authorized to roles other than ones intended, Golden_client in this case, and may result in a breach of the security policy. On the other hand, modifying the authorization rules blurs the distinction between genuine members and non- genuine members of roles. In our example, the genuine members are those who are authorized to role Golden_client by virtue of satisfying $rule_j$ in the original policy, while the non-genuine members are those who are temporarily authorized to the role due to the temporary change in the policy. This distinction is important in some situations such as when the security policy requires being a genuine member of Golden_client a prerequisite for being authorized to another role, say Platinum_client.

7.5.2.2 Hospital H

Hospital H has a policy that permits residents who are in their first year of residency to work as interns. However, it forbids them from working in the emergency room, where only senior residents (i.e. those in their second or third year of residency) or fully-trained doctors work. This is embodied in the following authorization rules:

$rule_1$: No. of years in residency $\leq 1 \Rightarrow$ intern

$rule_2$: No. of years in residency $\leq 1 \Rightarrow \neg$ ER_doctor

Naturally, during the holiday season large numbers of the medical staff take their yearly vacation. However, this period of the year witnesses a surge in the number of people admitted to the emergency room. Clearly, more medical staff is needed to handle this surge in demand of medical care. One way to handle this is to change the hospital policy by deleting $rule_2$. Alternatively, if PTP policy is enforced, we may add another rule that

authorizes interns to work as ER physicians. Similar to the first example, this course of action is not preferred for analogous reasons.

A better solution is to use *can_assume* relation. To illustrate, consider the second example, the SSO can add the following:

can_assume(intern, ER_doctor, *t*, *d*)

This authorizes interns to activate the role ER_doctor, i.e. to work in the emergency rooms starting at time *t* for a period *d*.

Using *can_assume* relation has the following merits:

a. It is a neat solution and easy to map elegantly to managerial decisions.
b. It allows overriding of the system implementing RB-RBAC but in a controlled manner to meet the needs posed by circumstances like the two examples given above.
c. When *can_assume* relation expires, users' authorization automatically reverts back to its original state before *can_assume* was specified by the SSO.

7.5.3 Specification

To enrich the RB-RBAC administrative model, we allow two forms of *can_assume* that provide two forms of granularity which authorize specific users to an attribute expression or a role. This authorization is temporary in nature and the system implementing RB-RBAC should be able to distinguish it from the authorization obtained by satisfying authorization rules. This results in two types of authorization:

a. Genuine authorization: Assuming we have the rule: $rule_i \Rightarrow r_g$, original membership of r_g is acquired either via satisfying $rule_i$, or via satisfying a rule, $rule_k$, such that $rule_k \geq rule_i$.

b. Non-genuine authorization: This membership is acquired when a user is authorized to a role via *can_assume* relation.

Clearly, this affects the value of URAuth as we have discussed in Chapter 3.

7.5.3.1 Coarse-granularity Form

This form allows temporary authorization of an attribute expression, say ae_i, to all the users who satisfy another attribute expression, say ae_j. The security officers could use this to make wholesale authorization of roles to a specific set of users who meet certain criteria.

Definition 19

can_assume $\subseteq$ AE $\times$ AE $\times$ T $\times$ D

where AE, T, and D are attributes expressions set, time set, and duration set respectively. We leave specifying the units of time and duration to the implementation. The semantic of *can_assume*(ae_i, ae_j, *t*, *d*) is that any user *u* such that (*u*,ae_i) $\in$ U_AE is authorized to any role r_g such that $r_g \in$ *RHS*(ae_j) starting at time *t* for a period *d*.

Let's modify the hospital example given earlier. Suppose we have the following situation:

- Five roles exist:
  a. In-floor: r_1
  b. In-Clinic: r_2
  c. ER-doctor: r_3
  d. Attending-doctor: r_4

e. Consultant: r_5

- Authorization rules:

a. Let ae_1 be: number_of_years_in_residency ≤ 1. Assume that $ae_1 \Rightarrow \{ r_1, r_2\}$

b. Let ae_2 be: number_of_years_in_residency > 1. Assume that $ae_2 \Rightarrow \{ r_1, r_2 , r_3, r_4\}$

c. Let ae_3 be: (fellow = True $\vee$ number_of_years_in_residency > 2). Assume that $ae_3 \Rightarrow \{ r_3, r_4, r_5\}$

If we want to authorize those who spent less that 1 year in residency all the roles of those who spent more than 1 year in residency, *can_assume* comes in handy.

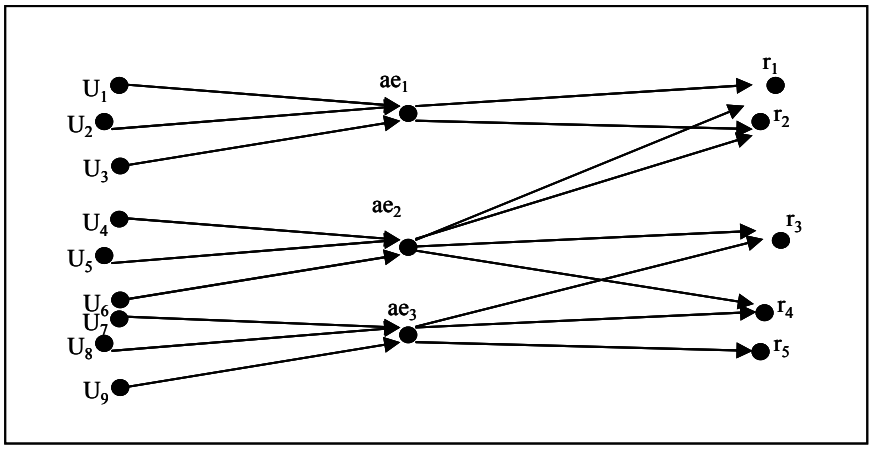

Figure 44: Users/Attribute Expressions/Roles Mapping

We specify *can_assume*(ae_1, ae_2, *t*, *d*). This results in authorizing users $\{u_1, u_2, u_3\}$ (Figure 44) to ae_2 and, consequently, causes them to become members of the following roles:

a. Genuine membership: $\{r_1 , r_2\}$

b. Non-genuine membership: $\{r_3, r_4\}$

7.5.3.2 Fine-granularity Form

This form of the relation gives the users, who are authorized to a specific role, a temporary authority over another role. This permits the security officer to make fine-granularity authorization when needed.

Definition 20

can_assume $\subseteq$ IR $\times$ IR $\times$ T $\times$ D

where IR, T, and D are roles set, time set, and duration set respectively

The semantics of *can_assume* (r_g, r_h, *t*, *d*) means that any user *u* such that (*u*, r_g) $\in$URAuth (i.e. *u* is authorized to r_g) is authorized to role r_h starting at time *t* for a period *d*. As a result, *u* has the following membership:

a. Genuine membership: {r_g}
b. Non-genuine membership: {r_h}

7.5.3.3 Fine-granularity Form with Cascade

Assume we have the following fine-granularity *can_assume* relations,

can_assume (Silver_client, Golden_client, t_1, d_1)

can_assume (Golden _client, Platinum_client, t_2, d_2)

Suppose that *u* and *v* are two users that are authorized to Silver_client and Golden_client respectively, i.e. (*u*, Silver_client) $\in$URAuth and (*v*, Golden_client) $\in$URAuth. Due to the relations above, *u* and *v* are non- genuine members of Golden _client and Platinum_client respectively. However, *u* is not authorized to Platinum_client because he is a non- genuine member of Golden _client. There are situations where it will be

desirable to cascade authorization where the authorization gained via a *can_assume* relation could be used as an input to another relation allowing further authorization. For this purpose, we introduce *can_assume_with_cascade* relation defined below.

Definition 21

can_assume_with_cascade $\subseteq$ IR $\times$ IR $\times$ T $\times$ D

where IR, T, and D are roles set, time set, and duration set respectively

The semeantics of *can_assume_with_cascade* (r_g, r_h, *t*, *d*) means that any user *u* such that (*u*, r_g) $\in$URAuth$^{\text{wit } can_assume}$ (Definition 7) is authorized to role r_h starting at time *t* for a period *d*. As a result, *u* has the following membership:

a. Genuine membership: {r_g}
b. Non-genuine membership: {r_h}

Suppose we have the following:

- *can_assume* (r_k, r_g, *t*, *d*)
- *can_assume_with_cascade* (r_g, r_h, *t*, *d*)

The users who are authorized to r_k are authorized to r_g by virtue of the first relation. They are also authorized to r_h because of *can_assume_with_cascade*.

7.5.3.4 *can_assume* Revocation

can_assume is revoked in the following ways:

a. By expiration: All authorizations obtained by a *can_assume* relation expire when the relation expires, i.e. when current time $\geq$ (*t* + *d*) specified in the relation.

b. By SSO action: If the SSO deletes the relation, then all authorizations specified by that relation are revoked.

c. The authorization given to u over role r_h via relation *can_assume*(r_g, r_h, t, d) is revoked if his authorization to r_g is revoked. Obviously, other beneficiaries of the relation are not affected.

d. The authorization given to u over role $r_h \in RHS(ae_j)$ *via* relation *can_assume*(ae_i, ae_j, t, d) is revoked if changes in authorization rules resulted in $r_h \notin RHS(ae_j)$.

e. If a change in authorization rules or users' attributes results in making a user unable to satisfy expression ae_i in relation *can_assume*(ae_i, ae_j, t, d).

7.5.4 *can_assume* and IRH

Assume we have the following fine-granularity *can_assume* relation:

$$can\_assume\ (r_g, r_h, t_1, d_1)$$

According to definition 4, $(r_g, r_h) \in$IRH means that $(u, r_g) \in$ URAuth $\rightarrow (u, r_h) \in$ URAuth.

Assume that the two roles are incomparable. Due to the above relation, r_h inherits all users of r_g. This flow of users' inheritance among the roles is the underlining semantics of IRH and based on that, we can say that $r_g \geq r_h$ in the IRH. We have discussed this in Chapter 3. Moreover, this relation affects $\mathcal{IRH}$. To show this, suppose that in the original IRH, we have the $r_h \geq r_g$. The above relation resulted in $r_g \geq r_h$. It makes IRH a quasi-order. Since r_g and r_h are mutually senior to each other, they introduce a new class to which they belong and, consequently, the $\mathcal{IRH}$ is modified.

Coarse-granularity *can_assume* also affects the IRH. Consider Figure 45 and the following relation:

can_assume(ae_2, ae_3, *t*, *d*)

Accordingly, ∀ *u* such that (*u*, ae_2) ∈ U_AE who is authorized to ∀ r_g ∈ *RHS*(ae_2), i.e. r_2 in Figure 45, will be authorized to ∀ r_h ∈ *RHS*(ae_3), i.e. r_3 in the figure. As such, the following holds:

(*u*, r_2) ∈ URAuth → (*u*, r_3) ∈ URAuth

This means that *can_assume* relation makes $r_2 \geq r_3$ in the new IRH. However, consider the following relation:

can_assume (ae_3, ae_2, *t*, *d*)

For this relation, (*u*, r_2) ∈ URAuth does not imply that (*u*, r_3) ∈ URAuth. This is so because of the *rule*$_4$: $ae_4 \Rightarrow r_3$.

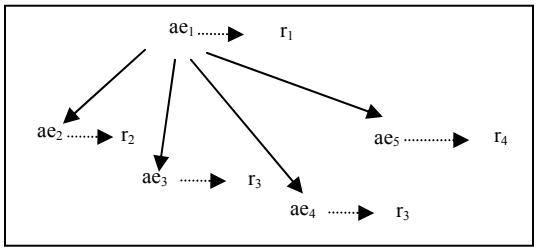

Figure 45: *can_assume* and IRH

Definition 22

IRH= {(r_g ,r_h) | (*u*, r_g) ∈URAuth → (*u*, r_h) ∈URAuth]

∨ ((*can_assume* (r_g, r_h, *t*, *d*)) ∨ (*can_assume_with_cascade*(r_g, r_h, *t*, *d*))

∨ (*can_assume*(ae_i, ae_j, *t*, *d*) Λ r_g∈*RHS*(ae_i) Λ r_h ∈*RHS*(ae_j) Λ

¬ (∃ae_k) [¬(ae_i→ ae_k) ∧ r_h ∈*RHS*(ae_k)]))

Λ *can_assume*/*can_assume_with_cascade* has not expired}

7.5.5 *can_assume* and GRH

Changes incurred by *can_assume* relation to the IRH might create discrepancy between IRH and GRH. The relation creates an artificial seniority among roles due to users' inheritance that is not accompanied by a matching permission inheritance. This discrepancy is hard to reconcile because the users' inheritance is not genuine and does not reflect a business practice that can reshape permissions inheritance among roles, which is the basis of the GRH.

7.6 Delegation

7.6.1 Introduction

Roles delegation is discussed in RBAC literature, where user *u*, who is assigned to role r_g, is allowed to delegate his membership in r_g to a specific user, say *v*, who is a member of role r_h. However, the recipient, in this case *v*, is not assigned to the role, and thus cannot activate the role, until the delegating user, *u* in this case, actively delegates his membership in r_g to *v*. So, merely adding a pair of roles to can_delegate relation does not guarantee that the potential recipient of the delegation will actually be able to assume the delegated role. Instead, the actual delegation is left to the discretion of the delegating party. This notion of delegation might be beneficial in certain circumstances, such as if one of an enterprise staff is leaving on a vacation and wants to delegate his role to his assistant.

Role delegation is an issue that has been strongly motivated in the literature [BS2000], [Bar2002], and [ZAC2003]. Our work is cast within the framework laid down by Barka [Bar2002] which was devised for the explicit user-role assignment of RBAC. We modify Barka's model to suit the requirement of the implicit user-role assignment specified using RB-RBAC.

7.6.2 Specification

7.6.2.1 *can_delegate* Relation

In this form of *can_delegate*, we modify the definition of *can_delegate* presented in RBDM0 where the unit of delegation is the role. This type is useful when it is desired to

allow individual users to delegate their membership in a specific role to other users who are members of another specific role. We differentiate between two types of membership:

i. Original membership: Assuming that $r_g \in RHS(ae_i)$, original membership of r_g is acquired via either satisfying $rule_i$, or satisfying a rule, $rule_k$, such that $rule_k \geq rule_i$.

ii. Delegated membership: This membership is acquired when a user is delegated membership to a role.

Definition 23

$can\_delegate \subseteq IR \times IR \times T \times D$

where IR, T, and D are roles set, time set, and duration set respectively.

For two users u and v, $(u, v) \in U$, $can\_delegate$ (r_g, r_h, t, d) means that user u such that $(u, r_g) \in$URAuth can delegate his membership in role r_g to $(v, r_h) \in$URAuth, starting at time t and for the duration d. Note that $(u, r_g) \in$URAuth means that u is an original members of r_g, which, by definition, confines the privilege of delegating a role to its original members.

7.6.2.2 *can_delegate_with_cascade* Relation

The recipient of a delegated membership to a role is authorized to further delegate it to other users. This type provides more flexibility than the one discussed above.

Definition 24

can_delegate_with_cascade $\subseteq$ IR $\times$ IR $\times$ T $\times$ D

where IR, T, and D are roles set, time set, and duration set respectively.

So for two users *u* and *v*, (*u*, *v*) $\in$U, *can_delegate_with_cascade* (r_g , r_h, *t*, *d*) means that *u* such that(*u*, r_g) $\in$ URAuth, i.e. *u* who is authorized to r_g, can delegate his membership in role r_g to *v* such that (*v*, r_h) $\in$ URAuth i.e. *v* who is authorized to r_h starting at time *t* and for the duration *d*. Furthermore, *v* can delegate his delegated membership of r_g to another user *w*, who is authorized to r_k provided either/both of the following holds:

a. *can_delegate* (r_h , r_k, *t*, *d*)

b. *can_delegate_with_cascade* (r_h , r_k, *t*, *d*)

7.6.2.3 Revocation of *can_delegate*

can_delegate is revoked in the following ways:

a. By expiration: All authorizations obtained by a *can_delegate* relation expire when the relation expires, i.e. when current time $\geq$ (*t* + *d*) specified in the relation.
b. By SSO action: If the SSO deletes the relation, then all authorizations specified by that relation are revoked.
c. The authorization given to a recipient of delegation is revoked if the authorization of the delegating user is revoked.
d. The authorization given to a recipient of delegation is revoked if the authorization he has over his own role is revoked.

7.6.3 Delegation Semantics and Users States

In recognizing various states of users, RB-RBAC gives a plethora of semantics to the *can_delegate* relation since a delegating user can be in P, D, or Act state *wrt* delegating role. Similarly, the recipient can be in P, D, or Act state *wrt* delegated role. For example, a user *v* who is in D state *wrt* r_h may receive delegation for role r_g membership from user *u* who is in P state *wrt* r_g but is in Act state *wrt* another role, r_k. This enriches the semantics of *can_delegate* by making it possible to specify different variations of *can_delegate*, each has a very specific semantics. To illustrate, assume that the policy prohibits any user authorized to a role r_g from delegating his membership to another user unless the original user has activated the delegated role at least once. This can be achieved by the following specification:

$(r_g, r_h, t, d) \in can\_delegate^{activated\ once}$ so that $(\forall u,v)$ $[(u,v) \in U, can\_delegate^{activated\ once}$ (r_g, r_h, t, d) means that a user *u* such that $(u, r_g) \in URA \cup URD$ can delegate his membership in role r_g to *v* such that $(v,r_h) \in URAuth]$.

7.7 Summary

In this chapter, we have presented ARB-RBAC, a rich model that provides administration of systems that implements RB-RBAC. Besides showing how to use ARB-RBAC for specifying traditional services provided by administrative models discussed in RBAC literature, new concepts such as *can_assume* have been introduced and analyzed. Figure 46 presents a summary of ARB-RBAC formal model.

1. The following is imported from RBAC96: AR and ARH, which are the set of administrative roles and the administrative role hierarchy. We assume that user-role assignment *wrt* administrative roles is explicitly performed.
2. The notion of role range, *rr* for short, is imported from ARBAC97.
3. 2^{Att} is the power set of all possible attributes.
4. A prerequisite condition is a Boolean expression using the usual $\wedge$ and $\vee$ operators on terms of the form x and x′ where x is a regular role (i.e., $x \in R$). A prerequisite condition is evaluated for a user *u* by interpreting *x* to be true if:
   - $x \in R$: $(\exists x' \geq_{GRH} x) . (u, x') \in$ URAuth

   and x′ to be true if:
   - $x \in R$: $(\forall x' \geq_{GRH} x) . (u, x') \notin$ URAuth
5. CR is the set of all possible prerequisite conditions
6. $r \in rr$ as defined in ARBAC97. This means that the roles that represent the end point of the range may or may not be within the range.
7. AuthorizationOutsideRoleRange or $AORR(u,rr) = \{(u,r) | (u, r) \in \text{URAuth} \wedge r \notin rr\}$. *AORR* returns the sets of roles outside *rr* to which *u* is authorized to activate.
8. Administrating an attribute means the following:
   a. Adding a new attribute to the user's attributes
   b. Modifying the values of an existing attribute
   c. Deleting an attribute
9. ARB-RBAC X model authorizes administering users' attributes via the relation
   $$can\_administer\_attributes \subseteq AR \times 2^{Att} \times CR \times 2^{R}$$
   The semantic of the relation *can_administer_attributes* (x, y, c, *rr*) is that a member of administrative role x (or a member of an administrative role that is senior to x) can administer the attributes of a user *u* provided that:
   a. The modified *u* 's attributes $\subseteq$ y,
   b. *u* satisfies prerequisite condition c, and
   c. $\forall ae_i$ that is satisfied by the resulting attributes the following holds:
      $RHS(ae_i) \in rr \wedge AORR(u,rr) = AORR'(u,rr)$.
      AORR (*u*,*rr*) and *AORR* ′(*u*,*rr*) are the sets of roles outside *rr* to which *u* is authorized to activate before and after the changes made by the JSO, respectively. This specification is required to ensure that the changes that a security officer makes may not cause a change in the user's authorization *wrt* roles outside the designated role range.
10. ARB-RBAC Y model authorizes administering users' attributes via the relation
    $$can\_revoke \subseteq AR \times 2^{Att} \times 2^{R}$$
    The semantic of *can_revoke* (x, y, *rr*) is that a member of administrative role x (or a member of an administrative role that is senior to x) can administer the attributes set of a user *u* if the following holds:
    1. $\alpha \subseteq$ y, where α is *u* 's attributes before the modification such that α satisfies $ae_i \wedge RHS(ae_i) \in rr$.
    2. If β is the set of resulting attributes, then $\forall ae_j$ that is satisfied by the β, the following holds:
       β satisfies $ae_j \rightarrow RHS(ae_j) \notin rr$
    3. $AORR(u,rr) = AORR'(u,rr)$
11. RB-RBAC model authorizes administering users' attributes via the relation *can_administer_rule* $\subseteq AR \times 2^{R}$
12. Coarse-Granularity *can_assume* is specified using the following: *can_assume* $\subseteq AE \times AE \times T \times D$
13. Fine-Granularity *can_assume* is specified using the following: *can_assume* $\subseteq IR \times IR \times T \times D$
14. *can_assume_with_cascade* is specified using the following: *can_assume_with_cascade* $\subseteq IR \times IR \times T \times D$
15. IRH= $\{(r_g, r_h) \mid (u, r_g) \in \text{URAuth} \rightarrow (u, r_h) \in \text{URAuth}$]
    $\vee$ ((*can_assume* (r_g, r_h, t, d)) $\vee$ (*can_assume_with_cascade*(r_g, r_h, t, d))
    $\vee$ (*can_assume*(ae_i, ae_j, t, d) Λ $r_g \in RHS(ae_i)$ Λ $r_h \in RHS(ae_j)$ Λ $\neg$ $(\exists ae_k)$ $[\neg(ae_i \rightarrow ae_k) \wedge r_h \in RHS(ae_k)]$)) Λ *can_assume*/*can_assume_with_cascade* has not expired}
16. *can_delegate* is specified using the following: *can_delegate* $\subseteq IR \times IR \times T \times D$
17. *can_ delegate_with_cascade* is specified using the following: *can_ delegate_with_cascade* $\subseteq IR \times IR \times T \times D$
18.

Figure 46: ARB-RBAC Formal Model

Chapter 8: Conclusion

This chapter lists the main contributions of this dissertation and discusses future work.

8.1 Contributions

The principal contributions of this dissertation are:

a. The formalization of a new family of models to automate user-role assignment based on a set of authorization rules. This family of models provides languages to express the authorization rules.

b. The identification of seniority relations that might hold among authorization rules.

c. Introducing and formalizing the concept of Induced Role Hierarchies (IRH) which is extracted from the seniority relations to represent the formal security policy of the enterprise.

d. The identification and analysis of possible discrepancies between the IRH which represent the formal security policy of the enterprise, and GRH which is the *de facto* security policy. Our work also provides insight about their probable reasons of discrepancy and how to reconcile them.

e. The introduction of negative authorization into RBAC context with the suitable syntax and semantics accompanied by the identification and analysis of possible

conflict among authorization rules and suggesting policies to resolve the conflict. Moreover, new conflict resolution policies are discussed.

f. Analysis of 3 classes of prohibition constraints, their semantics in RB-RBAC context, and their impact on IRH. New types of constraints were introduced.

g. Allowing specifying both local (rule-specific) and global constraints (invariants) in the same model which improves functionality, implementation and security of the system that implements RB-RBAC. Three methods to specify constraints within RB-RBAC are presented, compared and contrasted to determine their relative strengths and weaknesses. Policies to resolve conflict that might rise among the three methods of specifying constraints are suggested.

h. The suggestion of new semantics for the cardinality semantics in the presence of GRH

i. The formalization of a companion administrative model, namely ARB-RBAC, whose specifications are based on users' attributes. It allows the authorized individuals to perform administrative tasks, such as the administration of the users' attributes, authorization rules, and delegation.

j. Introducing novel concepts to RBAC administration such as *can_assume* relation which gives security officers a new scope of authority over the user-role assignment process.

k. Specifying *can_delegate* relation that allows users to delegate roles if permitted by the security policy. Our work allows new semantics for delegation based on users' states.

8.2 Future Work

RB-RBAC model can be extended in several different directions:

8.2.1 Cross-domain RB-RBAC

RB-RBAC is designed to provide access control for a single domain which we define as a collection of resources and services under a single point of access control. However, RB-RBAC could be extended to provide access control to multi-domains, which requires among other things:

i. Integrating RB-RBAC with emerging technologies such as SAML and XML to facilitate the communication among interacting RB-RBAC implementations.
ii. Utilizing trust management protocols such as the work in the area of *Automated Trust Negotiation* to help in trust negotiation among interacting RB-RBAC implementations.

8.2.2 Enforcement architectures

Several possible enforcement architectures could be built to support RB-RBAC. We may start by a basic architecture to support Model A, and then extend this architecture to support models B and C. The analysis we provided gives some insight to factors that need to be considered when weighing different architectural alternatives.

8.2.3 Role parameterization

To reduce the number of roles which, as a result, reduces the administrative overhead, RB-RBAC could be extended to allow role parameterization.

Bibliography

Bibliography

[Ahn1999] G. Ahn. "*THE RCL 2000 Language FOR Specifying Role-Based Authorization Constraints"*, A Dissertation Submitted to George Mason University, 1999.

[AS2002] M. Al-Kahtani and R. Sandhu, "*A Model for Attribute-Based User-Role Assignment*", In Proceedings of the 18th Annual Computer Security Applications Conference, Las Vegas, Nevada, December 9-13, 2002.

[AS2003] M. Al-Kahtani and R. Sandhu, *"Induced Role Hierarchies with Attribute-Based RBAC"*, In Proceedings of the 8th ACM Symposium on Access Control Models and Technologies (SACMAT), Villa Gallia, Como, Italy, June 2-3, 2003.

[AS2001] G. Ahn and Michael Shin, "*Role-based Authorization Constraints Specification Using Object Constraint Language*", 10th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises **,** June 20 - 22, 2001, Massachusetts

[Bar2002] E. Baraka, "*Framework for Role-Based Delegation Models*", A Dissertation A Dissertation Submitted to George Mason University, 2002.

[BB2001] E. Bertino and P Bonatti, "*TRBAC, A Temporal Role-Based Access Control Model*", In ACM Transactions of Information and System Security, Vol. 4, No. 3, August 2001, Pages 191-223.

[BCFP2003] E. Bertino, B. Catania, E. Ferrari and P. Perlasca, "*A logical Framework for Reasoning About Access Control Models*", In the ACM Transactions on Information and System Security (TISSEC) ,Vol. 6 , No. 1, February 2003, Pages: 71 - 127

[BJWW2002] C. Bettini, S. Jajodia, X. S. Wang, and D. Wijesekera, "*Provisions and Obligations in Policy Management and Security Applications*", In Proceedings of the 28th VLDB conference, Hong Kong, China, 2002.

[BMY2002] J. Bacon, K. Moody, and W. Yao, "*A Model of OASIS Role-Based Control and its Support for Active Security*", In the ACM Transactions on Information and System Security (TISSEC) , Volume 5 , Issue 4 (November 2002) , Pages: 492 - 540

[BS2000] E. Baraka and R. Sandhu "*Framework for Role-Based Delegation Models*", In Proceedings of 16th Annual Computer Security Applications Conference (ACSAC'00)**,** December 11 - 15, 2000 New Orleans, Louisiana

[BSJ1993] E. Bertino, P. Samarati, and S. Jajodia, "*Authorizations in Relational Database Management Systems"*, In Proceedings of the 1st ACM Conference on Computer and Communications Security (Fairfax, VA. Nov. 3–5). ACM, New York, pp. 130–139.

[BSJ1997] E. Bertino, P. Samarati, and S. Jajodia, "*An Extended Authorization Model for Relational Databases*", In IEEE Transactions On Knowledge and Data Engineering, Vol. 9, No. 1, January-February 1997.

[CS1995] F. Chen and R. Sandhu, "*Constraints for Role Based Access Control*", In Proceedings of 1st ACM Workshop on Role-Based Access Control, pages 39{46, Gaithersburg, MD, November 1995.

[CW1987] D. D. Clark and D. R. Wilson, "*A Comparison of Commercial and Military Computer Security Policies*", In Proceedings of IEEE Symposium on Security and Privacy, pages 184{194, April 1987.

[FBK1999] 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 Systems Security, 2(1):34-64, February 1999.

[FK1995] D. Ferraiolo, and R. Kuhn, "*Role Based Access Control: Features and Motivations*", In Annual Computer Security Applications Conference. IEEE Computer Society Press, 1995.

[FSGK2000] D. Ferraiolo, R. Sandhu, S. Gavrila , and R. Kuhn, "*Proposed NIST Standard for role-based access control: towards a unified standard*", In ACM Transaction on Information and System Security (TISSEC), Vol. 4, Number 3, August 2001.

[HMM2000] A. Herzberg, Y. Mass, and J. Mihaeli, "*Access Control Meets Public Key Infrastructure, Or: Assigning Roles to Strangers*", In Proceedings of the 2000 IEEE Symposium on Security and Privacy, 2000.

[JSS1997] S. Jajodia, P. Samarati and V.S. Subrahmanian, "*A logical Language for Expressing Authorizations*", In Proceedings of the 1997 IEEE Symposium on Security and Privacy, 1997.

[JSSS2001] S. Jajodia, P. Samarati, M. Sapino and V. Subrahmanian, "*Flexible Support for Multiple Access Control Policies*" , In ACM Transactions on Database Systems, Vol. 26, No. 2, June 2001.

[Ker2002] A. Kern, "*Advanced Features for Enterprise-Wide Role-Based Access Control*", In Proceedings of the 18th Annual Computer Security Applications Conference, Las Vegas, Nevada, USA, pages 333-342, December, 2002.

[KKSM2002] A. Kern, M. Kuhlmann, A. Schaad and J. Moffett, "*Observations on the Role Life-Cycle in the Context of Enterprise Security Management*", In Proceedings of the seventh ACM symposium on Access control models and technologies, Monterey, California, June 2002.

[KSM2003] A. Kern, A. Schaad and J. Moffett, "*An Administration Concept for the Enterprise Role-Based Access Control Model*", SACMAT'03, June 1-4, Como, Italy.

[KSS2003] M. Kuhlmann, D. Shohat, and G. Schimpf, "*Role Mining-Revealing Business Roles for Security Administration using Data Mining Technology"*, SACMAT'03, June 1-4, Como, Italy.

[Kuhn1997] R. Kuhn, "*Mutual Exclusion of Roles as a Means for Implementing Separation of Duty in Role-Based Access Control Systems*", Second ACM

Workshop on Role-Based Access Control, Fairfax, Virginia, United States,1997, Pages: 23-30

[LDAP1997] Lightweight Directory Access Protocol (v3), RFC2251, (December 1997).

[LDAP2001] Dynamic Groups for LDAPV3 draft-haripriya-dynamicgroup-00.txt, (October 2001).

[Mar2002] A. Marshall, "*A Financial Institution's Legacy Mainframe Access Control System in Light of the Proposed NIST RBAC Standard*", In Proceedings of the 18th Annual Computer Security Applications Conference, Las Vegas, Nevada, USA, pages 382-390, December, 2002.

[NO1999] M. Nyanchama and S. Osborn, "*The Role Graph Model and Conflict of Interest*", In ACM Transactions on Information and System Security (TISSEC) Volume 2 , Issue 1 (February 1999) , Pages: 3 - 33

[OSM2000] 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, vol. 3, No. 2, May 2000, pages 85-106.

[OSZ2003] S. Oh, R. Sandhu, and X. Zhang, "*An Effective Role Administration Model Using Organization Structure*", ACM Transactions on Information and System Security, *to be printed*.

[PSA2001] J. Park, R. Sandhu and G. Ahn, "*Role-based Access Control on the Web*", In ACM Transactions on Information and System Security, Vol. 4, No 1, 2001.

[PSG99] Joon S. Park, Ravi Sandhu, and SreeLatha Ghanta. *"RBAC on the Web by Secure Cookies*." In Proceedings of the IFIP WG11.3 Workshop on Database Security. Chapman & Hall, July, 1999.

[RS1998] C. Ramaswamy and R. Sandhu, "*Role-Based Access Control Features in Commercial Database Management Systems*", NISSC 1998.

[San1993] Ravi Sandhu, "*Lattice-Based Access Control Models*", IEEE Computer, Volume 26, Number 11 (Cover Article), November 1993

[San2000] R. Sandhu,"*Engineering Authority and Trust in Cyberspace, The OM-AM and the RBAC way"*, In Proceeding of RBAC 2000, Berlin, Germany.

[SB1999] R. Sandhu and V. Bhamidipati, "*Role-Based Administration of User-Role Assignment: The URA97 Model and its Oracle Implementation*", Journal of Network and Computer Applications, 22(3), July 1999.

[SBM1999] R. Sandhu, V. Bhamidipati, and Q. Munawer, "*The ARBAC97 Model for Role-based Administration of Roles*". ACM Transactions on Information and System Security. Vol.2, No.1, Feb. 1999, pages 105-135.

[SCFY1996] R. Sandhu, E. Coyne, H. Feinstein and C. Youman, "*Role-Based Access Control Model",* IEEE Computer, 29(2), Feb. 1996.

[SFK2000] R. Sandhu, D. Ferraiolo, and R. Kuhn, "*The NIST Model for Role-Based Access Control:Ttowards a Unified Standard*", In Proceedings of the fifth ACM workshop on Role-based access control table of contents, Berlin, Germany Pages: 47 - 63, Year of Publication: 2000

[She2000] General Henry H. Shelton, Chairman of the Joint Chiefs of Staff, *Office of the Assistant Secretary of Defense Reserve Affairs. Data as of September 30, 2000.* http://www.defenselink.mil/ra/rfpb/chapter_5.html

[SM1998] Ravi Sandhu and Qamar Munawer, "*How to Do Discretionary access Control Using Roles*", In Proceedings of 3rd ACM Workshop on Role-Based Access Control, Fairfax, Virginia, October 22-23,1998.

[WL2002] W. Winsborough and N. Li, "*Towards Practical Automated Trust Negotiation*", 3rd International Workshop in Policies for Distributed Systems and Networks (POLICY'02), June 05-07, 2002, Monterey, California.

[YMB2001] W. Yao, K. Moody, and J. Bacon, "*A Model of OASIS Role-Based Control and its Support for Active Security*", SACMAT'01, May 3-4, 2001, Chantilly, Virginia.

[ZAC2003] L. Zhang, G. Ahn, and B. Chu, "*A Rule-Based Framwork for Role-Based Delegation and Revocation*", In ACM Transactions on Information and System Security, Vol. 6, No. 3, August 2003, Pages 404-441.

[ZBM2001] Y. Zhong, B. Bhargava, and M. Mahoui, "*Trustworthiness Based Authorization on WWW*", In IEEE workshop on "Security in Distributed Data Warehousing", New Orleans, Oct. 2001.

Curriculum Vitae

Mohammad al-Kahtani was born on 7/9/1381 H, in Saudi Arabia, and is a Saudi citizen. He received a B.S. in Computer Science from King Saud University in Riyadh, Saudi Arabia, in 1410 H. Mohammad received a M.S. in Software Engineering from the George Mason University in Fairfax, Virginia, in May of 1995.

He joined the Laboratory for Information Security Technology in 2001. His research interests include access control, network security, and secure information systems.