THE *RCL 2000* LANGUAGE FOR SPECIFYING ROLE-BASED AUTHORIZATION CONSTRAINTS

by

Gail-Joon Ahn

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. Daniel Menascé
____________________	Dr. Prasanta Bose
____________________	Dr. Stephen G. Nash, Associate Dean for Graduate Studies and Research
____________________	Dr. Lloyd J. Griffiths, Dean, School of Information Technology and Engineering
Date: ____________________	Fall 1999 George Mason University Fairfax, Virginia

The *RCL 2000* Language for Specifying Role-Based Authorization Constraints

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

By

Gail-Joon Ahn

B.S., SoongSil University, Seoul, KOREA, February 1994
M.S., George Mason University, Fairfax, VA, January 1997

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

Fall 1999
George Mason University
Fairfax, Virginia

DEDICATION

To my God
"A man's heart deviseth his way: but the LORD directeth his steps." – Proverbs 16:9

To my parents, Dr. Young Ro Ahn and Kyung Ok Paik, my wife Mi Hye Ahn, my sister Ae Kyung Ahn, my brother Bong Joon Ahn, and my lovely son Benjamin Soohyun Ahn.

–Without their prayers and love, this work could not have been started.–

ACKNOWLEDGMENTS

I would like to sincerely express my gratitude and appreciation to my dissertation director, Professor Ravi Sandhu, who has been so graceful all the way and 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 Daniel Menasce, and Professor Prasanta Bose. I am thankful for their valuable comments on my dissertation.

I am particularly thankful to my parents, Dr. Young Ro Ahn and Kyung Ok Paik, to my wife Mi Hye Ahn, to my grandmother Im D. Park, to my uncle Elder Won K. Paik, and to my aunt Eun D. Kim for their prayers and love.

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

I acknowledge the financial support from National Security Agency (NSA), Naval Research Laboratory (NRL), National Science Foundation (NSF), and Lockheed Martin Corporation during my doctoral study.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

THE *RCL 2000* LANGUAGE FOR SPECIFYING ROLE-BASED AUTHORIZATION CONSTRAINTS

Gail-Joon Ahn, Ph.D.

George Mason University, 1999

Dissertation Director: Dr. Ravi S. Sandhu

Authorization constraints (also simply called constraints) are an important aspect of role-based access control (RBAC), since they can be argued to be one of the principal motivations behind RBAC. Although the importance of constraints in RBAC has been recognized for a long time, they have not received much attention in research literature, while role hierarchies have been practiced and discussed at considerable length. Most prior work has focused on separation of duty (SOD) constraints enumerating many variations. In this dissertation, we describe a framework for specifying authorization constraints in role-based systems. To specify these constraints, we need an appropriate language as well as some system functions. We propose a simple and intuitive language, *RCL 2000* (Role-based Constraints Language 2000), to specify constraints in an intuitive and useful way in role-based systems. The formal semantics for this language is based on its translation to a restricted form of first order predicate logic.

With this language we show how we can express the previous SOD constraints discovering newly identified properties, such as permission-centric constraints. We also define new forms of SOD, especially with role hierarchies. To illustrate the power of

RCL 2000 we specify constraints which have been identified in simulations of Lattice-based access control, Chinese Wall, and Discretionary access control policy in RBAC. Moreover, we separate role-based constraints into two major classes: *Prohibition Constraints* and *Obligation Constraints.* We characterize a subset of these classes from our specification of role-based constraints.

Our work also shows that it is futile to try to enumerate all constraints because there are too many possibilities and variations; instead, we should pursue an intuitively simple yet rigorous language, such as *RCL 2000*, for specifying constraints.

Chapter 1

INTRODUCTION

Role-based access control (RBAC) has emerged as a widely accepted alternative to classical discretionary and mandatory access controls [SCFY96]. Several models of RBAC have been published and several commercial implementations are available. RBAC regulates the access of users to information and system resources on the basis of activities that users need to execute in the system. It requires the identification of roles in the system. A role can be defined as a set of actions and responsibilities associated with a particular working activity. Then, instead of specifying all the access each user is allowed to execute, access authorizations on objects are specified for 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. Because roles within an organization typically have overlapping permissions, RBAC models include features to establish role hierarchies, where a given role can include all of the permissions of another role. Another fundamental aspect of RBAC is authorization constraints (also simply called constraints). Although the importance of constraints in RBAC has been recognized for a long time, they have not received much attention in the research literature, while role hierarchies have been practiced and discussed at considerable length [SA98a, FGS94, SA98b, Jan98, AS98, AS99b, Ahn99a, Ahn99b].

In this dissertation we focus on constraints in RBAC. We may consider several issues such as constraint specification, constraint analysis, and constraint enforcement. In this dissertation our focus is on constraint specifications, i.e, on how constraints can be expressed. Constraints can be expressed in natural languages, such as English, or in more formal languages. Natural language specification has the advantage of ease of comprehension by human beings, but may be prone to ambiguities. Natural language specifications do not lend themselves to the analysis of properties of the set of constraints. For example, one may want to check if there are conflicting constraints in the set of access constraints for an organization. We opted for a formal language approach to specify constraints. The advantages of this method include : i) a formal way of reasoning about constraints, ii) a framework for identifying new types of constraints, iii) a classification scheme for types of constraints (e.g., prohibition constraints and obligation constraints), and iv) a basis for supporting optimization and specification techniques on sets of constraints.

Also we may need to study how to enforce constraints which are identified in the role-based systems. These two research issues, constraints analysis and enforcement, remain to be done in future work.

This dissertation describes a framework for specifying constraints in role-based systems. To specify these constraints we introduce the specification language *RCL 2000* (for Role-based Constraints Language 2000, pronounced *Rı́ckle* 2000). The user of this language will be mainly the security researcher who needs to think and reason about role-based authorization constraints. The objective of this language is to specify role-based constraints and help the security researcher to investigate the useful constraints including laying out an organizational policy.

1.1 Role-Based Access Control

This section gives a brief overview of a well-known family of models for RBAC, commonly known as RBAC96 [SCFY96]. This model has become a widely-cited authoritative reference and is the basis of a standard currently under development by the National Institute of Standards and Technology [SB97, SBC$^+$97, SBM99, Giu95, FK92]. This model was also used to develop implementations of various RBAC models and mechanisms using Unix, Oracle, Windows NT, and so on [SA98a, SA98b, AS98, SB97, GS96, Rut97, Sut97, Gri97]. *RCL 2000* is defined in context of RBAC96.

Intuitively, a user is a human being or an autonomous agent, a role is a job function or job title within an organization with some associated semantics regarding the authority and responsibility conferred on a member of the role, and a permission is an approval of a particular mode of access (operation) to one or more objects in the system or some privilege to carry out specified actions. Roles are organized in a partial order or hierarchy, so that a senior role inherits permissions from junior roles, but not vice versa. A user can be a member of many roles and a role can have many users. Similarly, a role can have many permissions and the same permission can be assigned to many roles. Each session relates one user to possibly many roles. Intuitively, a user establishes a session (e.g., by signing on to the system) during which the user activates some subset of roles that he or she is a member of. The permissions available to the users are the union of permissions from all roles activated in that session. Each session is associated with a single user. This association remains constant for the life of a session. A user may have multiple sessions open at the same time, each in a different window on the workstation screen, for instance. Each session may have a different combination of active roles. The concept of a session equates to the traditional notion of a subject in access control. A subject is a unit of access control, and a user may have multiple subjects (or sessions) with different permissions active at the same time.

Figure 1.1 shows the described RBAC96 model. RBAC96 is comprised of a family of four conceptual models. $RBAC_0$ is the base model and the minimum requirement for any system to support RBAC. $RBAC_1$ and $RBAC_2$ both include $RBAC_0$ and each adds an independent feature to it. $RBAC_1$ has the additional feature of role hierarchies so roles can inherit the permissions from other roles. $RBAC_2$ has the additional feature of constraints which impose restrictions on configuration of the components of RBAC as shown by the dashed lines. $RBAC_3$ is the consolidated model which includes $RBAC_1$ and $RBAC_2$. Formal definitions for the various component of RBAC96 are given in figure 1.2. RBAC96 does not define constraints formally. In this dissertation we will give a formal definition of constraints in terms of the *RCL 2000* language used to define them.

1.2 Authorization Constraints

Constraints are an important aspect of access control and are a powerful mechanism for laying out a higher level organizational policy [Dep85]. Consequently, the specification of constraints needs to be considered [Hay98]. Unfortunately, this scope has not received enough attention in the area of role-based access control. [1] And most prior works have focused on separation of duty constraints, which is a foundational principle in computer security as described below.

[1]In this dissertation, we are focusing on the research works which deal with constraints in the context of role-based access control. Also, there are several works such as [BF99, HBM98, VCP98, DHTK93, HW89, FB98, JSS97, JSSB97, CCSC98, CS96, FJ95, MS92, Mic98] which are just related to constraints in other area.

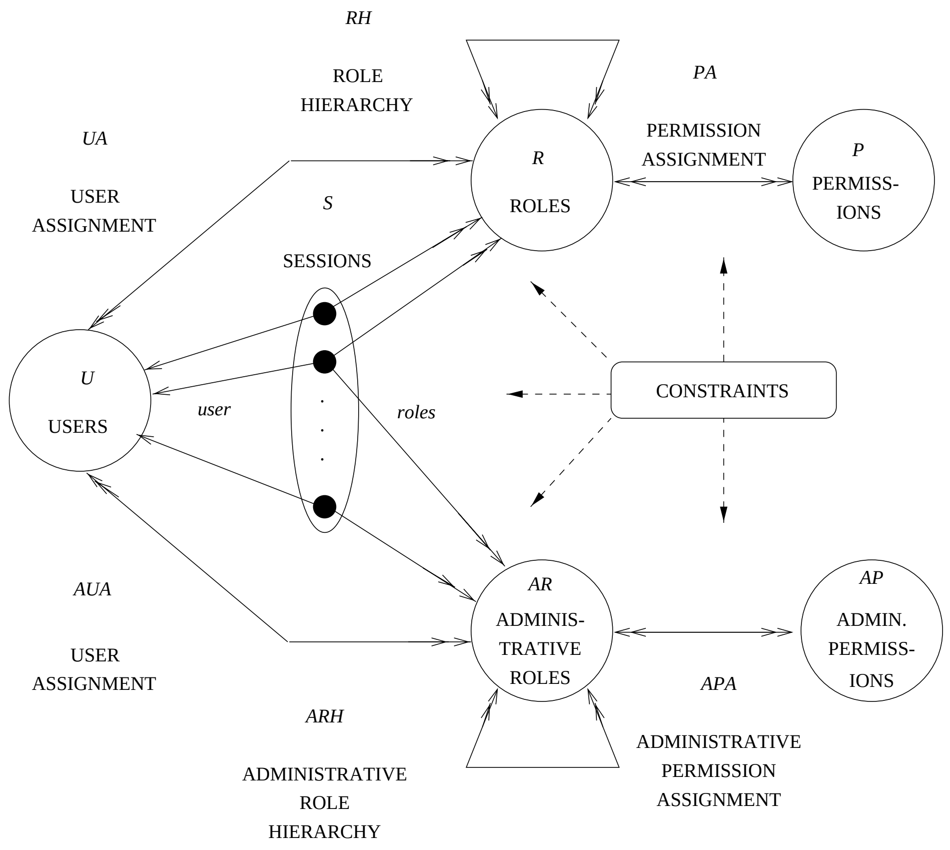


Figure 1.1: RBAC96 Model

- U, a set of users
  R and AR, disjoint sets of (regular) roles and administrative roles
  P and AP, disjoint sets 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 ordered role hierarchy
  $ARH \subseteq AR \times AR$, partially ordered 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 and administrative roles
  $roles(s_i) \subseteq \{r \mid (\exists\, r' \geq r)[(user(s_i), r') \in UA \cup AUA]\}$ (which can change with time)
  session s_i has the permissions $\cup_{r \in roles(s_i)} \{p \mid (\exists\, r'' \leq r)[(p, r'') \in PA \cup APA]\}$
- There is a collection of constraints stipulating which values of the various components enumerated above are allowed or forbidden.

Figure 1.2: Summary of the RBAC96 Model

1.2.1 Separation of Duty Constraints

Separation of duty (SOD) constraints are a major class of role-based authorization constraints. In this dissertation we focus on SOD constraints to present a framework for specification of constraints in role-based systems. SOD is a fundamental technique for preventing fraud and errors, known and practiced long before the existence of computers. Once certain roles are declared mutually exclusive, there's less concern about assigning individual users to roles. User assignment can be delegated and decentralized without fear of compromising the organization's overall policy objectives. A common 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. Several definitions of SOD have been given in the literature as enumerated in table 1.1. We give our own definition of SOD below and also extend it to apply in a role-based environment.

> **Separation of duty** *reduces the possibility of fraud or significant errors which can cause damage to an organization by partitioning of tasks and associated privileges required to complete a task or set of related tasks.*
> **Role-Based separation of duty** *enforces SOD in a role-based environment by controlling membership in and use of roles, as well as permission assignment.*

Although separation of duty is intuitively easy to understand, so far there is no systematic and rigorous basis for expressing this principle in computerized information systems. There are several papers in the literature which deal with separation of duty [CW87, San88, Bal90, NP90, PM94, FCK95, SZ97, Kuh97, FBK99, NO99, GGF98]. These papers have identified numerous forms of SOD, but there has been little work so far on specifying SOD policies in a comprehensive way. We provide a brief overview of these early works in chapter 2.

1.3 Outline of the Dissertation

In this dissertation we develop a framework for specification of authorization constraints identified in role-based systems. In order to specify constraints in an intuitive and useful way in role-based systems, we propose a simple and intuitive language *RCL 2000*. The formal semantics for this language is based on its translation to a restricted form of first order predicate logic.

Table 1.1: Definition of SOD

Separation of Duty
"No user of the system, even if authorized, may be permitted to modify data items in such a way that assets or accounting records of the company are lost or corrupted. Essentially there are two mechanisms at the heart of fraud and error control: the well-formed transaction, and separation of duty among employees. The most basic separation of duty rule is that any person permitted to create or certify a well-formed transaction may not be permitted to execute it. This rule ensures that at least two people are required to cause a change in the set of well-formed transactions." (from [CW87])
"Separation of duty is enforced by the rule that for transient objects different transactions must be executed by distinct users." (from [San88])
"NPDs make it possible to express separation of duties policies that help prevent fraud by prohibiting certain privileges from being active simultaneously." (from [Bal90])
"Separation of duty attempts to ensure the correspondence between data objects within a system and the real world objects they represent. This correspondence cannot normally be certified directly. Rather, the correspondence is endured indirectly by separating all operations into several subparts and requiring that each subpart be executed by a different person." (from [NP90])
"The same individual is not permitted to belong to both roles, because this creates a possibility for committing fraud. This well-known, time-honored principle is separation of duty." (from [SCFY96])
"It is a security principle used to formulate multi-person control policies, requiring that two or more different people be responsible for the completion of a task or set of related tasks. The purpose of this principle is to discourage fraud by spreading the responsibility and authority for an action or task over multiple people, thereby raising the risk involved in committing a fraudulent act by requiring the involvement of more than one individual." (from [SZ97])
"Separation of duty is an important requirement in many commercial systems, and one of the most desired features of RBAC systems. One means of implementing separation of duty policies is with mutual exclusion of roles." (from [Kuh97])
"Its purpose is to ensure that failures of omission or commission within an organization are caused only by collusion among individuals and, therefore, are riskier and less likely, and that chances of collusion are minimized by assigning individuals of different skills or divergent interests to separate tasks." (from [GGF98])
"It refers to the partitioning of tasks and associated privileges among different mutually-exclusive roles associated with a single user to prevent the action of users from interfering or colluding with one another." (from [FBK99])
"In complex environments where the actions of ill-intentioned users can create financial or other damage to a company, it is common to identify combinations of operations which should not be authorized to a single user. Policies which deal with preventing such fraud are called separation of duties policies." (from [NO99])

With this language we show how we can express the previous SOD constraints discovering newly identified properties, such as permission-centric constraints. We also define new forms of SOD, especially with role hierarchies. To illustrate the power of *RCL 2000* we specify constraints which have been identified in simulations of Lattice-based access control, Chinese Wall, and Discretionary access control policy in RBAC. Moreover, we separate role-based constraints into two major classes: *Prohibition Constraints* and *Obligation Constraints*. We characterize a subset of these classes from our specification of role-based constraints.

Our work also shows that it is futile to try to enumerate all constraints because there are too many possibilities and and variations; instead, we should pursue an intuitively simple yet rigorous language, such as *RCL 2000*, for specifying constraints.

Chapter 2

RELATED WORK

Authorization constraints (also simply called constraints) are an important aspect of access control and are a powerful mechanism for laying out higher level organizational policy. Consequently, the specification of constraints needs to be considered. So far, this topic has not received much attention in research literature. There are a few papers such as [CS95, GI96] that deal with constraints in the context of role-based access control. These papers, however, are preliminary and tentative, and also need substantial further work. Also most of the prior work has focused on separation of duty constraints which is a foundational principle in computer security, as described in section 1.2.1. In this chapter we briefly review the related works which deal with constraints in the context of role-based access control.

2.1 Authorization Constraints

As we mentioned earlier, authorization constraints are an important aspect of access control. This issue has received surprisingly little attention in research literature. There is some work such as [CS95, GI96] that deal with constraints in the context of role-based access control. These papers, however, are preliminary and tentative, and also need substantial further development. Chen and Sandhu [CS95] presented the initial description which *RCL 2000* builds on. Even though their description is preliminary, it suggests how constraints can be specified. Giuri and Iglio [GI96] defined a new model to provide the capability of defining constraints on roles. In their model, a role is defined as a *named set of constrained protection domains* (NSCPD) that is activatable only if the corresponding constraint is satisfied. Their description focused on the activation of roles. But we should also consider that constraints can be applied to other components in RBAC.

2.2 Separation of Duty Constraints

In this section, we discuss prior work on separation of duty which is a foundational principle in computer security.

Clark and Wilson [CW87] called attention to separation of duty as one of the major mechanisms to counter fraud and error while ensuring the correspondence between data objects within a system and the real world objects they represent. The Clark-Wilson scheme includes the requirement that the system maintain the separation of duty requirement expressed in the access control triples. It calls for certification by the security officer that these tuples provide adequate separation. It is therefore a static concept that is realized at design time.

Dynamic separation of duty provides greater flexibility by allowing a user to carry out conflicting operations, but only on distinct objects. Sandhu introduced notation for dynamic separation of duty in Transaction Control Expressions [San88]. Roles were used to specify who can issue which transaction steps. In Sandhu's model each user executing a step in a transaction had to be different. To enforce this, the history of the execution of each transaction sequence had to be maintained. The constraints specifying the roles that could execute each step were associated with an object. These constraints turned into the history specifying which user executed each step on that object. A weighted voting syntax allowed the specification of multiple person authorizations on a particular step on a particular object.

Baldwin [Bal90] introduced Named Protection Domains (NPDs) as a named, hierarchical grouping of database privileges and users. To help enforce separation of duty, a user could have only one of these NPDs activated at any time. The security administrator determined which NPDs could be activated, but there were no further restrictions on the graph of NPDs (other than it be acyclic). Thus, one activatable NPD could contain multiple activatable and non-activatable NPDs. While the activation restriction meant that a user could be in only one role (NPD) at a time, the security administrator could set up arbitrarily complex roles.

A number of new issues around separation of duty was raised by Nash and Poland's paper [NP90] of a portable security device used in the commercial world. Nash and Poland also proposed the notion of object based separation of duty, which forced every transaction against an object to be by a different user. They suggested using Sandhu's Transaction Control Expressions [San88, San90] to maintain the history of an object's transactions.

Ferraiolo et al. [FCK95, FBK99] defined three kinds of separation of duty in their formal model of RBAC. The first two were static separation of duty and dynamic separation of duty. These variants were presented in previous work. The third kind was operational separation of duty which introduced the notion of a business function and the set of operations required for that function; a business function resembles the notion of task and task unit introduced by Thomas and Sandhu [TS94]. The formal definition of operational separation of duty stated that no role can contain the permissions to execute all of the operations necessary to a single business function. This forces all business functions to require at least two roles to be used for their completion. The informal description of operational separation of duty assumes the roles involved have disjoint memberships (static separation of duty), so that no single person has access to all the operations in a business function.

Kuhn's paper [Kuh97] focussed on the time when exclusion is introduced, and the degree to which two roles conflict. As far as time is concerned, mutual exclusion can be defined at role authorization time, or at run time. As he observed, these correspond to static and dynamic separation of duties respectively. Kuhn also distinguished between complete and partial mutual exclusion of roles.

Simon and Zurko [SZ97] enumerated different kinds of separation of duty such as static separation of duty (or strong exclusion), dynamic separation of duty (or weak exclusion), and object-based separation of duty. Simon and Zurko's enumeration of kinds of conflict of interest also includes four more kinds which all have to do with complex tasks involving several interrelated steps, say in a workflow management system. They tried to enumerate all the variations of SOD that have been called out in one source or another but their description of SOD was informal.

Gligor et al. [GGF98] enumerated several forms of SOD properties using first order predicate logic which causes difficulties to understand the properties. They missed the important SOD properties in RBAC such as session-based SOD and SOD in role hierarchies. To constrain sessions, we should consider each session of a set of sessions as well as a set of sessions. Also, SOD should be applied with the role hierarchies.

Nyanchama and Osborn [NO99] discussed a taxonomy of conflict of interest types such as user-user conflicts and privilege-privilege conflicts. They also discussed such conflicts in great detail with respect to their role graph model.

2.3 Summary

We briefly reviewed the related work which is mostly related to role-based constraints and separation of duty constraints. These early works in constraints and separation of duty focused on mechanisms that were easy to understand and are often rigid and unrealistic. In this dissertation, we describe a framework for specifying constraints in RBAC which has not been practiced in the literature. To specify these constraints, we need an appropriate language as well as some system functions. We propose a simple and intuitive language to specify constraints in an intuitive and realistic way in role-based systems. The formal semantics for this language is based on its translation to first order predicate logic.

Chapter 3

RCL 2000 LANGUAGE

In this chapter we define a new formal language called *RCL 2000* (for Role-based Constraint Language, pronounced *Ríckle* 2000), actually is language for specifying role-based constraints. To develop this language, we need a model for role-based systems. A general RBAC model, commonly called RBAC96 [SCFY96, San97] has become a widely cited reference in this arena. For the most part, *RCL 2000* components are built upon RBAC96 [Ahn99b, AS99a]. RBAC96 model was required in section 1.1. Here we use a slightly augmented form of RBAC96 illustrated in figure 3.1. We decompose permission into operations and object to enable formulation of certain forms of SOD. Also in figure 3.1 we drop the administrative roles of RBAC96 since they are not germane to *RCL 2000*.

Our work also builds upon SOD properties analyzed in [SZ97] and formalized in [GGF98]. Even though these papers lay out significant ground work they has several shortcomings. As we address these shortcomings, we also introduce the motivation for our work. In particular these previous papers do not have the notion of role hierarchies. Also, they miss the concept of session-based SOD which deals with SOD property in a single session. This form of dynamic SOD is required for simulating Lattice-based access control and Chinese Walls in RBAC [San93, San96]. Conflicting users and privileges are also not dealt with. From these observations, we are led to identify other significant SOD properties which have not been previously identified in the literature.

The rest of this chapter is organized as follows. In section 3.1 we introduce the basic elements on which *RCL 2000* is based and introduce notation and definitions that will be used throughout this dissertation, including non-deterministic functions which are newly proposed functions in this work. These non-deterministic functions are the core concepts of this work. They eliminate use of explicit quantifiers resulting in an intuitive language. give informal and intuitive definition of *RCL 2000*. Section 3.2 describes the formal syntax and semantics of *RCL 2000*. For the syntax we use usual Backus Normal Form. For semantics of *RCL 2000* we identify a restricted form of first order predicate logic which is exactly equivalent to *RCL 2000*. In section 3.3 we introduce the package method between *RCL 2000* expression which allows us to compose and reuse constraints.

3.1 Basic Elements and System Functions

In this section we introduce the basic elements on which *RCL 2000* is based and introduce notation and definitions that will be used throughout this dissertation. Also, we introduce the basic constructs of our specification language *RCL 2000*. We will show in subsequent chapters how these constructs can be used to specify various separation of duty properties.

3.1.1 Basic Elements and System Functions: from RBAC96

The basic elements on which *RCL 2000* is based and system functions that will be used in the rest of this dissertation are defined in figure 3.2. Figure 3.1 shows the RBAC96 model which is the context for these definitions.

RCL 2000 has six entity sets called users (`U`), roles (`R`), objects (`OBJ`), operations (`OP`), permissions (`P`), and sessions (`S`). These are interpreted as in RBAC96. A user is a

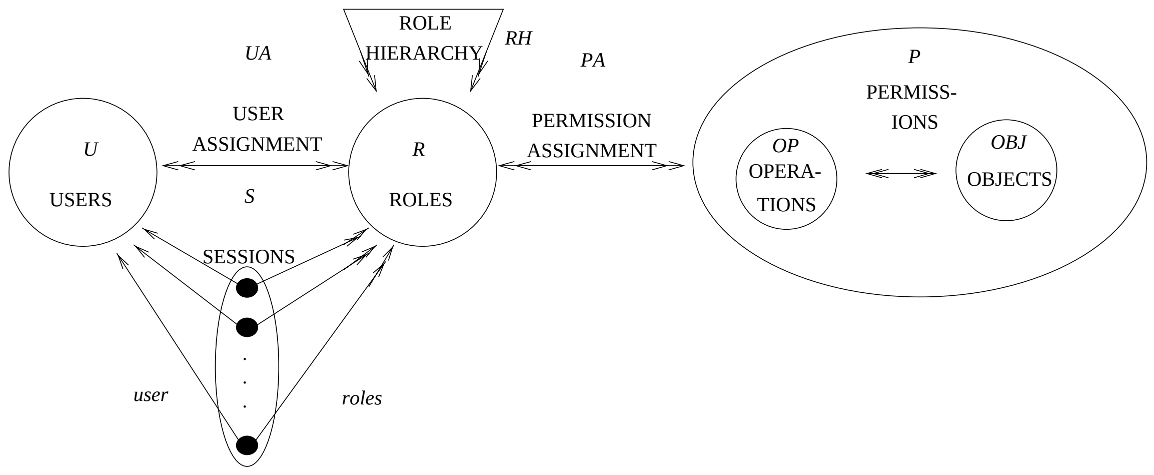


Figure 3.1: Basic Elements and System Functions : from RBAC96 Model

human being. A role is a named job function within the organization that describes the authority and responsibility conferred on a member of the role. Objects are passive entities that contain or receive information. An operation is an executable image of a program, which upon execution causes information flow between objects. A permission is an approval of a particular mode of operation to one or more objects in the system.

A session is a mapping between a user and an activated subset of the set of roles the user is assigned to. The function `user` gives us the user associated with a session and `roles` gives us the roles activated in a session. Both functions do not change during the life of a session.[1]

Hierarchies are a natural means for structuring roles to reflect an organization's lines of authority and responsibility (see Figure 3.3). By convention, senior roles are shown toward the top of this diagram and junior roles toward the bottom. Mathematically,

[1]This is a slight simplification from RBAC96 which does allow roles in a session to change. We adopt this simplification since our objective is the constraints specification instead of constraints enforcement which may need to maintain all activities in sessions.

- $\mathtt{U}$ = a set of users, $\{u_1, ..., u_n\}$; $\mathtt{R}$= a set of roles, $\{r_1, ..., r_m\}$;
- $\mathtt{OP}$ = a set of operations, $\{op_1, ..., op_o\}$; $\mathtt{OBJ}$ = a set of objects, $\{obj_1, ..., obj_r\}$;
- $\mathtt{P} = \mathtt{OP} \times \mathtt{OBJ}$, a set of permissions, $\{p_1, ..., p_q\}$; and
- $\mathtt{S}$ = a set of sessions, $\{s_1, ..., s_r\}$.
  - $\mathtt{user} : \mathtt{S} \rightarrow \mathtt{U}$, a function mapping each session s_i to the single user.
  - $\mathtt{roles} : \mathtt{S} \rightarrow 2^{\mathtt{R}}$, a function mapping the set $\mathtt{S}$ to a set of roles $\mathtt{R}$.
    $\mathtt{roles}(s_i) \subseteq \{r \in \mathtt{R} \mid (\mathtt{sessions}\ (s_i), r) \in \mathtt{UA}\}$
- $\mathtt{RH} \subseteq \mathtt{R} \times \mathtt{R}$ is a partial order on $\mathtt{R}$ called the role hierarchy or role dominance relation, written as $\preceq$.
- $\mathtt{UA} \subseteq \mathtt{U} \times \mathtt{R}$, a many-to-many user-to-role assignment relation.
- $\mathtt{PA} \subseteq \mathtt{P} \times \mathtt{R} = \mathtt{OP} \times \mathtt{OBJ} \times \mathtt{R}$, a many-to-many permission-to-role assignment relation.
- $\mathtt{user} : \mathtt{R} \rightarrow 2^{\mathtt{U}}$, a function mapping each role r_i to a set of users.
  $\mathtt{user}(r_i) = \{u \in \mathtt{U} \mid (u, r_i) \in \mathtt{UA}\}$
- $\mathtt{roles} : \mathtt{U} \cup \mathtt{P} \cup \mathtt{S} \rightarrow 2^{\mathtt{R}}$, a function mapping the set $\mathtt{U}$ and $\mathtt{P}$ to a set of roles $\mathtt{R}$.
  $\mathtt{roles}^* : \mathtt{U} \cup \mathtt{P} \cup \mathtt{S} \rightarrow 2^{\mathtt{R}}$, extends $\mathtt{roles}$ in presence of role hierarchy
  $\mathtt{roles}(u_i) = \{r \in \mathtt{R} \mid (u_i, r) \in \mathtt{UA}\}$ $\mathtt{roles}^*(u_i) = \{r \in \mathtt{R} \mid (\exists\, r' \succeq r)[(u_i, r') \in \mathtt{UA}]\}$
  $\mathtt{roles}(p_i) = \{r \in \mathtt{R} \mid (p_i, r) \in \mathtt{PA}\}$ $\mathtt{roles}^*(p_i) = \{r \in \mathtt{R} \mid (\exists\, r' \preceq r)[(p_i, r') \in \mathtt{PA}]\}$
  $\mathtt{roles}(s_i)$ = defined above $\mathtt{roles}^*(s_i) = \{r \in \mathtt{R} \mid (\exists\, r' \succeq r)[r' \in \mathtt{roles}(s_i)]\}$
- $\mathtt{sessions} : \mathtt{U} \rightarrow 2^{\mathtt{S}}$, a function mapping each user u_i to a set of sessions.
- $\mathtt{permissions} : \mathtt{R} \rightarrow 2^{\mathtt{P}}$, a function mapping each role r_i to a set of permissions.
  $\mathtt{permissions}^* : \mathtt{R} \rightarrow 2^{\mathtt{P}}$, extends $\mathtt{permissions}$ in presence of role hierarchy
  $\mathtt{permissions}(r_i) = \{p \in \mathtt{P} \mid (p, r_i) \in \mathtt{PA}\}$
  $\mathtt{permissions}^*(r_i) = \{p \in \mathtt{P} \mid (\exists\, r \preceq r_i)[(p, r_i) \in \mathtt{PA}]\}$
- $\mathtt{operations} : \mathtt{R} \times \mathtt{OBJ} \rightarrow 2^{\mathtt{OP}}$, a function mapping each role r_i and object obj_i to a set of operations.
  $\mathtt{operations}(r_i, obj_i) = \{op \in \mathtt{OP} \mid (op, obj_i, r_i) \in \mathtt{PA}\}$
- $\mathtt{object} : \mathtt{P} \rightarrow 2^{\mathtt{OBJ}}$, a function mapping each permission p_i to a set of objects.

Figure 3.2: Basic Elements and System Functions : from the RBAC96 Model

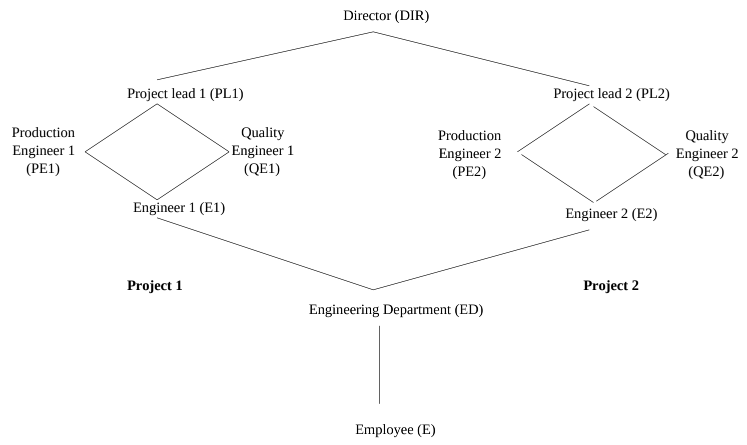


Figure 3.3: Example of role hierarchies

these hierarchies are partial orders. A partial order is a reflexive, transitive, and antisymmetric relation, so that if $y \prec x$ then role x inherits the permissions of role y, but not vice versa. In figure 3.3, the junior-most role is that of employee. The engineering department role is senior to employee and thereby inherits all permissions from employee. The engineering department role can have permissions besides those it inherited. Permission inheritance is transitive, for example, the engineer1 role inherits permissions from both the engineering department and employee roles. Engineer1 and engineer2 both inherit permissions from the engineering department role, but each will have different permissions directly assigned to it.

The user assignment relation `UA` is a many-to-many relation between users and roles. Similarly the permission-assignment relation `PA` is a many-to-many relation between permissions and roles. Users are authorized to use the permissions of roles to which they are assigned. This is the essence of RBAC.

The remaining functions defined in figure 3.2 are built from the sets, relations and functions discussed above. In particular, note that the `roles` function can have different types of arguments so we are overloading this symbol. Also the definition

- $\texttt{CR}$ = a collection of conflicting role sets, $\{cr_1, ..., cr_s\}$, where $cr_i = \{r_i, ..., r_t\} \subseteq \texttt{R}$
- $\texttt{CP}$ = all conflicting permission sets, $\{cp_1, ..., cp_u\}$, where $cp_i = \{p_i, ..., p_v\} \subseteq \texttt{P}$
- $\texttt{CU}$ = all conflicting user sets, $\{cu_1, ..., cu_w\}$, where $cu_i = \{u_i, ..., u_x\} \subseteq \texttt{U}$
- $\texttt{oneelement}(\mathrm{X}) = x_i$, where $x_i \in \mathrm{X}$
- $\texttt{allother}(\mathrm{X}) = \mathrm{X} - \{\texttt{OE}(\mathrm{X})\}$

Figure 3.4: Basic Elements and Non-deterministic Functions: beyond RBAC96 Model

of `roles`* is carefully formulated to reflect the role inheritance with respect to users and sessions going downward and with respect to permissions going upward. In other words a permission in a junior role is available to senior roles, and activation of a senior role makes available permissions of junior roles. This is a well-accepted concept in the RBAC literature and is a feature of RBAC96. Using a single symbol `roles`* simplifies our notation so long as we keep this duality of inheritance in mind.

In *RCL 2000*, for simplicity, we assume that the role hierarchy, user-assignment, and permission-assignment relations do not change. We consider one snapshot in a system at a time and SOD properties are applied only to that snapshot.[2]

3.1.2 Basic Elements and Non-deterministic Functions: beyond RBAC96

Additional elements and system functions used in *RCL 2000* are defined in figure 3.4. For mutually disjoint organizational roles such as those of purchasing manager and accounts payable manager, the same individual is generally not permitted to belong to both roles. We defined these mutually disjoint roles as conflicting roles. We assume

[2]In order to handle such changes we can consider several alternatives. For example, we can simply apply such changes to all the current sessions which are active under certain constraints and deactivate the sessions which violate the constraints. Or, we just apply such changes to new sessions which are activated after the changes. We can apply either alternative based on the organization's policy.

that there is a collection `CR` of sets of roles which have been defined to conflict.

The concept of conflicting permissions defines conflict in terms of permissions rather than roles. Thus, the permission to issue purchase orders and the permission to issue payments are conflicting, irrespective of the roles to which they are assigned. We denote sets of conflicting permissions as `CP`. As we will see, defining conflict in terms of permissions offers greater assurance than defining it in terms of roles. Conflict defined in terms of roles allows conflicting permissions to be assigned to the same role by error (or malice). Conflict defined in terms of permissions eliminates this possibility.

In the real world, conflicting users should be also considered. For example, for the process of preparing and approving purchase orders, it might be company policy that members of the same family cannot prepare the purchase order and also be a user who approves that order. This kind of SOD property is not discussed in [GGF98, SZ97]. The concept is considered in [NO99], but in a general way without identification of specific properties in this class. We denote sets of conflicting users as `CU`.

These sets, such as `CR`, `CP`, and `CU`, are based on SOD properties. In order to specify other constraints, we also may need the additional sets which can be derived from the basic sets described in figure 3.2. We call this type of set administrative set (`AS`). Our language *RCL 2000* also allows us to use this administrative set. From this moment, we just assume that if the new set other than basic set is introduced and the elements of such set should also be elements of one of basic sets, the new set is `AS`. We do not explicitly mention as such whenever the new set is introduced.

RCL 2000 has two non-deterministic functions, `oneelement` and `allother` (first introduced by Chen and Sandhu [CS95]). These are introduced to replace explicit quantifiers. The elimination of explicit quantifiers from our language keeps it simple and intuitive. The `oneelement`(X) function allows us to get one element x_i from set

X. We usually write `oneelement` as `OE`. Multiple occurrences of $\texttt{OE}(\mathrm{X})$ in a single *RCL 2000* statement all select the same element x_i from X. With $\texttt{allother}(\mathrm{X})$ we can get a set by taking out one element. We usually write `allother` as `AO`.

These two non-deterministic functions are related by context, because for any set S, $\{\texttt{OE}(S)\} \cup \texttt{AO}(S) = S$, and at the same time, neither is a deterministic function. In order to illustrate how to use these two functions to specify SOD properties, we take the requirement of static separation of duty property which is the simplest variation of SOD. For the moment assume there is no role hierarchy.

> `Requirement`: *No user can be assigned to two conflicting roles.* In other words, conflicting roles cannot have common users. We can express this requirement as below.
>
> `Expression`: $\mid \texttt{roles}(\texttt{OE}(\texttt{U})) \cap \texttt{OE}(\texttt{CR}) \mid \leq 1$

$\texttt{OE}(\texttt{CR})$ means a conflicting role set and the function $\texttt{roles}(\texttt{OE}(\texttt{U}))$ returns all roles which are assigned to a single user $\texttt{OE}(\texttt{U})$. $\texttt{AO}(\texttt{OE}(\texttt{CR}))$ means conflicting roles, excluding one role which is from $\texttt{OE}(\texttt{OE}(\texttt{CR}))$. Therefore this statement ensures that a single user cannot have more than one conflicting role from the specific role set $\texttt{OE}(\texttt{CR})$. We can interpret the above expression as saying that if a user has been assigned to one conflicting role, that user cannot be assigned to any other conflicting role. We can also specify this property in many different ways using *RCL 2000*, such as $\texttt{OE}(\texttt{OE}(\texttt{CR})) \in \texttt{roles}(\texttt{OE}(\texttt{U})) \implies \texttt{AO}(\texttt{OE}(\texttt{CR})) \cap \texttt{roles}(\texttt{OE}(\texttt{U})) = \phi$ or $\texttt{user}(\texttt{OE}(\texttt{OE}(\texttt{CR}))) \cap \texttt{user}(\texttt{AO}(\texttt{OE}(\texttt{CR}))) = \phi$

RCL 2000 system functions do not include a time or state variable in their structure. So we assume that each function considers the current time or state. For example, if we use `sessions` function in the expression, this function maps a user u_i to a set of current sessions which are established by user u_i.

As a general notational device we have the following convention.

- For any set valued function f defined on set X,

  We understand $f(\mathrm{X}) = f(x_1) \cup f(x_2) \cup ... \cup f(x_n)$, where X=$\{x_1, x_2, x_3, ..., x_n\}$.

For example, we want to get all users who are assigned to a set of roles $\mathtt{R} = \{employee, engineer1, engineer2\}$. We can express it using the function `user` such as `user(R)`. And `user(R)` is equivalent to $\mathtt{user}(employee) \cup \mathtt{user}(engineer1) \cup \mathtt{user}(engineer2)$.

In this section we have described the basic components of specification language *RCL 2000*. This language is built on RBAC96 components and has two non-deterministic functions `OE` and `AO`. Most of the basic elements are described in set theory [End77, Kec95, Vau95, Gol96]. The following section discusses the syntax which *RCL 2000* should follow to specify role-based authorization constraints.

3.2 Formal Syntax and Semantics of *RCL 2000*

We now provide the formal syntax and semantic of *RCL 2000*. For the syntax we use usual Backus Normal Form (BNF). For semantics of *RCL 2000* we identify a restricted form of first order predicate logic which is exactly equivalent to *RCL 2000*.

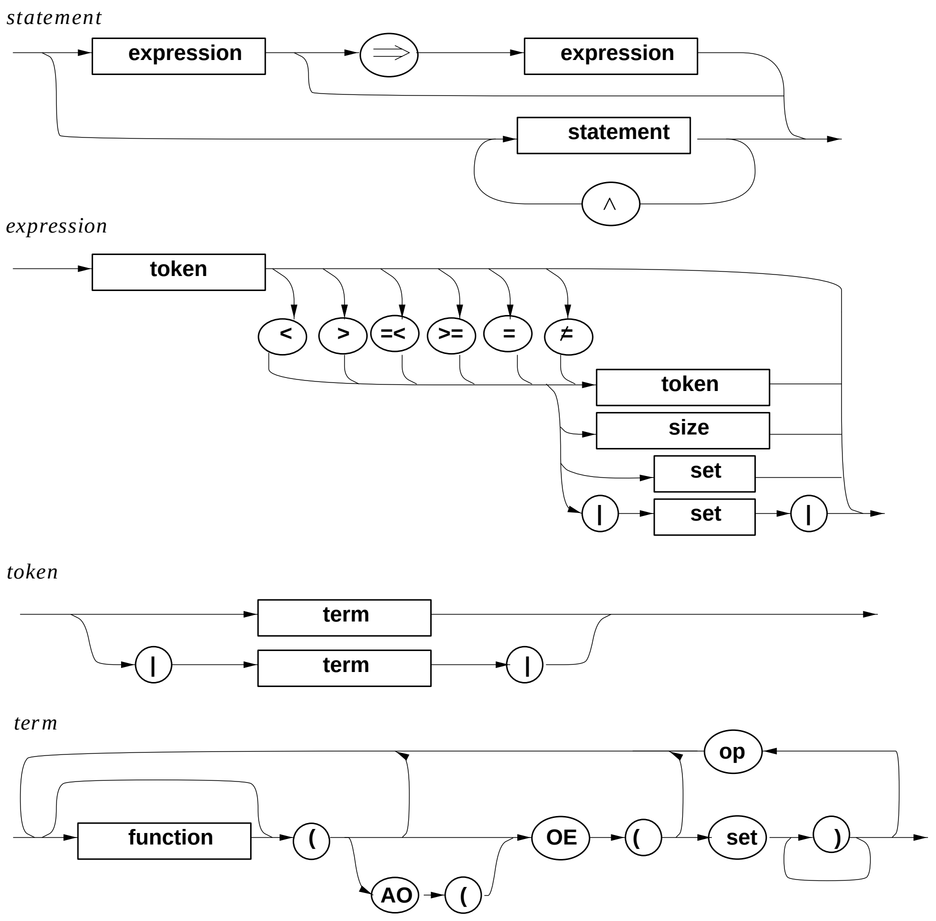
3.2.1 The Syntax

The syntax of *RCL 2000* is defined by the syntax diagram and grammar given in figure 3.5. The rules take the form of flow diagrams. The possible paths represent the possible sequence of symbols. Starting at the beginning of a diagram, a path is followed either by transferring to another diagram if a rectangle is reached or by reading a basic symbol contained in a circle. Backus Normal Form (BNF) is also used to describe the grammar of *RCL 2000* as shown in the bottom of figure 3.5. The symbols of this form are: ::= meaning "is defined as" and | meaning "or." Figure 3.5 shows that *RCL 2000* statements consist of the expression followed by implication ($\Longrightarrow$) and another expression, or expression itself. Also *RCL 2000* statements can be recursively used with logical AND operator ($\wedge$). Each expression consists of the token followed by a comparison operator and token, size, set, or set with cardinality. Also token itself can be expression. Each token can be just a term or a term with cardinality. Each term consists of functions and sets including set operators. Those sets and system functions described in section 3.1 are allowed in this syntax. Also, we denote `oneelement` and `allother` as `OE` and `AO` respectively. We assume that the type of arguments of functions should follow the function descriptions presented in section 3.

3.2.2 Formal Semantics

Next, we discuss the formal semantics for *RCL 2000*. Any property written in *RCL 2000*, called *RCL 2000* expression, can be translated to an equivalent expression which is written in a restricted form of first order predicate logic which we call RFOPL. The syntax of RFOPL is described at the end of this section. The translation algorithm we developed, namely *Reduction*, converts a *RCL 2000* expression to an equivalent RFOPL expression. This algorithm is outlined in figure 3.6. *Reduction* algorithm eliminates `AO` function(s) from *RCL 2000* expression in the first step. Then we translate `OE` terms iteratively into an element introducing universal quantifiers from left to right. If we have nested `OE` functions in *RCL 2000* expression, translation will be started from innermost `OE` terms. The analysis of the running time depends on the number of `OE` terms. Therefore, this algorithm can translate *RCL 2000* expression to RFOPL expression in time $\mathcal{O}$(n), supposing that the number of `OE` term is n.

For example, the following expression can be converted to RFOPL expression accord-



op ::=∈| ∩ | ∪

size ::= ϕ | 1 | ... | N

set ::= `U` | `R` | `OP` | `OBJ` | `P` | `S` | `AS`

function ::= `user` | `roles` | `roles`* | `sessions` | `permissions` | `permissions`* |
`operations` | `object` | `OE` | `AO`

Figure 3.5: Syntax of Language

```
*Reduction* **Algorithm**
Input: *RCL 2000* expression ; Output: RFOPL expression

Let Simple-OE term be either OE(set), or OE(function(element)). Where
set is an element of {U, R, OP, OBJ, P, S, CR, CU, CP, T, HU, HS, cr, cu, cp}. function
is an element of {user, roles, roles*, sessions, permissions, permissions*,
operations, object}

1. AO elimination
        replace all occurrences of AO(expr) with (expr - {OE(expr)});

2. OE elimination
   While There exists Simple-OE term in RCL 2000 expression
       choose Simple-OE term;
       call reduction procedure;
   End

   Procedure reduction
       case (i) Simple-OE term is OE(set)
        create new variable $x$;
        put $\forall x \in set$ to right of existing quantifier(s);
        replace all occurrences of OE(set) by $x$;

      case (ii) Simple-OE term is OE(function(element))
        create new variable $x$;
        put $\forall x \in$ function(element) to right of existing quantifier(s);
        replace all occurrences of OE(function(element)) by $x$;
   End
```

Figure 3.6: Reduction

```
Construction Algorithm
Input: RFOPL expression ; Output: RCL 2000 expression

1. Construction RCL 2000 expression from RFOPL expression
   While There exists a quantifier in RFOPL expression
       choose the rightmost quantifier $\forall\, x \in$ X;
       pick values $x$ and X from the chosen quantifier;
       replace all occurrences of $x$ by OE(X);
   End

3. Replacement of AO
   if there is (expr - {OE(expr)}) in RFOPL expression
    replace it with AO(expr);
```

Figure 3.7: Construction

ing to the sequences below.

Example 1

$\texttt{OE}(\texttt{OE}(\texttt{CR})) \in \texttt{roles}(\texttt{OE}(\texttt{U})) \Longrightarrow \texttt{AO}(\texttt{OE}(\texttt{CR})) \cap \texttt{roles}(\texttt{OE}(\texttt{U})) = \phi$

---

1. $\texttt{OE}(\texttt{OE}(\texttt{CR})) \in \texttt{roles}(\texttt{OE}(\texttt{U})) \Longrightarrow (\texttt{OE}(\texttt{CR}) - \{\texttt{OE}(\texttt{OE}(\texttt{CR}))\}) \cap \texttt{roles}(\texttt{OE}(\texttt{U})) = \phi$

2. $\forall\, cr \in \texttt{CR}$: $\texttt{OE}(cr) \in \texttt{roles}(\texttt{OE}(\texttt{U})) \Longrightarrow (cr - \{\texttt{OE}(cr)\}) \cap \texttt{roles}(\texttt{OE}(\texttt{U})) = \phi$

3. $\forall\, cr \in \texttt{CR}, \forall\, r \in cr$: $r \in \texttt{roles}(\texttt{OE}(\texttt{U})) \Longrightarrow (cr - \{r\}) \cap \texttt{roles}(\texttt{OE}(\texttt{U})) = \phi$

4. $\forall\, cr \in \texttt{CR}, \forall\, r \in cr, \forall\, u \in \texttt{U}$: $r \in \texttt{roles}(u) \Longrightarrow (cr - \{r\}) \cap \texttt{roles}(u) = \phi$

Example 2

$\texttt{OE}(\texttt{OE}(\texttt{CR})) \in \texttt{roles}(\texttt{OE}(\texttt{sessions}(\texttt{OE}(\texttt{U}))))\Longrightarrow \texttt{AO}(\texttt{OE}(\texttt{CR}))\cap\texttt{roles}(\texttt{OE}(\texttt{sessions}(\texttt{OE}(\texttt{U}))))=\phi$

---

1. $\texttt{OE}(\texttt{OE}(\texttt{CR})) \in \texttt{roles}(\texttt{OE}(\texttt{sessions}(\texttt{OE}(\texttt{U})))) \Longrightarrow (\texttt{OE}(\texttt{CR}) - \{\texttt{OE}(\texttt{OE}(\texttt{CR}))\})$

   $\cap\ \texttt{roles}(\texttt{OE}(\texttt{sessions}(\texttt{OE}(\texttt{U}))))=\phi$

2. $\forall\, cr \in \texttt{CR}$: $\texttt{OE}(cr) \in \texttt{roles}(\texttt{OE}(\texttt{sessions}(\texttt{OE}(\texttt{U})))) \implies (cr - \{\texttt{OE}(cr)\})$ $\cap\ \texttt{roles}(\texttt{OE}(\texttt{sessions}(\texttt{OE}(\texttt{U}))))=\phi$

3. $\forall\, cr \in \texttt{CR}$, $\forall\, r \in cr$: $r \in \texttt{roles}(\texttt{OE}(\texttt{sessions}(\texttt{OE}(\texttt{U})))) \implies (cr - \{r\}) \cap \texttt{roles}(\texttt{OE}(\texttt{sessions}(\texttt{OE}(\texttt{U})))) = \phi$

4. $\forall\, cr \in \texttt{CR}$, $\forall\, r \in cr$, $\forall\, u \in \texttt{U}$: $r \in \texttt{roles}(\texttt{OE}(\texttt{sessions}(u)) \implies (cr - \{r\}) \cap \texttt{roles}(\texttt{OE}(\texttt{sessions}(u)) = \phi$

5. $\forall\, cr \in \texttt{CR}$, $\forall\, r \in cr$, $\forall\, u \in \texttt{U}$, $\forall\, s \in \texttt{sessions}(u)$, $r \in \texttt{roles}(s) \implies (cr - \{r\}) \cap \texttt{roles}(s) = \phi$

Example 3

$|\ \texttt{roles}(\texttt{OE}(\texttt{U})) \cap \texttt{OE}(\texttt{CR})\ | \leq 1$

---

1. $\forall\, u \in \texttt{U}$, $|\ \texttt{roles}(u) \cap \texttt{OE}(\texttt{CR})\ | \leq 1$

2. $\forall\, u \in \texttt{U}$, $\forall\, cr \in \texttt{CR}$, $|\ \texttt{roles}(u) \cap cr\ | \leq 1$

The resulting RFOPL expression will have the following general structure.

1. The RFOPL expression has a (possibly empty) sequence of universal quantifiers as a left prefix, and these are the only quantifiers it can have. We call this sequence the *quantifier part*.

2. The quantifier part will be followed by a predicate separated by a colon (:), i.e., *universal quantifier part* **:** *predicate*

3. The predicate has no free variables or constant symbols. All variables are declared in the quantifier part, e.g., $\forall\, r \in \texttt{R}$, $\forall\, u \in \texttt{U}$: $r \in \texttt{roles}(u)$.

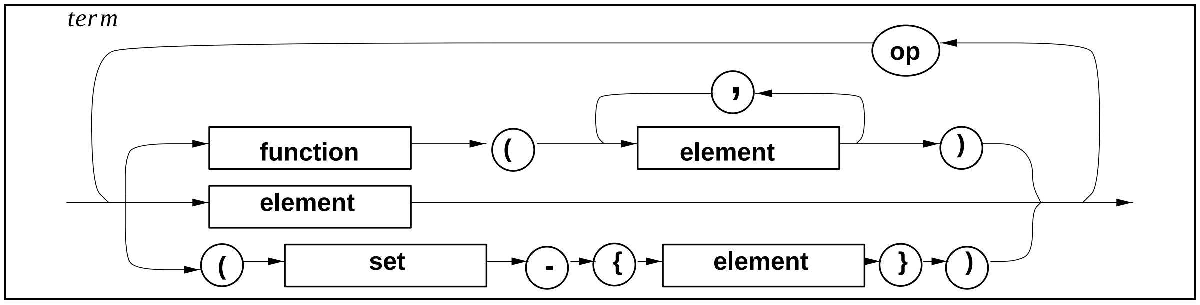


Figure 3.8: Syntax of restricted FOPL expression

4. The order of quantifiers is determined by the sequence of `OE` elimination. In some cases this order is important so as to reflect the nesting of `OE` terms in the *RCL 2000* expression. For example, in $\forall\, cr \in$ `CR`, $\forall\, r \in cr$,$\forall\, u \in$ `U` : *predicate*; the set cr, which is used in the second quantifier, must be declared in a previous quantifier as an element, such as cr in the first quantifier.

5. *Predicate* follows most of rules in the syntax of *RCL 2000* except *term* syntax in figure 3.5. Figure 3.8 shows the syntax which *predicate* should follow to express *term*.

The above discussion defines the syntax of RFOPL. A complete formal definition can be given from these observation and is omitted for simplicity.

Through reduction algorithm $\mathcal{R}$ we may have several RFOPL expressions that are equivalent to a *RCL 2000* expression. We introduce the following lemma, where $\mathcal{R}$(*expr*) denotes the RFOPL expression translated by *Reduction* algorithm.

Lemma 1 *If* $\mathcal{R}(\alpha)$ *gives us* β_1 *and* β_2, $\beta_1 \neq \beta_2$ *then* $\beta_1 \equiv \beta_2$.

Proof: Given the *RCL 2000* expression α, we reduce the `OE` term from it, replacing it with a variable. This variable in β_1 and β_2 might be different because **While-End** loop in reduction algorithm has non-deterministic choice for reduction of `OE` term. Clearly variable names do not matter for equivalence. And the arrangement of quantifiers might also be different but it does not affect the structure of RFOPL expression β_1 and β_2 because the reduction of `OE` term is always from the innermost one first. The order of quantifiers is important for nested `OE` term otherwise order does not matter. Therefore, we can get β_1 which is logically equivalent to β_2. A formal induction based on these arguments can be developed. $\square$

3.2.3 Soundness and Completeness

Next, we discuss the algorithm *Construction* that constructs a *RCL 2000* expression from a RFOPL expression which has the syntax given above. The algorithm is described in figure 3.7. Firstly, this algorithm repeatedly chooses the rightmost quantifier in RFOPL expression and constructs the corresponding `OE` term by eliminating the variable of that quantifier. After all quantifiers are eliminated the algorithm constructs `AO` terms according to the formal definition of `AO` function. The running time of the algorithm obviously depends on the number of quantifiers in RFOPL expression.

For example, the following RFOPL expression can be converted to *RCL 2000* expression according to the sequences described below.

RFOPL expression:

$\forall\, cr \in \texttt{CR},\ \forall\, r \in cr,\ \forall\, u \in \texttt{U},\ \forall\, s \in \texttt{sessions}(u) : r \in \texttt{roles}(s) \Longrightarrow (cr - \{r\}) \cap \texttt{roles}(s) = \phi$

RCL 2000 expression :

1. $\forall\, cr \in \texttt{CR},\ \forall\, r \in cr,\ \forall\, u \in \texttt{U}$: $r \in \texttt{roles(OE(sessions}(u))) \Longrightarrow (cr - \{r\})$ $\cap\ \texttt{roles(OE(sessions}(u))){=}\phi$

2. $\forall\, cr \in \texttt{CR},\ \forall\, r \in cr$: $r \in \texttt{roles(OE(sessions(OE(U))))} \Longrightarrow (cr - \{r\})$

$\cap$ `roles(OE(sessions(OE(U))))`$=\phi$

3. $\forall\, cr \in$ `CR`: `OE(`cr`)` $\in$ `roles(OE(sessions(OE(U))))` $\Longrightarrow$ $(cr - \{$`OE(`cr`)`$\})$
   $\cap$ `roles(OE(sessions(OE(U))))`$=\phi$

4. `OE(OE(CR))` $\in$ `roles(OE(sessions(OE(U))))` $\Longrightarrow$ (`OE(CR)` $-$ $\{$`OE(OE(CR))`$\}$)
   $\cap$ `roles(OE(sessions(OE(U))))`$=\phi$

5. `OE(OE(CR))` $\in$ `roles(OE(sessions(OE(U))))` $\Longrightarrow$ `AO(OE(CR))`
   $\cap$ `roles(OE(sessions(OE(U))))`$=\phi$

Unlike the reduction algorithm we can observe the following lemma, where $\mathcal{C}(expr)$ denotes the *RCL 2000* expression constructed by *Construction* algorithm.

Lemma 2 *$\mathcal{C}(\beta)$ always gives us the same* RCL 2000 *expression α.*

Proof: Construction algorithm always choose the rightmost quantifiers to construct *RCL 2000* expression from RFOPL expression. This procedure is deterministic. Therefore, given RFOPL expression β, we will always get the same *RCL 2000* expression α. $\square$

We introduced two algorithms, namely *Reduction* and *Construction*, that can reduce and construct *RCL 2000* expression. In order to show the soundness and completeness, we introduce lemmas and theorems regarding translation between *RCL 2000* and RFOPL expressions. The relationship between *RCL 2000*-RFOPL translations above is expressed by Soundness and Completeness theorems. We also define the expression which can be generated during reduction and construction as an intermediate expression called *IE*. This expression will have the mixture form of *RCL 2000* and RFOPL expression, that is it will contain quantifiers as well as `OE` terms.

3.2.3.1 Soundness Theorem

In order to show the soundness of *RCL 2000*, we introduce the following lemma.

Lemma 3 *If the intermediate expression γ is derived from* RCL 2000 *expression α by reduction algorithm in k iterations then construction algorithm applied to γ will terminate in exactly k iterations.*

Proof: It is obvious that γ has k quantifiers because the reduction algorithm generates exactly one quantifier for each iteration. Now the construction algorithm eliminates exactly one quantifier per iteration, and will therefore terminate in k iterations.

$\square$

Based on the lemma above, we introduce the following theorem. We define all the occurrences of same `OE` term in the expression as a distinct `OE` term.

Theorem 1 *Given* RCL 2000 *expression α, α can be translated into RFOPL expression β. Also α can be reconstructed from β.*

$$\mathcal{C}(\mathcal{R}(\alpha)) = \alpha$$

Proof: We will prove the stronger result that $\mathcal{C}^n(R^n(\alpha)) = \alpha$ by induction on the number of iterations in reduction $\mathcal{R}$ (or, $\mathcal{C}$). We define $\mathcal{C}^n$ as n iterations of reduction algorithm, and $\mathcal{R}^n$ as n iterations of reduction algorithm.

Basis: If the number of iteration n is 0, the theorem follows trivially.

Inductive Hypothesis: We assume that if n=k, this theorem is true.

Inductive Step: Consider the intermediate expression γ translated by reduction algorithm in $k + 1$ iterations. Let $\gamma^{'}$ be the expression and intermediate expression translated by reduction algorithm in the k^{th} iteration. γ differs from $\gamma^{'}$ in having an additional rightmost quantifier and one less distinct `OE` term. Applying the construction algorithm to γ, eliminates this rightmost quantifier and brings back the same

`OE` term in all its occurrences. Thus the construction algorithm applied to γ gives us $\gamma^{'}$. From this intermediate expression $\gamma^{'}$, we can construct α due to the inductive hypothesis. This completes the inductive proof. $\square$

3.2.3.2 Completeness Theorem

In order to show the completeness of *RCL 2000*, we also introduce the following lemmas.

Lemma 4 *If the intermediate expression γ is derived from RFOPL expression β by construction algorithm in k iterations then reduction algorithm applied to γ will terminate in exactly k iterations.*

Proof: It is obvious that γ has k distinct `OE` terms because the construction algorithm generates exactly one distinct `OE` term for each iteration. Now the reduction algorithm eliminates exactly one distinct `OE` term per iteration, and will therefore terminate in k iterations. $\square$

Let us say two intermediate expressions γ_1 and γ_2 are equivalent if $\mathcal{R}(\gamma_1) \equiv \mathcal{R}(\gamma_2)$.

Lemma 5 *Lemma 1 extends to the intermediate expression. if $\mathcal{R}^k(\alpha) = \gamma_1$ and $\mathcal{R}^k(\alpha) = \gamma_2$, where $\mathcal{R}^k$ is defined as k iterations of reduction algorithm, then γ_1 and γ_2 are equivalent.*

Proof: Follows from **Lemma 1** $\square$

Lemma 6 *There exists an execution of $\mathcal{R}$ such that $\mathcal{R}(\mathcal{C}(\beta)) = \beta$*

Proof: We prove the stronger result that there is an execution of $\mathcal{R}$ such that $\mathcal{R}^n(C^n(\beta)) = \beta$ by induction on the number of iterations in construction $\mathcal{C}$ (or,

$\mathcal{R}$). We define $\mathcal{C}^n$ as n iterations of reduction algorithm, and $\mathcal{R}^n$ as n iterations of reduction algorithm.

Basis: If the number of iteration n is 0, the theorem follows trivially.

Inductive Hypothesis: We assume that if n=k, this theorem is true.

Inductive Step: Consider the intermediate expression γ constructed by construction algorithm in $k+1$ iterations. Let $\gamma^{'}$ be intermediate expression after the k^{th} iteration. γ differs from $\gamma^{'}$ in having one less distinct `OE` term and one more distinct `OE` term. Applying one iteration of the reduction algorithm to γ, eliminates this particular `OE` term and introduce the same variable in the new rightmost quantifier. This gives us $\gamma^{'}$. By inductive hypothesis from $\gamma^{'}$ there is an execution of $\mathcal{R}^k$ that will give us β.

□

Now, we introduce the theorem which shows the completeness of *RCL 2000*, relative to RFOPL.

Theorem 2 *Given RFOPL expression β, β can be translated into* RCL 2000 *expression α. Also $\beta^{'}$ which is logically equivalent to β can be reconstructed from α.*

$$\mathcal{R}(\mathcal{C}(\beta)) = \beta^{'}$$

Proof: **Lemma 2** states that $\mathcal{C}(\beta)$ gives us a unique result. Let us call it α. **Lemma 6** states there is an execution of $\mathcal{R}$ that will go back exactly to β from α. **Lemma 1** states that all executions of for α will give an equivalent RFOPL expression. The theorem follows. □

Theorem 1 establishes soundness of *RCL 2000* and theorem 2 establishes its completeness relative to RFOPL.

3.3 Packaging Method

Using *RCL 2000* we can specify constraints in role-based systems. We may need to combine the identified constraints together while we can identify the wide variety of constraints. This kind of composition between constraints is helpful to implement a stronger property.

For example, we have two constraints such as *Constraint1* and *Constraint2*. *Constraint1* requires that conflicting roles cannot have common users. And *Constraint2* requires the following requirement: conflicting permission cannot be assigned to common roles. From *Constraint2* constraint we can have a stronger property called *Constraint3* which requires that conflicting permission cannot be assigned to common roles and conflicting roles cannot have common users. This composition of *Constraint1* and *Constraint2* allows us to find another useful and stronger property.

In this section we introduce the *packaging* method for constraints composition. We consider four criteria to provide constraints packaging. Those criteria are as follows:

1. We need to specify the name of property
2. We need to embed other property
3. We need to declare sets which are used in property
4. We need to declare sets which are used in embedded property

From this observation, the packaging construction follows the format as shown below.

```
Property [property name] is
         Set:
         Do:
         Include: none or property name
End [property name]
```

Each property has three parts : `Set` which is the set name used in property, `Do` which is the constraint statement, and `Include` which is the name of the property to be composed. Our usage of packaging method is limited within similar properties.

Let's take the example described above. The requirement of *Constraint1* is conflicting roles cannot have common users. In order to express this constraint, we need conflicting role sets and constraint expression as below. `CR` set is required in constraint statement and this property does not include any other property in it.

```
Property Constraint1 is
          Set: CR = $\{cr_1, ..., cr_n\}$, where $cr_i = \{r_i, ..., r_k\} \subseteq$ R
          Do: $|$ roles(OE(U)) $\cap$ OE(CR) $| \leq 1$
          Include: none
End Constraint1
```

And *Constraint2* requires conflicting roles cannot have common users. In order to express this constraint, we need conflicting permission sets and constraint expression as below. `CP` set is required in constraint statement.

```
Property Constraint2 is
          Set: CP = $\{cp_1, ..., cp_n\}$, where $cp_i = \{p_i, ..., p_k\} \subseteq$ P
          Do: $|$ permissions(OE(R)) $\cap$ OE(CP) $| \leq 1$
          Include: none
End Constraint2
```

As we mentioned earlier, *Constraint3* needs *Constraint1* and *Constraint2* property. *Constraint3* requires the following requirement: conflicting permission cannot be assigned to common roles and conflicting roles cannot have common users. Therefore, Constraint3 property means $| \texttt{roles(OE(U))} \cap \texttt{OE(CR)} | \leq 1 \wedge | \texttt{permissions(OE(R))} \cap \texttt{OE(CP)} | \leq 1$. We can express this using the packaging method. *Constraint3* can derived from *Constraint2*, adding *Constriant1*. The following statement is the same expression as *Constraint2* except for `Include:` part.

```
Property Constraint3 is
         Set: CP = $\{cp_1, ..., cp_n\}$, where $cp_i = \{p_i, ..., p_k\} \subseteq$ P
         Do: $|$ permissions(OE(R)) $\cap$ OE(CP) $| \leq 1$
         Include:  Constraint1
End Constraint3
```

Constraint3 property requires two sets CP and CR. We came up with one reason such as that *Constraint3* property includes *Constraint1* property but *Constraint3* might need a different set(s) from sets which are used in *Constraint1* property. For example, *Constraint3* property requires CP and CR sets. This CR set is inherited from *Constraint1* property. For organization policy, *Constraint3* property might need to modify this CR set. In order to support this, we introduce instantiation method. This method allows us to modify the embedded set such as CR set in *Constraint3* property. The following shows the structure of instantiation method.

```
Instantiation [Property Name] is
         Set : set_name = [property_name.set_name [$\cup$|$\cap$|-]] {...}
End [Property Name]
```

From this structure, *Constraint3* can modify CR set with one of three major ways, as below.

- Set : CR = Constraint1.CR

  This statement means that *Constraint3* just uses *Constraint1*'s CR set within *Constraint1* property.

- Set : CR = $\{r_\alpha, r_\beta\}$

  This statement implies that *Constraint3* requires different CR set from *Constraint1* property.

- `Set : CR` $=$ `SSOD.CR` $\cup \{r_\alpha, r_\beta\}$

  This instantiation statement adds two more roles in *Constraint1*'s `CR` set which is used in *Constraint3* property. The following shows an example to express this instantiation.

```
Instantiation Constraint3 is
        Set : CR = Constraint1.CR ∪ {r_α, r_β}
End SSOD
```

Using this packaging method, constraint specifiers can embed one constraint in other constraints providing flexibility of sets to be selected.

3.4 Summary

We have described the specification language *RCL 2000*. This language is built on RBAC96 components and has two non-deterministic functions `OE` and `AO`. Most of the basic elements are described in set theory.

In this chapter we have given a formal syntax and semantics for *RCL 2000* and have demonstrated its soundness and completeness. Any property written in *RCL 2000* could be translated to an expression which is written in a restricted form of first order predicate logic, which we call RFOPL. During the analysis of this translation, we proved two theorems which support the soundness and completeness of the specification language *RCL 2000* and RFOPL respectively. In the final section we described the packaging method between *RCL 2000* expression which allows us to compose and reuse constraints.

Chapter 4

SPECIFICATION OF SOD CONSTRAINTS

Separation of duty (SOD) reduces the possibility of fraud or significant errors which can cause damage to an organization by partitioning of tasks and associated privileges required to complete a task or set of related tasks. Role-based separation of duty enforces SOD requirements in role-based environment by controlling membership in and use of roles as well as permission assignment.

As we mentioned earlier, we assume that the role hierarchy, user-assignment, and permission-assignment relation do not change in *RCL 2000* for simplicity. We consider one snapshot in a system at a time and SOD properties are applied only to that snapshot.

In this chapter, we show how *RCL 2000* can be used to specify various role-based separation of duty properties. Our work builds upon SOD properties analyzed in [SZ97] and formalized in [GGF98]. However, these papers do not have the notion of role hierarchies. They miss the concept of session-based SOD which deals with SOD property in a single session. This form of dynamic SOD is useful for simulating Lattice-based access control and Chinese Walls in RBAC [San93, San96]. Conflicting users and privileges are also not dealt with. From these observations, we are led to identify other significant SOD properties which have not been previously identified in the literature. Many of the SOD properties discussed in this chapter are recognized in [GGF98]. Properties not recognized in [GGF98] are explicitly identified as such. Our specification language can be used to express SOD properties in different ways. We usually show only one specification in each case. We also show a few SOD properties with conflicting users and privileges.

4.1 SOD Properties ($RBAC_2$)

In this section we first look at the properties recognized in [GGF98] and specify these properties in *RCL 2000*. In order to show that *RCL 2000* can specify [GGF98]'s properties, we begin specifying properties without the notion of role hierarchies so we are in $RBAC_1$ of RBAC96 family. We discuss SOD properties with the notion of conflicting roles and we call these properties role-centric SOD. We also discuss SOD properties with the notion of conflicting users and conflicting permission and we call these properties user-centric and permission-centric SOD, respectively.

4.1.1 Role-centric SOD

In this section we specify a variety of SOD properties based on the notion of conflicting roles.

Static SOD

1. Simple Static SOD

   Static SOD is the simplest variation of SOD. This simple static SOD property requires that no user should be assigned to two conflicting roles. In other words, conflicting roles cannot have common users. This requirement is expressed as follows:

   $|\,\texttt{roles(OE(U))} \cap \texttt{OE(CR)}\,| \leq 1$

   Alternatively, this property can be stated as

   $\texttt{UR(OE(OE(CR)))} \cap \texttt{UR(AO(OE(CR)))} = \phi$

We can specify this property in many different ways using our language. For simplicity, we use the simple expressions in this dissertation.

$\texttt{OE}(\texttt{CR})$ means a conflicting role set cr_i which is an element of a collection of conflicting role sets $\texttt{CR}$. $\texttt{roles}(\texttt{OE}(\texttt{U}))$ returns all roles which are assigned to a single user $\texttt{OE}(\texttt{U})$. We can interpret this expression as follows: *if a user has been assigned to one conflicting role, that user cannot be assigned to any other conflicting role.*

2. Strict Static SOD

   Strict static SOD requires that conflicting roles are not authorized to perform operations on the same object in addition to property 1. This property is a stronger version of property 1 by adding the requirement that the target object sets of two conflicting roles be disjoint. This requirement is expressed as follows:

   $(1) \wedge$
   $\texttt{operations}(\texttt{OE}(\texttt{OE}(\texttt{CR})), \texttt{OE}(\texttt{OBJ})) \cap \texttt{operations}(\texttt{AO}(\texttt{OE}(\texttt{CR})), \texttt{OE}(\texttt{OBJ})) = \phi$

   $\texttt{operations}(\texttt{OE}(\texttt{OE}(\texttt{CR})), \texttt{OE}(\texttt{OBJ}))$ returns all operations which one conflicting role $\texttt{OE}(\texttt{OE}(\texttt{CR}))$ can perform on an object $\texttt{OE}(\texttt{OBJ})$. $\texttt{operations}(\texttt{AO}(\texttt{OE}(\texttt{CR})), \texttt{OE}(\texttt{OBJ}))$ means all operations which other conflicting roles $\texttt{AO}(\texttt{OE}(\texttt{CR}))$ can perform on an object $\texttt{OE}(\texttt{OBJ})$. That is, this statement says that at most one conflicting role can perform operations on an object.

3. 1-step Strict Static SOD

   1-step strict static SOD make strict SOD property stronger by adding the requirement that each conflicting role execute only one operation on an object. In other words, this property requires that conflicting roles are authorized to perform, at most, one operation in addition to property 1 and 2. This requirement is expressed as follows:

$(1) \wedge (2) \wedge | \texttt{operations}(\texttt{OE}(\texttt{OE}(\texttt{CR})), \texttt{OE}(\texttt{OBJ})) | \leq 1$

In this statement, $| \texttt{operations}(\texttt{OE}(\texttt{OE}(\texttt{CR})), \texttt{OE}(\texttt{OBJ})) |$ means the number of operations which one conflicting role $\texttt{OE}(\texttt{OE}(\texttt{CR}))$ can perform on an object $\texttt{OE}(\texttt{OBJ})$. This number should be less than 1 to satisfy the requirement of this property.

Dynamic SOD

Clark and Wilson [CW87], and then others [GGF98], defined several dynamic separation of duty properties. With *RCL 2000* we can specify these properties given as below.

4. Simple Dynamic SOD

   The simple dynamic SOD requires that there are no users with two conflicting roles enabled. In other words, conflicting roles may have common users but users cannot activate roles which are conflicting with each other. This requirement is expressed as follows:

   $| \texttt{roles}(\texttt{sessions}(\texttt{OE}(\texttt{U}))) \cap \texttt{OE}(\texttt{CR}) | \leq 1$

   $\texttt{OE}(\texttt{CR})$ means a conflicting role set cr_i which is an element of a collection of conflicting role sets $\texttt{CR}$. $\texttt{roles}(\texttt{sessions}(\texttt{OE}(\texttt{U})))$ returns all roles which are activated in sessions invoked by a single user $\texttt{OE}(\texttt{U})$. We can interpret this expression as follows : *if a user activates one conflicting role, that user cannot activate any other conflicting role.*

5. Session-based Dynamic SOD (not in [GGF98])

   Session-based dynamic SOD requires that there are no users with two conflicting roles enabled in a session. This newly identified property ensures that no single session has two conflictings roles activated. This requirement is expressed as follows:

   $| \texttt{roles}(\texttt{OE}(\texttt{sessions}(\texttt{OE}(\texttt{U})))) \cap \texttt{OE}(\texttt{CR}) \mid \leq 1$

   $\texttt{roles}(\texttt{OE}(\texttt{sessions}(\texttt{OE}(\texttt{U}))))$ returns all roles which are activated in a single session invoked by a single user $\texttt{OE}(\texttt{U})$. As we will use in chapter 5, this property is required to simulate Lattice-based access control and Chinese Walls in RBAC.

Object-based SOD

Nash and Poland [NP90] introduced object-based SOD as a more flexible and realistic alternative to the static SOD. Gligor et al [GGF98] try to remove ambiguities in the Nash and Poland paper by formalizing their intuitive definitions. The following two properties are static variants of object-based SOD.

6. Object-based Static SOD

   This property requires that no collection of conflicting roles with a common user is authorized to perform more than one operation on each object. This variant of static SOD is based on a set of roles with common users. This requirement is expressed as follows:

   $| \texttt{operations}(\texttt{OE}(\texttt{OE}(\texttt{CR})) \cap \texttt{roles}(\texttt{OE}(\texttt{U})), \texttt{OE}(\texttt{OBJ})) \mid \leq 1$

   $\texttt{operations}(\texttt{OE}(\texttt{OE}(\texttt{CR})) \cap \texttt{roles}(\texttt{OE}(\texttt{U})), \texttt{OE}(\texttt{OBJ}))$ means that all operations can be performed by users on a single object with a conflicting role.

7. Per-Role Object-based Static SOD

   This property requires that no conflicting roles are authorized to perform more than one operation on an object. Unlike the above property, this property is based on a single roles instead of a set of roles with common users. This requirement is expressed as follows:

   $\mid \texttt{operations}(\texttt{OE}(\texttt{OE}(\texttt{CR})), \texttt{OE}(\texttt{OBJ})) \mid \leq 1$

Operational SOD

Another variation of SOD is the operational SOD.

8. Operational Static SOD

   Operational static SOD requires that any collection of conflicting roles with a common user cannot perform all operations. This requirement is expressed as follows:

   $\mid \texttt{operations}(\texttt{OE}(\texttt{OE}(\texttt{CR})) \cap \texttt{roles}(\texttt{OE}(\texttt{U})), \texttt{OBJ}) \mid < \mid \texttt{OP} \mid$

   $\texttt{operations}(\texttt{OE}(\texttt{OE}(\texttt{CR})) \cap \texttt{roles}(\texttt{OE}(\texttt{U})), \texttt{OBJ})$ means that all operations can be performed by users on a single object with a conflicting role.

9. Per-Role Operational Static SOD

   This property requires that no role can perform all operations. This variant of operational SOD is obtained by applying operational SOD to single roles. This requirement is expressed as follows:

   $\mid \texttt{operations}(\texttt{OE}(\texttt{OE}(\texttt{CR})), \texttt{OBJ}) \mid < \mid \texttt{OP} \mid$

4.1.2 User-centric SOD

In this section we specify SOD properties with the notion of conflicting users. These properties have not been formally discussed in the literature.

10. Static SOD (not in [GGF98])

    Static SOD requires that conflicting users cannot have a common role. This requirement is expressed as follows:

    $\texttt{roles}(\texttt{OE}(\texttt{OE}(\texttt{CU}))) \cap \texttt{roles}(\texttt{AO}(\texttt{OE}(\texttt{CU}))) = \phi$

    $\texttt{roles}(\texttt{OE}(\texttt{OE}(\texttt{CU})))$ means all roles which are assigned to a conflicting user while $\texttt{roles}(\texttt{AO}(\texttt{OE}(\texttt{CU})))$ means all roles which are assigned to other conflicting users in the same conflicting user set cu_i.

11. Dynamic SOD (not in [GGF98])

    Dynamic SOD requires that there are no roles enabled with two conflicting users in sessions. This requirement is expressed as follows:

    $\texttt{roles}(\texttt{sessions}(\texttt{OE}(\texttt{OE}(\texttt{CU})))) \cap \texttt{roles}(\texttt{sessions}(\texttt{AO}(\texttt{OE}(\texttt{CU})))) = \phi$

    $\texttt{roles}(\texttt{sessions}(\texttt{OE}(\texttt{OE}(\texttt{CU}))))$ means that all roles which a conflicting user activates in sessions. $\texttt{roles}(\texttt{sessions}(\texttt{AO}(\texttt{OE}(\texttt{CU}))))$ means all roles which are activated by other conflicting users in the same conflicting users set cu_i.

4.1.3 Permission-centric SOD

In this section we specify SOD properties with the notion of conflicting privileges (permissions). These properties also have not been discussed in the literature.

12. Static SOD (not in [GGF98])

    This static SOD requires that no role may contain two privileges which have been defined to conflicts. In other words, conflicting privilege cannot be assigned to a common role. This requirement is expressed as follows:

    $\texttt{roles}(\texttt{OE}(\texttt{OE}(\texttt{CP}))) \cap \texttt{roles}(\texttt{AO}(\texttt{OE}(\texttt{CP}))) = \phi$

    $\texttt{roles}(\texttt{OE}(\texttt{OE}(\texttt{CP})))$ means all roles which have a conflicting privilege, and $\texttt{roles}(\texttt{AO}(\texttt{OE}(\texttt{CP})))$ stands for all roles which have other conflicting privileges in the same conflicting privileges set cp_i.

13. Dynamic SOD (not in [GGF98])

    Dynamic SOD requires that no roles with conflicting privileges cannot activate in sessions. This requirement is expressed as follows:

    $\mid \texttt{permissions}(\texttt{roles}(\texttt{sessions}(\texttt{OE}(\texttt{U})))) \cap \texttt{OE}(\texttt{OE}(\texttt{CP})) \mid \leq 1$

    This statement ensures that a role cannot perform more than one conflicting privilege from conflicting privileges.

4.2 SOD Properties (RBAC$_3$)

Several recent papers [San98, GB98, Mof98] discuss role hierarchies in terms of separation of duty. In this section we specify properties discussed in 4.1 with the notion of role hierarchies. This section is organized as follows: Section 4.2.1 revisits the notion of role hierarchies and functions described in chapter 3.

We can specify all properties identified in section 4.1 with those functions. In section 4.2.2 we show a few examples since we replace roles by roles* in the most of the case, in order to apply the notion of role hierarchies.

4.2.1 Additional Components

Hierarchies are a natural means for structuring roles to reflect an organization's lines of authority and responsibility. Mathematically, these hierarchies are partial orders. A partial order is a reflexive, transitive, and antisymmetric relation. The formal definition of role hierarchies follows.

- $\mathtt{RH} \subseteq \mathtt{R} \times \mathtt{R}$ is a partial order on $\mathtt{R}$ called the role hierarchy or role dominance relation, written as $\preceq$.

  In addition to this component above, we also need the additional function. *RCL 2000* components `roles` and `permissions` was defined in section 4.1 and are as below.[1]

- $\mathtt{roles}^*(u_i) = \{r \in \mathtt{R} \mid (\exists\, r \preceq r')[(u_i, r') \in \mathtt{UA}]\}$; get all roles assigned to this user in `UA` and roles junior to these.

- $\mathtt{roles}^*(p_i) = \{r \in \mathtt{R} \mid (\exists\, r' \preceq r)[(p_i, r') \in \mathtt{PA}]\}$; get all roles assigned to this permission in `PA` and roles junior to these.

- $\mathtt{roles}^*(s_i) = \{r \in \mathtt{R} \mid (\exists\, r \preceq r')[r' \in \mathtt{roles}(s_i)]\}$; get all roles holding this sessions and roles junior to these.

- $\mathtt{permissions}^*(r_i) = \{p \in \mathtt{P} \mid (\exists\, r \preceq r_i)[(p, r_i) \in \mathtt{PA}]\}$; get all permissions assigned to this role and roles junior to this role.

With theses components, we can specify SOD properties with the notion of role hierarchies. The main changes to our specification from 4.1 is the usage of these new components. For simplicity, we just explain the requirements and describe our expression. These properties have not been discussed in the literature.

[1] In RBAC_3, a function `roles` will get a set of roles which is explicitly assigned to a user.

4.2.2 Role-centric SOD

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

Static SOD

1. Simple Static SOD

   `Requirement`: conflicting roles cannot have common users This requirement is expressed as follows:

   $\mid \texttt{roles}^*(\texttt{OE(U)}) \cap \texttt{OE(CR)} \mid \leq 1$

Dynamic SOD

2. Session-based Dynamic SOD

   `Requirement`: there are no users with two conflicting roles enabled in a session. This requirement is expressed as follows:

   $\mid \texttt{roles}^*(\texttt{OE(sessions(OE(U))))} \cap \texttt{OE(CR)} \mid \leq 1$

Object-based SOD

3. Object-based Static SOD `Requirement`: no collection of conflicting roles with a common user can perform more than one operation on each object. This requirement is expressed as follows:

   $\mid \texttt{operations(OE(OE(CR))} \cap \texttt{roles}^*(\texttt{OE(U))), OE(OBJ))} \mid \leq 1$

Operational SOD

4. Operational Static SOD

   `Requirement`: any collection of conflicting roles with a common user cannot perform all operations. This requirement is expressed as follows:

   $\mid \texttt{operations}(\texttt{OE}(\texttt{OE}(\texttt{CR})) \cap \texttt{roles}^*(\texttt{OE}(\texttt{U})), \texttt{OBJ}) \mid < \mid \texttt{OP} \mid$

4.2.3 User-centric SOD

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

5. Static SOD

   `Requirement`: conflicting users cannot have a common role. This requirement is expressed as follows:

   $\texttt{roles}^*(\texttt{OE}(\texttt{OE}(\texttt{CU}))) \cap \texttt{roles}^*(\texttt{AO}(\texttt{OE}(\texttt{CU}))) = \phi$

4.2.4 Permission-centric SOD

In this section we specify SOD properties with the notion of conflicting privileges (permissions).

6. Static SOD

   `Requirement`: no role can contain two privileges which have been defined to conflicts. This requirement is expressed as follows:

   $\texttt{roles}^*\texttt{OE}(\texttt{OE}(\texttt{CP})) \cap \texttt{roles}^*(\texttt{AO}(\texttt{OE}(\texttt{CP}))) = \phi$

In section 4.1 and 4.2 we have shown how we can specify SOD constraints in a role-based environment using *RCL 2000*. In section 4.3 we discuss SOD properties in a more practical and systematic manner.

4.3 Discussion

SOD is a well-known principle for preventing fraud by identifying conflicting roles—such as Purchasing Manager and Accounts Payable Manager—and ensuring that the same individual can belong to, at most, one conflicting role. Static SOD applies to the user-assignment relation and dynamic SOD applies to the activated roles in session(s). In section 4.1 and 4.2 we started specifying constraints based on SOD properties analyzed in [SZ97] and formalized in [GGF98]. Also we identified new properties such as session-based properties and properties with the notion of role hierarchies, conflicting users, and conflicting privileges.

In this section we discuss the properties which are practical formulations with tradeoff between assurance and flexibility in the context of role-based access control. Also we discuss how *RCL 2000* can be used to specify the various separations of duty properties in a practical way.

4.3.1 Static SOD

Static SOD (SSOD) is the simplest variation of SOD. In table 4.1 we show our expression of several forms of SSOD. These include new forms of SSOD which have not previously been identified in the literature. This demonstrates how *RCL 2000* helps us in understanding SOD and discovering new basic forms of it.

Property 1, $| \texttt{roles}^*(\texttt{OE(U)}) \cap \texttt{OE(CR)} | \leq 1$, is the most straightforward property. The SSOD requirement is that no user should be assigned to two roles which are conflicting with each other. In other words, it means that conflicting roles cannot have common users. *RCL 2000* can clearly express this property. This property is the classic formulation of SSOD, which is identified by several papers including [GGF98, Kuh97, SCFY96]. It is a role-centric property.

Property 2, $| \texttt{permissions}(\texttt{roles}^*(\texttt{OE(U)})) \cap \texttt{OE(CP)} | \leq 1$, follows the same intuition as property 1, but is permission-centric. Property 2 says that a user can have, at most, one conflicting permission acquired through roles assigned to the user. Property 2 is a stronger formulation than property 1, which prevents mistakes in role-permission assignment. This kind of property has not been previously mentioned in the literature. *RCL 2000* helps us discover such omissions in previous work. In retrospect, property 2 is an "obvious property" but there is no mention of this property in over a decade of SOD literature. Even though property 2 allows more flexibility in role-permission assignment, since the conflicting roles are not predefined, it can also generate roles which cannot be used at all. For example, two conflicting permissions can be assigned to a role. Property 2 simply requires that no user can be assigned to such a role or any role senior to it, which makes that role quite useless. Thus, property 2 prevents certain kinds of mistakes in role-permissions but tolerates others.

Property 3, $\mid \texttt{permissions}^*(\texttt{OE(R)}) \cap \texttt{OE(CP)} \mid \leq 1$, eliminates the possibility of useless roles with an extra condition, $\mid \texttt{permissions}^*(\texttt{OE(R)}) \cap \texttt{OE(CP)} \mid \leq 1$. This condition ensures that each role can have, at most, one conflicting permission without consideration of user-role assignment.

With this new condition, we can extend property 1 in the presence of conflicting permissions as property 4. In property 4 we have another additional condition, which is that conflicting permissions can only be assigned to conflicting roles. In other words, non-conflicting roles cannot have conflicting permissions. The net effect is that a user can have one conflicting permission via roles assigned to the user.

Property 4, $\mid \texttt{permissions}^*(\texttt{OE(R)}) \cap \texttt{OE(CP)} \mid \leq 1 \wedge \texttt{permissions}(\texttt{OE(R)}) \cap \texttt{OE(CP)} \neq \phi \Longrightarrow \texttt{OE(R)} \cap \texttt{OE(CR)} \neq \phi$, can be viewed as a reformulation of property 3 in a role-centric manner. Property 3 does not stipulate a concept of conflicting roles. However, we can interpret conflicting roles to be those that happen to have conflicting permissions assigned to them. Thus for every cp_i we can define $cr_i = \{r \in \texttt{R} \mid$

$cp_i \cap \texttt{permissions}(r) \neq \phi\}$. With this interpretation, properties 2 and 4 are essentially identical. The viewpoint of property 3 is that conflicting permissions get assigned to distinct roles which, thereby, become conflicting, and therefore cannot assigned to the same user. Which roles are deemed conflicting is not determined a priori but is a side-effect of permission-role assignment. The viewpoint of property 4 is that conflicting roles are designated in advance and conflicting permissions must be restricted to conflicting roles. These properties have different consequences on how roles get designed and managed but essentially achieve the same objective with respect to separation of conflicting permissions. Both properties achieve this goal with much higher assurance than property 1. Property 2 achieves this goal with similar high assurance but allows for the possibility of useless roles. Thus, even in the simple situation of static SOD, we have a number of alternatives offering differing degrees of assurance and flexibility.

Property 5, $\mid \texttt{user}(\texttt{OE}(\texttt{CR})) \cap \texttt{OE}(\texttt{CU}) \mid \leq 1$, is a very different property and is also new to the literature. With a notion of conflicting users, we identify new forms of SSOD. Property 5 says that two conflicting users cannot be assigned to roles in the same conflicting role set. This property is useful because it is much easier to commit fraud if two conflicting users can have different conflicting roles in the same conflicting role set. This property prevents this kind of situation in role-based systems. A collection of conflicting users is less trustworthy than a collection of non-conflicting users, and therefore should not be mixed up in the same conflict role set. This property has not been previously identified in the literature.

We also identify a composite property which includes conflicting users, roles and permissions. Property 6 combines property 4 and 5 so that conflicting users cannot have conflicting roles from the same conflict set, while assuring that conflicting roles have, at most, one conflicting permission from each conflicting permission set. This property supports SSOD in user-role and role-permission assignment with respect to conflicting users, roles, and permissions.

4.3.2 Dynamic SOD

In RBAC systems, a dynamic SOD (DSOD) property with respect to the roles activated by the users requires that no user can activate two conflicting roles. In other words, conflicting roles may have common users but users cannot simultaneously activate roles which are conflicting with each other. From this requirement we can express user-based Dynamic SOD as property 1.

Property 1, $| \mathtt{roles}^*(\mathtt{sessions}(\mathtt{OE}(\mathtt{U}))) \cap \mathtt{OE}(\mathtt{CR}) | \leq 1$, ensures that each user cannot simultaneously activate conflicting roles in sessions. $\mathtt{roles}^*(\mathtt{sessions}(\mathtt{OE}(\mathtt{U})))$ returns all roles activated by a user in sessions. We can also identify a Session-based Dynamic SOD property which can apply to the single session as property 3.

Property 3, $| \mathtt{roles}^*(\mathtt{OE}(\mathtt{sessions}(\mathtt{OE}(\mathtt{U})))) \cap \mathtt{OE}(\mathtt{CR}) | \leq 1$, applies property 1 to the single session. $\mathtt{roles}^*(\mathtt{OE}(\mathtt{sessions}(\mathtt{OE}(\mathtt{U}))))$ means all roles activated by a user in a single session. Therefore, this property ensures that each user cannot simultaneously activate conflicting roles in a session.

We can also consider these properties with conflicting users, such as property 2 and 4. Property 2, $| \mathtt{roles}^*(\mathtt{sessions}(\mathtt{OE}(\mathtt{OE}(\mathtt{CU})))) \cap \mathtt{OE}(\mathtt{CR}) | \leq 1$, requires that conflicting users cannot simultaneously activate roles which are conflicting with each other. $\mathtt{roles}^*(\mathtt{sessions}(\mathtt{OE}(\mathtt{OE}(\mathtt{CU}))))$ means all roles activated by a conflicting user in sessions. We can also identify another Session-based Dynamic SOD property which can apply conflicting users to the single session as property 4.

Property 4, $| \mathtt{roles}^*(\mathtt{OE}(\mathtt{sessions}(\mathtt{OE}(\mathtt{OE}(\mathtt{CU}))))) \cap \mathtt{OE}(\mathtt{CR}) | \leq 1$, ensures that each conflicting user cannot simultaneously activate conflicting roles in a session.

SSOD Properties	**Expressions**
1. SSOD-CR	$\mid \texttt{roles}^*(\texttt{OE(U)}) \cap \texttt{OE(CR)} \mid \leq 1$
2. SSOD-CP	$\mid \texttt{permissions}(\texttt{roles}^*(\texttt{OE(U)})) \cap \texttt{OE(CP)} \mid \leq 1$
3. Variation of 2	$(2) \wedge \mid \texttt{permissions}^*(\texttt{OE(R)}) \cap \texttt{OE(CP)} \mid \leq 1$
4. Variation of 1	$(1) \wedge \mid \texttt{permissions}^*(\texttt{OE(R)}) \cap \texttt{OE(CP)} \mid \leq 1$ $\wedge\ \texttt{permissions}(\texttt{OE(R)}) \cap \texttt{OE(CP)} \neq \phi \Longrightarrow \texttt{OE(R)} \cap \texttt{OE(CR)} \neq \phi$
5. SSOD-CU	$(1) \wedge \mid \texttt{user}(\texttt{OE(CR)}) \cap \texttt{OE(CU)} \mid \leq 1$
6. Yet another variation	$(4) \wedge (5)$
DSOD Properties	**Expressions**
1. User-based DSOD	$\mid \texttt{roles}^*(\texttt{sessions}(\texttt{OE(U)})) \cap \texttt{OE(CR)} \mid \leq 1$
2. Variation of 1	$\mid \texttt{roles}^*(\texttt{sessions}(\texttt{OE}(\texttt{OE(CU)}))) \cap \texttt{OE(CR)} \mid \leq 1$
3. Session-based DSOD	$\mid \texttt{roles}^*(\texttt{OE}(\texttt{sessions}(\texttt{OE(U)}))) \cap \texttt{OE(CR)} \mid \leq 1$
4. Variation of 3	$\mid \texttt{roles}^*(\texttt{OE}(\texttt{sessions}(\texttt{OE}(\texttt{OE(CU)})))) \cap \texttt{OE(CR)} \mid \leq 1$

Table 4.1: Static & Dynamic Separation of Duty

In section 4.3.1 and 4.3.2, we have discussed how *RCL 2000* can be used to specify the various separations of duty properties in a practical way. These static and dynamic properties are summarized in table 4.1.

4.4 Summary

In this chapter we showed how *RCL 2000* can be used to specify the various separation of duty properties. Our specification is primarily based on properties in [GGF98]. We specified those properties with several different perspectives, such as conflicting users, conflicting roles, and conflicting permissions, while identifying several properties which are not identified in the literature. Our specification also included the extension to role hierarchies with $\texttt{roles}^*$ and $\texttt{permissions}^*$.

We have shown that *RCL 2000* allows us to investigate nuances of static SOD and dynamic SOD in a way that has not been possible so far. This has led to formulation of static SOD properties that have not identified in a decade of literature on SOD. During this discussion we found the practical formulation with tradeoff between assurance and flexibility. Our work showed that there are many alternate formulations of even the simplest SOD properties, with varying degree of flexibility and assurance.

Chapter 5

CASE STUDIES

In this chapter we specify constraints identified in simulation of Lattice-based access control, Chinese Wall policy, and Discretionary access control in RBAC using *RCL 2000.* Role-based access control is a promising alternative to traditional discretionary access control (DAC) and mandatory access control (MAC). Sandhu has earlier shown how to simulate several variations of MAC in RBAC [San96]. Sandhu and Munawer have recently shown how to simulate a variety of DAC policies in RBAC [SM98]. These results were of theoretical interests because it relates RBAC to the most dominant form of access control. In this chapter we show how to express these simulations with *RCL 2000*, particularly based on Sandhu's LBAC-RBAC simulation [San96] and DAC simulation [SM98]. Sandhu has also shown that the Chinese Wall policy is just another Lattice-based information policy which is also known as mandatory access control [San92]. In this chapter we also discuss the simulation of the Chinese Wall policy in RBAC.

The rest of this chapter is organized as follows. Section 5.1 overviews the LBAC-RBAC simulation of [San96]. We also specify some of these simulations with *RCL 2000.* In section 5.2 we discuss the simulation of the Chinese Wall policy in doing more direct construction than [San92]. Section 5.3 we overview the DAC-RBAC simulation of [SM98] and specify some of these constructions with *RCL 2000.*

5.1 Lattice-Based Access Control

As we noticed in [San96], several constraints are required to simulate Lattice-based access control in RBAC. In this section we briefly overview the Lattice-based access control (LBAC) and show how we can specify LBAC constraints using *RCL 2000*.

5.1.1 Lattice-Based Access Control

Lattice-based access control is concerned with enforcing one directional information flow in a lattice of security labels [HW89, Den76, San96]. LBAC is also known as mandatory access control (MAC) or multilevel security.[1] Depending upon the nature of the lattice, the one-directional information flow enforced by LBAC can be applied for confidentiality, integrity, confidentiality and integrity together, or for aggregation policies, such as Chinese Walls [San93].

The mandatory access control policy is expressed in terms of security labels attached to subjects and objects. A label on an object is called a *security classification*, while a label on a user is called a *security clearance*. It is important to understand that a Secret user may run the same program, such as a text editor, as a Secret subject, or as an Unclassified subject. Even though both subjects run the same program on behalf of the same user, they obtain different privileges due to their security labels. It is usually assumed that the security labels on subjects and objects, once assigned, cannot be changed (except by the security officer). This last assumption, that security labels do not change, is known as *tranquility*. The security labels form a lattice structure as defined below.

Definition 1 (Security Lattice) There is a finite lattice of security labels $\mathcal{SC}$ with

[1]LBAC is typically applied in addition to classical discretionary access controls (DAC) [SS94] but for our purposes we will focus only on the MAC component. DAC can be accommodated in RBAC as an independent access control policy, just as it is done in LBAC.

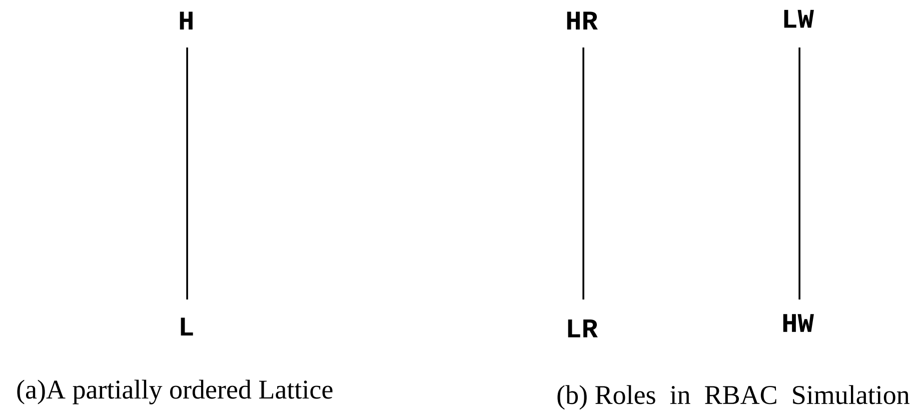


Figure 5.1: A Partially Ordered Lattice

a partially ordered dominance relation $\succeq$ and a least upper bound operator. $\square$

A simple example of a security lattice is shown in figure 5.1(a) with H > L. Information is only permitted to flow upward in the lattice. In this example, H and L, respectively, denote high and low. This is a typical confidentiality lattice where information can flow from low to high but not vice versa.

The specific mandatory access rules usually specified for a lattice, where λ signifies the security label of the indicated subject or object, are as follows:

Definition 2 (Simple Security) Subject s can read object o only if $\lambda(s) \geq \lambda(o)$. $\square$

Definition 3 (Liberal *-property) Subject s can write object o only if $\lambda(s) \leq \lambda(o)$. $\square$

Definition 4 (Strict *-property) Subject s can write object o only if $\lambda(s) = \lambda(o)$. $\square$

We now show how LBAC constraints can be simulated in RBAC using *RCL 2000*. In order to simulate LBAC constraints we adopt LBAC-RBAC construction in [San96] and we formulate it with *RCL 2000*.

5.1.2 Constraints Specification in *RCL 2000*

We begin by considering the example lattice of Figure 5.1(a) with the liberal *-property. Subjects with labels higher up in the lattice have more power with respect to read operations but have less power with respect to write operations. Thus, this lattice has a dual character. In role hierarchies subjects (sessions) with roles higher in the hierarchy always have more power than those with roles lower in the hierarchy. To accommodate the dual character of a lattice for LBAC we will use two dual hierarchies in RBAC, one for read and one for write. These two role hierarchies for the lattice of Figure 5.1(a) are shown in Figure 5.1(b). Each lattice label x is modeled as two roles xR and xW for read and write at label x, respectively. The relationship among the two read roles and the two write roles is respectively shown on the left and right hand sides of Figure 5.1(b). The duality between the left and right lattices is obvious from the diagrams.

To complete the construction we need to enforce appropriate constraints to reflect the labels on subjects in LBAC. Each user in LBAC has a unique security clearance. This is enforced by requiring that each user in RBAC be assigned to exactly one role xR and the role LW. An LBAC user can login at any label dominated by the user's clearance. This requirement is captured in RBAC by requiring that each session has exactly two matching roles yR and yW. The condition that x $\succeq$ y, that is that the user's clearance dominates the label of any login session established by the user, is not explicitly required because it is directly imposed by the RBAC model anyway.

LBAC is enforced in terms of read and write operations. In RBAC this means our permissions are reads and writes on individual objects written as (o,r) and (o,w), respectively. An LBAC object has a single sensitivity label associated with it. This is expressed in RBAC by requiring that each pair of permissions (o,r) and (o,w) be assigned to exactly one matching pair of xR and xW roles, respectively. By assigning permissions (o,r) and (o,w) to roles xR and xW respectively, we are implicitly setting the sensitivity label of object o to x.

The above construction is formalized below [San96].

Example 1 *(*Liberal *-Property*)*

- *R = {HR, LR, HW, LW}*
- *RH as shown in Figure 5.1(b)*
- *P = {(o,r), (o,w) | o is an object in the system}*
- *Constraint on UA: Each user is assigned to exactly two roles xR and LW*
- *Constraint on sessions: Each session has exactly two roles yR and yW*
- *Constraints on PA:*
  - *(o,r) is assigned to xR iff (o,w) is assigned to xW*
  - *(o,r) is assigned to exactly one role xR* □

With *RCL 2000* we need additional administrative sets to specify these constraints because these constraints require that each user should have exactly two roles xR and LW and each session should have exactly two roles yR and yW. From this observation, we introduce the following new administrative sets: active role sets (`AR`), assignment role sets (`ASR`), write roles (`WR`), and read roles (`RR`). Active roles mean that those roles should be active at the same session and assignment roles are roles which a user should have during user-role assignment. *RCL 2000* includes administrative sets shown in Figure 5.2, including all elements of *RCL 2000*. With these sets, we can specify the formalization mentioned above as below.

RCL 2000 Specification: (*Liberal *-Property*)

- $\texttt{R} = \{\text{HR, HW, LR, LW}\}$
- $\texttt{OBJ} = \{obj_1, obj_2, ..., obj_n\}$
- $\texttt{OP} = \{read, write\}$
- $\texttt{P} = \{rp, wp\}$

  $rp = \{(read, obj_1), (read, obj_2), ..., (read, obj_n)\}$ and

  $wp = \{(write, obj_1), (write, obj_2), ..., (write, obj_n)\}$
- $\texttt{RR} = \{\text{HR, LR}\}$
- $\texttt{WR} = \{\text{HW, LW}\}$
- $\texttt{AR}=\{ ar_1,\ ar_2\}$

  $ar_1 = \{\text{HR, HW}\}$

  $ar_2 = \{\text{LR, LW}\}$
- $\texttt{ASR}=\{ asr_1,\ asr_2\}$

  $asr_1 = \{\text{HR, LW}\}$

  $asr_2 = \{\text{LR, LW}\}$

Given these sets, such as `R`, `OBJ`, `OP`, `P`, `RR`, `WR`, `AR`, and `ASR`, we specify the above constraints described as below.

- Constraint on *UA*:

  $\texttt{roles(OE(U))} = \texttt{OE(ASR)}$

- Constraint on sessions:

  $\texttt{roles(OE(sessions(OE(U))))} = \texttt{OE(AR)}$

- Constraints on *PA*:

  $\texttt{objects(OE(}wp\texttt{))} = \texttt{objects(OE(}rp\texttt{))} \Longrightarrow$

  $\texttt{roles(OE(}wp\texttt{))} \cup \texttt{roles(OE(}rp\texttt{))} = \texttt{OE(AR)} \wedge$

  $\texttt{roles(OE(}rp\texttt{))} \cap \texttt{WR} = \phi \wedge$

  $\mid \texttt{roles(OE(}rp\texttt{))} \cap \texttt{RR} \mid = 1$

$\square$

From example 1, we can have two sets, ar_1 and ar_2 because both HR and HW roles should be activated, and both LR and LW roles should also be in a session. Also asr_1 and asr_2 are required because each user should have HR and LW (or, LR and LW) in user-role assignment. Therefore, we can express UA constraint and session constraint using `ASR` and `AR` sets, respectively. In expression of UA constraints, $\texttt{roles(OE(U))} = \texttt{OE(OE(ASR))}$ ensures that a user should have roles exactly equal to one of the `ASR` sets. In expression of session constraints, `roles(OE(sessions(OE(U))))` denotes all roles which a user activates in a single session. The constraint ensures that the roles activated by a user in a session consist of exactly one of the `AR` sets. Constraint specification of PA ensures that each permission should be assigned to a single object and roles which have read-write permissions on a single object can be invoked in a session. Unlike SOD constraints, we have just shown that we need additional administrative sets of *RCL 2000* to specify LBAC constraints and that these constraints need to be forced to activate the predefined roles at the same time.

- $\mathtt{AR}$ = a collection of active role sets, $\{ar_1, ..., ar_n\}$, where $ar_i = \{r_i, ..., r_k\} \subseteq \mathtt{R}$
- $\mathtt{ASR}$ = a collection of assignment role sets, $\{asr_1, ..., asr_n\}$, where $asr_i = \{r_i, ..., r_k\} \subseteq \mathtt{R}$
- $\mathtt{RR}$ = a collection of read roles, where $\mathtt{RR} = \{r_i, ..., r_k\} \subseteq \mathtt{R}$
- $\mathtt{WR}$ = a collection of write roles, where $\mathtt{WR} = \{r_i, ..., r_k\} \subseteq \mathtt{R}$

Figure 5.2: Administrative Sets in LBAC simulation

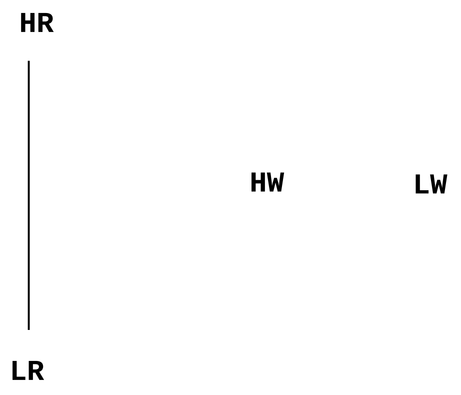


Figure 5.3: Role Hierarchies for Strict *-property

Variations in LBAC can be accommodated by modifying this basic construction in different ways. In particular, the strict *-property retains the hierarchy on read roles but treats write roles as incomparable to each other, as shown in figure 5.3.

Example 2 *(*Strict *-Property*) Identical to example 1 except RH is as shown in figure 5.3.*

We have shown how we can specify constraints identified during simulation of LBAC in RBAC. In the next section, we consider *RCL 2000* specification of the Chinese Wall policy.

5.2 Chinese Wall Policy

The Chinese Wall policy is intuitively simple and easy to describe. In this section we describe this policy by adapting the description of Brewer and Nash [BN89] and Sandhu [San92]. Chinese Wall policy can be reduced to LBAC so the previous construction can be used. In this section we consturct Chinese Wall-RBAC simulation in more direct manner.

It is important to keep in mind that we are deliberately ignoring all discretionary access control issues. In practice, the Chinese Wall policy as described here would be the mandatory component of a larger policy which includes additional discretionary controls (and possibly additional mandatory controls). We begin by distinguishing public information from company information. There are no mandatory controls on reading public information. Reading company information, on the other hand, is subjected to mandatory controls, which we will discuss in a moment. The policy for writing public or company information is indirectly determined by its impact on providing indirect read access contrary to the mandatory read controls. We will consider mandatory controls on writing information following our discussion of the read controls. The motivation for recognizing public information is that a computer system used for financial analysis will inevitably have large public databases of financial information for use by consultants. Moreover, public information allows for desirable features, such as public bulletin boards and electronic mail, which users expect to be available in any modern computer system. Public information can be read by all users, principals, and subjects in the system (restricted only by discretionary controls which, as we have said, we are ignoring). Company information is categorized into mutually disjoint conflict of interest classes as shown in figure 5.4.

Each company belongs to exactly one conflict of interest (COI) class. The Chinese Wall policy requires that a consultant should not be able to read information for

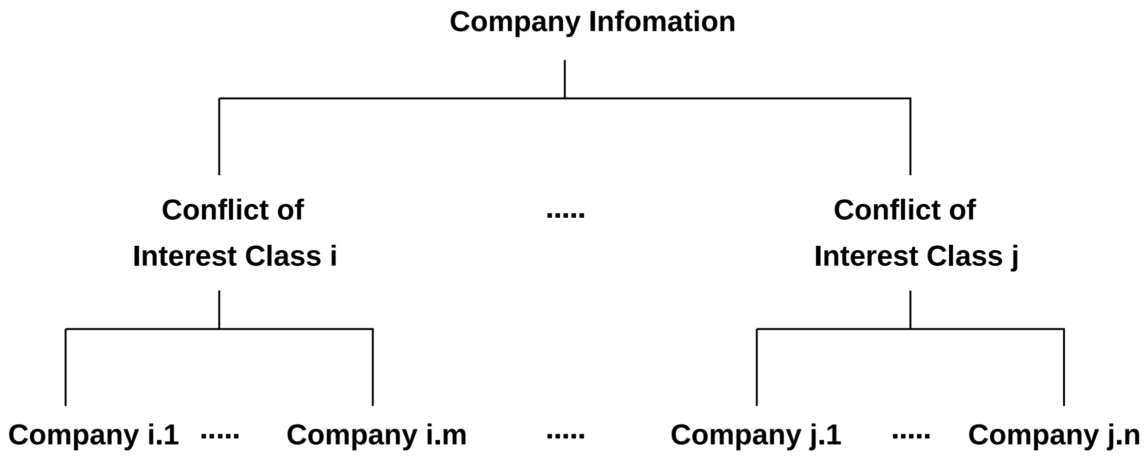


Figure 5.4: Company Information in the Chinese Wall

more than one company in any given COI class. To be concrete, let us say that COI class i consists of banks and COI class j consists of oil companies. The Chinese Wall stipulation is that the same consultant should not have read access to two or more banks or two or more oil companies. The Chinese Wall policy has a mix of free choice and mandated restrictions. So long as a consultant has not yet been exposed to any company information about banks, that consultant has the potential to read information about any bank. The moment this consultant reads, say, bank A information, thereafter, that consultant is to be denied read access to all other banks. The free choice of selecting the first company to read in a COI class can be exercised once and is then forever gone. So long as we have focused on read access the Chinese Wall policy has been easy to state and understand. When we turn to write access the situation becomes more complicated and subtle.

We begin by taking several variations to specify Chinese Wall policy. Figure 5.5 shows a simple example of a Chinese Wall which is used in the rest of this section.

5.2.1 Variation 1

We assume that each company can have one role which can do everything in a company. We can express this variant of Chinese wall policy as follow:

Constraints:

User should activate only one role in a dataset.

This constraint means that each company has one role which can do the whole business function. That is, this role can read and write on objects in a company. In order to specify this constraint, we just declare proper role for each session and we have to enforce that only one role should be activated in sessions which are invoked by a single user (or consultant).

Specification:

- `R` = { `Oil-X, Oil-Y, Bank-A, Bank-B` }

`roles(sessions(OE(U))) = OE(R)`

We have four roles since we have four companies, as shown in figure 5.5. We just use the company name as a role name for simplicity. This specification ensures that each user should activate only one role in a dataset. For example, we activate `Oil-X` role in a session, and then he/she cannot activate any other roles in role set `R`.

5.2.2 Variation 2

In this variation, we consider two roles such as, read and write, for each company. Then user can activate read role but he cannot activate the write role if the write role is not in the same company.

Constriants:

If user wants to activate the write role, only the matched read role should be activated.

This constraint requires that there is a pair of roles in a company and that those roles should be activated simultaneously. As we defined in section 5.1, we use the predefined roles to specify this constraint.

Specification:

- `SR` = { sr_1, sr_2, sr_3, sr_4 }
  - sr_1 = { `Bank-A-read, Bank-A-write` }
  - sr_2 = { `Bank-B-read, Bank-B-write` }
  - sr_3 = { `Oil-X-read, Oil-X-write` }
  - sr_4 = { `Oil-Y-read, Oil-Y-write` }

`roles(sessions(OE(U)))` = `OE(SR)`

We have four predefined roles sets since we have four companies, as shown in figure 5.5. We use only the company name with read and write notion as role name for simplicity. This specification ensures that only two matched roles in a company can be activated in sessions which a user invoked. For example, a user activates `Oil-X-read`, and then he can activate only the matched write role `Oil-X-write` when he wants to activate a write role.

5.2.3 Variation 3

In this variation, we consider two roles, such as read and write, for each company. A user can activate a read role but he cannot activate a write role if the write role is not in the same company. In addition, a user can read other objects in different company sets but he cannot write at all.

Constraints:

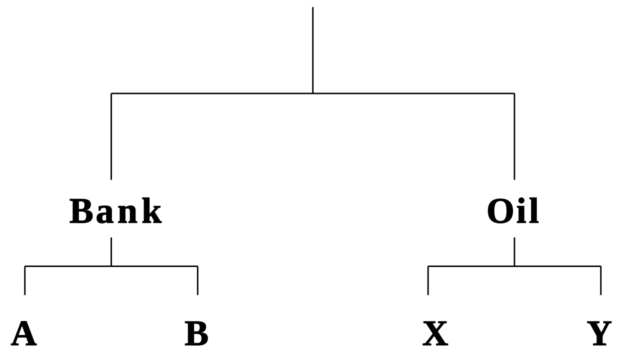


Figure 5.5: Example of Chinese Wall

- read

  User can read object(s) in different datasets but cannot write at all.

- write

  Same as variation 2.

This constraint requires two groups of role sets, such as read and write role sets. As we defined in section 5.1, we use the predefined roles to specify this constraint.

Specification:

In addition to variation 2,

- `GR` = { `read`, `write` }

  - `read` = { `Oil-Read`, `Bank-Read` }

    `Oil-R` = { `Oil-X-Read`, `Oil-Y-Read` }

    `Bank-R` = { `Bank-A-Read`, `Bank-B-Read` }

  - `write` = { `Oil-Write`, `Bank-Write` }

    `Oil-W` = { `Oil-X-Write`, `Oil-Y-Write` }

$$\texttt{Bank-W} = \{\ \texttt{Bank-A-Write},\ \texttt{Bank-B-Write}\ \}$$

We have two groups of roles sets and four predefined roles sets since we have two constraints–read and write constraints–and four companies, as shown in figure 5.5. We just use the company name with read and write notion as the role name for simplicity. With this set, our specification is as follows:

read constraint:

$\texttt{roles(sessions(OE(U)))} = \texttt{OE(OE(read))} \cup \texttt{OE(AO(read))} \wedge$

$\texttt{roles(sessions(OE(U)))} \cap \texttt{OE(OE(write))} = \phi$

write constraint:

$\texttt{roles(sessions(OE(U)))} = \texttt{OE(GR)}$

This specification ensures that each user can read object(s) in different dataset but he cannot write all. In a write constraint, only a matched write role should be activated in sessions which a user invoked.

We have shown how we can specify constraints identified during simulation of Chinese Wall policy in RBAC. In the next section, we consider the *RCL 2000* specification of constraints which can be recognized in the simulation of the discretionary access control in RBAC.

5.3 Discretionary Access Control

In this section we discuss DAC policies and specifications of the constraints using *RCL 2000*. The main idea of DAC is that the owner of an object has discretionary authority over who else can access that object [SS94, SS97]. There are many variations of DAC policy, particularly concerning how the owner's discretionary power can be delegated to other users and how access is revoked. This has been raised since the earliest formulations of DAC [Lam91, GD72].

Our approach in this section is to express constraints in *RCL 2000* which are identified in DAC simulation. The DAC simulation which we used in this section has been given by [SM98]. Their work is an intuitive, but well-founded, justification for the claim that DAC can be simulated in RBAC.

The DAC policies we consider in this section all share the following characteristics which are introduced in [SM98]:

- The creator of an object becomes its owner.
- There is only one owner of an object. In some cases, ownership remains fixed with the original creator, whereas in other cases, it can be transferred to another user.
- Destruction of an object can only be done by its owner.

With this in mind, we adopt the following two variations of DAC with respect to the granting of access.

1. **Strict DAC**

   Strict DAC requires that the owner is the only one who has discretionary authority to grant access to an object and that ownership cannot be transferred. For example, suppose Alice has created an object (Alice is owner of the object) and grants read access to Bob. Strict DAC requires that Bob cannot propagate access to the object to another user.

2. **Liberal DAC**

   Liberal DAC allows the owner to delegate discretionary authority for granting access to an object to other users. Sandhu and Munawer introduced three variations of liberal DAC such as one level grant, two level grant, and multilevel grant [SM98]. We take one variation, *one level grant*, of these variations given below.

- **One Level Grant:**

  The owner can delegate grant authority to other users but they cannot further delegate this power. So Alice being the owner of object O can grant access to Bob who can grant access to Charles. But Bob cannot grant Charles the power to further grant access to Dorothy.

In RBAC96 the behavior described above would be enforced by the constraints mechanism. In next section, we demonstrate how to simulate these variations of DAC in RBAC trying to specify these constraints with *RCL 2000*.

5.3.1 Simulations in RBAC

To specify the above variation in RBAC it suffices to consider DAC with one operation, which we choose to be the read operation. Similar constructions for other operations such as write, execute and append, are easily possible.

Before considering specific DAC variations, we first adopt the well-defined approach of [SM98] as shown in figure 5.6. Their approach introduced three administrative roles and one regular role. Figure 5.6 indicates that the role `OWN` can add users to the role `PARENTGRANT` which in turn can add users to the role `PARENT` and so on. This diagram also shows the seniority relation between the three administrative roles.[2]

Basic elements which can be used to this simulation are as follows:

[2]We slightly modify the name of roles which their approach introduced.

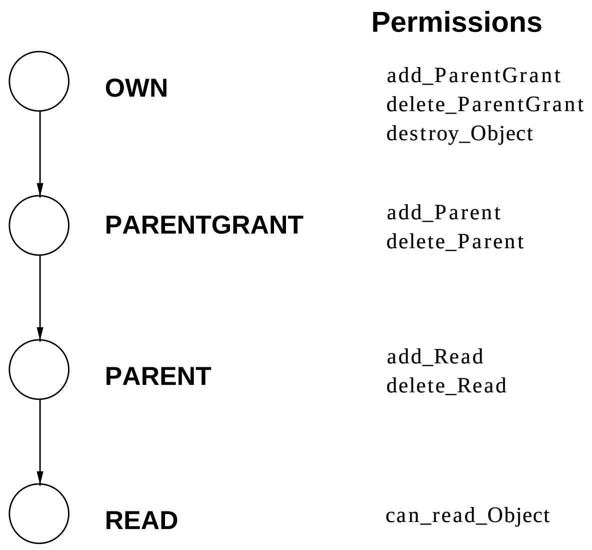


Figure 5.6: Roles and Permissions in DAC

- R = { OWN, PARENT, PARENTGRANT, READ }

Three administrative roles :

- OWN = { Own_O_1, ... , Own_O_n }
- PARENTGRANT = {$ParentGrant\_O_1$, ... ,$ParentGrant\_O_n$}
- PARENT = {$Parent\_O_1$, ... , $Parent\_O_n$ }

One regular role :

- READ = {$Read\_O_1$, ... , $Read\_O_n$ }

We separate the permissions which OWN role can perform as *administrative permissions* and *regular permissions* (because we do not need to assign users to role PARENTGRANT in strict DAC).

Administrative (`OWNAPM`) and regular (`OWNRPM`) permissions which `OWN` role can perform:

- `OWNAPM` $= \{add\_ParentGrant,\ delete\_ParentGrant\ \}$
- `OWNRPM` $= \{destroy\_O_1,\ ...\ ,\ destroy\_O_n\ \}$

Permissions which `PARENTGRANT` role can perform:

- `PGPM` $= \{add\_Parent,\ delete\_Parent\ \}$

Permissions which `PARENT` role can perform:

- `PPM` $= \{add\_Read,\ delete\_Read\ \}$

Permissions which `READ` role can perform:

- `RPM` $= \{can\_read\_O_1,\ ...\ ,\ can\_read\_O_n\}$

1. **Strict DAC**

   In strict DAC only the owner can grant/revoke read access to/from other users. The creator is the owner of the object. By virtue of membership in PARENT role and PARENTGRANT role, the owner can change assignments of the role READ. Membership of the three administrative roles cannot change, so only the owner will have this power. This policy could be simulated using just two roles OWN and READ, and giving the add_Read and delete_Read permissions directly to role OWN at the creation of object.

   In order to specify this constraint, we make the list of concerns which we may take.

(a) Owner needs to add a user to READ role.

(b) When a user creates an object, he has OWN role.

(c) And he has three permissions such as OWN role's regular permission `OWNRPM` and PARENT role's permissions `PPM`.

(d) And READ role has obviously `RPM` permission.

We can specify the above as below.

Specification:

- $\mid \mathtt{OE(PARENTGRANT)} \cup \mathtt{AO(PARENTGRANT)} \mid = 0 \wedge$
  $\mid \mathtt{OE(PARENT)} \cup \mathtt{AO(PARENT)} \mid = 0$
- $\mathtt{permissions(OE(OWN))} = \mathtt{OE(PPM)} \cup \mathtt{OE(OWNRPM)} \wedge$
  $\mathtt{object(permissions(OE(OWN)))} = \mathtt{object(OE(PPM))} \wedge$
  $\mathtt{object(permissions(OE(OWN)))} = \mathtt{object(OE(OWNRPM))} \wedge$
- $\mathtt{permissions(OE(READ))} = \mathtt{OE(RPM)} \wedge$
  $\mathtt{object(permissions(OE(READ)))} = \mathtt{object(OE(RPM))}$

2. **Liberal DAC with one-level grant**

The owner can delegate grant authority to other users but they cannot further delegate this power. So Alice being the owner of object O can grant access to Bob who can grant access to Charles. But Bob cannot grant Charles the power to further grant access to Dorothy. The one-level grant DAC policy can be simulated by using three roles OWN, PARENT, and READ.

In order to specify this constraint, we make the list of concerns which we may take.

(a) Owner needs to assign a user to PARENT role.

(b) When a user creates an object, he can have OWN role for that object.

(c) And he has three permissions such as such as OWN role's regular permission `OWNRPM` and two permissions from `PGPM`.

We can specify the above as below.

Specification:

- $\mid \texttt{OE(PARENTGRANT)} \cup \texttt{AO(PARENTGRANT)} \mid = 0$
- $\texttt{permissions(OE(OWN))} = \texttt{OE(PGPM)} \cup \texttt{OE(OWNRPM)} \wedge$

  $\texttt{object(permissions(OE(OWN)))} = \texttt{object(OE(PGPM))} \wedge$

  $\texttt{object(permissions(OE(OWN)))} = \texttt{object(OE(OWNRPM))} \wedge$
- $\texttt{permissions(OE(READ))} = \texttt{OE(RPM)} \wedge$

  $\texttt{object(permissions(OE(READ)))} = \texttt{object(OE(RPM))}$

We just showed that we can specify the constriants in *RCL 2000* which are identified in DAC-RBAC simulation of [SM98], particularly in strict DAC policy and one level grant DAC policy.

5.4 Summary

In this chapter, we have shown that *RCL 2000* can specify the constraints identified in the simulations of Lattice-based access control, Chinese Wall policy, and Discretionary access control. In order to specify these constraints, we defined the predefined roles to support the requirement of each constraint. These constraints have the notion of obligation when each constraint is addressed. In chapter 6 we look at this scope with generalization and characterization.

Chapter 6

PROHIBITION AND OBLIGATION CONSTRAINTS

Based on the constraints specifications in the previous chapters, we analyze the constraints. In order to do that, we define and identify the major classes of constraints in RBAC such as *Prohibition Constraints* and *Obligation Constraints*. We characterize these classes of constraints based on the *RCL 2000* expression. This distinction is a pragmatic approach rather than theoretical completeness. Our reasoning will help system security officers to think and design a system practically.

The rest of this chapter is organized as follows. Section 6.1 defines two major classes of constraints in RBAC such as *Prohibition Constraints* and *Obligation Constraints*. In section 6.2 we characterize these classes of constraints. This characterization is based on *RCL 2000* specifications which have been discussed in this dissertation, such as SOD constraints, LBAC simulation and the Chinese Wall policy construction. From these specifications, we characterize each class of constraints with an intuitive and a sharp distinction rather than with exhaustive analysis.

6.1 Role-based Constraints

As we mentioned the above, we identify two major classes of constraints in RBAC. In order to define these classes we introduce two rules.

Rule 1. *cannot_do* $(X,\ C_i)$

This *cannot_do* rule implies that RBAC component X is not allowed to do (be) something under C_i. If a certain constraint, such as C_k, does not allow RBAC component to do something, this constraint C_k satisfies the following: *cannot_do* $(X,\ C_i) =$ *True*

Rule 2. *must_do* $(Y,\ C_j)$

This rule implies that the RBAC component Y should do (be) something under C_j. If a certain constraint, such as C_k, forces the RBAC component to do something, this constraint C_k satisfies the following: *must_do* $(X,\ C_j) =$ *True*

Intuitively, these rules can be used to define the major class constraints in RBAC.

6.1.1 Prohibition Constraints

In an organization, we need to prevent a user from doing (or being) something that he is not allowed to do (or be), based on organizational policy. We call this class of requirements *Prohibition Constraints. Prohibition Constraints* are constraints that forbid the RBAC component from doing (or being) something which it is not allowed to do (or be). Therefore, this class of constraints satisfies *cannot_do* $(X,\ C_i) =$ *True.* Most of the separation of duty constraints specified in chapter 4 are included in this class of constraints.

6.1.2 Obligation Constraints

We also need to force a user to do (or be) something that he is allowed to do (or be) based on organizational policy. We derive another class of constraints from this motivation called *Obligation Constraints. Obligation Constraints* are constraints that force the RBAC component to do (or be) something. Therefore, this class of constraints satisfies *must_do* $(Y,\ C_j) =$ *True.* This category includes most of the constraints identified in the simulations of Lattice-based access control and Chinese Wall policy in role-based access control. As described in chapter 5, we need to enforce appropriate constraints to reflect the labels on subjects in LBAC in order to complete LBAC construction in RBAC. Each user in LBAC has a unique security clearance. This is enforced by requiring that each user in RBAC is assigned to exactly one role xR and the role LW. An LBAC user can login at any label dominated by the user's clearance. This requirement is captured in RBAC by requiring that each session has exactly two matching roles yR and yW. In our specification we use the active role set and assignment role set for these purposes. In the simulation of Chinese Wall policy, we assumed that each company has one role which can do everything in a company. This variation requires that the user should activate only one role in a dataset. In order to specify this constraint, we declared a proper role for each session and we needed to enforce that only one role should be activated in sessions which are invoked by a single user.

In this section we briefly looked at the identified classes of constraints in RBAC. These definitions are intuitive and more formal characterization of these constraints is discussed in the subsequent section, based on our specifications from case studies described in the previous chapters.

6.2 Constraints Characterization

In this section, we characterize the classes of constraints in RBAC. This characterization is based on *RCL 2000* specifications which have been discussed in this dissertation, such as SOD constraints, LBAC simulation and the Chinese Wall policy construction. From these specifications, we try to characterize each class of constraints with an intuitive and a sharp distinction rather than with exhaustive analysis.

Figure 6.1 indicates that there are two major classes of constraints, such as *Prohibition Constraints* and *Obligation Constraints*. Even though we believe that *RCL 2000* helps us discover useful constraints, this dissertation cannot list all of the existing con-

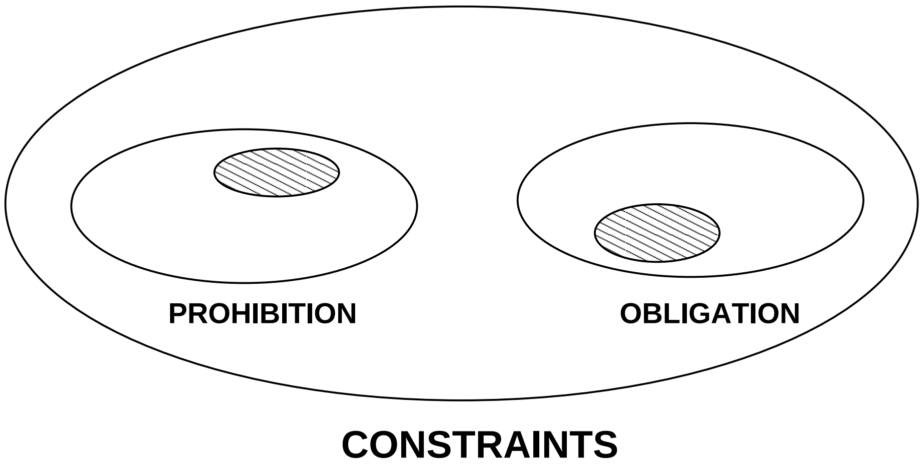


Figure 6.1: Constraints Characterization

straints. We assert that our specifications in this dissertation cover a subset of each class of constraints. We simply call our characterizations *Simple Prohibition Constraints* class and *Simple Obligation Constraint* class. In figure 6.1, a small set in the inner circle represents a group of constraints which we identified in this dissertation.

6.2.1 Simple Prohibition Constraints

The identified constraints which are classified as prohibition constraints are generally expressed in several similar forms. We generalize these forms to characterize prohibition constraints.

Simple prohibition constraints include:

- Type 1:
  $| \mathit{expr} | \leq 1$
  An example of this type of constraints is the simple static SOD (see Chapter 4.1.1, property 1). The specification of the simple static SOD is as follows: $| \texttt{roles}(\texttt{OE}(\texttt{U})) \cap \texttt{OE}(\texttt{CR}) | \leq 1$

- Type 2:

  $expr = \phi$ or $\mid expr \mid = 0$

  An example of this type of constraints is the static SOD with conflicting users (see Chapter 4.1.2, property 16). The expression of the static SOD with conflicting users is as follows: $\texttt{roles}(\texttt{OE}(\texttt{OE}(\texttt{CU}))) \cap \texttt{roles}(\texttt{AO}(\texttt{OE}(\texttt{CU}))) = \phi$

- Type 3:

  $\mid expr \mid < \mid expr \mid$

  An example of this type of constraints is per-role operational static SOD (see Chapter 4.1.1, property 9). The specification of this constraint is as follows: $\mid \texttt{operations}(\texttt{OE}(\texttt{OE}(\texttt{CR})), \texttt{OBJ}) \mid < \mid \texttt{OP} \mid$

Most of these forms are identified in the SOD constraints and some can also be found in other case studies.

6.2.2 Simple Obligation Constraints

Obligation constraints which are identified in this dissertation have several unique forms. We generalize these forms to characterize prohibition constraints.

Simple obligation constraints include:

- Type 1:

  $expr \neq \phi$ or $\mid expr \mid > 0$

  An example of this type of constraints is the variations of SOD constraints (see Chapter 4.3.1, property 4). The specification of this type is as follows: $\texttt{permissions}(\texttt{OE}(\texttt{R})) \cap \texttt{OE}(\texttt{CP}) \neq \phi$

- Type 2:

  $set\ X = set\ Y$

One example of this type of constraints occurs in the constraint in LBAC simulation (see Chapter 5.1.2, Constraint on UA). The expression of the constraint in LBAC is as follows: $\texttt{roles}(\texttt{OE}(\texttt{U})) = \texttt{OE}(\texttt{PR})$

- Type 3:

  obligation constraints $\Longrightarrow$ *obligation constraints*

  An example of this type of constraints can be found in LBAC simulation (see Chapter 5.1.2, Constraint on PA). The specification of this constraint is as follows:

  $\texttt{objects}(\texttt{OE}(wp)) = \texttt{objects}(\texttt{OE}(rp)) \Longrightarrow \texttt{roles}(\texttt{OE}(wp)) \cup \texttt{roles}(\texttt{OE}(rp)) = \texttt{OE}(\texttt{SR})$

- Type 4:

  $\mid expr \mid = 1$

  An example of this type of constraint is the constraints in LBAC simulation (see Chapter 5.1.2, Constraint on PA). The specification of this constraint is as follows: $\mid \texttt{roles}(\texttt{OE}(wp)) \cap \texttt{WR} \mid = 1$

  This form can be re-expressed with the forms which are identified in simple prohibition constraints and simple obligation constraints as below.

  $\mid expr \mid = 1 \equiv \mid expr \mid \leq 1 \wedge \mid expr \mid > 0$

As we mentioned earlier, most of the constraints which are identified in the simulation of LBAC and Chinese Wall policy in RBAC have these types of forms.

6.3 Summary

In this section we identified the major classes of constraints in RBAC such as *Prohibition Constraints* and *Obligation Constraints*. We also characterized these classes

Prohibition Constraints	**Obligation Constraints**
$\mid expr \mid \leq 1$	$\mid expr \mid > 0$
$\mid expr \mid = 0$	$set \text{ X} = set \text{ Y}$
$\mid expr \mid < \mid expr \mid$	$obligation \Longrightarrow obligation$
	$\mid expr \mid = 1$

Table 6.1: Characterizations of Constraints

of constraints based on the specifications described in our case studies (as shown in table 6.1. Our distinction is a pragmatic, but it is not theoretically complete. We believe that this distinction will help system security officers to think and to design a system more practically.

Chapter 7

CONCLUSION

This chapter lists the contributions of this dissertation and discusses into future directions. The contributions of this dissertation are presented in section 7.1 and section 7.2 gives the future research directions.

7.1 Contributions

Role hierarchies and constraints are two fundamental aspects of role-based access control. Although the importance of constraints in RBAC has been recognized for a long time, they have not received much attention in research literature while role hierarchies have been practiced and discussed at considerable length.

In this dissertation we developed the specification language *RCL 2000*. We have shown that our language can be applied to express constraints such as separation of duty constraints and constraints arising in the RBAC simulations of Lattice-based access control, Chinese Wall policy, and Discretionary access control. We have shown that *RCL 2000* allows us to investigate nuances of static SOD and dynamic SOD in a way that has not been possible so far. This has led to formulation of static SOD properties that have not identified in a decade of literature on SOD. Our work showed that there are many alternate formulations of even the simplest SOD properties, with varying degree of flexibility and assurance.

Based on these specifications, we analyze the constraints defining and identifying the major classes of constraints in RBAC such as *Prohibition Constraints* and *Obligation Constraints*. We characterized these classes of constraints based on the *RCL 2000* expression. This distinction is a pragmatic approach rather than theoretical completeness. Such classification of constraints is the first attempt in role-based security. This kind of characterization will help system security officers to think and design a system more practically.

Although the ease of expressing constraints in proposed language is an important aspect of the language, the primary contribution of the proposed language is its function in supporting a constraint specification facility for role-based systems rather than the enumeration of constraints.

We assert that our work shows that it is futile to try to enumerate all role-based constraints properties, because there are too many possibilities. Instead, we should pursue a rigorous language such as *RCL 2000*, for this purpose. That is, this work gives us direction in how we should deal with constraints in role-based systems. Also, we are convinced that our language has expressive power and extensibility.

We believe that the constraint language described above is a useful vehicle for expressing the sorts of constraints envisioned in role-based systems. Our language can be given a rigorous formal definition and is more expressive than the current rather *ad hoc* collection of constraint clichés defined for Role-based system. We believe that we can express most constraints currently expressible within RBAC and many more within a context which is more rigorous.

7.2 Future Work

Based on the research work in this dissertation, we propose the following future research directions and issues.

7.2.1 Extension of *RCL 2000*

In this dissertation we proved that *RCL 2000* can specify useful constraints which have not been identified in the literature. Our work is still the first step in role-based constraints. In the future, we would like to extend *RCL 2000*, investigating the utility of our language by applying it to the formalization of some realistic security policies. We may also need to think how we can enforce the constraints specified by *RCL 2000*.

7.2.2 Implementation Issues

We would like to build a tool that can check the syntax and the semantics of the specification, and we would like to make the specification language *RCL 2000* more efficient by implementing a tool that provides visualization support for constraint specifications. In this visualization, the system displays all components that can be used in constraint specifications. Also it shows which components are used and are available for constraint specifications. Using this visualization tool, security researchers can easily know the current status of components in role-based systems.

7.2.3 Update Problem

The basic component such as role hierarchy, user-assignment, and permission-assignment relations may be changed. In order to handle such changes we can consider several alternatives. For example, we can simply apply such changes to all the current sessions which are active under certain constraints and deactivate the sessions which violate the constraints. Or, we just apply such changes to new sessions which are activated after the changes. We can apply either alternative based on the organization's policy. We intend to investigate this type of issues including changes of constraints that are rarely modified.

BIBLIOGRAPHY

BIBLIOGRAPHY

[Ahn99a] Gail-Joon Ahn. Hierarchical administration in network information services. In *17th IAoM Annual International Conference on Computer Science*, pages 424–429. ACTA Press, August 6-8 1999.

[Ahn99b] Gail-Joon Ahn. The RSL99 language for specifying constraints in role-based access control. In *Technical Report*. GMU, Laboratory for Information Security Technology, 1999.

[AS98] Gail-J. Ahn and Ravi Sandhu. Security architecture of DCOM and its integration with RBAC. In *Proceedings of 1998 International Computer Symposium*, pages 71–78, N.C.K.U., Tainan, Taiwan, R.O.C., December 1998.

[AS99a] Gail-Joon Ahn and Ravi Sandhu. The RSL99 language for role-based separation of duty constraints. In *Proceedings of 4th ACM Workshop on Role-Based Access Control*, pages 43–54. ACM, 1999.

[AS99b] Gail-Joon Ahn and Ravi Sandhu. Towards role-based administration in network information services. *Journal of Network and Computer Applications*, 22(3):199–213, 1999.

[Bal90] Robert W. Baldwin. Naming and grouping privileges to simplify security management in large database. In *Proceedings of IEEE Symposium on Research in Security and Privacy*, pages 61–70, Oakland, CA, April 1990.

[BF99] Elisa Bertino and Elena Ferrari. The specification and enforcement of authorization constraints in workflow management systems. *ACM Transactions on Information and Systems Security*, 2(1):65–104, February 1999.

[BN89] D.F.C. Brewer and M.J. Nash. The Chinese Wall security policy. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 215–228, Oakland, CA, May 1989.

[CCSC98] Frederic Cuppens, Laurence Cholvy, Clarire Saurel, and Jerome Carrere. Merging security plocies: analysis of a practical example. In *Proceedings of IEEE Computer Security Foundations Workshop*, pages 123–136. IEEE, 1998.

[CS95] Fang Chen and Ravi 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.

[CS96] Frederic Cuppens and Clarire Saurel. Specifying a security policy: A case study. In *Proceedings of IEEE Computer Security Foundations Workshop*, pages 123–134. IEEE, 1996.

[CW87] 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.

[Den76] D.E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[Dep85] Department of Defense National Computer Security Center. *Department of Defense Trusted Computer Systems Evaluation Criteria*, December 1985. DoD 5200.28-STD.

[DHTK93] S.A. Demurjian, M.-Y. Hu, S. Ting, and D. Kleinman. Towards an authorization mechanism for user-based security in an object-oriented design model. In *Proceedings of Twelfth Annual International Phoenix Conference on Computers and Communications*, pages 195–202. IEEE, 1993.

[End77] H. B. Enderton. *Elements of Set Theory.* Academic Press, 1977.

[FB98] Tinothy Fraser and Lee Badger. Ensuring continuity during dynamic security policy reconfiguration in dte. In *Proceedings of IEEE Symposium on Research in Security and Privacy*, pages 15–26. IEEE, 1998.

[FBK99] David F. Ferraiolo, John F. Barkley, and D. Richard 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.

[FCK95] David Ferraiolo, Janet Cugini, and Richard Kuhn. Role-based access control (RBAC): Features and motivations. In *Proceedings of 11th Annual Computer Security Application Conference*, pages 241–48, New Orleans, LA, December 11-15 1995.

[FGS94] Eduardo B. Fernandez, Ehud Gudes, and Haiyan Song. The role graph model and conflict of interest. *IEEE Transactions on Knowledge and Data Engineering*, 6(2):275–292, April 1994.

[FJ95] Simon N. Foley and Jeremy Jacob. Specifying security for cscw systems. In *Proceedings of IEEE Computer Security Foundations Workshop*, pages 136–145. IEEE, 1995.

[FK92] David Ferraiolo and Richard Kuhn. Role-based access control. In *Proceedings of NIST-NCSC National Computer Security Conference*, pages 554–563, 1992.

[GB98] Cheh Goh and Adrian Baldwin. Towards a more complete model of role. In *Proceedings of 3rd ACM Workshop on Role-Based Access Control*, Fairfax, VA, October 1998.

[GD72] G.S. Graham and P.J. Denning. Protection – principles and practice. In *AFIPS Spring Joint Computer Conference*, pages 40:417–429, 1972.

[GGF98] Virgil D. Gligor, Serban I. Gavrila, and David Ferraiolo. On the formal definition of separation-of-duty policies and their composition. In *Proceedings of IEEE Symposium on Research in Security and Privacy*, pages 172–183, Oakland, CA, May 1998.

[GI96] Luigi Giuri and Pietro Iglio. A formal model for role-based access control with constraints. In *Proceedings of IEEE Computer Security Foundations Workshop*, pages 136–145, Kenmare, Ireland, June 1996.

[Giu95] Luigi Giuri. A new model for role-based access control. In *Proceedings of Annual Computer Security Application Conference*, pages 249–255. IEEE, 1995.

[Gol96] Derek Goldrei. *Classic Set Theory.* Chapman & Hall, 1996.

[Gri97] Richard Grimes. *Professional DCOM Programming.* Wrox Press Ltd., 1997.

[GS96] Simon Garfinkel and Eugene Spafford. *Practical Unix and Internet Security (2nd edition).* O'Reilly & Associates, Inc., 1996.

[Hay98] Ian J. Hayes. Expressive power of specification language. *Formal Aspects of Computing*, 10:187–192, April 1998.

[HBM98] R.J. Hayton, J.M. Bacon, and K. Moody. Access control in an open distributed environment. In *Proceedings of IEEE Symposium on Research in Security and Privacy*, pages 3–14. IEEE, 1998.

[HW89] Maurice P. Herlihy and Jeannette M. Wing. Specifying security constraints with relaxation lattices. In *Proceedings of IEEE Computer Security Foundations Workshop*, pages 47–53, 1989.

[Jan98] W.A. Jansen. Inheritance properties of role hierarchies. In *Proceedings of NIST-NCSC National Computer Security Conference*, pages 476–485, 1998.

[JSS97] Sushil Jajodia, Pierangela Samarati, and V.S. Subrahmanian. A logical language for expressing authorizations. In *Proceedings of IEEE Symposium on Research in Security and Privacy*, pages 31–42, 1997.

[JSSB97] Sushil Jajodia, Pierangela Samarati, V.S. Subrahmanian, and Elisa Bertino. A unified framework for enforcing multiple access control policies. In *Proceedings of the ACM SIGMOD international conference on management of data*, pages 474–485, 1997.

[Kec95] Alexander S. Kechris. *Classical Descriptive Set Theory*. Springer-Verlag, 1995.

[Kuh97] D. Richard Kuhn. Mutual exclusion of roles as a means of implementing separation of duty in role-based access control systems. In *Proceedings of 2nd ACM Workshop on Role-Based Access Control*, Fairfax, VA, October 1997.

[Lam91] B.W. Lampson. Protection. In *4th Princeton Symposium on Information Science and Systems*, pages 427–443, 1991.

[Mic98] Brian Michalowski. A constraint-based specification for box layout in CSS2. In *Technical Report UW-CSE-98-06-03, University of Washington*, 1998.

[Mof98] Jonathan D. Moffett. Control principles and role hierarchies. In *Proceedings of 3rd ACM Workshop on Role-Based Access Control*, Fairfax, VA, October 1998.

[MS92] J.A. McDermid and Qi Shi. Secure composition of system. In *Proceedings of Annual Computer Security Application Conference*, pages 112–122. IEEE, 1992.

[NO99] Matunda Nyanchama and Sylvia Osborn. The role graph model and conflict of interest. *ACM Transactions on Information and Systems Security*, 2(1):3–33, February 1999.

[NP90] M.N. Nash and K.R. Poland. Some conundrums concerning separation of duty. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 201–207, Oakland, CA, May 1990.

[PM94] Rita Pascale and Joseph McEnerney. Using THETA to implement access controls for separation of duties. In *Proceedings of 17th NIST-NCSC National Computer Security Conference*, pages 47–55, 1994.

[Rut97] Charles B. Rutstein. *Windows NT Security.* McGraw-Hill, 1997.

[SA98a] Ravi Sandhu and Gail-J. Ahn. Decentralized group hierarchies in UNIX: An experiment and lessons learned. In *Proceedings of 21st NIST-NCSC National Information Security Conference*, pages 486–502, Crystal City, VA, October 1998.

[SA98b] Ravi Sandhu and Gail-J. Ahn. Group hierarchies with decentralized user assignment in Windows NT. In *Proceedings of IASTED Conference on Software Engineering*, pages 352–355, Las Vegas, NV, October 1998.

[San88] Ravi Sandhu. Transaction control expressions for separation of duties. In *Proceedings of 4th Aerospace Computer Security Conference*, pages 282–286, Orlando, FL, December 1988.

[San90] Ravi Sandhu. Separation of duties in computerized information systems. In *Proceedings of the IFIP WG11.3 Workshop on Database Security*, pages 18–21, Halifax, U.K., September 1990.

[San92] Ravi Sandhu. Lattice-based enforcement of chinese walls. *Computers & Security*, 11(8):753–763, December 1992.

[San93] Ravi S. Sandhu. Lattice-based access control models. *IEEE Computer*, 26(11):9–19, November 1993.

[San96] Ravi S. Sandhu. Role hierarchies and constraints for lattice-based access controls. In Elisa Bertino, editor, *Proc. Fourth European Symposium on Research in Computer Security.* Springer-Verlag, Rome, Italy, 1996. Published as *Lecture Notes in Computer Science, Computer Security–ESORICS96.*

[San97] Ravi Sandhu. Rationale for the RBAC96 family of access control models. In *Proceedings of the 1st ACM Workshop on Role-Based Access Control*. ACM, 1997.

[San98] Ravi Sandhu. Role activation hierarchies. In *Proceedings of 3rd ACM Workshop on Role-Based Access Control*, Fairfax, VA, October 1998.

[SB97] Ravi Sandhu and Venkata Bhamidipati. The URA97 model for role-based administration of user-role assignment. In T. Y. Lin and Xiaolei Qian, editors, *Database Security XI: Status and Prospects*. North-Holland, 1997.

[SBC$^+$97] Ravi Sandhu, Venkata Bhamidipati, Edward Coyne, Srinivas Ganta, and Charles Youman. The ARBAC97 model for role-based administration of roles: preliminary description and outline. In *Proceedings of 2nd ACM Workshop on Role-Based Access Control*, pages 41–50, Fairfax, VA., October 1997.

[SBM99] Ravi Sandhu, Venkata Bhamidipati, and Qamar Munawer. The ARBAC97 model for role-based administration of roles. *ACM Transactions on Information and Systems Security*, 2(1):105–135, February 1999.

[SCFY96] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, February 1996.

[SM98] Ravi Sandhu and Qamar Munawer. How to do discretionary access control using roles. In *Proceedings of 3rd ACM Workshop on Role-Based Access Control*, pages 47–54, 1998.

[SS94] Ravi S. Sandhu and Pierangela Samarati. Access control: Principles and practice. *IEEE Communications*, 32(9):40–48, 1994.

[SS97] Ravi Sandhu and Pierangela Samarati. Authentication, access control and intrusion detection. In Elisa Bertino, editor, *The Computer Science and Engineering Handbook*, pages 1929–1948. CRC press, 1997.

[Sut97] Stephen A. Sutton. *Windows NT Security Guide*. Addison Wesley Developers Press, 1997.

[SZ97] R. T. Simon and M. E. Zurko. Separation of duty in role-based environments. In *Proceedings of IEEE Computer Security Foundations Workshop*, pages 183–194, Rockport, MA, December 1997.

[TS94] Roshan Thomas and Ravi S. Sandhu. Conceptual foundations for a model of task-based authorizations. In *Proceedings of IEEE Computer Security Foundations Workshop*, pages 66–79, Franconia, NH, June 1994.

[Vau95] Robert L. Vaught. *Set Theory: An Introduction.* Birkhauser, 1995.

[VCP98] Vijay Varadharajan, Chris Crall, and Joe Pato. Authorization in enterprise-wide distributed system: A practical design and application. In *Proceedings of 14th Annual Computer Security Application Conference*, pages 178–189. IEEE, 1998.

VITA

Gail-Joon Ahn was born on January 18, 1969, in Korea, and is Korean citizen. He received a B.S. in Computer Science from SoongSil University in Seoul, Korea, in Februrary of 1994. SoongSil was the first university in Korea to have a Computer Science Department. Ahn received a M.S. in Computer Science from the George Mason University in Fairfax, Virginia, in January of 1996.

He joined the Laboratory for Information Security Technology in 1996, and has been actively involved in research in information security. In 1999, he received a doctoral fellowship to continue his Ph.D. studies from the School of Information Technology and Engineering at George Mason University. His research interests include access control, security, distributed objects, and secure information systems.

This dissertation was typeset with LaTeX[‡] by the author.

[‡]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.