SCALABLE ROLE & ORGANIZATION BASED ACCESS CONTROL AND ITS ADMINISTRATION

by

Zhixiong Zhang
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:

[illegible] Dr. Ravi S. Sandhu, Dissertation Co-Director

[illegible] Dr. Daniel Menascé, Dissertation Co-Director

[illegible] Dr. Duminda Wijesekera, Committee Member

[illegible] Dr. Xinyuan Wang, Committee Member

[illegible] Dr. Robert P. Simon, Committee Member

[illegible] Dr. Daniel Menascé, Associate Dean for Research and Graduate Studies

[illegible] Dr. Lloyd J. Griffiths, Dean, The Volgenau School of Information Technology and Engineering

Date: 04/29/2008

Spring Semester 2008
George Mason University
Fairfax, VA

Scalable Role & Organization Based Access Control and Its Administration

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

By

Zhixiong Zhang
Master of Engineering
Shanghai Jiao Tong University, 1986
Bachelor of Science
Fudan University, 1983

Co-Director: Ravi S. Sandhu, Professor
Department of Information and Software Engineering
Co-Director: Daniel Menascé, Professor
Department of Computer Science

Spring Semester 2008
George Mason University
Fairfax, VA

DEDICATION

To my parents, Meiqin Sheng and Xiaogen Zhang, who gave me unconditional love and encouraged me to pursue my life-long dream.

To my lovely wife, Jianyu, who unwaveringly supported and shared with me all the difficult times throughout this academic journey with great love and sacrifices.

To my lovely sons, JJ and Ben, who are the constant sources of joy and pride in my life.

To my sisters and brother, who supported and encouraged me during this journey.

ACKNOWLEDGEMENTS

I would like to express my sincere appreciation and gratitude to my dissertation director, Professor Ravi Sandhu, who has provided very valuable guidance and encouragement during my doctoral research. Without his insightful guidance, it would have been impossible to make this work go so smoothly.

My special appreciation and thanks go to my dissertation co-director, Professor Daniel Menascé, who stepped in to rescue me from starting over again and provided insight and advice on my dissertation work.

I am also grateful to my dissertation committee members, Professor Duminda Wijesekera, Professor Xinyuan Wang, and Professor Robert Simon for their valuable comments and suggestions.

My gratitude also goes to Dr. Kenneth Hintz for his kind support and advice during the earlier years of my doctoral study and during my family emergency.

My appreciation also goes to many friends at George Mason University for their help and friendship. Special thanks go to Bing Wang, Xinwen Zhang, and Gregory McIntyre.

Many thanks go to Kathy Wang who helped me proof-read my dissertation in spite of a short time notice. I also thank the administrative staff at GMU for their support. Special thanks go to Lisa Nolder who helped me go through some administrative obstacles.

I would also like to thank my current employer, The College Board, for the financial support during my study.

I am particularly thankful to my parents, my wife, my two sons, my sisters, and my brother for their support, patience, tireless encouragement, and their faith in me throughout the years.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

SCALABLE ROLE & ORGANIZATION BASED ACCESS CONTROL AND ITS ADMINISTRATION

Zhixiong Zhang, Ph.D.

George Mason University, 2008

Dissertation Co-director: Dr. Ravi S. Sandhu

Dissertation Co-director: Dr. Daniel Menascé

In Role Based Access Control (RBAC), roles are typically created based on job functions inside an organization. Traditional RBAC does not scale up well for modeling security policies spanning multiple organizations. To solve this problem, a family of extended RBAC models called Role and Organization Based Access Control (ROBAC) models and its administrative models are proposed and formalized in this dissertation. Two examples are used to motivate and demonstrate the usefulness of ROBAC. Comparison between ROBAC and other RBAC extensions are given. I show that ROBAC can significantly reduce the administrative complexities of applications involving a large number of similar organizational units. The applicability and expressive power of ROBAC are discussed. By showing that any given ROBAC model can be modeled by a RBAC model and vice versa, I prove that the expressive power of ROBAC is equal to that of traditional RBAC.

A comprehensive role and organization based administrative model called AROBAC07 is developed. It has five sub-models dealing with various administrative tasks in ROBAC. I show that the AROBAC07 model provides an intuitive and controlled way to decentralize administrative tasks in ROBAC based systems. A concept called application compartment (ACom) in ROBAC is introduced and its usage in ROBAC is discussed. AROBAC07 scales up very well for ROBAC based systems involving many organizational units.

Two ROBAC variants, manifold ROBAC (ROBAC^M) and pseudo ROBAC (ROBAC^P), are presented and formalized. Their corresponding administrative models are also proposed. The usefulness of manifold ROBAC is demonstrated in secure collaboration via a ROBAC^M based secure collaboration schema which avoids many problems resulted from role-mapping, role-translation, or role exporting. The usefulness of pseudo ROBAC is demonstrated in a web based on-demand movie service case study.

Chapter 1. Introduction

With the widespread Internet usage in our society, security and privacy issues become more important than ever. In the last decade, role based access control (RBAC) has been generating a considerable amount of interest among researchers and practitioners. In RBAC, roles are defined based on job functions, permissions are associated with roles, and users are made members of appropriate roles, thereby acquiring the roles' permissions. This indirect association between users and permissions greatly simplifies the management of user's permissions. RBAC has several attractive features, such as policy neutrality, principle of least privilege, and ease of management. Several well-known RBAC models, such as RBAC96 [SCY96], the role graph model [NO99], and the NIST model [FSGKC01], were developed during the last decade. Those models form the basis for the ANSI RBAC standard [ANSI04]. As a powerful alternative to traditional discretionary and mandatory access control, the adoption of RBAC in commercial software and enterprises has rapidly increased in recent years [RTI02].

The complexity of an RBAC system can be defined on the basis of the number of roles, the number of permissions, the size of the role hierarchy, the constraints on user-role and permission-role assignments, etc. [SFK00]. For existing large-scale RBAC systems, the number of roles and the number of permissions are in the order of thousands. Beyond this magnitude, the performance of RBAC may degrade and its management becomes too difficult to handle correctly. Researchers and practitioners [GI97, Tho97, PS01, PCND04] have proposed several RBAC extensions to scale up RBAC systems from various perspectives. To achieve decentralized administration of RBAC, some role-based administrative models have also been proposed [SBM99, CL03, OSZ06, BJBG04].

Much of the previous work addresses RBAC in the context of a single organization and is mainly motivated by B2E (Business to Employee) applications. On the other hand, B2B (Business to Business) and B2C (Business to Consumer) applications often involve a large number of organizations such as corporations, schools, families, etc. Typically, users from different organizations with the same role name have slightly different access privileges due to privacy considerations. For example, a user in the parent role of family A has permission to view the progress records of family A's kids but not the progress records of other families' kids. Using standard RBAC naively in these situations can result in an enormous number of roles and permissions, well into the order of millions. To scale up RBAC for applications involving many organizational units is the problem we address in this dissertation.

1.1 RBAC Models

ANSI RBAC reference model [ANSI04] includes core (flat) RBAC (no role hierarchy), hierarchal RBAC (with role hierarchy), and constrained RBAC (with Separation of Duty constraints). Figure 1 shows a classic RBAC, which is based on the well-known RBAC96 [SCY96] plus permission definition from ANSI RBAC.

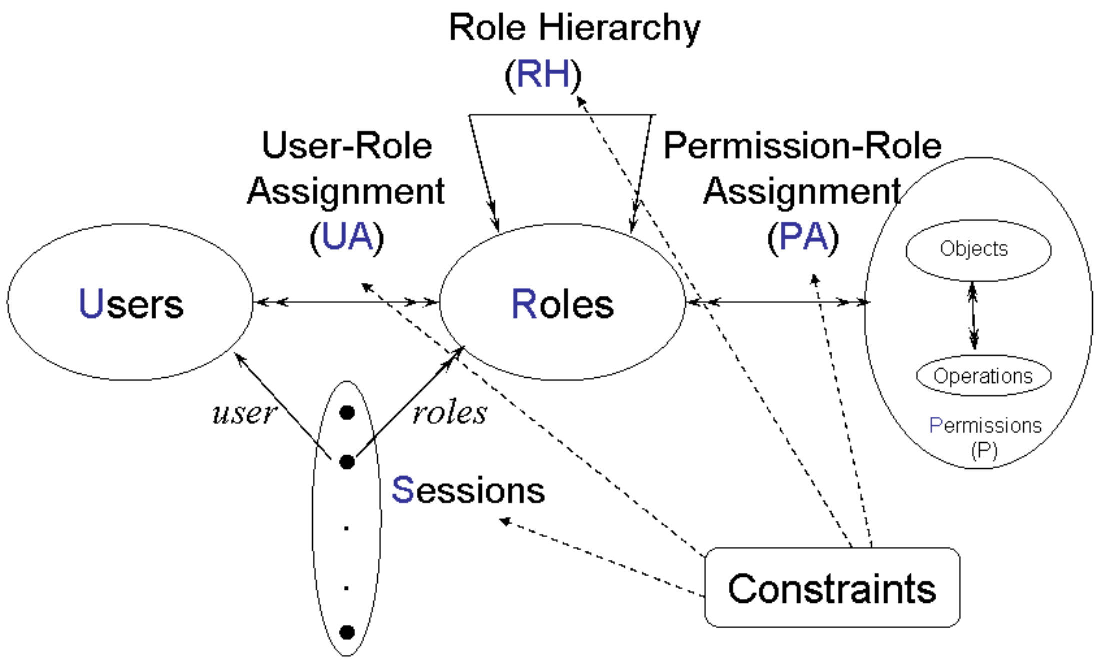


Figure 1. Classic RBAC

Here we use the term classic RBAC to refer the typical RBAC models proposed in [SCY96, NO99, FSGKC01]. As you can see, permissions and users are both assigned to roles. Users acquire permissions via their memberships in roles during one or more sessions. The permissions in standard RBAC are defined as operations over objects. Here, objects represent any resources that need to be protected in the system. Assigning permissions to roles and assigning roles to users are two separate administrative tasks. How to define roles and permissions depends on the desired security policy in an organization. RBAC models have been extended from various aspects (temporal, spatial, location-aware, or context-aware) [BBF01, CLSDAA01, KKC02, BCDP05, JBLG05, RKY06] to better meet the needs of the real world. Due to indirect assignment between users and permissions, role based systems are more flexible and scale up better than traditional MAC and DAC based system with respect to the number of users. However, RBAC does not scale up well with respect to the number of roles and permissions. Beyond the magnitude of a thousand roles, the management of RBAC is very error prone. Several approaches have been proposed to scale up RBAC systems. Giuri and Iglio [GI97] extend RBAC by introducing the concept of role templates with parameterized privileges to achieve content-based access control. Thomas [Tho97] proposes Team Based Access Control (TMAC) to scale up permission assignment with fine-grained run-time permission activation at the level of individual users and objects. Perwaiz and Sommerville [PS01] describe a mechanism for viewing role-permission relationships in the context of organizational structures, which reduces the number of roles in an RBAC implementation. Park et al. [PCND04] propose a composite RBAC for large and complex organizations. Park's approach categorizes roles into different classes and maps roles between them to achieve role class reusability and scalability.

1.2 Problem Statement

It is very common in B2B (Business to Business) and B2C (Business to Consumer) to have applications that involve a large number of organizational units such as corporations, branches, schools, families, etc. The classic RBAC does not scale up well with respect to the number of roles and permissions. Directly applying classic RBAC to applications involving a large number of organizations can result in a large number of roles and permissions due to local variations and privacy concerns. Most existing RBAC models either do not scale up well or lack systematic administrative models. Better models are needed to address the issue of scalability when applying RBAC to applications involving many organizational units. The new models should scale up well, be easy to use, and inherit the good features of RBAC. This dissertation seeks to solve this RBAC scalability problem by proposing a family of new RBAC extensions and its corresponding administration models.

1.3 Summary of Contributions

This dissertation contains the following contributions.

- A family of extended RBAC models called Role and Organization Based Access Control (ROBAC) models is proposed and formalized. It includes
    - $ROBAC_0$ – an extension to core (flat) RBAC and acts as a base model
    - $ROBAC_1$ – an extension to hierarchal RBAC and has both role hierarchy and organization hierarchy
    - $ROBAC_2$ – an extension to constrained RBAC, which includes $ROBAC_0$ plus constraints

    - $ROBAC_3$ – a combined model, which includes $ROBAC_0$ plus role hierarchy, organization hierarchy, and constraints
- The applicability and expressive power of ROBAC are discussed.
- A comprehensive role and organization based administrative model for ROBAC, called AROBAC07, is developed. AROBAC07 includes the following five sub-models:
    - UROA07 (user to role and organization pair assignment '07) is concerned with user to role and organization pair assignment;
    - PRA07 (permission to role assignment '07) deals with permission-role assignment;
    - RRA07 (role to role assignment '07) manages roles and role hierarchy;
    - OOA07 (organization to organization assignment '07) handles organizations and organization hierarchy; and
    - ROA07 (role to organization assignment '07) controls applicable association between roles and organizations.
- A concept called application compartment (ACom) in ROBAC is introduced and its usage in ROBAC is discussed
- Two ROBAC variants, manifold ROBAC ($ROBAC^M$) and pseudo ROBAC ($ROBAC^P$) are presented.
    - A $ROBAC^M$ based secure collaboration schema is proposed and the usefulness of manifold ROBAC is demonstrated in secure collaboration.

- The usefulness of pseudo ROBAC is demonstrated in a ROBAC^P case study.

1.4 Organization of the dissertation

Chapter 2 gives two motivating examples for developing ROBAC models. One is abstracted from a B2B application, and the other is abstracted from a B2C application. Chapter 3 presents a family of Role and Organization Based Access Control (ROBAC) models. Section 3.2 describes the formal definitions of ROBAC models. The applicability and expressive power of ROBAC are shown in section 3.3. The comparison between ROBAC and other related RBAC models is discussed in section 3.4. After discussing the administrative issues in ROBAC, a comprehensive role and organization based administrative model for ROBAC, called AROBAC07, is introduced and formalized in Chapter 4. The application compartment concept and its usage in ROBAC/AROBAC07 are discussed in section 4.3. The related administrative RBAC models are discussed in section 4.4. Two ROBAC variants, manifold ROBAC (ROBAC^M) and pseudo ROBAC (ROBAC^P), and their usefulness are presented in Chapter 5. The formal definition of ROBAC^M is given in section 5.1. A ROBAC^M based secure collaboration schema and its related administrative models are proposed in section 5.2. The formal definition of ROBAC^P and its administrative model are described in section 5.3. A ROBAC^P based case study is presented in section 5.4. Chapter 6 concludes the dissertation and presents some future research directions.

Chapter 2. Motivating Examples

Most existing RBAC models address problems in the context of one organization. Many B2B and B2C applications involve a large number of organizations and often have privacy requirements such as users in one organization are only allowed to access the resources related to that organization and are not allowed to access other organizations' resources. To show the motivation for our new models, let us start with two examples in the contexts of B2B and B2C, respectively. These are abstracted from our experience with similar real-world applications.

2.1 B2B Example

This example considers access control policy for a web based report delivery system, which only allows authorized users to access specific reports. The users are educational professionals from schools, districts, and states in USA. There are on order of 10,000 schools participating in the system. Reports are classified into types based on sensitivity and nature of the content. Because some report types include privacy-sensitive data such as student test results and personal information, only an authorized user, say, School_1's official, can view School_1's reports but cannot view School_2's reports. There are many different types of reports, each of which may have up to three different levels of information (state level, district level, and school level). Some sample report types are listed in Table 1.

Table 1. Sample report types in B2B example

Type A Report (school level, district level, and state level)
Type B Report (school level only)
Type C Report (school level only)
Type D Report (school level only)
Type E Report (school level and district level)
Type F Report (district level and state level)
…

States, districts, and schools usually form a management hierarchy. Figure 2 shows an example of a possible management hierarchy among states, districts, and schools.

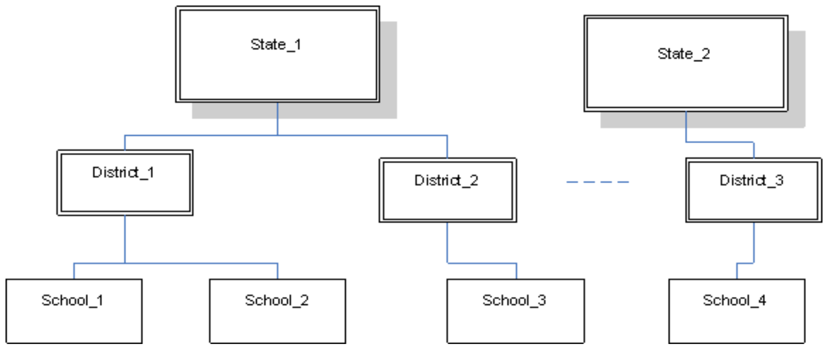


Figure 2. A sample organization hierarchy

Informal descriptions of some security policies of the system may include:

- Users from a school are only allowed to access the reports related to that school.
- Users from a district education office are allowed to access the reports related to that district and the schools under it.

- Users from a state education office are allowed to access the reports related to that state and the districts and schools under it.
- School principles can view type A and type B reports.
- School teachers can view type B and type E reports.
- Officials from a district's board of education office can view type A and type B reports but cannot view type D reports

Under the above policies, an authorized school level user (say School_1 teacher) can only access certain types of the user's own school's reports, but is not allowed to access other types of reports, and furthermore, cannot access another school's reports, or those from any district or state level. Here we are assuming that access not explicitly allowed by the stated policies is denied. An authorized district level user can access certain types of the user's own district's reports (district level) and may also access the same types of its subordinate schools' reports. For example, an authorized District_1 official can access District_1's district level Type_A reports and school level Type_A report for School_1 and School_2 since School_1 and School_2 are under District_1.

Assuming there are 10,000 organizations and 10 types of reports, if we use standard RBAC to model this problem directly, we have to define about 100,000 (10,000 x 10) permissions because viewing School_1's Type_A report is different from viewing School_2's Type_A report. We also need to define 100,000 different roles because a role that can view a School_1_Type_A report is different from a role that can view a

School_2_Type_A report. Table 2 and Table 3 show some samples of the possible permissions and roles in this example.

Table 2. Sample permissions in B2B example (with RBAC)

p1: View School_1 Type A Report
p2: View School_2 Type A Report
p3: View District_1 Type A Report
…

Table 3. Sample roles in B2B example (with RBAC)

r1: School_1 Type A Report Viewer with permission p1.
r2: School_2 Type A Report Viewer with permission p2.
r3: District_1 Type A Report Viewer with permission p3.
…

The goal here is not so much to define a complete and coherent policy for this example but rather to illustrate the issues and complexities involved.

2.2 B2C Example

Consider an online subscription-based tutoring system, where customers are families that have children in elementary schools. Parents pay subscription fees for their children and are authorized to create/update the family’s profile and view their children’s progress reports. Students that have subscribed to the service can take classes on the web and view their progress reports and family profiles. Here, family profiles and student progress reports need to be protected against unauthorized access. There are potentially millions of families, and even tens or hundreds of millions. The informal description of some security policies of this system may include:

- Parents can only view their own children's progress reports.
- Parents can create/update/view their family's profile.
- A student can view his/her own progress report and view his/her family's profile.

Suppose we use standard RBAC to model these policies. Because Family_1's parent is only allowed to access Family_1's profile and Family_1's children's progress reports, Family_1's parents have slightly different permissions from that of Family_2's parents. Table 4 and Table 5 show some samples of the possible permissions and roles when using classic RBAC in this B2C example.

Table 4. Sample permissions in B2C example (with RBAC)

p1: Update Family_1's Profile
p2: View Family_1's Kids' Progress Reports
p3: View Family_1's Profile
p4: Update Family_2's Profile
p5: View Family_2's Kids' Progress Reports
p6: View Family_2's Profile
……

Table 5. Sample roles in B2C example (with RBAC)

r1: Family_1 Parents with permission p1 and p2.
r2: Family_1 Student with permission p2 and p3.
r3: Family_2 Parents with permission p4 and p5.
r4: Family_2 Student with permission p5 and p6.
…

We can see that the administrative complexity is very high in applying RBAC directly to the above two examples. These scenarios are quite typical for B2B and B2C applications. In practice, security and application engineers usually work around this problem by combining RBAC with other access control mechanisms such as context-based or attribute-based access control. The result is an ad hoc access control model with a specialized administrative tool for each application [GMPT01, SMJ01].

Chapter 3. Role and Organization Based Access Control (ROBAC)

To address the issue where classic RBAC does not scale up well for applications involving multiple organizations where privacy issue is the main concern, we extend RBAC to ROBAC (Role and Organization Based Access Control) by basing access decision on both role and organization.

For ease of comparison with classic RBAC, we define ROBAC models one by one based on the increasing security functionality of the models ($ROBAC_0$, $ROBAC_1$, $ROBAC_2$, $ROBAC_3$) in direct correspondence with the four models of well-known RBAC96 family ($RBAC_0$, $RBAC_1$, $RBAC_2$, $RBAC_3$). $ROBAC_0$ is a base model. $ROBAC_1$ is $ROBAC_0$ plus role hierarchy and organization hierarchy. $ROBAC_2$ is $ROBAC_0$ plus constraints. $ROBAC_3$ is $ROBAC_0$ plus role hierarchy, organization hierarchy and constraints. Figure 3 shows the relationship of the ROBAC models and Figure 4 portrays their essential characteristics.

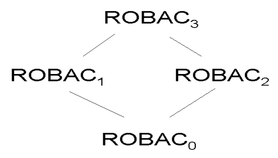


Figure 3. Relationship among ROBAC models

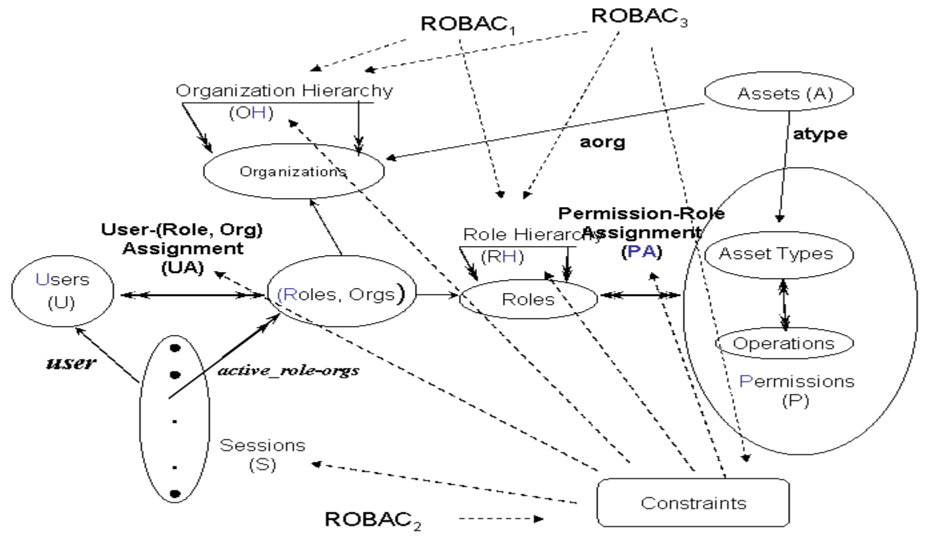


Figure 4. A family of ROBAC models

3.1 Informal Description

The central idea behind ROBAC is quite simple. Instead of only using role related information, ROBAC utilizes both role and organization information during the authorization process. Specifically, in ROBAC a user is assigned to role and organization pairs rather than roles only. Permissions in ROBAC are defined as operations over object types instead of operations over objects. A user can access an object if and only if the user is assigned to a role and organization pair, where the role has the right to access the object's type and the object is related to the organization. In the following sections, we show that the number of roles and permissions for the above B2B and B2C examples can be reduced significantly if we use ROBAC to model them. This demonstrates that ROBAC can significantly reduce administrative complexity for applications involving a large number of similar organizational units.

3.2 Formal Description

We focus our formal definitions on $ROBAC_0$, $ROBAC_1$, and $ROBAC_2$ as the $ROBAC_3$ is just a combination of $ROBAC_1$ and $ROBAC_2$.

3.2.1 $ROBAC_0$

$ROBAC_0$ is a base model in ROBAC family. The roles and organizations in $ROBAC_0$ are flat.

Definition 3.1: $ROBAC_0$ has the following components:

- U -- a set of users (same as U in RBAC96);
- S -- a set of sessions (same as S in RBAC96);
- R -- a set of roles (same as R in RBAC96);
- O -- a set of organizations;
- Op -- a set of operations;
- A -- a set of assets;
- At -- a set of asset types;
- $P \subseteq Op \times At$ -- a set of permissions;
- $RO \subseteq R \times O$ -- a set of applicable role and organization associations;
- $PA \subseteq P \times R$ -- a many-to-many permission-to-role assignment relation;
- $UA \subseteq U \times RO$ -- a many-to-many user-to-role-and-organization assignment relation;
- *user*: $S \rightarrow U$ -- a function mapping a session *s* to a single user *user(s)* (same as *user* in RBAC96);
- *atype*: $A \rightarrow At$ -- a function mapping an asset to its type;
- *aorg*: $A \rightarrow O$ -- a function mapping an asset to the organization it belongs to;

- *assigned_role-orgs*: $U \rightarrow 2^{RO}$ -- a function mapping a user to a set of role-organization pairs assigned to the user; formally: *assigned_role-orgs(u)* = { (*r,o*) | (*u*, (*r,o*)) ∈ UA };
- *active_role-orgs*: $S \rightarrow 2^{RO}$ -- a function mapping a session *s* to a set of active role-organization pairs such that *active_role-orgs(s)* ⊆ *assigned_role-orgs(user(s))*;
- *can_access*: S × Op × A → {true, false} -- a predicate defined as *can_access(s, op, a)* is true iff ∃ (*r, o*) ∈ *active_role-orgs(s)* ∧ *aorg(a) = o* ∧ ((*op*, *atype(a)*), *r*) ∈ PA ;

here the operations (Op) are similar to the operations or actions in the classic RBAC; the assets (A) are similar to objects; the *active_role-orgs* is used to model activation of role-organization pairs inside a session and it returns a subset of the role and organization pairs the *assigned_role-orgs* returns. Because certain roles are only meaningful for certain organizations – for instance, the School_Principle role is only meaningful for school type organizations --, we introduce the set of applicable role and organization pairs (RO) to model that requirement. Briefly, $ROBAC_0$ extends $RBAC_0$ by:

- introducing new sets: Organizations(O), Asset Types (At), and Role-Organization pairs (RO);
- introducing new functions: *atype, aorg;*
- extending *assigned_role* and *roles* (*session_role*) to *assigned_role-orgs* and *active_role-orgs*;
- redefining permissions (P) and user to role assignment (UA);
- introducing a predicate: *can_access(s, op, a)*.

Any access control system needs to answer the following question: Can a subject perform an operation over an object?

The newly introduced predicate *can_access* serves this purpose in ROBAC. If predicate *can_access(s, op, a)* is true, session *s* or user *user(s)* can perform operation *op* on asset *a* during the session. The definition of *can_access* in $ROBAC_0$ indicates that a user (*user(s)*) in a session *s* can perform an operation *op* over an asset *a* if and only if the user has an active role and organization pair (*r, o*) in that session, where *r* has permission to perform *op* over *a*'s type and *a* is related to *o*.

For the aforementioned B2C example, we can use $ROBAC_0$. Possible permissions and roles are listed in Table 6 and Table 7.

Table 6. Sample permissions in B2C example (with ROBAC)

p1: Update Family Profile
p2: View Kid's Progress Reports
p3: View Family Profile
…

Table 7. Sample roles in B2C example (with ROBAC)

r1: Parent which has permission p1 and p2.
r2: Student which has permission p2 and p3.
…

In this B2C example, ROBAC only creates two roles (parent and student) instead of a parent role and a student role for each family as in the classic RBAC.

3.2.2 ROBAC$_1$

ROBAC$_1$ is ROBAC$_0$ plus role hierarchy and organization hierarchy.

Definition 3.2: ROBAC$_1$ has the following components:

- U, S, R, O, Op, A, At, P, RO, PA, UA, *user, atype, aorg* -- the same as those from ROBAC$_0$.
- $OH \subseteq O \times O$ -- a partial order relation on O called organization hierarchy;
- $RH \subseteq R \times R$ -- role hierarchy (same as RH in RBAC96);
- *assigned_role-orgs*: $U \rightarrow 2^{RO}$ -- a function mapping a user to a set of role-organization pairs assigned to the user; formally: *assigned_role-orgs(u)* $= \{ (r,o) \mid \exists r' \geq r \wedge \exists o' \geq o \wedge (u, (r',o')) \in UA \}$;
- *active_role-orgs*: $S \rightarrow 2^{RO}$ -- a function mapping each session *s* to a set of active role-organization pairs such that *active_role-orgs(s)* $\subseteq$ *assigned_role-orgs(user(s))*;
- *can_access*: $S \times Op \times A \rightarrow \{true, false\}$ -- a predicate defined as *can_access(s, op, a)* is true iff $\exists (r, o) \in$ *active_role-orgs(s)* $\wedge$ *aorg(a)* $\leq o \wedge (\exists r' \leq r, ((op,$ *atype(a)*$), r') \in PA)$;

ROBAC$_1$ adds both OH (organization hierarchy) and RH (role hierarchy) and changes the *assigned_role-orgs* function and the *can_access* predicate from ROBAC$_0$.[1]

The definition of *can_access* in ROBAC$_1$ means that a user *user(s)* in a session *s* can perform an operation *op* over an asset *a* if and only if the user has an active role and organization pair (*r, o*) in the session where role *r* or any of its junior roles has permission to perform operation *op* over asset *a*'s type and asset *a* is related to organization *o* or any of its subordinate organizations.

For the aforementioned B2B example, we can use ROBAC$_1$ to model it very conveniently. We show some ROBAC elements differing from those in RBAC as follows.

- O= {State_1, State_2, District_1, District_2, District_3, School_1, School_2, School_3, School_4, …}
- OH = {(State_1, District_1), (State_1, District_2), (District_1, School_1), (District_1, School_2), (District_ 2, School_3), (State_2, District_3), (District_3, School_4), …}
- At = {Type_A_Report, Type_B_Report, …}
- RO = { (r1, District_1), (r2, District_1), (r1, School_1), (r2, School_2), (r3, School_1), (r4, School_1), … }

[1] In ROBAC$_1$, *assigned_role-orgs* function and *can_access* predicate consider both role hierarchy and organization hierarchy. We could have finer classification of ROBAC models wherein only one of these hierarchies is allowed in two subcases of ROBAC$_1$, one where organization hierarchies are allowed and one where role hierarchies are allowed.

Possible permissions and roles are listed in Table 8 and Table 9.

Table 8. Sample permissions in B2B example (with ROBAC)

p1: View Type A Report
p2: View Type B Report
p3: View Type C Report
p4: View Type D Report
p5: View Type E Report
p6: View Type F Report
…

Table 9. Sample roles in B2B example (with ROBAC)

r1: Type A Report Viewer which has permission p1.
r2: Type B Report Viewer which has permission p2.
r3: Type C Report Viewer which has permission p3.
r4: Type D Report Viewer which has permission p4.
r5: Type E Report Viewer which has permission p5.
r6: Type F Report Viewer which has permission p6.
…

In this B2B example, ROBAC only creates one role called Type_A_Report_Viewer with only one permission called View_Type_A_Report for viewing type A report, whereas a classic RBAC must create a role for each organization’s type A report viewer, such as School_1_Type_A_Report_Viewer with View_School_1_Type_A_Report permission, School_2_Type_A_Report_Viewer with View_School_2_Type_A_Report permission, etc.

Based on the security policies, r1 and r2 can have role-organization pairs within all levels of organizations but r3 and r4 can only have role-organization pairs within school level organizations.

3.2.3 ROBAC$_2$

ROBAC$_2$ is ROBAC$_0$ plus constraints. There are many different types of constraints in RBAC. A detailed discussion of RBAC related constraints can be found in Ahn's dissertation [Ahn99]. Instead of listing all types of constraints in ROBAC, we only emphasize select differences between RBAC constraints and ROBAC constraints via certain examples. In ROBAC, there are two levels of constraints, *global constraints* and *local constraints*. We explain these two levels of constraints by using the two most common UA constraints: separation of duty and cardinality.

Separation of duty (SoD) constraint I

A separation of duty constraint I, SoD_I, is a subset of $(2^R \times N)$. For an element *(rs, n)* $\in$ SoD_I, where *rs* is a role set and *n* is a natural number greater than 1, it means that no user can be assigned to *n* or more roles from the set *rs* within any same organization in O.

Separation of duty constraint II

A separation of duty constraint II, SoD_II, is a subset of $(2^{RO} \times N)$. For an element *(ros, n)* $\in$ SoD_II, where *ros* is a role-organization pair set and *n* is a natural number greater than 1, it means that no user can be assigned to *n* or more role-organization pairs from the set *ros*.

Cardinality constraint I

A cardinality constraint, *cardinality*: R → N (natural number set), is a function that maps a role to a natural number. For $r \in$ R, *cardinality*(r) means the maximum number of users who can be assigned to the role r within each organization in O.

Cardinality constraint II

A cardinality constraint, *cardinality*: RO → N, is a function that maps a role and organization pair to a natural number. For $(r, o) \in$ RO, *cardinality*((r, o)) means the maximum number of users who can be assigned to role r in organization o .

In **Separation of duty constraint I** and **Cardinality constraint I**, constraints are defined on role set R only. They have the same syntax as in the RBAC constraints and apply to every organization within O. We classify these types of constraints as *global constraints* because they apply to all organizations. In **Separation of duty constraint II** and **Cardinality constraint II**, constraints are defined on role and organization pair set RO. We subsequently classify these types of constraints as *local constraints* because they only apply to specified organizations.

A global constraint is equivalent to a set of local constraints. For example, a global SoD constraint ($\{r_i, r_j\}$, 2) can be represented by the following set of local SoD constraints:

$$\bigcup_{o_k \in O} \{ (\{(r_i, o_k), (r_j, o_k)\}, 2) \}$$

If we use wild character '?' to denote the above set as ($\{(r_i, ?), (r_j, ?)\}$, 2), we can use a unified syntax to represent both global SoD constraints and local SoD constraints.

Definition 3.3: Separation of Duty (SoD) constraints in ROBAC: $\text{SoD} \subseteq (2^{\text{RO+}} \times \text{N})$ where $\text{RO}^+ \subseteq \text{R}\times\text{O}^+$; $\text{O}^+ = \text{O} \cup \{ ?, * \}$, N is a natural number set such that $\forall (ros, n) \in \text{SoD}, |ros| \geq n \geq 2$.

The interpretation of SoD depends on whether it is static SoD (SSD) or dynamic SoD (DSD). The meaning of select sample static SoD notations is shown in Table 10.

Table 10. Separation of Duty Constraints (SoD) in ROBAC

Element in SoD	Meaning
$(\{ (r_i, ?), (r_j, ?)\}, 2)$	no user can be assigned to both r_i and r_j for any same organization in O (global SoD).
$(\{ (r_i, o_k), (r_j, o_l)\}, 2)$	no user can be assigned to both r_i in organization o_k and r_j in organization o_l (local SoD).
$(\{ (r_i, o_k), (r_j, ?)\}, 2)$	no user can be assigned to r_i in organization o_k while the user has role r_j in any organization.
$(\{ (r_i, o_k), (r_j, *)\}, 2)$	same as above.
$(\{ (r_i, *), (r_j, *) \}, 2)$	no user can be assigned to both r_i and r_j in any organizations in O.

The difference between wild character '?' and '*' is that all occurrences of '?' will take the same value from O while the different occurrences of '*' can take different values from O. Therefore, $(\{ (r_i, ?), (r_j, ?)\}, 2)$ is less strict than $(\{ (r_i, *), (r_j, *) \}, 2)$ in

ROBAC's SoD. There will be no difference between '?' and '*' if there is only one occurrence in the notation.

Similar to RBAC, the *static SoD (SSD)* in ROBAC is a constraint on user to role and organization pair assignment. The presence of global static SoD signifies that the same individual user can never hold mutually exclusive roles within one organization at any time. Whereas the presence of local static SoD means that the same individual user can never hold mutually exclusive role-organization pairs. To model static SoD, we need to add a condition on set UA such that the UA cannot be in a state which violates static SoD constraints. Formally:

$$\forall (ros, n) \in \text{SSD}, \forall \text{t} \subseteq ros: |t| \geq n \geq 2 \Rightarrow \bigcap_{(r,o) \in t} assigned\_users((r,o)) = \phi.$$

where *assigned_user*(*(r,o)*) is all the users assigned to the role-org pair *(r, o)*. That is:

$$assigned\_users((r,o)) = \{ u \in \text{U} \mid (u, (r,o)) \in \text{UA} \}$$

The global *dynamic SoD* (*DSD*) in ROBAC means that the same user cannot activate mutually exclusive roles for any same organization in a session. The local *DSD* in ROBAC means that the same user cannot activate mutually exclusive role-organization pairs in a session. To model the dynamic SoD, the function *active_role-orgs* needs to be constrained to reflect the DSD constraints. Formally:

$\forall (ros, n) \in \text{DSD}, \forall s \in \text{S}, \forall ro\_subset \in 2^{\text{RO}}, ro\_subset \subseteq active\_role\text{-}orgs(\text{s}),$

$ro\_subset \subseteq ros \Rightarrow |ro\_subset| < n.$

Similarly, we can unify cardinality constraints in ROBAC**.**

Definition 3.4: cardinality constraints in ROBAC, *cardinality*: $RO^{+} \rightarrow N$

where $RO^{+} \subseteq R \times O^{+}$; $O^{+} = O \cup \{ ?, * \}$; and N is a natural number set;

$\forall\ (r, o) \in RO$, $|assigned\_users((r,o))| \leq cardinality((r,o))$

Here the wild character '?' and '*' have the same meaning as those in SoD constraints. Because there is only occurrence in cardinality constraints, there is no difference between '?' and '*' in cardinality constraints. For instance, $cardinality((r,*)) = 5$ is same as $cardinality((r,?)) = 5$, which means that the maximum number of users who can be assigned to the role *r* in any organization is 5.

Other types of constraints on PA (permission assignment) and role activation (session) in ROBAC can be defined along similar lines.

In addition to the constraints on user assignment, permission assignment, and role activation, we can also define constraints on the RO set in ROBAC. Before we formally define RO constraints, we need to introduce organization type concept. Usually, organizations have different types. Some roles can only be associated with certain types of organizations. For example, School_Principle role is only meaningful when it is associated to school type organizations. Here we give a definition of RO constraints in ROBAC.

Definition 3.5: *RO constraints* in ROBAC is based on $ROBAC_0$ plus

- Ot -- a set of organization types;

- *otype*: O $\rightarrow$ Ot -- a function mapping an organization to its type;
- ROC $\subseteq$ R $\times$ Ot -- a set of RO constraints; a *(r, ot)* $\in$ ROC means that the role *r* cannot associate any organizations with organization type *ot*. Formally,

  $\forall (r, o) \in \text{RO} \Rightarrow (r, otype(o)) \notin \text{ROC}$

For the aforementioned B2B example, we can use RO constraint to model the problem instead of listing all role-organization pairs in RO explicitly.

Ot = { School, District, State }

ROC = { (r3, District), (r3, State), (r4, District), (r4, State), (r5, State), (r6, School), … }

Thereby r3 and r4 are roles associated only with Schools, whereas r5 can be associated with District and School but not State.

3.2.4 ROBAC$_3$

ROBAC$_3$ is the base model ROBAC$_0$ plus role hierarchy, organization hierarchy, and constraints. The *assigned_role-orgs*, *active_role-orgs*, and *can_access* in ROBAC$_3$ are same as those in ROBAC$_1$. The constraints in ROBAC$_3$ are slightly different from those in ROBAC$_2$ because we need to consider the impact of both role hierarchy and organization hierarchy on constraints in ROBAC$_3$. Ultimately, ROBAC$_3$ is a combination of ROBAC$_1$ and ROBAC$_2$ with modified definition of *assigned_users* function plus possible constraints on role hierarchy and organization hierarchy.

The *assigned_users*(*(r,o)*) function used in the static SoD definition in ROBAC$_2$ only includes the users directly assigned to *(r, o)* pair. In ROBAC$_3$, the *assigned_users*(*(r,o)*) needs to include both directly assigned users and indirectly assigned users via role hierarchy and organization hierarchy. Formally,

$$assigned\_users((r,o)) = \{ u \in \text{U} \mid \forall (r', o') \in \text{RO},\ \ (r', o') \geq (r, o),\ \ (u, (r', o')) \in \text{UA} \}$$

Where $(r', o') \geq (r, o)$ means $r' \geq r$ in the role hierarchy and $o' \geq o$ in the organization hierarchy. With this modified *assigned_users* definition, all constraints discussed in ROBAC$_2$ are applicable to ROBAC$_3$.

In addition to the constraints defined in ROBAC$_2$, we may define new constraints on role hierarchy (RH) and organization hierarchy (OH) in ROBAC$_3$. For example, we can define constraints on RH to restrict the permission inheritance during the role activation. We may also define constraints to invalidate a particular parent-child relationship in OH during a particular session in ROBAC$_3$.

3.3 Applicability and Expressive Power of ROBAC

What is the best scenario for using ROBAC? Here we try to categorize ROBAC from the applicability perspective.

Definition 3.6: Given a ROBAC based system (U, S, R, RH, O, OH, Op, At, A, RO, P, PA, UA, *atype*, *aorg*), let us define the following notations:

- $o_i, o_j \in \text{O},\ r \in \text{R}$, we say o_i is *compatible* to o_j on r iff $(r, o_i) \in \text{RO} \wedge (r, o_j) \in \text{RO}$. We denote that as $o_i \equiv_r o_j$.

- $o_i, o_j \in$ O, Rc $\subseteq$ R, we say o_i is *compatible* to o_j on Rc iff $\forall r \in$ Rc, $(r, o_i) \in$ RO $\wedge$ $(r, o_j) \in$ RO, (alternately, iff o_i is *compatible* to o_j on all $r \in$ Rc). We denote that as $o_i \equiv_{\text{Rc}} o_j$.
- *compatible_O*: R $\rightarrow 2^{\text{O}}$ -- a function mapping a role to a set of organizations in which all organizations are compatible on the given role; formally, *compatible_O*(r) = { o | $(r, o) \in$ RO }.
- *compatible_O**: $2^{\text{R}} \rightarrow 2^{\text{O}}$ -- a function mapping a set of roles to a set of organizations in which all organizations are compatible on the given set of roles; formally, for a Rc $\subseteq$ R and Rc $\neq \varnothing$, *compatible_O**(Rc) = { o | $\forall r \in$ Rc, $(r, o) \in$ RO }. We define *compatible_O**($\varnothing$) = $\varnothing$.
- *applicable_R*: O $\rightarrow 2^{\text{R}}$ -- a function mapping a organization to a set of roles in which all roles are applicable to the organization; formally, *applicable_R*(o) = { r | $(r, o) \in$ RO }
- ROBAC is *homogeneous* on r, if *compatible_O*(r) = O.
- ROBAC is *heterogeneous* on r, if |*compatible_O*(r)| = 1.
- ROBAC is *partially homogeneous* on Rc, if 1 < | *compatible_O**(Rc)| < |O|.
- ROBAC is *totally homogeneous* on Rc, if *compatible_O**(Rc) = O.
- ROBAC is *heterogeneous* on Rc, if |*compatible_O**(Rc)| = 1.
- *homogeneous index*, *hindex*: $2^{\text{R}} \rightarrow [0, 1]$ – a function mapping a role set to number between 0 to1 (including 0 and1); formally,

  hindex(Rc) = |*compatible_O**(Rc)| / |O|.

The *homogeneous index* function measures the degree of compatibility of a given role set in ROBAC. If a ROBAC is totally homogeneous on Rc, *hindex*(Rc) = 1. If a ROBAC is heterogeneous on Rc, *hindex*(Rc) = 1/|O|. If a ROBAC is partially homogeneous on Rc, the *hindex*(Rc) is between 1/|O| and 1. For completeness, we define *hindex*($\varnothing$) = 0.

In the aforementioned B2B example, the $compatible\_O^*(\{r1, r2\}) = O$. Therefore,

$$hindex(\{r1,r2\}) = |\ compatible\_O^*(\{r1, r2\})| \,/\, |O| = 1$$

which means that the ROBAC in the B2B example is *totally homogeneous* on {*r1, r2*}. Similarly, the ROBAC in the B2C example is *totally homogeneous* on {*parent, kid*}. From the above definition, we can derive some properties of *hindex*.

Theorem 3.1: For any two non-empty subsets Rc_1 , Rc_2 of role set R in ROBAC, if $Rc_1 \subseteq Rc_2$, then $hindex(Rc_1) \geq hindex(Rc_2)$.

Poof: Given condition: $Rc_1 \neq \varnothing$ and $Rc_1 \subseteq Rc_2$, $\forall r \in Rc_1 \Rightarrow r \in Rc_2$.

For any $o \in compatible\_O^*(Rc_2) \Rightarrow \forall r \in Rc_2, (r, o) \in RO \Rightarrow \forall r \in Rc_1, (r, o) \in RO$ $\Rightarrow o \in compatible\_O^*(Rc_1)$. That is: any *o* in $compatible\_O^*(Rc_2)$ is also in $compatible\_O^*(Rc_1)$. So $|\ compatible\_O^*(Rc2)\ | \leq |\ compatible\_O^*(Rc1)\ |$. That means $hindex(Rc_1) = |\ compatible\_O^*(Rc_1)\ | \,/\, |O| \geq |\ compatible\_O^*(Rc_2)\ | \,/\, |O| = hindex(Rc_2)$.

□

Theorem 3.2: If a ROBAC is *totally homogeneous* on Rc, then the ROBAC is *totally homogeneous* on all non-empty subsets of Rc.

Poof: Given the condition and Definition 3.6, we have $compatible\_O^*(Rc) = O$. That means: $hindex(Rc) = | compatible\_O^*(Rc) | / |O| = 1$.

From Theorem 3.1, $\forall Rc' \subseteq Rc \wedge Rc' \neq \emptyset$, $hindex(Rc') \geq hindex(Rc) = 1$ (1)

By definition, $hindex(Rc') \leq 1$ (2)

From (1) and (2), we have $hindex(Rc') = 1$. That is, $compatible\_O^*(Rc') = O$ which means that the ROBAC is *totally homogeneous* on Rc'. □

The higher the *hindex*(R) value is, the more the benefits we can gain by applying ROBAC. The best scenario is that the ROBAC is *totally homogeneous* on the role set R, that is *hindex*(R) = 1. The worst scenario for ROBAC is that the ROBAC is *heterogeneous* on all subsets of the role set R, that is, $\forall Rc \subseteq R$ and $Rc \neq \emptyset$

$hindex(Rc) = 1/|O|$

The *hindex* function quantifies how well the ROBAC model is used for a given problem.

ROBAC models change RBAC models in several ways. Do the ROBAC models increase (or decrease) the expressive power of RBAC models? The following theorem shows that ROBAC and RBAC have the same expressive power. We prove the theorem by using a constructive approach. We first show that for any RBAC based system, a ROBAC based system can be constructed to simulate the behavior of the RBAC system.

Then we show that for any ROBAC based system, a RBAC based system can be constructed to simulate the behavior of the ROBAC system.

Lemma 3.1: Any given RBAC based system can be simulated by a ROBAC based system.

Proof: For any given RBAC based system (U, S, R, RH, Op, A, P, PA, UA), where

- U -- a set of users;
- S -- a set of sessions;
- *user*: S $\rightarrow$ U -- a function mapping a session *s* to a single user *user(s)*
- R -- a set of roles;
- RH $\subseteq$ R $\times$ R -- role hierarchy;
- Op -- a set of operations;
- A = { a_i, i = 1,2, …, n } -- a set of assets;
- P $\subseteq$ Op $\times$ A -- a set of permissions;
- PA $\subseteq$ P $\times$ R -- a many-to-many permission-to-role assignment relation;
- UA $\subseteq$ U $\times$ R -- a many-to-many user-to-role assignment relation;

We can construct a corresponding ROBAC based system (U', S', user', R', RH', Op', A', At', O', RO', P', PA', UA', *atype, aorg*) where

- U', S', user', R', RH', Op', A' are same as U, S, user, R, RH, Op, A in the above RBAC system;

- At’ = { at_i | i = 1, 2, …, n } – asset type set; for each a_i in A, we add a unique at_i in At’ and define $atype(a_i) = at_i$.
- O’ = { o } – a set of organization which only has one organization; $\forall a_i \in$ A, we define $aorg(a_i) = o$;
- RO’ = { (r, o) | $\forall r \in$ R }
- P’ $\subseteq$ Op’ $\times$ At’ -- for each $(op, a_i) \in$ P, we add (op, at_i) in P’;
- PA’ $\subseteq$ P’ $\times$ R’ -- for each $((op, a_i), r) \in$ PA, we add $((op, at_i), r)$ in PA’;
- UA’ $\subseteq$ U’ $\times$ RO’ -- for each $(u, r) \in$ UA, we add $(u, (r,o))$ in UA’.

In the constructed ROBAC system, there is only one organization *o*; all assets are related to *o*; *atype* is a one to one mapping between A and At. A permission (op, a_i) for a role *r* in RBAC is mapped to a permission (op, at_i) for the role *r* in ROBAC where $at_i = atype(a_i)$. Assigning a user to a role *r* in RBAC is mapped to assigning a user to *(r, o)* in ROBAC where *o* is the only organization in O.

A user *u* can perform operation *op* over asset *a* in RBAC means that the user *u* is assigned some role *r* which has some permission *(op, a)*. That is equivalent to that user *u* is assigned to a role-org pair *(r, o)* where *r* has permission *(op, atype(a))* and asset *a* is related to organization *o* in the constructed ROBAC. So this same user *u* can also perform operation *op* over asset *a* in the constructed ROBAC system because of the logic in *can_access* predicate.

Lemma 3.2: Any given ROBAC based system can be simulated by a RBAC based system.

Proof: For any given ROBAC based system (U, S, user, R, RH, O, OH, Op, At, A, RO, P, PA, UA, *atype, aorg*), we can construct a RBAC based system (U', S', user', R', RH', Op', A', P', PA', UA') where

- U', S', user', Op', A' are same as U, S, user, Op, A in the above ROBAC system;
- R' -- a set of roles; for each role-org pair $(r_i, o_j) \in$ RO, we add a unique role r_{ij}' in the R';
- RH' = $\{ (r_{ij}', r_{kl}') \mid r_i \leq r_k$ in RH $\wedge\ o_j \leq o_l$ in OH $\}$;
- P' $\subseteq$ Op' $\times$ A' -- a set of permissions; for each permission (op_i, at_k) in P,

  P' = P' $\cup$ $\{ (op_i, a_k) \mid atype(a_k) = at_k \}$;
- PA' $\subseteq$ P' $\times$ R' -- for each $((op_i, at_k), r_i)$ in the PA,

  PA' = PA' $\cup$ $\{ ((op_i, a_k), r_{im}') \mid atype(a_k) = at_k \wedge (r_i, o_m) \in$ RO $\}$;
- UA' $\subseteq$ U' $\times$ R' -- for each $(u, (r_i, o_j)) \in$ UA, we add (u, r_{ij}') in UA'.

Each role-org pair in ROBAC is mapped to a unique role in the constructed RBAC. Each permission in ROBAC is mapped to a subset of permissions in the RBAC. Each permission to role assignment in ROBAC is mapped to a subset of the permission to role assignments in the RBAC. Each user to role-org pair assignment in ROBAC is mapped to a user to role assignment in the RBAC.

If a user *u* can perform operation *op* over asset *a* in ROBAC, then the user *u* is assigned some role-org pair *(r_i,o_j)* in which r_i has some permission *(op, at)* and *atype*(*a*) = *at* and *aorg*(*a*) = o_j. That is equivalent to a situation where user *u* is assigned to a role r_{ij}*'* which has permission *(op, a)* in the constructed RBAC. So this same user *u* can also perform operation *op* over asset *a* in the constructed RBAC system.

Theorem 3.3: RBAC and ROBAC have same expressive power.

Poof: Lemma 3.1 shows that ROBAC is at least as expressive as RBAC while Lemma 3.2 shows that RBAC is at least as expressive as ROBAC. So RBAC and ROBAC have same expressive power. □

For a ROBAC system with M number of roles and a large number of organizations, say N organizations, a corresponding RBAC may have M×N roles. Using ROBAC to model the problems involving large number of similar organizations can significantly reduce the number of roles and permissions. Although ROBAC and RBAC have the same theoretical-expressive power, the practical benefits of using ROBAC over RBAC for applications involving a large number of similar organizations are obvious. A detail comparison between ROBAC and RBAC is given in the next section.

3.4 Discussion and Related Work

The organization concept in ROBAC introduces a powerful abstraction that can be coupled quite naturally with the traditional concept of roles. For example, we can treat the divisions or project teams in an enterprise as organizations. So ROBAC can also be used in many B2E applications. A user with an assigned role and organization pair *(r, o)* indicates that the user can act as role *r* within the organization *o* and its subordinate organizations. Because ROBAC performs access control based on both roles and association relations between users and protected resources (assets), it is suitable to model privacy-related policies in applications involve a large number of similar organizational units. A preliminary version of ROBAC model was published in [ZZS06].

As the decision logic (*can_access* predicate) in ROBAC is deterministic, we believe that it is viable to develop a general-purpose authorization engine and an administration tool based on our proposed ROBAC models.

In this section we first compare ROBAC with classic RBAC, and subsequently with some existing access control models, which extend RBAC with motivations that are, to some degree, similar to those of ROBAC.

3.4.1 Comparison with RBAC

Users in ROBAC are assigned to role-organization pairs while users in classic RBAC are assigned to roles. Permissions in ROBAC are defined as a subset of $Op \times At$ while permissions in classic RBAC are defined as a subset of $Op \times A$. Usually, $|At|$ is much smaller than |A|. In the above B2B example, $|A| \approx 10{,}000 \times |At|$. In general, the relationship between the number of possible roles in RBAC ($|Rc|_{RBAC}$) and the number of possible roles in ROBAC ($|Rc|_{ROBAC}$) for a same set of job functions can be summarized by the following formula:

$$|Rc|_{RBAC} = |O| \times [\ 1 + (\ |Rc|_{ROBAC} - 1\ ) \times \textit{hindex}(Rc)\ ]$$

For a totally homogenous ROBAC, *hindex*(R) = 1,

$$|R|_{RBAC} = |O| \times [\ 1 + (\ |R|_{ROBAC} - 1\ ) \times 1\ ] = |O| \times |R|_{ROBAC}$$

So the number of possible roles in classic RBAC is |O| times more than the number of possible roles in ROBAC for a totally homogeneous ROBAC.

For heterogeneous ROBAC, hindex(R) = 1 / |O|

Because none of the organizations shares job functionality, we can treat the ROBAC as having one big organization. That is,

$$|O| = 1,\ |R|_{RBAC} = |O| + |R|_{ROBAC} - 1 = |R|_{ROBAC}$$

So the number of possible roles in RBAC and ROBAC are same for a heterogeneous ROBAC.

Compared to RBAC, the number of roles and permissions in ROBAC are reduced dramatically in the above mentioned B2B and B2C examples because the large homogeneous index values in these two examples. The set of applicable role and organization pairs (RO) is a newly introduced concept in ROBAC. The size of RO may become large when there are a large number of organizations involved. Instead of creating RO explicitly, we can define RO implicitly by using some rules or using RO constraints. For example, in the previously mentioned B2B example, we can use the following rules to establish RO implicitly:

- r1 (Type A Report Viewer) and r2 (Type B Report Viewer) can associate with any organizations.
- r3 (Type C Report Viewer) and r4 (Type D Report Viewer) can only associate with school type organizations.

Assuming that there are 8,950 schools, 1,000 districts, and 50 states in the B2B example, we can calculate the *hindex* for the above roles.

hindex({r1, r2}) = 1; *hindex*({r3, r4}) = 8,950 / 10,000 = 0.895

The resulted ROBAC is *totally homogeneous* on subset {r1, r2} but is *partially homogeneous* on subset {r3, r4}. Based on the Theorem 3.1, *hindex* (R) ≤ *hindex*({r3, r4}) where R is the role set including all roles in the ROBAC. So the ROBAC in the B2B example is *partially homogeneous* on R.

For the B2C example, we allow any role to associate with any organization. The resulted ROBAC is *totally homogeneous* on the role set R because the *hindex*(R) = 1. So the B2C example is one of the best scenarios for ROBAC. While the size of RO may be large, the administrative work for RO is small. Detailed comparison between ROBAC and RBAC is listed in Table 11.

Table 11. Comparison between RBAC and ROBAC (with N organizations and M asset types for totally homogenous ROBAC)

	RBAC	**ROBAC**
Number of permissions	$N \times M$	M

Number of roles	N×M	M
Organization hierarchy	N/A	Yes
Role hierarchy	Yes	Yes
Constraints	Yes	Yes
User-role-(org) assignment	Yes	Yes
Permission-role assignment	Yes	Yes
Role administration	Yes	Yes
Number of role-org pairs	N/A	≤ N×M
Role-org pairs administration	N/A	Yes

Because the number of roles and permissions in ROBAC is much smaller than that in RBAC under situations similar to the above two examples, the administrative complexity in Permission-to-Role assignment is significantly reduced. Therefore, using ROBAC to model problems similar to the above illustrated B2B and B2C examples is more succinct and intuitive than using RBAC, and at the same time still keeps RBAC's benefits such as policy neutrality, principle of least privilege, separation of duty, and ease of management. Although ROBAC and RBAC have the same theoretical-expressive power, the practical benefits of using ROBAC over RBAC for applications involving a large number of similar organizations are quite evident. It is more beneficial to use ROBAC where privacy is a major concern and in situations involving many organizational units across which job functions are similar (higher homogeneous index value). It is not worthwhile to use ROBAC when there is no job function similarity across organizational units (lowest homogeneous index value).

3.4.2 Comparison with Role Templates

In the perspective of restricting access to a subset of contents, the role templates [GI97] proposed by Giuri and Iglio has similar effectiveness as ROBAC. In a role template, a parameterized privilege is defined as ($am, o, exp(v_1, ..., v_n)$) where *am* is an access mode (same as an operation in ROBAC) that can be performed on object *o* which is similar to the asset concept in ROBAC. The $exp(v_1, ..., v_n)$ is a logical expression with unbound variables $v_1, ...v_n$. The parameterized privilege means to perform operation *am* over *o* while $exp(v_1, ..., v_n)$ is true. The unbound variables in the parameters of the corresponding role template need to be bound when a role is assigned to a user. Although role templates can model security policies in our B2C example by defining organization, objects, and users as unbound variables and "related to" as a logical expression, the approach is not as straightforward as with ROBAC.

Because the values of the variables in role templates are un-structured, it is difficult to use role templates to model the aforementioned B2B example. In general, role templates can achieve the modeling capability of $ROBAC_0$ with similar complexity, but it is very difficult for role templates to achieve the modeling capability of $ROBAC_1$ due to the flexibility introduced by organization hierarchies used in these ROBAC models. We believe that the administrative tasks in ROBAC are much simpler than those in role templates approach.

3.4.3 Comparison with Team-based Access Control (TMAC)

In TMAC [Tho97], the notion of "team" is proposed as an abstraction that groups users in specific roles with the objective of accomplishing a specific task or goal. As recognized in TMAC, it is very difficult for classic RBAC to enforce the following two requirements at the same time: scalable permission assignment and fine-grained, run-time permission activation at the level of individual users and objects. The notions of "team" in TMAC and "organization" in ROBAC are different. Team in TMAC or its variants [GMPT01] and [Wang99] represents a group of users and has roles and permissions derived from the roles and permissions of the users in the group while organization in ROBAC does not have roles or permissions. Team members share a common goal and may share a default set of permissions for their cooperative work. The teams in TMAC are flat while the organizations in ROBAC can be structured. In TMAC, an access decision is based on a team's permissions, the user's contexts, and the object's contexts, while in ROBAC a decision is based on user's roles and the indirect association between the user and the assets via organizations. Both TMAC and ROBAC realize the importance to distinguish object (asset) type and object instance (asset) for scalable permission assignment in RBAC. With TMAC we find that it is possible to model the B2C example, but it is very complex to model the B2B example due to the presence of organization hierarchy.

Further, we can simulate most components (except for the special role *team header*) of TMAC with a ROBAC$_0$ model directly. For the *team header* role in TMAC, we can simply create a role with special permissions in ROBAC to model it.

3.4.4 Comparison with Organizational Units

Perwaiz and Sommerville [PS01] utilize organizational context to restrict the permissions of roles. An organization unit (OU) in their paper has its own permissions. The maximum permissions of a role in an organization unit are the intersection of the role's permissions and the organization unit's permissions. So the OU in Perwaiz's approach is more like the "team" notion in the TMAC than to the "organization" in ROBAC, since an organization in ROBAC does not associate with permissions. The access control decision process in ROBAC is also different from that in [PS01]. In the perspective of reducing the number of roles in RBAC, Perwaiz and Sommerville's approach renders some effects similar to those yielded by ROBAC. But the administration of ROBAC is simpler since permissions are not assigned to organizations.

3.4.5. Comparison with Group Based RBAC (GB-RBAC)

The GB-RBAC model [LZQX06] incorporates the component of groups into the RBAC96 model. Roles in GB-RBAC are divided as group roles and user roles. Groups are assigned to group roles. Users are assigned to user roles. When a user become a member of a group, the user acquires all group roles that group has. So the group concept in GB-RBAC is more like team concept in TMAC than the organization concept in ROBAC.

3.4.6. Comparison with Organization Based Access Control (OrBAC)

In OrBAC model [KBBBCDMST03], Kalam et al. consider roles that subjects, actions or objects are assigned in an organization. The role of a subject is simply called a role as in the RBAC model. The role of an action is called an activity whereas the role of an object is called a view. An organization specifies security policy by defining security rules which specify that some roles are permitted or prohibited to carry out some activities on some views. These security rules may be activated depending on contextual conditions. Security rules are modeled using a 6-places predicate:

- **security_rule**(**type, org, role, activity, view, context**) where **type** is either *permission* or *prohibition*.

For example, **security_rule**(*permission*, *a_hosp*, *nurse*, *consult*, *medial_record*, *urgency*) means that in organization a*_hosp*, a nurse is permitted to consult a medical record in the context of urgency.

The organizational policy is then used to automatically derive concrete configurations of Policy Enforcement Point. OrBAC model uses the following 3-places predicates to associate subjects, actions, and objects with roles in organizations:

- **empower(org, subject, role)**: means that **subject** is empowered with **role** in organization **org**.
- **consider(org, action, activity)**: means that **action** is considered as an implementation of **activity** in organization **org**.
- **use(org, object, view)**: means that **object** is used in **view** in organization **org**.

The contextual condition is modeled by the following predicate:

- **hold(org, subject, action, object, context)**: means that **subject** performs **action** on **object** in context **context** within organization **org**.

The general principle of deriving concrete privileges that apply to subject, action, and object from organizational security rules is:

concrete_privilege(*type, s, act, obj*) :-
security_rule(*org, type, r, a, v, cxt*),
empower(*org, s, r*),
use(*org, obj, v*),
consider(*org, act, a*),
hold(*org, s, act, obj, cxt*)

that is a subject *s* has a concrete privilege of type *type* to perform an action *act* on an object *obj* if

(1) organization *org* assigns to role *r* a security rule of type *type* to perform activity *a* on view *v* in context *cxt* and

(2) organization *org* empowers subject *s* in role *r* and

(3) *org* uses object *obj* in view *v* and

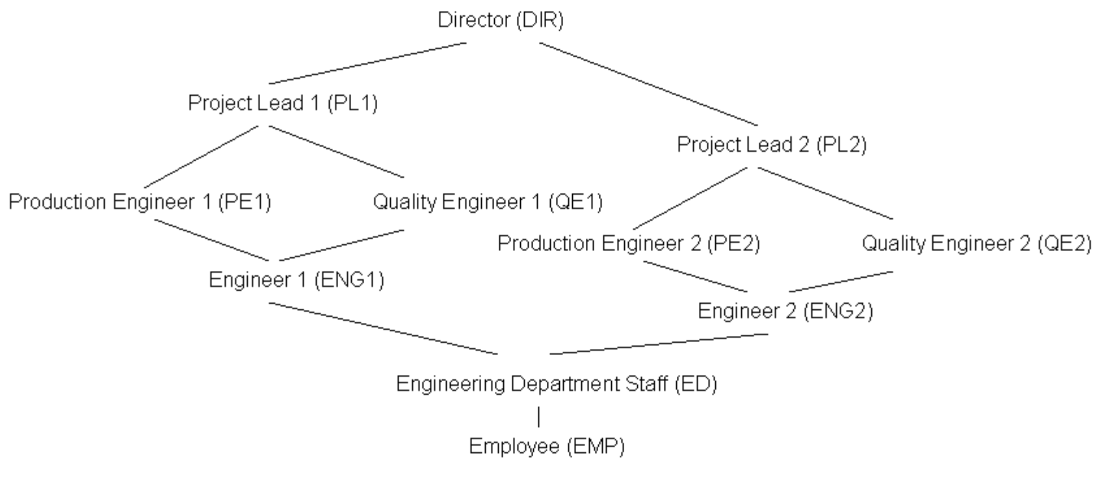
(4) *org* considers that action *act* implements the activity *a* and

(5) context *cxt* is active when subject *s* is performing action *act* on object *obj* in *org*.

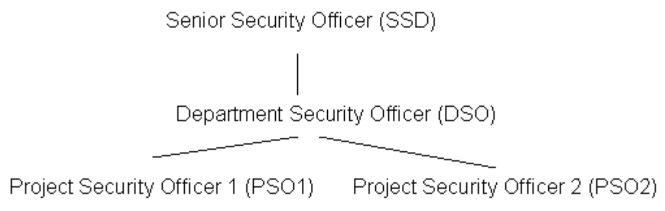
In OrBAC, organization is used as an actor while organization in ROBAC is passive. The organizations in OrBAC are flat (no relationship among the organizations) while the organizations in ROBAC are structured. The **security_rule** predicate in OrBAC has an effect similar to assigning permissions to role within an organization. But a role in OrBAC may have completely different privileges in different organizations. The **empower** predicate in OrBAC is similar to assigning a user (subject) to a role-org pair in ROBAC. Similar to ROBAC, OrBAC restricts role privileges with an organization. The concrete privilege derivation logic in OrBAC has a similar effect as *can_access* predicate in ROBAC but the logic in *can_access* is much simpler. For each organization, OrBAC need define its own security rules while ROBAC share the same role definition. ROBAC is developed to solve the RBAC scalability problem due to many similar organizations while OrBAC is developed to address organization bounded security policy modeling problem (more abstract from implementation). For aforementioned B2B example, OrBAC has to define security rules for each organization. So OrBAC does not scale up well for applications involving many organizations.

Chapter 4. Administrative ROBAC Model

In any security system, administrative actions need to be controlled. Using a role-based method to control RBAC administrative tasks is often a preferred way because it can share the underline authorization mechanism. There are two main approaches to perform RBAC administration. One is centralized such as Gavrila and Barkley's NIST model [GB98] and Nyanchama and Osborn's role graph model [NO99] where one or more security administrators perform all administrative tasks. Another is decentralized such as Sandhu et al's ARBAC97 model [SBM99], Crampton and Loizou's SARBAC model [CL03], Oh et al's ARBAC02 model [OSZ06], and Bhatti et al's X-GTRBAC admin [BJBG04] where administrative tasks are distributed among many different administrators in a controlled manner. These role-based decentralized approaches usually add a separate administrative role hierarchy in the original RBAC model. Figure 5 shows an example of regular role hierarchy and administrative role hierarchy created in ARBAC97 model for an engineering department within an enterprise.



(a) An example of regular role hierarchy


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

(b) An example of administrative role hierarchy

Figure 5. Examples of role hierarchy using classic administrative RBAC

where Department Security Officer (DSO) can perform administrative tasks on the department level and Project Security Officer (PSO) can perform administrative tasks on the project level. Each project not only has its own instance of Project Leader role, Production Engineer role, and Quality Engineer role in the regular role hierarchy, but also has its own instance of Project Security Officer role in the administrative role hierarchy. If there are a large number of projects, say in the degree of hundreds or more, in an enterprise, both the regular role hierarchy and administrative role hierarchy will become very clumsy and hard to manage correctly. Many classic administrative RBAC models, such as ARBAC97, do not scale up well when a large number of similar organizational units are involved.

As we mentioned earlier, the organization concept in ROBAC should not be treated literally. For business to employee (B2E) applications, we may treat divisions or project teams in an enterprise as organizations. ROBAC model is suitable to be used in large enterprises where there are many similar organizational units (higher homogeneous index value). A decentralized administrative approach is preferred for large enterprises. Based on the observation that administrative tasks are very similar in many enterprises, we believe that it is an effective approach to utilize the role and organization based access control concept to manage administrative tasks in ROBAC. The main topic of this chapter is to present a comprehensive model for role and organization based administration of ROBAC.

4.1 Administrative Issues in ROBAC

In classic RBAC, major administrative tasks include assigning users to roles, assigning permissions to roles, and adjusting role hierarchy. Thus, some role based administrative models, such as ARBAC97, have three separate sub-models: URA97 (user-role assignment), PRA97 (permission-role assignment), and RRA97 (role-role assignment), to deal with these three major administrative tasks. There are more components in ROBAC than those in RBAC, making the administration of ROBAC more multifaceted than

RBAC. Following the ARBAC97 approach, we divide administrative tasks into the following categories: assigning users to role-organization pairs, assigning permissions to roles, managing roles and role hierarchy, managing organization and organization hierarchy, and managing role and organization association. The reason is that these administrative activities in ROBAC affect user's access rights in different ways.

In most ROBAC based systems, organizations O, organization hierarchy OH, roles R, and role hierarchy RH are expected to be mostly static and change very slowly. They are typically available in advance and will not change frequently after the ROBAC is set up. Therefore, the major administrative effort after initial setup is on the permission-to-role assignment and user-to-role-and-organization assignment.

In the next section, a comprehensive role and organization based administrative model for ROBAC is proposed. We call this new model AROBAC07 (administrative ROBAC'07).

4.2 AROBAC07 Model

AROBAC07 has five sub-models.

1. UROA07 (user to role and organization pair assignment '07) is concerned with user to role and organization pair assignment;
2. PRA07 (permission to role assignment '07) deals with permission-role assignment;
3. RRA07 (role to role assignment '07) manages roles and role hierarchy;

4. OOA07 (organization to organization assignment ’07) handles organizations and organization hierarchy; and

5. ROA07 (role to organization assignment ’07) controls applicable association between roles and organizations.

The development of AROBAC07 is inspired by ARBAC97 [SBM99], SARBAC [CL03], and ARBAC02 [OSZ06]. Our AROBAC07 model will be presented in the context of ROBAC$_1$. Its interpretation for ROBAC$_0$, ROBAC$_2$, and ROBAC$_3$ is straightforward.

ARBOAC07 adds some additional sets, relationships, and functions to the ROBAC model. Similar to ROBAC, an administrative user is assigned to administrative role and organization pairs instead of administrative roles only. An administrative decision is made based on both role and organization information. The common elements of AROBAC07 are described in Definition 4.1 and Figure 6 shows some relationship among the elements of AROBAC07.

Definition 4.1: AROBAC07 has the following components:

- U, S, O, OH, Op, A, At, P, RO, PA, UA, *user, atype, aorg, assigned_role-orgs*, *active_role-orgs, can_access* -- same as those from ROBAC;
- RR -- a set of regular roles (renamed R in ROBAC);
- RRH $\subseteq$ RR $\times$ RR – regular role hierarchy (renamed RH in ROBAC);
- AR -- a set of administrative roles (same as AR in ARBAC97), where

$RR \cap AR = \emptyset$.

- $ARH \subseteq AR \times AR$ -- administrative role hierarchy (same as ARH in ARBAC97);
- $R = RR \cup AR$ -- the set of all roles;
- $ARRA \subseteq AR \times RR$ -- a many-to-many administrative role to regular role assignment;
- $RH = RRH \cup ARH$ -- a combined role hierarchy;
- $UO \subseteq U \times O$ -- a set of user-organization affiliations;
- $PO \subseteq P \times O$ -- a set of applicable permission-organization associations;
- CRU – a set of applicable prerequisite conditions for users;
- CRP – a set of applicable prerequisite conditions for permissions;
- CAN_ASSIGN_USER $\subseteq$ ARRA $\times$ CRU - an association defines the constraints when assigning users to role-organization pairs;
- CAN_REVOKE_USER $\subseteq$ ARRA $\times$ CRU - an association defines the constraints when revoking users from role-organization pairs;
- *can_assign_user*: $S \times U \times RO \rightarrow \{true, false\}$ – a predicate which indicates that if *can_assign_user(s, u, (r,o))* is true then user *u* can be assigned to the role-org pair *(r,o)* within the session *s* (the definition is described in UROA07);

- *can_revoke_user*: $S \times U \times RO \rightarrow \{true, false\}$ – a predicate which indicates that if *can_revoke_user(s, u, (r,o))* is true then user *u* can be revoked from role-org pair *(r,o)* within the session *s* (the definition is described in UROA07);
- CAN_ASSIGN_PERMISSION $\subseteq$ ARRA $\times$ CRP - an association defines the constraints when assigning permissions to roles;
- CAN_REVOKE_PERMISSION $\subseteq$ ARRA $\times$ CRP - an association defines the constraints when revoking permissions from roles;
- *can_assign_permission*: $S \times P \times RR \rightarrow \{true, false\}$ – a predicate which indicates that if *can_assign_permission(s, p, r)* is true then the permission *p* can be assigned to the regular role *r* within the session *s* (the definition is described in PRA07);
- *can_revoke_permission*: $S \times P \times RR \rightarrow \{true, false\}$ – a predicate which indicates that if *can_revoke_permission(s, p, r)* is true then the permission *p* can be revoked from the regular role *r* within the session *s* (the definition is described in PRA07);
- *can_modify_R*: $S \times 2^{RR} \rightarrow \{true, false\}$ -- a predicate which indicates that if *can_modify_R*(*s, rset*) is true then the user *user(s)* can modify the roles and their relationship inside the role set *rset* within the session *s* (the definition is described in RRA07);
- *can_modify_O*: $S \times 2^{O} \rightarrow \{true, false\}$ -- a predicate which indicates that if *can_modify_O*(*s, oset*) is true then the user *user(s)* can modify the organizations

and their relationship inside the organization set *oset* within the session *s* (the definition is described in OOA07);

- *can_modify_RO*: S × R × O → {true, false} -- a predicate which indicates that if *can_modify_RO*(*s, r, o*) is true then the user *user(s)* can associate or disassociate role *r* with organization *o* within the session *s* (the definition is described in ROA07);

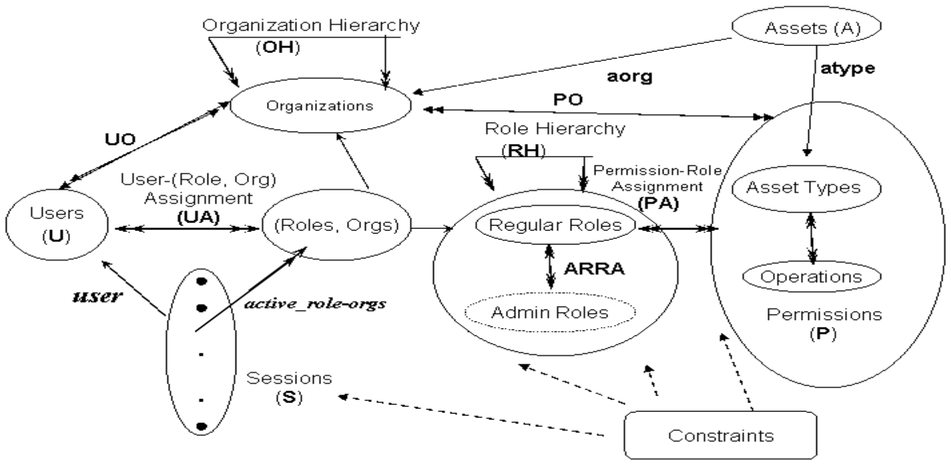


Figure 6. AROBAC07 Model

In AROBAC97, UO defines user's organization affiliation. A user may be affiliated with multiple organizations. UO is usually pre-determined by Human Resource (HR) departments of individual organizations. The applicable permission-organization association set PO defines permissions applicable to organizations. Similar to permission set P, PO is pre-determined via joint efforts between HR and Information Technology (IT) departments. So we do not include the management of UO, P, and PO in our model. The set ARRA can be considered as the administrative role's permission over the regular roles. Some constraints need to be enforced when creating or modifying the role hierarchy (RH) or organization hierarchy (OH) such as no circular reference. The detailed descriptions of the prerequisite condition sets CRU and CRP, the predicates *can_assign_user, can_revoke_user, can_assign_permission, can_revoke_permission, can_modify_R, can_modify_O,* and *can_modify_RO* are discussed in the following corresponding subsections.

4.2.1 The UROA07 Model

The UROA07 model deals with managing user to role-organization pair assignment. It provides two predicates to determine whether the current session can grant a user membership in a role-organization pair (or simply role-org pair) or revoke a user membership in a role-org pair. Before introducing the details of UROA07 model we need some definitions.

Definition 4.2: *user prerequisite condition (upc)* - a *upc* is a boolean expression using the usual $\wedge$ and $\vee$ operators on terms of form of $(r, ?)$, $\neg(r, ?)$, (r, o), and $\neg(r, o)$ where (r, o) is a role-org pair belongs to RO. A *user prerequisite condition* is evaluated for a user u by interpreting (r, o) to be true if $(\exists r' \geq r, \exists o' \geq o)$ $(u, (r', o')) \in$ UA and $\neg(r, o)$ is true if *(r, o)* is not true. Here "*?*" is a place holder for any $o \in O$, and *(r, ?)* is true for user *u* if $(\exists r' \geq r,\ \exists o' \geq ?,\ (u, (r', o')) \in UA)$ and $\neg(r, ?)$ is true if *(r, ?)* is not true. CRU is a set including all applicable *upc*s plus a *null* element. The *null* is interpreted as true for any user.

Note: The *(r, ?)* expression represents a condition template where the value of "*?*" is set to *o* when the system is asked whether a user can be assigned to a role-organization pair *(r, o)*. We will explain it in an example later.

Definition 4.3: *omembers*: $O \rightarrow 2^U$, is a function mapping an organization to a set of users who are affiliated with the organization; formally, $omembers(o) = \{ u \mid (u, o) \in UO \}$; $omembers^*(o) = \{ u \mid \exists o' \leq o, (u, o') \in UO \}$

Note: *omembers*(*o*) is the set of all users affiliated with organization *o* and *omembers**(*o*) is the set of all users affiliated with organization *o* or its subordinate organizations.

Definition 4.4: *apermissions*: $AR \rightarrow 2^{RR}$, is a function mapping an administrative role to a set of regular roles which the administrative role has administrative privilege over; formally, $apermissions(ar) = \{ r \mid (ar, r) \in ARRA \}$; $apermissions^*(ar) = \{ r \mid \exists ar' \leq ar, (ar', r) \in ARRA \}$

Note: *apermisions*(*ar*) is the set of regular roles where the administrative role *ar* has administrative privilege and *apermissions**(*ar*) is the set of regular roles where administrative role *ar* or its junior administrative roles has administrative privilege.

Definition 4.5: *may_manage_user*: AR $\times$ U $\times$ RO $\times$ CRU $\rightarrow$ {true, false} - a predicate defined as *may_manage_user(ar, u, (r,o), c)* is true iff ($r \in$ *apermissions**(ar)) $\wedge$ c $\wedge$ ($u \in$ *omembers**(o)).

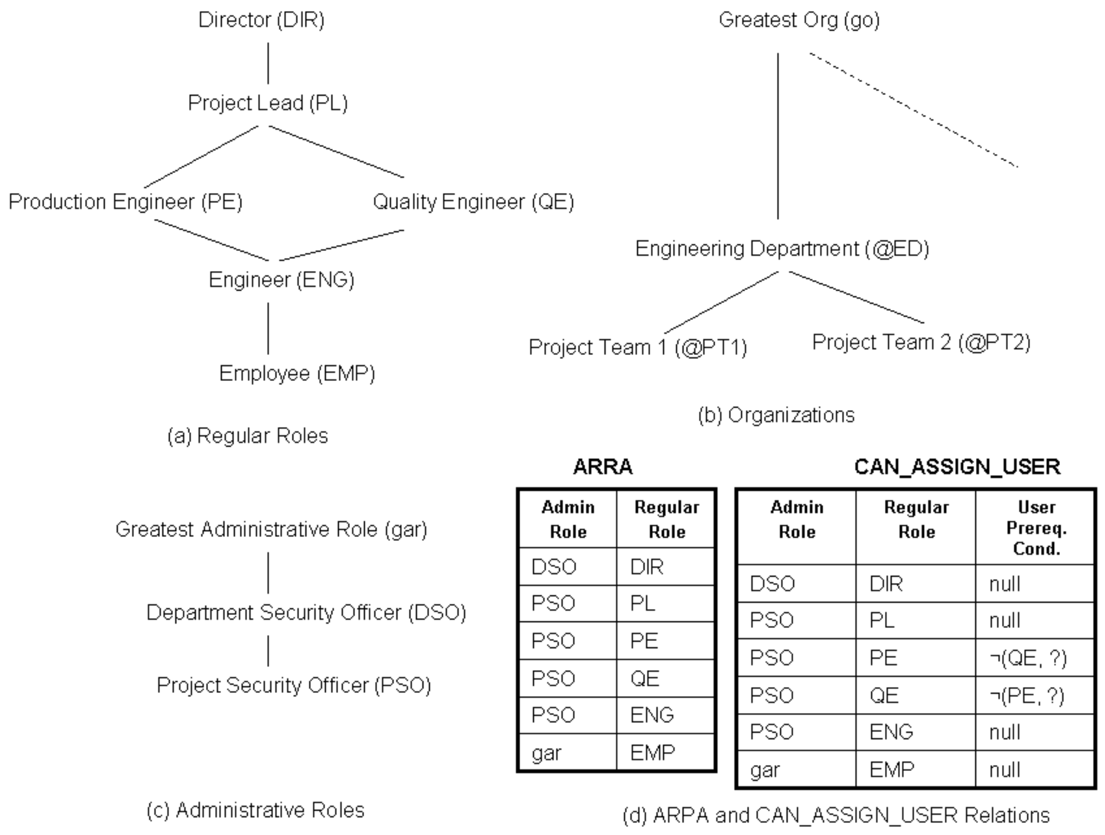
Note: The definition of *may_manage_user(ar, u, (r,o), c)* indicates that a user with administrative role *ar* may manage the user *u* with respect to the role-org pair *(r, o)* if and only if the user *u* satisfies the user prerequisite condition *c* and is affiliated to the organization *o* or its subordinate organizations and the administrative role *ar* or its junior administrative roles can perform administrative tasks on role *r*. The *may_manage_user* predicate is used as a sub-routine in the following UROA07 Grant Model and UROA07 Revoke Model.

Definition 4.6: The UROA07 Grant Model – *can_assign_user* predicate controls whether a user can be assigned to a role-org pair within a session. Formally, *can_assign_user(s, u, (r,o))* is true iff ($\exists o' \geq o$, $\exists (ar, o') \in$ *active_role-orgs*(s)) $\wedge$ ($\forall c \in$CRU, *((ar, r),c)* $\in$ CAN_ASSIGN_USER $\wedge$ *may_manage_user(ar, u, (r,o), c)*).

The definition of *can_assign_user(s, u, (r,o))* in UROA07 indicates that a user (*user(s)*) in a session *s* can assign a user *u* to a role-org pair *(r, o)* if and only if *user(s)* has an active role-org pair *(ar, o)* (explicitly or implicitly via organization hierarchy) in that session and user *u* satisfies all related user prerequisite conditions defined in CAN_ASSIGN_USER and is affiliated to the organization *o* or its subordinate organizations and the administrative role *ar* or its junior administrative roles can perform administrative tasks on role *r*.

The user prerequisite condition in UROA07 is likely to be empty in most cases. However, because it may be used to model more complex policies, we include it in the model.

To appreciate the benefits behind the UROA07 model, let us remodel the aforementioned engineering department example in Figure 5 using AROBAC07. Figure 7 shows the AROBAC07 model for the engineering department problem in Figure 5. Here we treat a project team as an organization. We further assume that roles in different project teams within the engineering department perform similar tasks and a team member can only access the resource related to the team he/she is in. This assumption will usually hold because most enterprises need to enforce a unified security policy across multiple teams.



ARRA

Admin Role	Regular Role
DSO	DIR
PSO	PL
PSO	PE
PSO	QE
PSO	ENG
gar	EMP

CAN_ASSIGN_USER

Admin Role	Regular Role	User Prereq. Cond.
DSO	DIR	null
PSO	PL	null
PSO	PE	¬(QE, ?)
PSO	QE	¬(PE, ?)
PSO	ENG	null
gar	EMP	null

Figure 7. Role and organization hierarchies using administrative ROBAC

We can see that both the regular role hierarchy (Figure 7(a)) and the administrative role hierarchy (Figure 7(c)) in AROBAC07 are simpler. For the moment please ignore the *Greatest Administrative Role* (*gar*) and the *Greatest Organization* (*go*). We will explain these later. Here we prefix "@" in the front of organizations to distinguish them from roles. For example, a user with an active role-org pair (PSO, @PT1) is a security administrator in project team 1. *may_manage_user*(PSO*, u, (*PE, @PT1*),* ¬(QE, ?)) is true if user *u* is affiliated with project team 1 (@PT1) and *u* is not a QE inside the project team 1. Based on the UROA07 grant-model, the user with active role-org pair (PSO, @PT1) can assign membership of roles: PL, PE, QE, and ENG within project team 1, to users affiliated with the project team 1 but he/she cannot assign these users to roles within project team 2 and cannot assign users not affiliated to project team 1 to any roles. He/she also cannot assign both (PE, @PT1) and (QE, @PT1) to the same user because of the user prerequisite conditions, ((PSO, PE), ¬(QE, ?)) and ((PSO, QE), ¬(PE, ?)) , defined in CAN_ASSIGN_USER at Figure 7(d), which represents a global separation of duty constraint in ROBAC.

If there are more project teams in the engineering department or there are more engineering departments, we need only to add them in the organization set O and organization hierarchy OH but do not need to change other settings in ROBAC. Even with hundreds or more project teams, the UROA07 model will scale up very nicely whereas previous models will become incomprehensible.

To complete the definition of the UROA07 model we define the Revoke Model as follows.

Definition 4.7: The UROA07 Revoke Model – *can_revoke_user* predicate controls whether a user can be revoked from a role-org pair within a session. Formally, *can_revoke_user(s, u, (r,o))* is true iff $(\exists o' \geq o, \exists (ar, o') \in active\_role\text{-}orgs(s)) \wedge (\forall c, ((ar, r), c) \in \text{CAN\_REVOKE\_USER} \wedge may\_manage\_user(ar, u, (r,o), c))$.

The concept of weak revocation and strong revocation in ARBAC97 can be considered in UROA07 too.

Definition 4.8: *weak revocation* – only the explicit membership of the specified role-org pair *(r, o)* in *can_revoke_user(s, u, (r,o))* for the user *u* is revoked.

Definition 4.9: *strong revocation* – all memberships (either explicit or implicit) of the specified role-org pair *(r, o)* in *can_revoke_user(s, u, (r,o))* for the user *u* are revoked.

The *weak revocation* only removes an existing entry *(u, (r,o))* from UA while the *strong revocation* removes all existing entries *(u, (r', o'))*, such that $r' \geq r$ and $o' \geq o$, from UA.

4.2.2 The PRA07 Model

The PRA07 model deals with managing permission to role assignment. Similar to UROA07, it also provides two predicates to determine whether a session can assign or revoke permission to or from a role. We give the following definitions in analogy to similar definitions for URAO07.

Definition 4.10: *permission prerequisite condition (ppc)* – a *ppc* is a boolean expression using the usual $\wedge$ and $\vee$ operators on terms of form of *r* and $\neg r$ where *r* is a role in RR. A *permission prerequisite condition* is evaluated for a permission *p* by interpreting *r* to be true if ($\exists r' \leq r,\ (p, r') \in$ PA) and $\neg r$ is true if ($\forall r' \geq r,\ (p, r') \notin$ PA. CRP is a set that includes all applicable *ppc*s plus a *null* element. The *null* is interpreted as true for any permission.

Definition 4.11: *opermissions*: O $\rightarrow$ 2^P, is a function mapping an organization to a set of permissions which are applicable to the organization; formally, *opermissions*(*o*) = { *p*: P | *(p, o)* $\in$ PO }; *opermissions**(*o*) = { *p*: P | $\exists$*o' ≤ o, (p, o')* $\in$ PO }

Note: *opermissions*(*o*) is the set of all permissions applicable to the organization *o* and *opermissions**(*o*) is the set of all permissions applicable to the organization *o* or its subordinate organizations.

Definition 4.12: *can_manage_permission*: RO $\times$ P $\times$ RR $\times$ CRP $\rightarrow$ {true, false} – a predicate defined as *can_manage_permission((ar, o), p, r, c)* is true iff (*r* $\in$ *apermissions**(*ar*)) $\wedge$ *c* $\wedge$ (*p* $\in$ *omembers**(*o*)).

Note: The definition of *can_manage_permission((ar, o), p, r, c)* indicates that a user who is a member of administrative role-org pair (*ar, o)* can manage the permission *p* for the role *r* if and only if permission *p* satisfies the permission prerequisite condition *c* and is applicable to the organization *o* or its subordinate organizations and the administrative role *ar* or its junior administrative roles can perform administrative tasks on the regular role *r*.

This leads to the following Grant Model.

Definition 4.13: The PRA07 Grant Model – *can_assign_permission* predicate controls whether a permission can be assigned to a role within a session. Formally, *can_assign_permission(s, p, r)* is true iff $\exists$*(ar, o)* $\in$ *active_role-orgs*(*s*) $\wedge$ ($\forall$*c*, *((ar, r), c)* $\in$ CAN_ASSIGN_PERMISSION $\wedge$ *can_manage_permission((ar, o), p, r, c)*.

The definition of *can_assign_permission(s, p, r)* in PRA07 indicates that a user (*user(s)*) in a session *s* can assign a permission *p* to a role *r* if and only if *user(s)* has an active role-org pair *(ar, o)* in that session, and the administrative role *ar* or its junior administrative roles have administrative right over the regular role *r,* and the permission *p* is applicable to the organization *o* or its subordinate organizations, and the permission *p* satisfies all specified permission prerequisite conditions defined in CAN_ASSIGN_PERMISSION. The permission prerequisite condition in PRA07 is optional.

Finally we have the following Revoke Model.

Definition 4.14: The PRA07 Revoke Model – *can_revoke_permission* predicate controls whether a permission can be revoked from a role within a session. Formally, *can_revoke_permission(s, p, r)* is true iff $(\exists(ar, o) \in active\_role\text{-}orgs(s) \wedge (\forall c, ((ar, r), c) \in$ CAN_REVOKE_PERMISSION $\wedge$ *can_manage_permission((ar, o), r, c)*).

4.2.3 The RRA07 Model

The RRA07 model deals with managing roles and role hierarchy. It provides one predicate called *can_modify_R* to determine whether the current session can add/remove a role or change role hierarchy during the session. To define *can_modify_R* predicate, we need to introduce some definitions.

Definition 4.15: *rjuniors*: $R \rightarrow 2^R$, is a function mapping a role to its junior roles; formally, $rjuniors(r) = \{ r' : R \mid r' < r \}$

Definition 4.16: *rseniors*: $R \rightarrow 2^R$, is a function mapping a role to its senior roles; formally, $rseniors(r) = \{ r' : R \mid r' > r \}$

Definition 4.17: *rfamily*: $R \rightarrow 2^R$, is a function mapping a role to a set of roles including itself and its junior role and senior roles; formally, $rfamily(r) = \{r\} \cup rjuniors(r) \cup rseniors(r)$

Definition 4.18: *rfamilies*: $2^R \rightarrow 2^R$, is a function mapping a set of roles to a set of roles including all families of its members; formally, $rfamilies(\{r_1, r_2, \ldots r_n\}) = rfamily(r_1) \cup rfamily(r_2) \cup \ldots \cup rfamily(r_n)$

It is worth noting that *rfamily* and *rfamilies* only include recursive direct family members and do not include siblings.

Definition 4.19: *permissible administrative role set*, *parset*: $AR \rightarrow 2^{RR}$, is a function mapping an administrative role to a set of regular roles in which the administrative role can modify the regular role hierarchy. Formally,

$$parset(ar) = \{ r : RR \mid (ar, r) \in ARRA \wedge rfamily(r) \subseteq apermissions^*(rfamily(ar)) \}$$

The above definition indicates that *parset* for an administrative role *ar* includes all of the regular roles it has administrative privilege such that the regular role's family is a part of the regular roles the *ar*'s family has administrative privilege over. For example *parset*(PSO) = { PL, PE }.

Because modifying role hierarchy affects all organizations in ROBAC, we should only allow the users at the highest organization level to perform these actions. We introduce an artificial organization called *greatest organization* (*go*) which is the ancestor for all organizations in O (see Figure 3). Now let us define the *can_modify_R* predicate.

Definition 4.20: *can_modify_R*: $S \times 2^{RR} \rightarrow$ {true, false} -- a predicate defined as *can_modify_R*(*s, rset*) is true iff $\exists$*(ar, go)* $\in$ *active_role-orgs*(*s*) $\wedge$ *rset* $\subseteq$ *parset*(*ar*).

The definition of *can_modify_R* means that a user *user(s)* in a session *s* can modify the relationship within the role set *rset* if and only if that the user has an active administrative role *ar* and paired with the greatest organization *go* in that session and the role set *rset* is a subset of the *permissible administrative role set* of *ar*. Here the modification within a set of roles means adding/deleting an edge or adding/removing a role. For example, according to Figure 7, PSO can remove the edge between PL and PE but cannot remove the edge between ENG and QE because QE and ENG are not in its *permissible administrative role set*.

The construction of *parset* in AROBAC07 is very similar to the concept of administrative scope in RH4 of SARBAC. We will discuss it further in the related work section.

4.2.4 The OOA07 Model

The OOA07 model deals with managing organizations and organization hierarchy. Similar to RRA07, it provides one predicate called *can_modify_O* to determine whether the current session can add/remove an organization or change organization hierarchy during the session. To define *can_modify_O* predicate, we also need to introduce some definitions.

Definition 4.21: *ojuniors*: O → 2^O, is a function mapping an organization to its subordinate organizations. Formally, *ojuniors*(*o*) = { *o'*: O | *o'* < *o* }

Definition 4.22: *oseniors*: O → 2^O, is a function mapping an organization to its parent organizations. Formally, *oseniors*(*o*) = { *o'*: O | *o'* > *o* }

Definition 4.23: *ofamily*: O → 2^O, is a function mapping an organization to a set of organizations including itself and its subordinate organizations and parent organizations. Formally, *ofamily*(*o*) = {*o*} ∪ *ojuniors*(*o*) ∪ *oseniors*(*o*)

Definition 4.24: *permissible administrative organization set*, *paoset*: O → 2^O, is a function mapping an organization to a set of organizations. Formally, *paoset*(*o*) = { *o'* : O | *o'* < *o* ∧ *ofamily*(*o'*) ⊆ *ofamily*(*o*) }

For example, *paoset*(@ED) = { @PT1, @PT2 } according to Figure 7(b).

Because modifying organization hierarchy has some global effects on access rights in ROBAC, we should only allow the most senior administrative role to modify the organization hierarchy. We can pre-define a most senior administrative role called *gar* (*greatest administrative role*) in AR. Here is the definition for *can_modify_O* predicate.

Definition 4.25: *can_modify_O*: S × 2^O → {true, false} -- a predicate defined as *can_modify_O*(*s, oset*) is true iff ∃*(gar, o)* ∈ *active_role-orgs*(*s*) ∧ *oset* ⊆ *paroet*(*o*).

The definition of *can_modify_O* means that a user *user(s)* in a session *s* can modify the relationship within the organization set *oset* if and only if that the user has the greatest administrative role *gar* in an organization *o* in that session and the organization set *oset* is a subset of the *permissible administrative organization set* of *o*. For example, a user assigned (gar, @ED) can remove @PT1 from the organization hierarchy or add a new organization say @PT3 under it.

4.2.5 The ROA07 Model

The ROA07 model deals with managing role and organization association in ROBAC. Similar to other models, it provides one predicate called *can_modify_RO* to determine whether the current session can associate / disassociate a role with an organization.

Definition 4.26: *can_modify_RO*: $S \times R \times O \rightarrow$ {true, false} -- a predicate defined as *can_modify_RO*(s, r, o) is true iff ($\exists o' \geq o, \exists (ar, o') \in$ *active_role-orgs*(s)) $\wedge\ r \in$ *apermissions**(ar).

The definition of *can_modify_RO* means that a user *user(s)* in a session *s* can associate or disassociate the role *r* with the organization *o* if and only if *user(s)* has an active role and organization pair *(ar, o')* in that session and *o* is *o'* or a subordinate of *o'* and the administrative role *ar* or its junior administrative roles can perform administrative tasks on the role *r*. For example, a user assigned (PSO, @PT1) can associate PE with @PT1 but cannot disassociate PE from @PT2.

The five sub-models in AROBAC07 decentralize the administrative tasks along the administrative role hierarchy and organization hierarchy. They control administrative tasks based on both administrative role permissions and organization hierarchy. This is a ROBAC approach to perform administrative work on ROBAC systems.

AROBAC07 model can not only decentralize the administrative tasks along organization hierarchy, but also decentralize the tasks along the administrative role hierarchy. Creating appropriate administrative roles and administrative role hierarchy is critical to achieving decentralized administration in ROBAC system. The following two sections try to address the issue of how to create administrative roles and build administrative role hierarchy.

4.3 Application Compartment in ROBAC

In the real world, an authorization service may need to support many applications in an enterprise. Each application usually has its own set of roles and these roles are only applicable to the application. Therefore, the regular role set may be partitioned based on which application the roles belong to. To formalize the idea of partitioning role set in ROBAC, we introduce the concept of application compartment. We denote **App** as a set of all applications controlled by a ROBAC system.

Definition 4.27: An *Application Compartment* (ACom) for an application $app_i \in \mathbf{App}$ in a ROBAC system is a tuple: (U_i, R_i O_i, P_i, RO_i, RH_i, OH_i, PA_i, UA_i), where:

- $U_i = U$ -- the same set of users as in ROABC;

- $R_i \subseteq R$ -- a subset of roles applicable to this application;
- $O_i \subseteq O$ -- a subset of organizations applicable to this application;
- $P_i \subseteq P$ -- a subset of permissions applicable to this application;
- $RO_i \subseteq R_i$ x O_i –- a set of applicable role and organization association in this application;
- $RH_i \subseteq R_i$ x R_i -- a partial order relation on R_i called role hierarchy;
- $OH_i \subseteq O_i$ x O_i -- a partial order relation on O_i called organization hierarchy;
- $PA_i \subseteq P_i$ x R_i -- a many-to-many permission-to-role assignment relation;
- $UA_i \subseteq U_i$ x RO_i -- a many-to-many user-to-role-and-organization assignment relation;

Let us use *acom*(*app_i*) to represent the ACom corresponding to application app_i and denote ACOM as a set of all application compartments in the ROBAC system. We can think of *acom*(app_i) as being controlled by a sub ROBAC system. Here we define a dominate relation DOM in ACOM.

Definition 4.28: DOM $\subseteq$ ACOM $\times$ ACOM -- is a partial order relation on ACOM such that (*acom*(app_i), *acom*(app_j)) $\in$ DOM iff $U_i \subseteq U_j \wedge R_i \subseteq R_j \wedge O_i \subseteq O_j \wedge P_i \subseteq P_j \wedge RO_i \subseteq RO_j \wedge RH_i \subseteq RH_j \wedge OH_i \subseteq OH_j \wedge PA_i \subseteq PA_j \wedge UA_i \subseteq UA_j$.

For administration purposes, we can create an administrative role ar_i for each $acom(app_i)$. That requires adding ar_i in AR, adding $\{ ar_i \} \times R_i$ into ARRA, and adding (ar_i , ar_j) in ARH iff $(acom(app_j), acom(app_i)) \in$ DOM.

The partitioning process may continue within an application. In general, we can form a hierarchy in AR based on the DOM relationship. The senior administrative role may delegate administrative tasks to its junior administrative roles. The idea of partitioning a system to smaller systems is not new; we simply apply this useful idea in the context of AROBAC07. ACom concept not only provides a way to partition the regular role set and construct administrative role hierarchy in AROBAC07, but also can be used to enforce application boundary. For example, when a user *u* enters an application, say app_i, the ROBAC system only actives the role-org pairs applicable to this application. That is, if a user *u* is inside app_i then

$$active\_roles\text{-}orgs(u) \subseteq assigned\_role\text{-}orgs(u) \cap acom(\mathrm{app_i}).\mathrm{RO_i}$$

where $acom(\mathrm{app_i}).\mathrm{RO_i}$ represents the role-org pairs applicable to the application app_i. You may notice that AROBAC07 does not explicitly talk about how to assign / revoke users to/from administrative role and organization pairs. In practice, a user with administrative role and organization pair *(ar, o)* can assign (or revoke) users to (or from) administrative role and org pair *(ar', o')* such that $ar' \leq ar$ and $o' \leq o$. It is not hard to develop a formal sub-model similar to the UROA07 model if more complex constraints are needed. So a user assigned with *(gar, @go)* can act like a super user in a ROBAC system. The syntax of assigning user to role and organization pair is same regardless of whether a role is a regular role or an administrative role. This feature makes AROBAC07 easy to implement in practice.

4.4 Discussion and Related Work

ROBAC models require that organizations have similar roles. Administrator roles across organizations tend to have greater similarity than the underlying regular roles may have. So the ROBAC concept is particularly well suited to administrative tasks. The ROBAC/AROBAC is *totally homogeneous* on the administrative role set AR. So it is the best scenario for using ROBAC approach to control administrative tasks. A user assigned with an administrative role and organization pair *(ar, o)* can perform administrative tasks the *ar* allowed within the organization *o* and its subordinate organizations. Administrative tasks in AROBAC07 can be delegated not only along the administrative role hierarchy but also along the organization hierarchy. A preliminary version of AROBAC07 has been accepted for publishing as a book chapter [ZZS07].

As we mentioned earlier, AROBAC07 model is inspired by the following three role-based administrative models: ARBAC97, ARBAC02, and SARBAC.

ARBAC97 is one of the most comprehensive role-based administrative models. It uses *role range (encapsulated range)* concept to define the scope of administrative permission. The *user prerequisite condition (upc)* and *permission prerequisite condition (ppc)* in AROBAC07 are extended versions of the corresponding prerequisite conditions in ARBAC97.

ARBAC02 enhances ARBAC97 by incorporating two external organization structures: *user organization structure* (OS-U) and *permission organization structure* (OS-P). URA02 and PRA02 sub-models in ARBAC02 modify the prerequisite conditions in URA97 and PRA97 of ARABC97 by using memberships of organizations to avoid some weakness, such as multi-step assignments, redundant assignment information, restricted hierarchy etc., in ARBAC97. ARBAC02 uses the organization structures only for constructing user pool and permission pool in prerequisite conditions but does not use it to control administrative permissions. UROA07 and PRA07 in AROBAC07 use the built-in organization hierarchy and implicitly enforce the user pool and permission pool constraints with respect to *can_assign_user* and *can_revoke_user* and *can_assign_permission* and *can_revoke_permission* predicates. So UROA07 and PRA07 have similar benefits of the user pool and permission pool concepts used in URA02 and PRA02 models without putting the user pool or permission pool as a part of the prerequisite conditions. Similar to ARBAC02, AROBAC07 avoids the weakness of ARBAC97 by using the built-in organization hierarchy, the user-organization affiliation information, and the applicable permission and organization association information.

SARBAC introduces a concept called *administrative scope*, which can be calculated for each role based on the role hierarchy. SARBAC tends to be simpler, more flexible, and more permissible than ARBAC97. The ARRA relation in AROBAC07 is similar to the **can_admin** relation in SARBAC. The current construction of *permissible administrative role set (parset)* concept in RRA07 is similar to the *administrative scope* concept in SARBAC. From that perspective, RRA07 is very similar to RH_4 sub-model in

SARBAC. So RRA07 has some similar benefits, such as flexibility, that SARBC enjoys. Unlike RH_4, RRA07 enforces strict separation between regular roles and administrative roles because we believe the separation of administrative duty from regular duty is a desired security policy in most cases. It is possible to construct *parset* differently, such as using the *encapsulated range* concept in RRA97 or other criteria. The OOA07 applies a similar concept on the role hierarchy.

X-GTRBAC admin [BJBG04] is a decentralized administrative model for the XML-based Generalized Temporal Role Based Access Control (X-GTRBAC) framework [Bha03]. It uses a concept called *admin domain* to tie roles (admin roles and regular roles), permissions (admin permissions and regular permissions), and users (admin users and regular users) together. It uses hard-coded Eligible Role (ER) for users in admin domain to put constraint on user to role assignment task. It uses Admin Permissions to model the possible administrative tasks. An administrator in an Admin Role is authorized to handle assignment of users to regular roles within a given domain. The administrative roles and admin domains in X-GTRBAC admin are both flat. If an admin domain is considered as an organization, X-GTRBAC admin's impact on X-GTRBAC framework is similar to the UROA07 of AROBAC07's impact on $ROBAC_0$ based system. It is hard for X-GTRBAC admin to achieve functionality similar to what AROBAC07 has when organization hierarchy or/and administrative role hierarchy exist.

The first three role-based administrative models control administrative tasks based on roles only while AROBAC07 controls administrative tasks based on both roles and organizations. From the perspective of administrative units, the admin domain concept in X-GTRBAC admin is similar to the organization concept in UROA07 for $ROBAC_0$ based systems, but AROBAC07 has more functionality, which is hard to achieve in X-GTRBAC admin.

In AROBAC07, we use the same ARRA relation across all sub-models. It may be desirable that we use a different version of ARRA in each sub-model when finer-grained control is needed. That is ARRA could be different for UROA07 and PRA07. The administrative role hierarchy in AROBAC07 tends to be simpler than these existing models when there are many organizational units, such as lots of branches or project teams, in an enterprise. With ROBAC/AROBAC07, security policies can be defined in a small scope first and then applied to all organizations.

Chapter 5. ROBAC Variants

In this chapter we propose two ROBAC variants and show their usefulness in the fields of secure collaboration and e-commerce.

5.1 Manifold ROBAC

In regular ROBAC models, an asset can only be tied to one organization and can only belong to one asset type. In the real world, it is sometimes desirable to have an asset be shared by many organizations and/or belong to more than one asset type. To meet this requirement, we propose a ROBAC variant called manifold ROBAC and denoted as ROBAC^M.

Definition 5.1: ROBAC_i^M (i=0,1,2,3) has the same elements of ROBAC_i except that the definitions of *aorg* , *atype*, and *can_access* are replaced with the following:

- *aorg*: $A \rightarrow 2^O$ – a function mapping an asset to a subset of the organization set;
- *atype*: $A \rightarrow 2^{At}$ – a function mapping an asset to a subset of the asset type set;
- The *can_access*(S, Op, A) in ROBAC_0^M is slightly different from that in ROBAC_1^M.

- In ROBAC_0^M, *can_access(s, op, a)* is true iff $\exists\ (r, o) \in active\_role\text{-}orgs(s) \wedge o \in aorg(a) \wedge (\exists at \in atype(a),\ ((op, at), r) \in \text{PA})$
- In ROBAC_1^M, *can_access(s, op, a)* is true iff $\exists\ (r, o) \in active\_role\text{-}orgs(s) \wedge (\exists o' \in aorg(a),\ o' \leq o) \wedge (\exists at \in atype(a), \exists r' \leq r, ((op, at), r') \in \text{PA})$

ROBAC^M models allow an asset to be related to more than one organization and/or an asset can belong to more than one asset type while the rule to make access decision remains the same as that in ROBAC regular models. That is, a user *user(s)* in a session *s* can perform an operation *op* over an asset *a* if and only if the user has an active role and organization pair (*r, o*) in that session and the role *r* (or any of its junior roles) has permission to perform the operation *op* over at least one of asset *a*'s types, while asset *a* is related to organization *o* (or any of its subordinate organizations).

If a new security policy, "Schools in the same district can view each other's reports", is need to be added in the aforementioned B2B example, we can easily model this new requirement by using ROBAC_1^M. The *aorg(a)* in ROBAC_1^M will include all organizations within the same district that organization *o* , to which asset *a* is originally related, is in. For example, we can define *aorg*(School_1_Type_B_Report) = {School_1, School_2}. Now School 1's Type B Report Viewer can see both School 1's Type B report and School_2's Type B report.

5.2 Secure Collaboration

5.2.1 Background

As our world becomes increasingly interwoven, collaboration among different organizations becomes common. How to model and enforce security policies during collaboration is an important issue. Gong and Qian [GQ96] proposed the following two principles of secure interoperation (collaboration):

- Autonomy Principle[2] – any allowed access in an individual system must also be allowed under secure interoperation
- Security Principle – any denied access in an individual system must also be denied under secure interoperation.

Here we use the words interoperation and collaboration interchangeably. In the last several years, some research efforts have been put into secure interoperation in multi-domain environments using RBAC [KACM00, JBBG04, PJ05, SJBG05, TAPH05, LZQX06]. Most of the previous work uses a "bottom-up" approach by performing role translation or role mapping between different domains. In [JBBG04, SJBG05], Joshi et al. identify and analyze three types of violations when integrating RBAC policies: user-specific separation of duty (SoD) violation, role-specific SoD violation, and role-assignment violation. In addition to handling those problems, the role mapping based

[2] Autonomy principle is reasonable but may not be universal. For example, some collaboration between organization o1 and o2 may require a role in o1 to give up some privilege due to conflict of interests. In this case, autonomy principle may not hold. Security principle should hold for collaboration.

approaches also have to pay special attention to the “covert role promotion” problem [KACM00], which appears when a user crosses domain boundaries and returns to a local domain with a role senior to his original roles in the domain, during the collaboration setup. In [LZQX06], Li et al. use Group-based RBAC to handle the ad-hoc collaboration among different groups. Instead of using role mapping, it uses role exporting, a permission-driven collaboration schema, to eliminate role-mapping problem but it introduces some new problems such as conflicts of role names and permission names, and conflicts of permissions of roles in different groups. To solve these problems, existing RBAC based secure collaboration approaches must use rather complex algorithms for granting, updating, and revoking collaboration.

In the following section, we describe a manifold ROBAC based approach for secure collaboration. Because our approach does not use role-mapping, role-translation, or role-exporting, all aforementioned problems are avoided. Our approach is much simpler than any existing RBAC based secure collaboration approaches.

5.2.2 Secure Collaboration with Manifold ROBAC

When applying the aforementioned two secure interoperation principles in the context of ROBAC, the *autonomy principle* means that security setup for some collaboration policy among different organizations should not reduce the original permissions a user has within the user’s original organization and the *security principle* means that the collaboration setup among the different organizations should not increase the original permissions a user has in the user's original organization.

5.2.2.1 Informal Description

The main idea is to create a virtual organization and set it as a subordinate of all participating organizations for the collaboration in the organization hierarchy in ROBAC. An administrator of a participating organization sets any of its want-to-be-shared assets as also related to the virtual organization. We can see that the want-to-be-shared assets are related to both the original organization and the newly created virtual organization. This is why we need to use manifold ROBAC to model secure collaboration; so that users with the same role but in different participating organizations can share assets without violating the two secure interoperation principles.

For example, in the aforementioned engineering department example (Figure 7) Project Team 1 (@PT1) and Project Team 2 (@PT2) need to collaborate on some work. Assume that the Engineer role (ENG) has permission to access some type X assets; the assets a_{11}, a_{12}, and a_{13} are type X and belong to @PT1 while assets a_{21}, a_{22}, and a_{23} are also type X and belong to @PT2. In this initial state, a user with (ENG, @PT1) membership (an engineer in Project Team 1) can access a_{11}, a_{12}, and a_{13} but cannot access a_{21}, a_{22}, and a_{23} while a user with (ENG, @PT2) membership (an engineer in Project Team 2) can access a_{21}, a_{22}, and a_{23} but cannot access a_{11}, a_{12}, and a_{13} based on the *can_access* predicate in ROBAC. If the collaboration work requires a_{13}, a_{21}, a_{23} to be shared among @PT1 and

@PT2, how do we setup the ROBAC to control the asset access while maintaining the two secure collaboration principles? Here is how we set up the corresponding ROBAC to achieve this collaboration requirement.

- Pre-collaboration state in the ROBAC system(note: only involved elements are listed here):
    - $\{ a_{11}, a_{12}, a_{13}, a_{21}, a_{22}, a_{23} \} \subseteq$ A;
    - $aorg(a_{1i}) = \{$ @PT1 $\}$, i = 1, 2, 3;
    - $aorg(a_{2i}) = \{$ @PT2 $\}$, i = 1, 2, 3;
    - $atype(a_{ij}) = \{ X \} \subseteq$ At, i = 1, 2; j = 1, 2, 3;
    - $\forall op \in$ Op, ((op, X), ENG) $\in$ PA;

  In the above state (before the collaboration), ROBAC *can_access* predicate guarantees:
    - (ENG, @PT1) can access asset a_{11}, a_{12}, and a_{13} but not a_{21}, a_{22}, and a_{23}
    - (ENG, @PT2) can access asset a_{21}, a_{22}, and a_{23} but not a_{11}, a_{12}, and a_{13}
- Collaboration Grant:
    - Create a virtual project team @VPT12 under @PT1 and @PT2, that is, O = O $\cup$ {@VPT12}, OH = OH $\cup$ { (@PT1, @VPT12), (@PT2, @VPT12) };
    - Assign a_{13} (shared asset in @PT1) to @VPT12, that is, $aorg(a_{13}) = aorg(a_{13}) \cup$ {@VPT12} = { @PT1, @VPT12 }

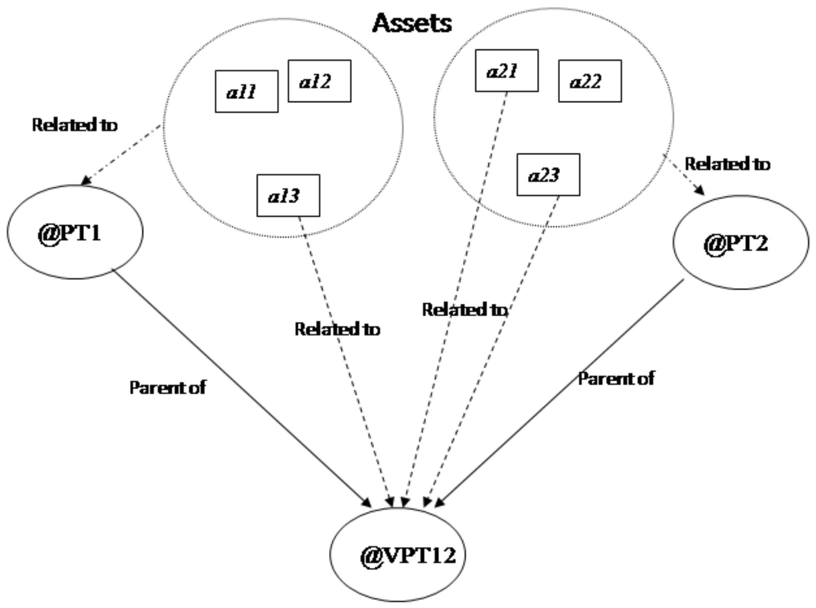
  - Assign a_{21}, a_{23} (shared assets in @PT2) to @VPT12, that is,

    $aorg(a_{21}) = aorg(a_{21}) \cup \{@VPT12\} = \{ @PT2, @VPT12 \}$,

    $aorg(a_{23}) = aorg(a_{23}) \cup \{@VPT12\} = \{ @PT2, @VPT12 \}$

- During the collaboration:
  - (ENG, @PT1) can access asset a_{11}, a_{12}, a_{13}, a_{21}, and a_{23}
  - (ENG, @PT2) can access asset a_{21}, a_{22}, a_{23}, and a_{13}
  - Assign all newly produced assets from the collaboration to @VPT12.
- Collaboration Revoke:
  - Remove anything related to @VPT12
- After collaboration:
  - System returns back to the pre-collaboration state.

Figure 8 shows the relationship among @PT1, @PT2, @VPT12, and the related assets.



Notes: @PT1 – Project Team 1; @PT2 – Project Team 2; @VPT12 – Virtual Project Team

Before collaboration:

a11, a12,a13 **are assets related to @PT1;** ***a21, a22, a23*** **are assets related to @PT2.**

During collaboration:

a11, a12,a13 **are assets related to @PT1;** ***a21, a22, a23*** **are assets related to @PT2;**
a13, a21, a23 are also related to @VPT12.

After collaboration:

a11, a12,a13 **are assets related to @PT1;** ***a21, a22, a23*** **are assets related to @PT2;**
@VPT12 and its related entries are removed from the ROBAC.

Figure 8. Virtual organization for secure collaboration in ROBAC

The ROBAC used in the above collaboration example is a manifold ROBAC. The only changes we made in the original ROBAC^M are to add a virtual organization in the organization hierarchy and allow the shared assets to also relate to the newly created virtual organization. There is no change to the other elements of ROABC^M. There is no role-mapping, role-translation, or role-exporting needed in our approach. The required shared assets a_{13}, a_{21}, and a_{23}, which are related to the virtual organization, are automatically shared by the users with (ENG, @PT1) or (ENG, @PT2) membership.

During the collaboration, the original security setting, such as permissions, constraints etc., of each participating organization is still in effect while additional permissions required by the collaboration are added. For the users who only need to access the shared assets, we can assign them a membership of (ENG, @VPT12).

Who has the authority to grant and setup collaboration? Who has the authority to setup shared resources for collaboration? Which role can associate with the virtual organization? Who should manage the newly created assets during the collaboration? These questions will be discussed and answered in the next two sections. Here we present some idea for the above collaboration example. It is quite reasonable to assume that the *greatest administrative role* (*gar*) in the senior organization of both project team 1 (@PT1) and project team 2 (@PT2) may grant a collaboration request between @PT1 and @PT2. So a member of (*gar*, @ED) has authority to grant a collaboration request between @PT1 and @PT2 by creating a virtual organization @VPT12 under @PT1 and @PT2. It seems appropriate that the update of *aorg* for the asset a_{13} is performed by *(gar, @PT1)* while the update of *aorg* for the asset a_{21} and a_{23} is performed by *(gar, @PT2)*. The roles applicable to @PT1 and/or @PT2 may be associated with the @VPT12. The newly produced assets in the collaboration may be managed by *(gar, @VPT12)*.

In regular ROBAC, functions *aorg* and *atype* are usually pre-determined during the ROBAC setup and do not change afterwards. That is why AROBAC07 does not include the management of *aorg* and *atype*. Updating *aorg* or/and *atype* usually will change the access permissions in ROBAC. So we treat the updating of *aorg* or/and *atype* as a kind of administrative action and therefore it needs to be controlled as well. In the following section, we present an administrative model to control the change of *aorg* and *atype* in ROBAC^M.

5.2.2.2 The Asset Management '07 Model

The Asset Management '07 (AM07) model addresses the asset management issue in ROBAC. AM07 determines who can modify *aorg* and *atype* for a given asset in ROBAC. It utilizes a predicate called *can_manage_A* to determine whether the current session can update the *aorg* and *atype* for a given asset.

Definition 5.2: *can_manage_A*: $S \times A \rightarrow \{\text{true, false}\}$ – a predicate defined as *can_manage_A*(s,a) is true iff $\exists o \in O$, $(gar, o) \in active\_role\text{-}orgs(s) \wedge a \in aorg(o)$.

Definition 5.3: Asset Management '07 model – if *can_manage_A*(s, a) is *true*, then *aorg*(a) and *atype*(a) can be modified in the session s.

The definition of AM07 means that a user *user(s)* in a session *s* can modify *aorg* or *atype* for a given asset *a* if and only if the *user(s)* has the active *greatest administrative role* (*gar*) in some organization *o* in the session *s* and the asset *a* is related to this organization *o*.

For example, a user with active (*gar*, @PT1) in a session can update $aorg(a_{1i})$, i = 1, 2, 3 and a user with active (*gar*, @PT2) in a session can update $aorg(a_{2j})$, j = 1, 2, 3 in the above collaboration example. You may notice that AM07 only allows a user with *(gar, o)* membership (a super user in the organization *o*) to manage the assets directly related to the organization and that it does not allow the super user to manage assets related to *o*'s subordinate organizations. For example, (*gar*, @PT1) cannot update $aorg(a_{21})$ although a_{21} is related to @VPT12 which is a subordinate of @PT1 during the collaboration. In this way, the original entry @PT2 inside $aorg(a_{21})$ will not be accidentally removed by a user with (*gar*, @PT1).

5.2.2.3 ROBAC^{M} Based Secure Collaboration Schema

Secure collaboration among a set of organizations within ROBAC means that some specified resources are shared among the participating organizations. Users with roles in the individual participating organization can also perform the same roles against the shared resources during the collaboration without any violation to the *autonomy principle* and the *security principle*. To simplify the description of ROBAC^{M} based secure collaboration, we first introduce some definitions.

Definition 5.4: *oaset*: $O \rightarrow 2^{A}$ -- a function mapping an organization to a set of assets related to the organization; formally: $oaset(o) = \{ a \mid o \in aorg(a) \}$.

Definition 5.5: *oaset_col*: $O \rightarrow 2^{A}$ -- a function mapping an organization *o* to a subset of *oaset*(*o*) which the organization authorizes to share during a collaboration. That is, $oaset\_col(o) \subseteq oaset(o)$.

Definition 5.6: A *secure collaboration request*, SecureColReq, in a ROBAC based system is a triple SecureColReq = (ROBAC, OColSet, AColSet) where ROBAC = (U, S, user, R, RH, O, OH, Op, At, A, RO, P, PA, UA, *atype, aorg*), OColSet $\subseteq$ O, a subset of O, wish to perform the collaboration; AColSet = { *oaset_col*(*o*) | *o* $\in$ OColSet}.

A *secure collaboration request* SecureColReq = (ROBAC, OColSet, AcolSet) may be generated by the joint effort among the organizations in OColSet and/or their parent organizations.

Definition 5.7: *ocommonseniors*: $2^O \rightarrow 2^O$, is a function mapping a set of organization to its common parent organizations. Formally, *ocommonseniors*(*oset*) = { *o'*: O | $\forall o \in$ *oset*, $o \leq o'$ }.

Note: For any oset $\in 2^O$, *ocommonseniors*(*oset*) is not empty because the *greatest organization* (*go*) is always inside *ocommonseniors*(*oset*).

Schema for secure collaboration in ROBAC:

- **Input**: given a collaboration request SecureColReq$_k$ = {ROBAC, OColSet$_k$, AColSet$_k$ } where
  OColSet$_k$ = $\{o_i \mid i = 1, 2, \ldots, m\}$, AColSet$_k$ = { *oaset_col*(o_i) | i = 1, 2, …, m}.
- **Collaboration grant**:
  - *Initialization*: a user with *(gar, o_k)* membership, where $o_k \in$ *ocommonseniors*(OColSet$_k$), can create a virtual organization vo_k for this collaboration request and put vo_k under all organizations in OColSet$_k$. Formally, O = O $\cup$ { vo_k }; $\forall o_i \in$ OColSet$_k$, OH = OH $\cup$ { (o_i, vo_k) }

- *Want-be-shared resources setup*: based on AM07, a user with *(gar, o_i)* membership can assign *oaset_col(o_i)* to vo_k. Formally,

  $\forall a_j \in oaset\_col(o_i), aorg(a_j) = aorg(a_j) \cup \{ vo_k \}.$

  Repeat this step for each o_i in $OColSet_k$

- *Additional privilege for collaboration*: in addition to all administrative roles, all roles in *applicable_R*(o_i), i =1, 2, …, m, can be associated with vo_k if needed. For example, a users with *(gar, vo_k)* membership can act as a super user in vo_k.

- **During the collaboration (collaboration update)**:
  - The users with existing role and organization pair *(r, o_i)* membership can continue doing their jobs in the o_i in addition to doing the same jobs against the shared assets in $AColSet_k$ via the virtual organization vo_k.
  - All newly produced assets from the collaboration will be related to the vo_k. The users with *(gar, vo_k)* membership can manage any assets produced during the collaborations based on AM07.
  - UROA07 can be used to assign users to *(r, vo_k)* if some users need to do work requiring role *r*'s privilege for the collaboration but he/she does not have the *r*'s privilege in any participating organizations.

- **Collaboration revoke**:
  - A user with *(gar, o_k)* membership, where $o_k \in$ *ocommonseniors*($OColSet_k$), can revoke the collaboration. Basically, it removes anything related to vo_k. Formally,

```
For each $o_i$ in $OColSet_k$
   Begin
      OH = OH - { $(o_i, vo_k)$ }
      For each $a_j$ in $oaset\_col(o_i)$
         Begin
            $aorg(a_j) = aorg(a_j)$ - { $vo_k$ }
         End
   End
O = O - { $vo_k$ }  //destroy the virtual organization.
```

The collaboration grant process requires several steps. Given a *secure collaboration request*, only a super user in some common parent organization for all participating organizations in the collaboration has the privilege to start the collaboration grant process in the *initialization* step and only a super user in each participation organization has the privilege to setup its own shared resources for the collaboration in the w*ant-be-shared resources setup* step. During the collaboration, the users' privileges in participating organizations do not change while enjoying the new privileges required by the collaboration via the newly created virtual organization. The collaboration revocation is also quite simple: just remove the virtual organization from O, OH, and *aorg*. As you can see, the ROBAC^M based secure collaboration schema is very simple and clean. It achieves the secure collaboration request without violating the two secure collaboration principles. There is no user-specific SoD violation, role-specific SoD violation, role assignment violation [SJBG05], covert role promotion [KACM00], conflicts of role names and permission names, and conflicts of permissions of roles in different groups [LZQX06] in the ROBAC^M based secure collaboration schema because it does not need to perform any role-mapping, role-translation, or role-exporting.

5.2.3 Related Work

The most important task in secure collaboration across multi-domains is to combine a consistent and conflict-free interoperation policy that governs all the inter-domain information and resource exchange. There are some research efforts on RBAC based secure collaboration in multi-domain environments in recent years [KACM00, JBBG04, SJBG05, PJ05, LZQX06]. A dynamic role translation model is proposed in [KACM00] to address the secure collaboration problem in multi-domain environment. It has several security risks, such as covert role promotion problem, during role translation. Joshi et al [JBBG04, SJBG05, PJ05] propose several approaches for multi-domain policy specification, integration, and interoperation based on an extended RBAC model: XML based RBAC and Generalized Temporal Role Based Access Control Model (GTRBAC). Integration and interoperation are achieved via role mapping between different domains. These role-mapping based approaches suffer the following three types of security violations when integrating RBAC policies due to the use of a “bottom-up” approach:

- user-specific separation of duty (SoD) violation
- role-specific SoD violation

- role-assignment violation

Most of these works pay special heed to the detection and resolution of the above mentioned violations in interoperation policies, thereby having to use rather complex algorithms in the process. In [LZQX06], Li et al propose a permission-driven collaboration schema which utilizes the concept of virtual group in GB-RBAC model. In Li's approach, the roles in the participating groups are exported to the virtual group instead of being mapped among them. Although it eliminates some aforementioned violation problems, it suffers the following conflicts: conflicts of user names, role names and permission names; conflicts of permissions of roles in different groups. To detect and resolve these conflicts, the GB-RBAC based collaboration schema has to also utilize somewhat complex algorithms for collaboration granting, updating, and revocation. In [CCC08], Cuppens et al. propose an OrBAC based approach, called O2O (Organization to Organization), to manage security policy interoperability. O2O uses Virtual Private Organization (VPO) and Role Single-Sign On (RSSO) concepts to enable any organization undertaking an inter-operation with other organizations to keep control over the resources accessed during the inter-operation and keep the same role when accessing to another organization but with privileges defined in VPO. It requires one VPO for each organization. While allowing the use of the same role name in organizations and their corresponding VPOs, the privilege of the role is redefined in each VPO.

In our ROBAC^M based collaboration schema, we treat our organizations as domains. Our approach does not need to use role-mapping, role translation, or role-exporting for secure collaboration due to the unique definitions of permissions, *can_access* predicate, and the organization hierarchy in ROBAC. Similar to GB-RBAC based collaboration schema, our schema avoids the violations resulted from role-mapping due to its “top-down” approach. Unlike the GB-RBAC method, our method also eliminates the conflicts resulting from role-exporting due to the organization hierarchy feature in ROBAC. All of this makes our collaboration granting, updating, revoking algorithms significantly simpler than those existing RBAC based collaboration methods. The redefinition of role for each VPO in O2O is similar to that of role mapping. In ROBAC^M based collaboration schema, we only need one virtual organization for a collaboration request while O2O need number of VPOs same as the number of organizations participating the collaboration. The collaboration setup in ROBAC^M based schema is much simpler than the OrBAC based collaboration schema.

5.3 Pseudo ROBAC

The organization in ROBAC can be used as a mechanism for establishing some kind of relationship between users and assets indirectly. For example, we can create artificial organizations and define *aorg* as a function of asset’s attributes and assign user to role-organization pairs based on the user’s attributes. Because the organizations are not real organizations, we call them pseudo organizations. The corresponding ROBAC models are called pseudo ROBAC, denoted as ROBAC^P. In ROBAC^P, the process of calculating *aorg* for a given asset and the process of assigning user to role-organization pairs may be automated using some rule based approaches while *can_access* predicate does not change. Implicitly calculating related organizations for assets and assigning users to role-organization pairs are useful because ROBAC usually involves a large number of organizations and external users. It is costly and error-prone to assign user to role-organization pairs manually. Further more, some external user's identifier may be unknown to a ROBAC system in advance if the ROBAC controlled application supports self registration.

5.3.1 Attributes in ROBAC^P

Utilizing an entity's attributes for access control purpose is a well-known technique in attribute based access control (ABAC) [WJ03, WWJ04], rule based RBAC (RB-RBAC) [Alk04], and usage control (UCON) [Par03, Zha06]. We assume that users, sessions, and assets in ROBAC^P have attributes associated with them. In ROBAC^P, we use the attribute definition in [Zha06]. That is, an attribute is modeled as a variable of a specific data type including a set of possible values (domain) and operators to manipulate them. We denote an attribute of an entity *e* as **entity**.*attr(e)* where **entity** can be **User**, **Session**, or **Asset**, $e \in U \cup S \cup A$, and *attr* is the attribute name. And we use *e.attr* as an abbreviation of **entity**.*attr(e)*. For example, a user Bob's age is represented as **User**.*age*(*Bob*) or *Bob.age*; how long a session *mySession* has lasted is represented as *mySession.lasted_time*; an asset *a*'s owner is represented as *a.owner*.

Definition 5.8: An *attribute predicate* in ROBAC^P is a Boolean expression on attributes in the form of

entity.$attr_1$ op_1 $attr_1$_constant $\wedge$ (or $\vee$) **entity**.$attr_2$ op_2 $attr_2$_constant $\wedge$ (or $\vee$) … $\wedge$ (or $\vee$) **entity**.$attr_n$ op_n $attr_n$_constant.

Here entity can be **User**, **Session**, or **Asset**, *$attr_i$_constant* is a value from attribute *$attr_i$*'s domain and *op_i* is a operator for attribute *$attr_i$* , i = 1, 2, …, n. For a given entity *e*, the *attribute predicate* is evaluated as *e*.$attr_1$ op_1 $attr_1$_constant $\wedge$ (or $\vee$) *e*.$attr_2$ op_2 $attr_2$_constant $\wedge$ (or $\vee$) … $\wedge$ (or $\vee$) *e*.$attr_n$ op_n $attr_n$_constant.

Here are some examples of *attribute predicates*:

- **User**.*age* $\geq$ 21
- **User**.*team* = 'Project_1' $\wedge$ **User** .*division* = 'Engineering Department'
- **Session**.*day* = 'Saturday' $\vee$ **Session**.*day* = 'Sunday'
- **Asset**.*price* > $100
- *null*

The first predicate is evaluated as true for any user whose age is equal or more than 21 years old. The second predicate is evaluated as true for any user who is in Project_1 team and in Engineering Department. The third predicate is evaluated as true for a session during the weekend (Saturday or Sunday). The fourth predicate is evaluated as true for an asset if the asset's price is greater than 100 dollars. The fifth predicate, *null*, is a special one and is evaluated as true for any entity's attribute.

Definition 5.9: A *cross-attribute predicate* in ROBAC^P is a Boolean expression from attributes, functions, and constants where attributes include user attributes, session attributes, and asset attributes.

Here are some examples of *cross-attribute predicates*:

- **User**.*credit* ≥ **Asset**.*price* + \$100
- **User**.*status* = *'gold'* ∧ **Session**.*duration* < 10

As we can see, the attribute predicates are a subset of the cross-attribute predicates.

What kinds of attributes to use is application specific. How an entity gets its attribute values and how to manage attributes are two very important issues need to be addressed. There are some efforts have been put in this area [HMMNR00, ZBM01, Ker02, KSM03, Alk04]. Usually, external users (strangers) may gain attributes via certificates or attribute authority (AA), or based on the users' activity inside applications. Internal users may gain attributes via an enterprise wide LDAP (Lightweight Directory Access Protocol) system [LDAP97, LDAP01] and/or via a human resources database. There is some nice treatment of attribute management in UCON [Par03, Zha06]. In UCON, attributes are categorized as "administrator-controlled" or "system-controlled" attributes. Administrator-controlled attributes can be updated only by explicit administrative actions and the system does not modify them automatically (immutability). For example a user's date of birth is an administrator-controlled attribute and cannot be modified without explicit administrative action. For administrator-controlled attributes, we may apply type-centric approach, organization-centric [KSM03], location-centric, or role-centric approaches to administrate them. For instance, we may use some model similar to ARB-

RBAC X model in RB-RBAC [Alk04] to manage attributes in pseudo ROBAC. System-controlled attributes can be updated by the system automatically as the side effects or results of the user's usage (mutability) and they do not require any administrative action for updates. For example, a user's credit balance is decreased by the value the user spent at the time of the usage. The approaches used in UCON to update system-controlled attributes may be applied in ROBAC^P. Some detail discussion of attribute management can be found in [Alk04].

5.3.2 Formal Definition of ROBAC^P

ROBAC^P has two important features which differentiate it from regular ROBAC. One is the artificial organization concept. Another is the rule based approach for calculating *aorg* of assets and for assigning / activating user to role-organization pairs implicitly.

With the understanding on the definition of *attribute predicate*, we can give a formal definition of ROBAC^P.

Definition 5.10: ROBAC^P has the same elements of ROBAC except for the following:

- AAP -- a set of applicable asset attribute predicates;
- UAP -- a set of applicable user attribute predicates;
- SAP -- a set of applicable session attribute predicates;
- UROA_Rule $\subseteq$ UAP $\times$ SAP $\times$ RO -- a set of rules used for activating role-organization pair in sessions;
- AOA_Rule $\subseteq$ AAP $\times$ O -- a set of rules used for asset to organization assignment;

- $aorg(a) = \{ o \mid \exists (aap, o) \in$ AOA_Rule, aap is *true* for $a\}$
- $active\_role\text{-}orgs(s) = \{ (r, o) \mid \exists (uap, sap, (r, o)) \in$ UROA_Rule, uap is *true* for $user(s) \wedge sap$ is *true for s* $\}$

The AAP includes applicable attribute predicates related to assets. An asset must satisfy at least one of these asset attribute predicates to be considered as related to an organization. The UAP includes applicable attribute predicates related to users and SAP includes applicable attribute predicates related to sessions. A user in a session must satisfy at least one of these user/session attribute predicates to be considered as a member of some role-organization pair. UROA_Rule is a *role-organization pair activation rule set* and is used in the definition of *active_role-orgs* function in ROBAC^P. AOA_Rule is an *asset to organization assignment rule set* and is used in the definition of *aorg* function in ROBAC^P.

Instead defining *aorg* function for all assets explicitly in regular ROBAC models, ROBAC^P uses the rules in AOA_Rule to determine the value of *aorg*(*a*) for a given asset *a* dynamically. In regular ROBAC, we require that *active_role-orgs(s)* $\subseteq$ *assigned_role-orgs(user(s))* and *assigned_role-orgs* is derived from UA (user to role-organization assignment relation). In ROBAC^P, we calculate that *active_role-orgs(s)* dynamically and do not explicitly assign users to role-org pairs. How the UA looks like in ROBAC^P is an interesting question. There are several options here. The first option is to use the same rules in UROA_Rule to populate UA implicitly when a session starts. So the content in *assigned_role-orgs(user(s))* will be same as the content in *active_role-orgs(s)*. Because the attribute values may change even for a same user in different sessions, the content in UA and *active_role-orgs(s)* may not be same in different sessions. We need to remove the entries, which are populated during a session startup, in UA after the session ends. This option keeps consistence between ROBAC^P and regular ROBAC. The second option is to remove the constraint *active_role-orgs(s)* $\subseteq$ *assigned_role-orgs(user(s))* in ROBAC^P. In this option, the UA is ignored. The consequence of ignoring UA is that we cannot use the constraints defined in ROBAC^2 to enforce UA constraints in ROBAC^P. We believe that the first option is a better choice from theoretical point of view. The second option may be useful in real world applications when UA constraints are not required.

For some pseudo ROBAC, the UROA_Rule may be decomposed to two separate rule sets: one for mapping users to roles and another for mapping user to organizations. We denote this kind of simplified pseudo ROBAC as ROBAC^{Ps}. A formal definition of ROBAC^{Ps} is given in the following definition.

Definition 5.11: A simplified pseudo ROBAC (ROBAC^{Ps}) has the same elements of ROBAC except for the following:

- AAP, UAP, SAP, AOA_Rule, *aorg* are same as those defined in Definition 5.10;
- URA_Rule $\subseteq$ UAP $\times$ SAP $\times$ R -- a set of rules used for activating roles in sessions;
- UOA_Rule $\subseteq$ UAP $\times$ SAP $\times$ O -- a set of rules used for activating organizations in sessions;

- *active_role-orgs(s)* = $\{ (r, o) \mid (r, o) \in$ RO $\wedge$ $\exists(uap, sap, r) \in$ URA_Rule, $\exists(uap', sap', o) \in$ UOA_Rule, $uap \wedge uap'$ are *true* for *user(s)* and $sap \wedge sap'$ is *true for s* $\}$

Due to the decoupling between roles and organizations in the rule sets, the size of the rule sets in ROBAC^{Ps} is much smaller than those in regular ROBAC^{P}.[3]

In ROBAC^{P}, the rules in the rule sets are quite straight forward because it does not support cross-over attribute predicates in rules. For example, **User**.attr$_i$ > **Asset**.attr$_j$ is not allowed to appear in the rules. This limitation reduces the expressive power of regular ROBAC^{P}. To overcome this limitation, we can re-define the rule sets in ROBAC^{P} by allowing *cross-attribute predicates* in rules. The resulted ROBAC is called pseudo ROBAC with cross-attributes predicate, denoted as ROBAC^{Pcap}. Its simplified version is denoted as ROBAC^{Pscap}.

Definition 5.12: ROBAC^{Pcap} has the same elements of ROBAC except for the following:

- AAP, AOA_Rule, and *aorg* are same as those in regular ROBAC^{P};
- USAAP – a set of applicable *cross-attribute predicates*;
- UROA_Rule $\subseteq$ USAAP $\times$ RO -- a set of rules used for activating role-organization pair in sessions;

[3] From now on, we will use ROBAC^{P} to refer any pseudo ROBAC and use regular ROBAC^{P} to refer the pseudo ROBAC defined in Definition 5.10.

- *active_role-orgs(s)* = $\{ (r, o) \mid \exists(usaap,\ (r, o)) \in \text{UROA\_Rule},\ \exists a \in \text{A},\ \ o \in aorg(a)\ \wedge usaap$ is *true* for the user *user(s),* session *s,* and the asset a }

Definition 5.13: ROBAC$^{\text{Pscap}}$ has the same elements of ROBAC except for the following:

- AAP, AOA_Rule, and *aorg* are same as those in regular ROBAC$^{\text{P}}$;
- USAAP – a set of applicable *cross-attribute predicates*;
- URA_Rule $\subseteq$ USAAP $\times$ R -- a set of rules used for activating roles in sessions;
- UOA_Rule $\subseteq$ USAAP $\times$ O -- a set of rules used for activating organizations in sessions;
- *active_role-orgs(s)* = $\{ (r, o) \mid (r, o) \in \text{RO} \wedge \exists(usaap,\ r) \in \text{URA\_Rule},\ \exists(usaap', o) \in \text{UOA\_Rule},\ \exists a \in \text{A},\ o \in aorg(a)\ \wedge\ usaap \wedge usaap'$ are *true* for *user(s),* session *s,* and the asset a }

There are two major differences between pseudo ROBAC and previous ROBAC models:

- The created artificial organizations are only used for the purpose of connecting users with assets.
- The user to role-organization pair assignment for a user in a session *s* and the asset to organization assignment for an asset *a* are refreshed automatically

based on the attributes of the user *user(s)*, the current session *s*, and the asset *a* each time when the session starts.

Pseudo ROBAC may be used in conjunction to regular ROBAC or manifold ROBAC.

5.3.3 Administration of ROBAC^P

The AAP, UAP, SAP in ROBAC^P are usually pre-determined for a given application. Any changes of UROA_Rule (or URA_Rule and UOA_Rule) may affect the user to role-org pair assignment perspective of ROBAC. The change of AOA_Rule may affect user's permissions. Therefore, any changes of UROA_Rule (or URA_Rule or UOA_Rule) and AOA_Rule needs to be controlled.

Definition 5.14: *can_manage_UROA_Rule*: S × UROA_Rule → {true, false} – a predicate controls whether a user can manage a given rule in UROA_Rule within a session. Formally, *can_manage_UROA_Rule*(*s, (uap, sap, (r, o))*) is true iff ($\exists o' \geq o$, $\exists(ar, o') \in$ *active_role-orgs*(s)) $\wedge$ ($r \in$ *apermissions**(ar)).

Note: The definition of *can_manage_UROA_Rule*(*s, (uap, sap, (r, o))*) indicates that a user (*user(s)*) in a session *s* can manage rule *(uap, sap, (r, o))* if and only if *user(s)* has an active role-org pair *(ar, o')* in that session, and the administrative role *ar* or its junior administrative roles have administrative right over the regular role *r,* and the organization *o* is same as the organization *o'* or one of its subordinate organizations.

Definition 5.15: *can_manage_AOA_Rule*: S $\times$ AOA_Rule $\rightarrow$ {true, false} -- a predicate defined as *can_manage_AOA_Rule*(*s, (aap, o)*) is true iff *(gar, o)* $\in$ *active_role-orgs*(*s*).

Note: The definition of *can_manage_AOA_Rule*(*s, (aap, o)*) indicates that only a super user in the organization *o* can manage the asset assignment rule *(aap, o)*.

Similarly, we can define the predicates for the rule sets in simplified pseudo ROBAC.

Definition 5.16: *can_manage_URA_Rule*: S $\times$ URA_Rule $\rightarrow$ {true, false} – a predicate controls whether a user can manage a given rule in URA_Rule within a session. Formally, *can_manage_URA_Rule*(*s, (uap, sap, r)*) is true iff ($\exists$*ar* $\in$ AR, *(ar, go)* $\in$ *active_role-orgs*(*s*) $\wedge$ (*r* $\in$ *apermissions**(*ar*)).

Note: The definition of *can_manage_URA_Rule*(*s, (uap, sap, r)*) means that only a user with administrator role *ar* membership in the highest organization level (*go*) and *ar* or its junior administrative roles have administrative right over the regular role *r* can manage the rule in URA_Rule because these rules affect all organizations in O.

Definition 5.17: *can_manage_UOA_Rule*: S $\times$ UOA_Rule $\rightarrow$ {true, false} – a predicate controls whether a user can manage a given rule in UOA_Rule within a session. Formally, *can_manage_UOA_Rule*(*s, (uap, sap, o)*) is true iff $\exists$*o'* $\geq$ *o*, *(gar, o')* $\in$ *active_role-orgs*(*s*).

Note: It makes sense that only a super user in the organization *o* or in its senior organization can manage the rule in UOA_Rule because these rules affect all roles in R.

Following the similar line, the administrative models for $\text{ROBAC}^{\text{Pcap}}$ and $\text{ROBAC}^{\text{Pscap}}$ can be developed easily.

The newly introduced *role-organization pair activation rule set* UROA_Rule (or its simplified version) and *asset to organization assignment rule set* AOA_Rule in the ROBAC^{P} provide great flexibility for modeling some security policy. We will see its usefulness in the next section.

5.4 A ROBAC^{P} Case Study

Let us consider some access control policies for a business that let users view or download movies on the web. It is common practice in the movie industry to control the content, date, and price of a release according to region. For example, DVD-Video discs are usually encoded with a region code intended to restrict the area of the world in which they can be played. This kind of "regional lockout" is achieved with the region coded DVD discs and region restricted DVD-players. The movies distributed in USA are rated based on the Motion Picture Association of America's (MPAA) film rating system. A movie may be rated as:

- G - General Audiences (all ages admitted)
- PG - Parental guidance suggested (some material may not be suitable for young children)

- PG-13 - Parents strongly cautioned (some material may be inappropriate for children under 13)
- R - Restricted (under 17 requires accompanying parent or adult guardian)
- NC-17 - No Children 17 and Under Admitted

A movie rated NC-17 should not be viewed by children under 18. The business wants to achieve similar "regional lockout' for the movies and prevent under-age users from viewing restricted materials on its website. The informal description of some security policies on this website are listed below:

- Movies released for a region can be viewed or downloaded by the users in that region and should not be viewed by users in other regions.
- Children under 13 can only view/download the movies rated G or PG.
- Children between 13 – 17 (including 13 and 17) can only view/download movies rated G, PG, or, PG-13.
- Persons age 18 or older can view/download any movies.

The goal here is to demonstrate the usefulness of ROBAC^P. So the security policies listed above are not exactly same as those used in real practice.

For the above security policies, we can use pseudo ROBAC to model them easily. We assume that a user has *age* attribute and a session has *residence* attribute which indicates where the session is originated. The asset set A includes all movies available on the website. Each movie (asset) has *code* attribute which represents the intended distribution area. The possible values for the code attribute is { Region_1, Region_2, Region_3 } assuming there are three regions on the world. The asset type set At = { G, PG, PG-13, R, NC-17 } where G means G rated movies, PG means PG rated movies, and so on. The operation set Op = {*access*}. Here 'access' means view or download. The permissions are shown in the Table 12 and the roles are shown in the Table 13.

Table 12. Permissions in the ROBAC^P

p1 = (*access*, G): access G rated movie
p2 = (*access*, PG): access PG rated movie
p3 = (*access*, PG-13): access PG-13 rated movie
p4 = (*access*, R): access R rated movie
p5 = (*access*, NC-17): access NC-17 rated movie

Table 13. Roles in the ROBAC^P

r1: Kid which has permission p1, p2.
r2: Teenage which has permission p1, p2, p3.
r3: Adult which has permission p1, p2, p3, p4, p5.

We define the organization set O = { Region_1, Region_2, Region_3}. Because all roles are applicable to all organizations, the resulted ROBAC is *totally homogeneous*. There is no restriction on RO. Further more, we can use a simplified pseudo ROBAC (ROBAC^{Ps}) to model this problem.

The set of applicable asset attribute predicates,

AAP = { **Asset**.*code* = Region_1, **Asset**.*code* = Region_2,

Asset.*code* = Region_3 }.

The set of applicable user attribute predicates,

UAP = { **User**.*age* $<$ 13, **User**.*age* $\geq$ 13 $\wedge$ **User**.*age* $\leq$ 17, **User**.*age* $\geq$ 18 }.

The set of applicable session attribute predicates,

SAP = { **Session**.*residence* in Region_1, **Session**.*residence* in Region_2,

Session.*residence* in Region_3 }.

The *role activation rule set,*

URA_Rule = { (**User**.*age* $<$ 13, *null*, *Kid*),

(**User**.*age* $\geq$ 13 $\wedge$ **User**.*age* $\leq$ 17, *null*, *Teenage*),

(**User**.*age* $\geq$ 18, *null*, *Adult*) }

The *organization activation rule set,*

UOA_Rule = { (*null*, **Session**.*residence* in Region_1, *Region_1*),

(*null*, **Session**.*residence* in Region_2, *Region_2*),

(*null*, **Session**.*residence* in Region_3, *Region_3*) }

The *asset to organization assignment rule set,*

AOA_Rule = { (**Asset**.*code* = Region_1, *Region_1*),

(**Asset**.*code* = Region_2, *Region_2*),

(**Asset**.*code* = Region_3, *Region_3*) }

If a 15 year old user from Canada signed into this website, the user will gain the membership of *(Teenage, Region_1)* based on the rules in the URA_Rule and UOA_Rule because Canada is in Region_1. If he/she requests to view some rated NC-17 movie, the request will be denied due to the *Teenage* role's lack of permission to view NC-17 rated movies. If he/she requests to download a PG rated movie intended for Region_2, the request will also be denied because he/she only has the *Teenage* role in Region_1 not in Region_2. However, his/her request to view a PG rated movie intended for Region_1 will be granted.

5.5 Discussion and Related Work

There are some efforts [HMMNR00, ZBM01, YMB01, LMW02, Alk04] to address the issue of how to assign users to roles in role based systems automatically. Herzberg et al. [HMMNR00] propose an approach to assign a user to roles based on the user's credentials, such as public key. Zhong et al. [ZBM01] propose a scheme for user-role assignment based on the "trustworthiness" of the users. Yao et al. [YMB01] use role activation rule and role membership rule to automate user role activation in Open Architecture for Secure Interworking Services (OASIS). In Li et al.'s RT framework [LMW02], each role specifies the roles that it contains and/or attributes that are required for membership. Al-Kahtani [Alk04] proposes a family of rule-based RBAC (RB-RBAC) models to allow the specification of automatic (implicit) user-role assignment. These previous works focus on how to automate user to role assignments in RBAC. Our work emphasizes how to use rule-based approach for linking users and assets via pseudo-organizations in the context of ROBAC. If a problem can be modeled by a simplified ROBAC^P, the number of roles and the number of rules in the ROBAC^P with respect to the problem are much smaller than those in RB-RBAC. Some detail treatment of rules, such as seniority relation among rules, induced role hierarchies etc., in RB-RBAC can be applied in ROBAC^P along the similar line.

Enforcing access control based on involved entities' attributes has been used extensively in attribute based access control (ABAC) [WJ03, WWJ04] and usage control (UCON) [Par03, Zha06]. In ABAC and UCON, the access control policy to a collection of services is specified only based on a collection of attributes possessed by the involved entities. Normally, ABAC and UCON treat role membership as one of the requester's attributes when some kind of role concept is involved. Due to the wide variety of attributes, the administration in ABAC and UCON is usually application specific, which often results in ad-hoc administrative models. The User and Asset attributes in ROBAC^P are similar to the Subject and Object attributes in UCON. The System attributes in UCON represent system wide attributes and are independent from Subject attributes and Object attributes. The Session attributes in ROBAC^P represent session specific context information such as current time, session duration etc. The System attributes can be simulated by Session attributes. UCON has two distinguishing features: continuity of access decision and the mutability of attributes. ROBAC^P does not address the continuity of access decision and the mutability of attributes. In ROBAC^P, we assume all attributes are pre-defined. Some schema of updating attribute values in UCON may be incorporated into ROBAC^P if needed. The attributes predicates in regular ROBAC^P are stricter than those in UCON. The best case scenario in using pseudo ROBAC is where the problem can be modeled by the simplified version of ROBAC^P and the resulted ROBAC is *totally homogeneous*. In this best case scenario, the number of rules in ROBAC^{P} is reduced significantly. While ROBAC^{Pcap} provides great flexibility and expressive power, its manageability is usually reduced. It is easy to show that any preA$_0$ model (an access control decision is made before the access and there is no attribute update before, during, or after the access) in UCON can be simulated by a ROBAC^{Pcap} model.

Chapter 6. Conclusions and future work

A family of extended RBAC models called Role and Organization Based Access Control (ROBAC) models and its corresponding administrative model called AROBAC07 are presented and formalized in this dissertation. The motivation behind ROBAC is to scale up RBAC for B2B and B2C applications where a large number of organizational units are involved. The advantages of ROBAC models over traditional RBAC models are shown via two examples. A comparison between ROBAC and RBAC has been given. We show that the benefits of using ROBAC depend on the *homogeneous index* (*hindex*) function for a given ROBAC. The larger the *hindex* value for a ROBAC model, the better off the ROBAC model. We show that the expressive power of ROBAC is the same as that of RBAC but ROBAC is more intuitive and succinct than many RBAC variants when used for situations involving a large number of similar organizational units and the privacy issue is a major concern.

Many serious security breaches are due to internal users. Hence, it is equally important to restrict and control administrative actions on access control systems. AROABC07 is developed for this purpose. AROBAC07 is a decentralized role and organization based administrative model for ROBAC. It has five sub-models:

- UROA07 is concerned with user to role and organization pair assignment;
- PRA07 deals with permission-role assignment;

- RRA07 manages roles and role hierarchy;
- OOA07 handles organizations and organization hierarchy;
- ROA07 controls applicable association between roles and organizations.

UROA07 and PRA07 retain the benefits of user pool and permission pool concepts in ARBAC02 without specifying user pool and permission pool explicitly. RRA07 has the similar advantage of RHA_4 in SARBAC over RRA97 in ARBAC97 without sacrificing the separation of duty between administrative roles and regular roles. OOA07 and ROA07 provide ways to decentralize the administrative tasks on organization hierarchy and applicable role and organization association. We claim that AROBAC07 scales up well and is better than existing role based administrative models by providing more controlled and decentralized approaches to perform administrative tasks on ROBAC.

ROBAC / AROBAC07 scales up classic RBAC systems for situations where many similar organizational units are involved. It inherits RBAC's beneficial features and provides a way to restrict access control within specified organizational units without introducing too much administrative burden on access control systems. It is quite suitable for modeling privacy related security policy.

A concept called application compartment (ACom) is introduced in the context of ROBAC / AROBAC07. We show how to use ACom when constructing administrative role hierarchy and activating role-organization pairs.

Finally, two ROBAC variants, manifold ROBAC ($ROBAC^M$) and pseudo ROBAC ($ROBAC^P$), are introduced. The usefulness of manifold ROBAC is demonstrated in secure collaboration. ROBAC models are well positioned to handle secure collaboration problems due to its built-in organization hierarchy. We show that using manifold ROBAC in secure collaboration is simpler and cleaner than most existing RBAC based approaches for cross-domain collaboration. The usefulness of pseudo ROBAC is shown in a case study for a web based on-demand movie service. We show that using pseudo ROBAC usually results in less number of rules than Rule Based RABC for applications involving many similar organizations.

This dissertation lays the groundwork for ROBAC models. Many research opportunities can be investigated further. Here is a list of future research topics related to ROBAC models:

- Detail the implication of *can_modify_R*, *can_modify_O*, and *can_modify_RO* predicates on individual administrative tasks such as add/delete nodes or edges;
- Define each administrative action using some formal specification language such as Z [PST91];
- Integrate general constraints in ROBAC;
- Present a reference implementation of ROBAC;
- Detail the implementation perspective of ROBAC^M based secure collaboration schema;
- How to enforce UA constraints in ROBAC^P?

- How attributes get their values in ROBAC^P? When and where to update system-controlled attributes in ROBAC^P? How administrator-control attributes are managed?
- Integrate regular ROBAC and pseudo ROBAC in same problem;
- Perform safety analysis of ROBAC, which is equivalent to the safety analysis of RBAC since any given ROBAC can be simulated by a RBAC though the resulted RBAC is more complex.

Bibliography

Bibliography

[Alk04] Mohammad Abdullah Al-Kahtani, "A Family of Models for Rule-Based User-Role Assignment", PhD Dissertation, George Mason University, Spring 2004.

[ANSI04] American National Standard Institute, “ANSI INCITS 359-2004 for Role Based Access Control”, 2004

[BBF01] E. Bertino, P. A. Bonatti, E. Ferrari, "TRBAC: A temporal role-based access control model", ACM Transactions on Information & System Security, 4(3), Aug.2001, p.191-233.

[BCDP05] Elisa Bertino, Barbara Catania, Maria Luisa Damiani, Paolo Perlasca, "Access control model I: GEO-RBAC: a spatially aware RBAC", Proceedings of the tenth ACM symposium on Access control models and technologies, June 2005.

[BJBG04] Rafae Bhatti, James Joshi, Elisa Bertino, Arif Ghafoor, "Role administration: X-GTRBAC admin: a decentralized administration model for enterprise wide access control", Proceedings of the ninth ACM symposium on Access control models and technologies, June 2004.

[CCC06] Frederic Cuppens, Nora Cuppens-Boulahia, and Celine Coma, “O2O: Virtual

Private Organizations to Manage Security Policy Interoperabilityl", in Aditya Bagchi and Vijayalakshmi Atluri Eds. Information Systems Security, Proceeding of Second International Conference, ICISS 2006, Kolkata, India, December 2006, LNCS 4332, Springer, pages 101-115.

[CL03] Jason Crampton and George Loizou, "Administrative Scope: A Foundation for Role-Based Administrative Models", ACM Transactions on Information and System Security, Volume 6, Number 2, May 2003, pages 201-231.

[CLSDAA01] Michael J. Covington, Wende Long, Srividhya Srinivasan, Anind K. Dev, Mustaque Ahamad, Gregory D. Abowd, "Securing context-aware applications using environment roles", Proceedings of the sixth ACM symposium on Access control models and technologies, May 2001.

[FBK99] David F. Ferraiolo, John F. Barkley, D. Richard Kuhn, "A role-based access control model and reference implementation within a corporate intranet," ACM Transactions on Information and System Security (TISSEC), Volume 2 Issue 1, February 1999

[FSGKC01] D. Ferraiolo, R. Sandhu, S. Gavrila, D. Kuhn, and R. Chandramouli, "Proposed NIST standard for role-based access control," ACM Transactions on Information and System Security (TISSEC), Volume 4 Issue 3, 2001

[GB98] Serban Gavrila and John Barkley, "Formal Specification for RBAC User/Role and Role/Role Relationship Management", Proceedings of Third ACM Workshop on Role-Based Access Control, Fairfax, VA, October 1998.

[GI97] Luigi Giuri and Pietro Iglio, “Role Templates for Content-Based Access Control”, Proceedings of Second ACM Workshop on Role-Based Access Control, November 1997

[GMPT01] Christos K. Georgiadis, Ioannis Mavridis, George Pangalos, and Roshan K. Thomas, "Flexible Team-Based Access Control Using Contexts", SACMAT’01, May 3-4, 2001, Chantilly, Vriginia, USA.

[GQ96] Li Gong and Xiaolei Qian, "Computational issues in secure interoperation", IEEE Transactions on Software and Engineering, 22(1):43-52, January 1996.

[HMMNR00] A. Herzberg, Y. Mass, J. Mihaeli, D. Naor, and Y. Ravid, “Access control meets public key infrastructure, or: assigning roles to strangers”, Proceedings. 2000 IEEE Symposium on Security and Privacy, 2000. S&P 2000. 14-17 May 2000 Page(s):2 – 14

[JBBG04] J. Joshi, R. Bhatti, E. Bertino, and A. Ghafoor, “Access control language for multi-domain environments”, IEEE Internet Computing, pages 40 - 50, November-December 2004.

[JBLG05] J. B. D. Joshi, E. Bertino, U. Latif, A. Ghafoor, "A generalized temporal role based access control model (GTRBAC)", IEEE Transaction on Knowledge and Data Engineering 17, 1 (Jan. 2005).

[KACM00] A. Kapadia, J. AI-Muhtdai, R. Campbell, and D. Mickunas, “IRBAC 2000: Secure interoperability using dynamic role translation”, In Technical Report: UIUCDCS-R-2000-2162, 2000.

[KBBBCDMST03] A. Abou El Kalam, R. E. Baida, P. Balbiani, S. Benferhat, F. Cuppens, Y. Deswarte, A. Miege, C. Saurel, and G. Trouessin, “Organization Based Access Control”, Policy’03, Como, Italie, June 2003.

[Ker02] A. Kern, "Advanced Features for Enterprise-Wide Role-Based Access Control", In Proceedings of the 18th Annual Computer Security Applications Conference, Las Vegas, Nevada, USA, pages 333-342, December, 2002.

[KKC02] Arun Kumar, Neeran Karnik, Girish Chafle, "Context sensitivity in role-based access control", ACM SIGOPS Operating Systems Review, Volume 36 Issue 3, July 2002.

[KSM03] A. Kern, A. Schaad and J. Moffett, "An Administration Concept for the Enterprise Role-Based Access Control Model", SACMAT'03, June 1-4, Como, Italy.

[LDAP97] Lightweight Directory Access Protocol (v3), RFC2251, December 1997.

[LDAP01] Dynamic Groups for LDAPV3 draft-haripriya-dynamicgroup-00.txt, October 2001.

[LMW02] Ninghui Li, John C. Mitchell, and William H. Winsborough, "Design of a role-based trust management framework", Proc. IEEE Symposium on Security and Privacy, Oakland, USA, May 2002.

[LZQX06] Qi Li, Xinwen Zhang, Sihan Qing, and Mingwei Xu, "Supporting Ad-hoc Collaboration with Group-based RBAC Model", CollaborateCom-2006, Atlanta, Georgia, USA, November 2006.

[NO99] Matunda Nyanchama and Sylvia Osborn, "The Role Graph Model and Conflict Of Interest", ACM Transactions on Information and System Security, Volume 2, Number 1, February 1999, pages 3-33.

[OSZ06] Sejong Oh, Ravi Sandhu, and Xinwen Zhang, "An Effective Role Administration Model Using Organization Structure", ACM Transactions on Information and System Security, Volume 9, Number 2, May 2006, pages 113-137.

[Par03] Jaehong Park, "Usage Control: A Unified Framework for Next Generation Access Control", PhD Dissertation, George Mason University, Summer 2003.

[PCND04] Joon S. Park, Keith P. Costello, Teresa M. Neven, Josh A. Diosomito, "A Composite RBAC Approach for Large, Complex Organizations", SACMAT'04, June 2-4, 2004, Yorktown Heights, New York, USA.

[PJ05] S. Piromruen and J. Joshi, "An RBAC framework for time constrained secure interoperation in multi-domain environments", In Proceedings of 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems(WORDS'05), pages 36-48, 2005.

[PS01] Najam Perwaiz and Ian Sommerville, "Structured Management of Role-Permission Relationships", SACMAT'01, May 3-4, 2001, Chantilly, Vriginia, USA.

[PST91] B. Potter, J. Sinclair, and D. Till, "An Introduction to Formal Specification and Z", Prentice-Hall, Ney York, NY, 1991.

[RKY06] Indrakshi Ray, Mahendra Kumar, and Lijun Yu, "LRBAC: A Location-Aware Role-Based Access Control Model", in Aditya Bagchi and Vijayalakshmi Atluri Eds. Information Systems Security, Proceeding of Second International Conference, ICISS 2006, Kolkata, India, December 2006, LNCS 4332, Springer, pages 147-161.

[RTI02] RTI International, "The Economic Impact of Role-Based Access Control", March 2002, http://www.nist.gov/director/prog-ofc/report02-1.pdf

[SBM99] Ravi Sandhu, Venkata Bhamidipati, and Qamar Munawer, "The ARBAC97 Model for Role-Based Administration of Roles", ACM Transactions on Information and Systems Security, Volume 2, Number, February 1999.

[SCY96] Ravi Sandhu, Edward Coyne, Hal Feinstein and Charles Youman, "Role-Based Access Control Models", IEEE Computer, Volume 29, Number 2, February 1996.

[SFK00] Ravi Sandhu, David Ferraiolo, and Richard Kuhn, "The NIST Model for Role-Based Access Control: Towards A Unified Standard, National Institute of Standards and Technology", December 2000, http://csrc.nist.gov/rbac/sandhu-ferraiolo-kuhn-00.pdf

[SJBG05] B. Shafiq, J. Joshi, E. Bertino, and A. Ghafoor, "Secure interoperation in a multi-domain environment employing RBAC policies", IEEE Transactions on Knowledge and Date Engineering, 17(11):1557-1577, November 2005.

[SMJ01] Andreas Schaad, Jonathan Moffett, Jeremy Jacob, "The Role-Based Access Control System of a European Bank: A Case Study and Discussion", SACMAT'01, May 3-4, 2001, Chantilly, Vriginia, USA.

[SP98] Ravi Sandhu , Joon S. Park, "Decentralized user-role assignment for Web-based intranets", Proceedings of the third ACM workshop on Role-based access control, p.1-12, October 22-23, 1998, Fairfax, Virginia, United States

[SRZ06] Ravi Sandhu, Kumar Ranganathan, and Xinwen Zhang, "Secure Information Sharing Enabled by Tusted Computing and PEI Models", ASIACCS '06, March 2006, Taipei, Taiwan.

[TAPH05] William Tolone, Gail-Joon Ahn, Tanusree Pai, and Seng-Phil Hong, "Access control in collaborative systems", ACM Computing Surveys, 37(1):29-41, May 2005.

[Tho97] R.K. Thomas, “Team-Based Access Control (TMAC): A Primitive for Applying Role-Based Access Controls in Collaborative Environments”, Proceedings of the Second ACM workshop on Role-based Access Control, Fairfax, VA, USA, 1997.

[WJ03] William H. Winsborough and Jay Jacobs, “Automated Trust Negotiation Technology with Attribute-based Access Control”, Proceedings of the DARPA Information Survivability Conference and Exposition (DISCEX’03), IEEE Computer Society, 2003.

[WWJ04] Lingyu Wang, Duminda Wijesekera, and Sushil Jajodia, "A Logic-based Framework for Attribute based Access Control", CCS’04, October 25-29, 2004, Washington, DC, USA.

[YMB01] W. Yao, K. Moody, and J. Bacon, "A Model of OASIS Role-Based Control and its Support for Active Security", SACMAT’01, Chantilly, Virginia, USA, May 3-4, 2001.

[ZBM01] Y. Zhong, B. Bhargava, and M. Mahoui, "Trustworthiness Based Authorization on WWW", In IEEE workshop on Security in Distributed Data Warehousing, New Orleans, USA, Oct. 2001.

[Zha06] Xinwen Zhang, "Formal Model And Analysis of Usage Control", PhD Dissertation, George Mason University, Spring 2006.

[ZZS06] Zhixiong Zhang, Xinwen Zhang, and Ravi Sandhu, “ROBAC: Scalable Role and Organization Based Access Control Models”, Proceedings of CollaborateCom-2006/TrustCol-2006, Atlanta, Georgia, USA, November 2006.

[ZZS07] Zhixiong Zhang, Xinwen Zhang, and Ravi Sandhu, “Towards a Scalable Role and Organization Based Access Control Model with Decentralized Security Administration”, in Manish Gupta and Raj Sharman edit: “Handbook of Research on Social and Organizational Liabilities in Information Security”, IGI Global publications. Accepted for publishing in April 2007.

CURRICULUM VITAE

Zhixiong Zhang received his Bachelor of Science in Computer Science from Fudan University in 1983. He received his Master of Engineering in Electronic Engineering from Shanghai Jiao Tong University in 1986. He also received Graduate Certificates in Software System Engineering and in Information Security and Assurance from George Mason University in 2000 and 2006 respectively. He worked as a research assistant and instructor in Shanghai Jiao Tong University from 1986 - 1993. He also worked as a software engineer for several companies in Japan and in the United States. He is currently a senior software developer for The College Board. He has been a practitioner in the Information Technology field for more than 20 years. His research interests include software engineering, information security, and artificial intelligence. He received his Ph.D. degree in Information Technology from George Mason University in 2008.