AN ACCESS CONTROL AND TRUST MANAGEMENT FRAMEWORK FOR LOOSELY-COUPLED MULTIDOMAIN ENVIRONMENTS

by

Yue Zhang

B.S. in Computer Science Department,

Nanjing University of Science and Technology, 2004

Submitted to the Graduate Faculty of

School of Information Sciences in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy.

University of Pittsburgh

2011

UNIVERSITY OF PITTSBURGH

SCHOOL OF INFORMATION SCIENCE

This thesis was presented

by

Yue Zhang

It was defended on

November, 4th, 2010

and approved by

Prashant Krishnamurthy, Associate Professor, Information Science Department

Adam J. Lee, Assistant Professor, Computer Science Department

Michael Spring, Associate Professor, Information Science Department

Vladimir Zadorozhny, Associate Professor, Information Science Department

Thesis Advisor: James B.D. Joshi, Associate Professor, Information Science Department

Copyright © by Yue Zhang

2011

AN ACCESS CONTROL AND TRUST MANAGEMENT FRAMEWORK FOR LOOSELY-COUPLED MULTIDOMAIN ENVIRONMENTS

Yue Zhang, PhD

University of Pittsburgh, 2011

Multidomain environments where multiple organizations interoperate with each other are becoming a reality as can be seen in emerging Internet-based enterprise applications. Access control to ensure secure interoperation in such an environment is a crucial challenge. A multidomain environment can be categorized as *tightly-coupled* and *loosely-coupled*. The access control challenges in the *loosely-coupled* environment have not been studied adequately in the literature.

In a *loosely-coupled* environment, different domains do not know each other before they interoperate. Therefore, traditional approaches based on users' identities cannot be applied directly. Motivated by this, researchers have developed several *attribute-based* authorization approaches to dynamically build trust between previously unknown domains. However, these approaches all focus on building trust between individual requesting users and the resource providing domain. We demonstrate that such approaches are inefficient when the requests are issued by a set of users assigned to a functional role in the organization. Moreover, preserving *principle of security* has long been recognized as a challenging problem when facilitating interoperations. Existing research work has mainly focused on solving this problem only in a *tightly-coupled* environment where a global policy is used to preserve the *principle of security*.

In this thesis, we propose a role-based access control and trust management framework for *loosely-coupled* environments. In particular, we allow the users to specify the interoperation requests in terms of requested permissions and propose several *role mapping* algorithms to map the requested permissions into roles in the resource providing domain. Then, we propose a **Simplify** algorithm to simplify the distributed proof procedures when a set of requests are issued according to the functions of some roles in the requesting domain. Our experiments show that our **Simplify** algorithm significantly simplifies such procedures when the total number of credentials in the environment is sufficiently large, which is quite common in practical applications. Finally, we propose a novel *policy integration* approach using the special semantics of *hybrid role hierarchy* to preserve the *principle of security*. At the end of this dissertation a brief discussion of implemented prototype of our framework is present.

TABLE OF CONTENTS

TABLE OF CONTENTS vi
LIST OF TABLES vii
LIST OF FIGURES viii
PREFACE x
1.0 INTRODUCTION 1
2.0 BACKGROUND AND RELATED WORK 11
2.1 ACCESS CONTROL MODELS IN SINGLE DOMAIN SYSTEMS 11
2.1.1 Role Based Access Control (RBAC) 13
2.1.2 Hybrid Hierarchy 16
2.1.3 Generalized Temporal Role Based Access Control (GTRBAC) 19
2.1.4 Role-based Administrative Models 24
2.2 SECURE INTEROPERATION IN MULTIDOMAIN ENVIRONMENTS 27
2.2.1 Global Policy Based Approaches 27
2.2.2 Trust Management Approaches in Multidomain Environments 29
2.2.3 Tightly and Loosely-Coupled Environments 32
3.0 ACCESS CONTROL CHALLENGES IN LOOSELY-COUPLED ENVIRONMENTS 33
3.1 TIGHTLY-COUPLED ENVIRONMENTS 33
3.2 LOOSELY-COUPLED ENVIRONMENTS 37
4.0 THE PROPOSED ACCESS CONTROL AND TRUST MANAGEMENT FRAMEWORK 44
4.1 OVERVIEW 45
4.2 THE ROLE MAPPING COMPONENT 48
4.3 THE TRUST MANAGEMENT COMPONENT 58
4.3.1 A Motivational Example 63
4.3.2 The Simplify() Algorithm 67
4.3.3 Proof Engine 93
4.3.4 Evaluation 94
4.4 THE POLICY INTEGRATION COMPONENT 112
4.4.1 Policy Integration 112
4.4.2 Administrative Model 117
4.5 PROTOTYPE 127
5.0 CONCLUSION AND FUTURE WORK 137
BIBLIOGRAPHY 141

LIST OF TABLES

Table 2. 1. Hierarchical Operations in SARBAC-RHA 26
Table 2. 2. User-Role operations in SARBAC-URA 26

Table 4. 1. Results of each step of Role-Mapping-1 53
Table 4. 2. Results of each step of Role-Mapping-2a 54
Table 4. 3. Results of each step of Role-Mapping-2b 56
Table 4. 4. Example of using Simplify() 84
Table 4. 5. Hierarchical Operations in SARBAC-HH 120
Table 4. 6. Success conditions for operations involved in the Policy Integration component. 127
Table 4. 7. An example test case of GTRBAC Implementation 135

LIST OF FIGURES

Figure 2. 1. Constraints and hierarchy in RBAC 13
Figure 2. 2. Permission assignments in (a) RBAC and (b) non-RBAC 14
Figure 2. 3. A Simple Role Hierarchy 15
Figure 2. 4. Derived relations in a hybrid hierarchy 18
Figure 2. 5. Example of (a) *Cyclic inheritance conflict*; (b) *Violation of SoD* 29

Figure 3. 1. An example of the *cyclic inheritance conflict* in a *tightly-coupled* environment 35
Figure 3. 2. An example of the *violation of SoD* in a *tightly-coupled* environment 37
Figure 3. 3. An example of the *cyclic inheritance conflict* in a *loosely-coupled* environment 43
Figure 3. 4. An example of the *violation of SoD* in a *loosely-coupled* environment 43

Figure 4. 1. Interaction and data flow among the components 44
Figure 4. 2. User Authorization Query Model 49
Figure 4. 3. An example RBAC policy to show the role mapping algorithms 50
Figure 4. 4. The algorithm to solve role mapping problem 1 52
Figure 4. 5. Algorithm for the role mapping problem 2(a) 54
Figure 4. 6. The algorithm to solve the role mapping problem 2(b) 55
Figure 4. 7. Role-Mapping(R, P_{RQ}) 57
Figure 4. 8. BuildAOT algorithm 71
Figure 4. 9. the algorithm to build the proof for a single role 72
Figure 4. 10. the algorithm to build the proof for a set of roles 73
Figure 4. 11. Simplify() Algorithm 80
Figure 4. 12. algorithm to simulate a multi-domain environment 96
Figure 4. 13. Comparison of $|A.R|$ when $(|B.R|, |A.R|)=(1,1)$ 107
Figure 4. 14. Comparison of $|A.R|$ when $(|B.R|, |A.R|)=(1,2)$ 107
Figure 4. 15. Comparison of $|A.R|$ when $(|B.R|, |A.R|)=(1,3)$ 107
Figure 4. 16. Comparison of $|A.R|$ when $(|B.R|, |A.R|)=(1,4)$ 108
Figure 4. 17. Comparison of $|B.R|$ when $(|B.R|, |A.R|)=(1,1)$ 109
Figure 4. 18. Comparison of $|B.R|$ when $(|B.R|, |A.R|)=(2,1)$ 109
Figure 4. 19. Comparison of $|B.R|$ when $(|B.R|, |A.R|)=(3,1)$ 109
Figure 4. 20. Comparison of $|B.R|$ when $(|B.R|, |A.R|)=(4,1)$ 110
Figure 4. 21. Comparison of $|A.R|$, $|B.R|$ when $(|B.R|, |A.R|)=(1,1)$ 110
Figure 4. 22. Comparison of $|A.R|$, $|B.R|$ when $(|B.R|, |A.R|)=(2,2)$ 111
Figure 4. 23. Comparison of $|A.R|$, $|B.R|$ when $(|B.R|, |A.R|)=(3,3)$ 111

Figure 4. 24. the use of access role and hybrid hierarchy to facilitate interoperation 114
Figure 4. 25. Using access role to prevent (a) cyclic inheritance conflicts; (b) violations of SoD 116
Figure 4. 26. Administrative Scope in SARBAC and SARBAC-HH.....119
Figure 4. 27. Parameters in AddRole..... 121
Figure 4. 28. The ChangeEdge operation won't succeed 123
Figure 4. 29. An example standard hierarchy 125
Figure 4. 30. Access control system in the individual domain 127
Figure 4. 31. The proposed GTRBAC engine 130

PREFACE

The research presented in this thesis has been supported by the US National Science Foundation – Intelligent Information Systems (IIS) CAREER award IIS-0545912.

1.0 INTRODUCTION

Multidomain environments in which multiple organizations interoperate with each other are becoming a reality, as seen in the emerging Internet-based enterprise applications. In these types of environments, it is a significant challenge to ensure that cross-domain accesses to facilitate information sharing are employed in a secure way. This is referred to as the multidomain *secure interoperation* problem [1]. Gong *et al.* introduce the following two principles for *secure interoperation* in multi-domain environments [1]:

- *Principle of Autonomy: If an access is permitted within an individual system, it must also be permitted under secure interoperation.*
- *Principle of Security: If an access is not permitted within an individual system, it must not be permitted under secure interoperation.*

A multidomain environment can be characterized into *tightly-coupled* environment and *loosely-coupled* environment. In a *tightly-coupled* environment, the access control and interoperation needs are typically predefined, and a global policy is created by integrating all the individual policies to facilitate those interoperation needs. In a *loosely-coupled* environment, different domains join and leave a multidomain environment dynamically and the interoperation

needs are dynamic and cannot be predefined. In the literature, several approaches have been proposed to address the access control challenges in *tightly-coupled* environments. Gong *et al.* [1] have studied the computational complexity of the global policy using Access Control Matrix (ACM) model. Bonati *et al.* [2] and Dawson *et al.* [3] have studied the policy integration problem applied to domains employing Multi-Level Security (MLS) models. Basit *et al.* [4] have studied how to specify a global policy by integrating different individual access control policies using Role Based Access Control (RBAC) model [5]. Unfortunately, the access control challenges in *loosely-coupled* environments have not been studied adequately in the literature. Piromruen *et al.* have proposed a secure interoperation framework focusing on how to establish secure interoperation between the *requesting domain* and the *providing domains* based on RBAC [42]. Their approach does not assume the existence of a global policy. However, they assume the naïve RBAC policy is used to make interoperation authorization decisions and do not consider the fact that user identities are usually not known to the resource providing domain. Shehab *et al.*'s SEcure Role mApping Technique (SERAT) focuses on finding the cross-domain authorization paths in a decentralized way [43]. However, they assume that a permitted interoperation set is pre-defined in the environment. There are also several research efforts on *trust management* [19, 20, 21, 22] which aim to make authorization decisions between previously unknown domains. However, they are not discussed in the context of a *loosely-coupled* environment and can only solve part of the challenges in *loosely-coupled* environments.

Role Based Access Control (RBAC) models have received much attention as a general approach to access control [5, 6, 7]. The survey conducted by NIST [8] shows that in many organizations the access control decisions is based on a person's roles and responsibilities within the organization, making role-based approach suitable for expressing security requirements. One important feature of RBAC is the role hierarchy. The use of role hierarchy can greatly simplify the policy specification task, since the administrators do not need to assign the permissions of the junior role to the senior role explicitly. Recently, many extensions of RBAC have been proposed to support the specification of more fine-grained policy requirements. Generalized Temporal Role Based Access Control (GTRBAC) model [9] is one of such RBAC extensions supporting temporal constraints on policies. In many situations, it is desirable to restrict the authorizations based on temporal constraints. For example, a user Alice may be assigned to DayNurse role only during daytime. Several researchers [10, 11] have also identified various limitations of the standard role hierarchy used in RBAC. Joshi *et al.* have proposed the notion of hybrid hierarchy [12] that overcomes some of those limitations of the standard role hierarchy. As a result, GTRBAC is a good choice for defining the local policies in each individual domain. However, the interoperation authorization decisions cannot be made directly on them since the user identities are not known in a *loosely-coupled* environment. In this thesis, we assume that GTRBAC is used in each individual domain to specify its local policy and the focus of this thesis is to study and propose solutions for access control challenges specific to the *loosely-coupled* environment.

In a *loosely-coupled* multidomain environment where each domain employs GTRBAC and hybrid hierarchy, there exist several specific access control challenges. In a traditional single domain system, users usually know the role structure of the organization and hence could request to assume the corresponding roles directly in order to perform the jobs. In a *loosely-coupled* environment, however, it is typically not practical to assume that users have already known the role structure of external domains. As a result, it is desirable to allow users to request the permissions directly. And the resource-providing domains need to identify a set of its local roles containing the requested permissions for the external users to assume.

Once the initial interoperation requests have been translated into a set of requested roles, the providing domain needs to make decisions on whether to authorize the requests or not based on their local policies and the interoperation requirements. Since the identity of the requesting users may not be known to the external domain, traditional identity-based access control approaches are not suitable [22]. A *trust management* approach is needed to facilitate access requests from previously unknown users. In role based multidomain environments, it is very common that several different users assigned to the same role (or a very small set of related roles) would request to acquire the same external resource several times in a period. In this thesis, we refer to such interoperation request scenario as *Role-based interoperation Access Requests (rar)*, and refer to the role(s) that requesting users assigned to as *requesting role*(*s*). In such a scenario, different users all request the same external resource because the functionality of the *requesting role* requires obtaining the external resource, and it is common that several users have been

assigned to the same role(s) (i.e. occupying the same position) in the same period. For example, assume Bob is travelling outside and needs to go to the emergency room in the local hospital. The assigned nurse there needs to obtain Bob's health information from his home hospital. Moreover, there might be several persons assigned to the nurse position (e.g. some during daytime, and some during night time) when taking care of Bob. They all need to acquire Bob's health information when they are on duty. From access control perspective, obviously it is not secure to allow the first nurse who has obtained Bob's health information to disclose it to the subsequent nurses. A more secure way is to require each nurse issuing a separate request and each request to be evaluated and authorized separately for each nurse. Here, we reach the *role-based interoperation request* scenario: different persons assigned to the same role (nurse) need to request the same external resource (Bob's health information) several times (when each person is taking the position) in a period (the time period when Bob is taken cared of). Using traditional role-based distributed proof approaches (e.g. DCCD), each single user needs to prove separately that he/she has the credentials required for the accessing requested resources. However, they will typically request the same external resources since they are assigned to the same requesting role(s) whose functionalities require acquiring those external resources. Unfortunately, few existing approaches have made use of this property to simplify the distributed proof procedures.

Lastly, several researchers have shown that the introduction of global policy in *tightly-coupled* environments could violate the *principle of security*. Although there is typically

no global policy in the *loosely-coupled* environment, the existence of multiple authorized interoperations could also violate the *principle of security*. Proper mechanisms need to be used to address such a problem.

In this thesis, we address the access control and trust management challenges in *loosely-coupled* environments as discussed above, and develop an access control and trust management framework consisting of three major components: (1) Role Mapping; (2) Trust Management; and (3) Policy Integration. First, we develop several role mapping algorithms to identify a set of roles that contain all the requested permissions. We show that it is more convenient to specify the interoperation requests in terms of requested permissions. Recall that in RBAC, permissions are made available through roles. Therefore, the resource providing domain needs to find out which of its local roles contain the requested permissions. This problem becomes more challenging when hybrid role hierarchy is used. Motivated by this, we propose 3 greedy role mapping algorithms to identify such roles according to the local policy. The proposed algorithms are able to handle three scenarios: (1) when exactly matched role set exists; (2) no exactly matched role set exists and the principle of *least privilege* is important; (3) no exactly matched role set exists and the *availability* is more important. Second, we develop a **Simplify** algorithm to simplify the role-based distributed proof procedure. In particular, we base our work on the role-based distributed proof procedure proposed by Li *et al.* (i.e. *RT* families of trust management language [22] and *Distributed Credential Chain Discovery* (DCCD) algorithm [26]). We first show that there is a common type of interoperation request in *loosely-coupled*

environment. That is, *role-based interoperation access request* where the access requests are issued according to the functional needs of the roles in the organization rather than from the individual behaviors. In this case, we show that DCCD approach is inefficient since it can only authorize the resource to the unknown users but not roles in unknown domains. Motivated by this, we propose a **Simplify** algorithm to simplify the distributed proof procedure as defined in DCCD approach by analyzing the policies of the requesting roles and requested roles. We conduct several experiments using simulation and the experimental results show that our approach significantly outperforms DCCD when the total number of credentials in the environment is sufficiently large, which is quiet common in *loosely-coupled* environments. Third, we develop a novel *policy integration* approach using the special semantic of *hybrid role hierarchy* to preserve the *principle of security*. Researchers have shown that violations of *principle of security* could be introduced in the global policy that is used to facilitate interoperations in *tightly-coupled* environments. They have proposed several solutions to detect and remove such violations in the global policy. We show that violations of *principle of security* could also be introduced in *loosely-coupled* environment although there is no global policy in it. And the existing approaches dealing with *tightly-coupled* environment cannot be applied. Motivated by this, we develop a novel policy integration approach that is able to preserve *principle of security* during interoperations. We use hybrid hierarchy to facilitate authorized interoperations and the special semantics of hybrid hierarchy guarantees that there is no violation of *principle of security*. To do this, we need to make several changes to the local GTRBAC

policy, which should be done properly by only the authorized administrators according to appropriate administrative models. However, as far as we know there are no existing administrative models in the literature that is able to deal with hybrid hierarchy. We also propose an administrative model for RBAC with hybrid hierarchy.

From all the above disscussions, the goal of this thesis is to propose an access control and trust management framework for *loosely-coupled* multidomain environment that is able to: (1) allow the users to specify the requested permissions directly; (2) simplify the trust management process assuming the user's requests are made according to the functionalities of their assigned roles ; (3) preserve the *principle of security* without using the global policy. In particular, the research presented in this thesis makes a number of contributions as follows:

- We clearly characterize the *tightly-coupled* and *loosely-coupled* environments, and analyze the access control challenges specific to each. Such analysis helps us to develop access control mechanisms that are especially suitable for *loosely-coupled* environments.
- We assume that users express the interoperation access requests in terms of requested permissions rather than requested roles in role-based multi-domain environments. Based on this, we develop three *role mapping* algorithms that are able to identify a set of roles containing the requested permissions according to the local policy of the resource providing domain. Such *role mapping* approaches are desirable in general RBAC systems and more so in multi-domain environments employing with RBAC.

- We show that a special type of interoperation requests – those issued according to the functionalities of the roles in an organization – is very common in role-based loosely-coupled environments and show that existing distributed proof systems are inefficient in dealing with such requests. Therefore, we propose a Simplify algorithm that significantly outperforms traditional role-based distributed proof procedures (in particular, DCCD approach [26]) when the total number of credentials is sufficiently large.
- We show violations of *principle of security* could be introduced in *loosely-coupled* environment and develop a Policy Integration component to prevent such violations. We also develop an administrative model for the required administrative operations involved in the Policy Integration component.

To our knowledge, no prior research has addressed the above issues in a unified manner, in the sense of analyzing the access control challenges for the *loosely-coupled* environment and developing an access control and trust management framework based on those identified challenges. Given the growing emphasis on interoperations over *loosely-coupled* multidomain environments, we believe the work presented in this thesis represents an important step towards addressing the access control issues in the *loosely-coupled* environments.

The rest of this thesis is organized as follows. In section 2, we present relevant background and related work on access control issues in multidomain environments. In section 3, we discuss the differences between *tightly-coupled* and *loosely-coupled* environments and

identify the access control challenges specific to each. In section 4, we present our access control and trust management framework for *loosely-coupled environments*. Finally, in section 5, we conclude our work and point out possible future directions.

2.0 BACKGROUND AND RELATED WORK

Access control is a fundamental security issue related to ensuring that only authorized accesses and activities are allowed in a computing environment. Authorizing an entity for accessing computing resources may involve satisfying complex policy rules. Recently, with the increased progress in large scale distributed applications, access control in multidomain environments has become a very significant challenge. In this section, we overview the general access control models in single domains, and the access control and trust management approaches in multidomain environments.

2.1 ACCESS CONTROL MODELS IN SINGLE DOMAIN SYSTEMS

Within a single domain, it is crucial to ensure that any access to its data and resources is properly authorized according to the access control policy. Several access control models have been proposed in the literature to specify and enforce various access control policy requirements in a single domain. Traditional access control approaches are broadly categorized as Discretionary Access Control (DAC) [44, 45, 46] and Mandatory Access Control (MAC) [46, 47, 48, 49]. In

DAC, the basic premise is that subjects have ownership over objects of the system and subjects can grant access rights to or revoke them from other subjects on the objects they own. It has been shown that the major problem of DAC is that it does not ensure information flow control [55]. In MAC, all subjects and objects are classified based on some predefined clearance/sensitivity levels that are used in an access decision [46, 49, 50]. These levels generally form a lattice structure, and hence a MAC policy is sometimes known as a lattice-based policy [49]. Unlike DAC, MAC provides deals with more specific security requirements, such as information flow control policy. However, enforcement of MAC policies is often a difficult task. In particular, for many commercial organizations [51], they do not provide viable solutions because they lack adequate flexibility. Furthermore, organizational security needs are often a mixture of policies that may need to use both DAC and MAC, which necessitates seeking solutions beyond those provided by DAC and MAC only [46].

Role Based Access Control (RBAC) approaches have been shown to offer many benefits over other models in terms of their applicability for a wider range of security requirements [5, 6, 7]. One feature of RBAC is the notion of role hierarchy. However, researchers have found some limitations of the standard role hierarchy supported in RBAC. Hybrid hierarchy has been proposed to overcome the shortcomings of the standard role hierarchy. Recently, the General Temporal Role Based Access Control (GTRBAC) model has been proposed to add temporal constraints into RBAC. In our proposal, we assume each individual domain employs the

GTRBAC model with hybrid hierarchy. Next, we will briefly overview RBAC, hybrid hierarchy, GTRBAC, and the role-based administrative models.

2.1.1 Role Based Access Control (RBAC)

In RBAC, users are assigned memberships to roles and these roles are in turn assigned permissions as shown in Figure 2.1. A user can acquire all the permissions of a role of which he is a member. A role-based approach naturally fits into an organizational context as users are assigned organizational roles that have well-defined duties and responsibilities, and are associated with user qualifications [8].

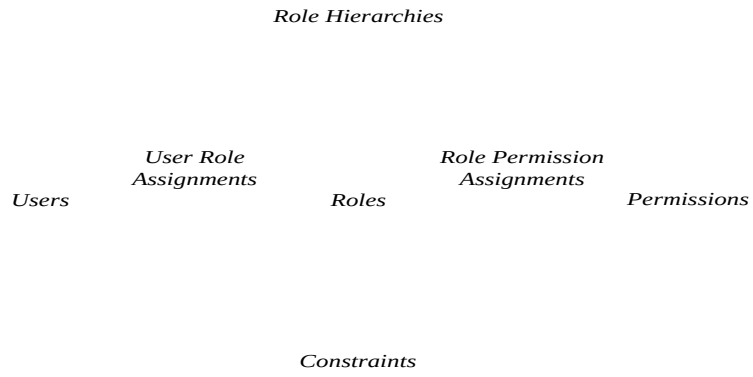


Figure 2. 1. Constraints and hierarchy in RBAC

According to a survey conducted by the US National Institute of Standards and Technology (NIST) [8], RBAC has been found to address many needs of the commercial and government sectors. This study shows that access control decisions in many organizations are

based on "*the roles that individual users take on as part of the organization.*" Many organizations surveyed indicate that they had unique security requirements and the available products did not have adequate flexibility to address them.

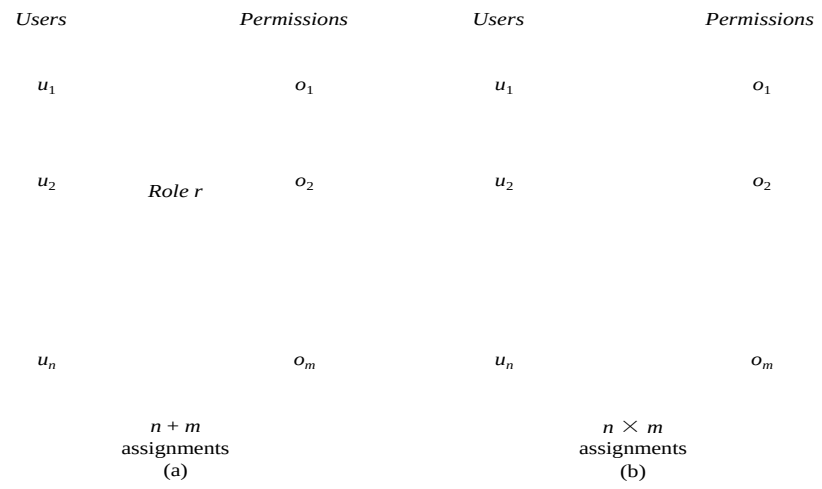


Figure 2. 2. Number of permission assignments in (a) RBAC and (b) non-RBAC

One of the key advantages of an RBAC model is the efficiency it provides in security administration. The *role in the middle* approach to access control removes the direct association of the users from the objects. This greatly simplifies management of authorization in RBAC systems. For example, when a user changes his role, all that needs to be done is to remove his membership from the current role and assign him to the new role. In case authorizations were specified in terms of direct associations between the users and the individual objects, this change would require revoking from the users all their permissions over the objects and explicitly granting the users the new permissions over the new set of objects. Figure 2.2 illustrates such

advantage of using RBAC approach. Using a role-based approach, the number of actual assignments is considerably reduced. Generally, a system has very large number of subjects and objects and hence using RBAC has benefits in terms of managing permissions.

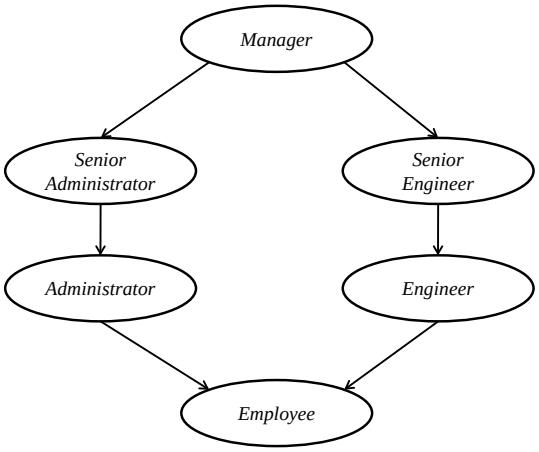


Figure 2. 3. A Simple Role Hierarchy

Another key advantage of RBAC is the use of role hierarchy. Role hierarchies that exist in many organizations based on the principle of generalization and specialization [41]. For example, in a company there may be several roles arranged in a role hierarchy as shown in Figure 2.3: *Employee, Engineer*, *Senior Engineer*, *Administrator*, *Senior Administrator*, and *Manager*. Since everyone is an employee, the *Employee* role models the generic set of access rights available to all. A *Senior Engineer* role will have all the permissions that an *Engineer* role will have, who in turn will have the permissions available to the *Employee* role. Thus, permission

inheritance relations can be organized in role hierarchies. This further simplifies management of access permissions.

Separation of Duty (SoD) has been considered a very desirable organizational security requirement [52, 53, 54]. SoD constraints are enforced mainly to avoid possible fraud in organizations. RBAC can be used to enforce such requirements easily – both statically and dynamically. For example, a user can be prevented from being assigned to two roles, one of which is related to authorizing a check and the other to cashing it, to prevent a possible fraud by using a *static* SoD which says that a user cannot be assigned to two roles,.

2.1.2 Hybrid Hierarchy

Standard role hierarchy supported in RBAC combines the semantics of permission inheritance and activation inheritance together. Several researchers have emphasized the need for separating the permission inheritance and activation inheritance semantics to provide flexibility in expressing fine-grained policies [10, 11]. Sandhu show that under the standard hierarchy semantics, certain *Separation of Duty* (SoD) constraints cannot be defined on hierarchically related roles, thus, restricting its effectiveness in supporting a broader set of fine-grained constraints and, in particular, in representing MAC policies [10]. To address such shortcomings, Sandhu has proposed the ER-RBAC96 model [10] that incorporates a distinction between a usage hierarchy that applies only the permission-inheritance semantics and activation hierarchy that uses the combined hierarchy semantics. Later, Joshi *et al.* [12] have established a clear

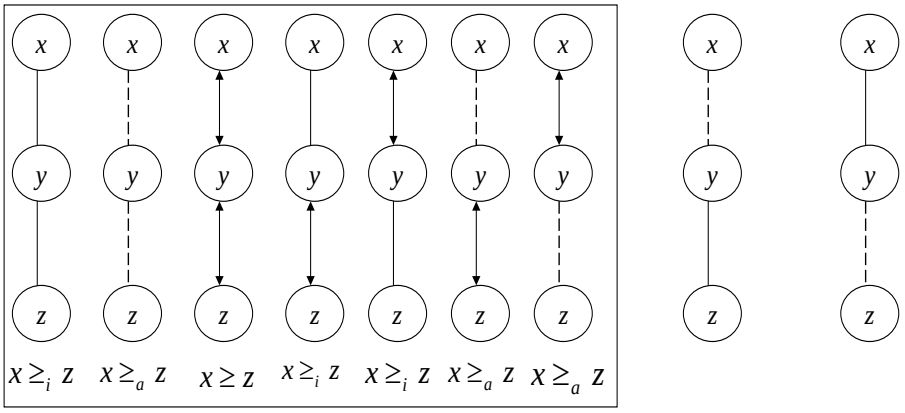
distinction among the following three types of hierarchical relations: *permission-inheritance-only* relation (*I*-relation, $\geq_i$), *activation-only* relation (*A*-relation, $\geq_a$), and the combined *permission-inheritance and activation* relation (*IA*-relation, $\geq$). They further propose the notion of hybrid hierarchy [12] where the above three hierarchical relations co-exist, while only *IA*-relations exist in the standard role hierarchy. Semantically, $x \geq_i y$ (read as x is *I*-senior to y) means that permissions available to y are also available to x; $x \geq_a y$ (read as x is *A*-senior to y) means that users who can activate x can also activate y; $x \geq y$ (read as x is *IA*-senior to y) means that permissions available to y are also available to x and users who can activate x can also activate y. It has been shown that such a fine-grained hierarchy can allow specification of a wide range of security requirements, including the specification of *Dynamic Separation of Duty* (DSoD), and *user-centric* as well as *permission-centric* cardinality constraints on roles [7, 12]. In their critique of the standard RBAC model, Li *et al.* have emphasized that such a distinction should be incorporated in the standard RBAC model to provide clearer semantics to support uniformity in implementations of the RBAC models [13].

Joshi *et al.* have shown that in hybrid hierarchy a hierarchical relation between any pair of roles which are not directly related could be derived [14]. It is obvious that the three hierarchy types are transitive. For instance, if ($x \geq y$) and ($y \geq z$) then it implies ($x \geq z$). Similarly, since *IA*-relation can be considered as both *I*-relation and *A*-relation, we have the following relations as shown in Figure 2.4(a): $(x <f_1> y) \wedge (y <f_2> z) \Rightarrow (x <f> z)$, *where,* $(<f_1> \in \{\geq\}) \vee (<f_2> \in \{\geq\})$ *and* $<f> = <f_1>$, *if* $<f_2> \in \{\geq\}$, *otherwise* $<f> = <f_2>$.

A special derivation relation in hybrid hierarchy is where an *A*-relation is followed by an *I*-relation, as shown in Figure 2.4(b). Here, a user assigned to *x* cannot acquire permissions of *z* by only activating *x*. However, any user assigned to *x* can acquire permissions of *z* by activating *y*, which means *x* can still "inherit" permissions of *z* even if there is no *I*-relation between them. In this situation, we say that *x* has a "conditioned" relation with *z*, written as $x[y] \geq_i z$ [14].

If an *I*-relation is followed by an *A*-relation as shown in Figure 2.4(c), there is not any derived relation between *x* and *z*. In this case, a user assigned to *x* cannot acquire permissions of *z*, since he can only acquire permissions of *y* but cannot activate *y*. To summarize, we define the derived relation between any pair of roles *x* and *y* as follows:

Definition 2.1 (Derived Relation)*: Let x and y be roles such that* $(x \geq_d y)$*, that is, x has a derived relation with y. Then the following holds:* $(x \geq_i y) \vee (x \geq_a y) \vee (x \geq y) \vee (\exists a \in R, x[a] \geq_i y)$



(a) Unconditioned relations (b) $x[y] \geq_i z$ (c) No relation

Figure 2. 4. Derived relations in a hybrid hierarchy

Lemma 2.1: *Let r_1 and r_2 be a pair of roles in the hybrid hierarchy; then users assigned to r_1 can acquire permissions of r_2, iff. the following holds:*

There exists at least one hierarchical path from r_1 to r_2 such that no I-relation precedes an A-relation in the path.

The proof of Lemma 2.1 follows directly from the semantics of hybrid hierarchy as discussed above.

2.1.3 Generalized Temporal Role Based Access Control (GTRBAC)

The GTRBAC model is an extension of RBAC that support temporal constraints [9]. Such a flexibility of supporting various temporal constrains is very helpful in our framework. Nearly all access decisions and related policy updates should be restricted by proper temporal constraints. For example, in our Trust Management components, we add a new RT_0 rule in the providing domain to facilitate the simplified proof. Such new rules should not exist forever. If any of the related trust policy changes, the simplified proof may not be valid and needs to be reevaluated. In such a case, we can use trigger feature in GTBAC to disable the new rule once the relevant policies have changed. Another example, in the Policy Integration component we need to update the local RBAC policy to facilitate the authorized interoperation, that is, creating an access role and connecting them between requesting roles and requested roles (details in Section 4). Such new roles and hierarchical relations should be restricted by proper temporal constraints too. If the authorized interoperation is only authorized for a period of time, obviously the corresponding

access roles and hierarchical relations should also be valid for that period only. The duration constraints in GTRBAC can be used to specify such restrictions. We assume each individual domain employs GTRBAC in our framework. Specifically, there are 6 types of temporal constraints defined in GTRBAC [9]:

Periodicity Constraints

A periodicity constraint contains a periodicity expression and an event expression. For example, given a periodicity constraint <*Monday*, enable Doctor> (Hereafter we will use a slightly different format without changing its semantic for each type of the constraint compared to the original paper for better readability), the system should enable Doctor on every Monday, and disable it on any other day.

Duration Constraints

A duration constraint contains a duration expression and an event expression. For example, *<2 hours*, enable Doctor> means that the system should disable the Doctor role 2 hours after it is enabled. When applying a duration constraint to role activation, it takes four different formats: (1) *total role activation duration per role*: for example, *<10 hours*, activate Doctor> means that the total activation time of Doctor role is 10 hours; (2) *total role activation duration per user-role*: for example, *<10 hours*, activate Doctor by *Alice*> means that the total activation time of Doctor by *Alice* is 10 hours; (3) *maximum role duration per activation per role*: for example, *<10 hours*, activate Doctor per session> means that the Doctor could be activated for at most 10 hours in a single session; (4) *maximum role duration per activation per user-role*: for example, *<10 hours*, activate Doctor by *Alice* per session> means that Doctor could only be activated for at most 10 hours in any of Alice's sessions.

Cardinality Constraint

The cardinality constraint is used to restrict the number of activations. It can be applied in four different scenarios: (1) *total number of activations per role*: for example, *<10*, activate Doctor>means that Doctor can be activated for at most 10 times; (2) *total number of activations per user-role*: for example, *<10*, activate Doctor by *Alice*> means that Doctor can be activated by *Alice* for at most 10 times; (3) *max number of concurrent activations per role*: for example, *<10*, concurrent activate Doctor> means that at any time Doctor should occur in no more than 10 sessions; (4) *max number of concurrent activations per user role*: for example, *<10*, concurrent activate Doctor by *Alice*> means that at any time Doctor should be activated in no more than 10 of Alice's sessions.

Constraints on Constraints

For each of the periodicity, duration, and cardinality constraints, we can add periodicity or duration constraint on the constraint itself. Periodicity Constraint on constraints specifies the enabling time of the corresponding constraints. For example, <*Weekends*, *<2 hours*, enable Doctor>> specifies that the inner duration constraint is enabled only during weekends. Duration Constraint on Constraints specifies how long a constraint is valid. For example, *<10 hours*, *<10*, activate Doctor>> means that once the inner constraint is enabled, it should be disabled after 10 hours.

Run-time Requests

GTRBAC supports administrators and users to issue run-time requests to change the system state. An administrator can issue all the 10 types of event expressions. For example, an administrative run-time request <enable constraint, <*Monday,* enable Doctor>, after *10 min*> enables the inner constraint after 10 minutes. A user can only issue role activation and role de-activation events as run-time requests.

Triggers

A trigger consists of a precondition and a body, both of which are a set of event expressions. If all the operations in the precondition occur, all the operations in the body should be issued. For example, a trigger <enable Doctor → enable DoctorInTraining> specifies that once Doctor is enabled DoctorInTraining should also be enabled.

As shown in Figure 1.1, the GTRBAC engine enforces the GTRBAC policy by updating the RBAC policy according to the semantics of the temporal constraints. For example, enforcing a periodical constraint <Monday, enable Doctor> involves automatically updating the enabling state of the role Doctor in the RBAC policy. In our work, we assume a GTRBAC engine is always running and updating the RBAC policy at fixed frequency. Whenever we mention an "RBAC policy", we mean the current RBAC policy.

2.1.4 Role-based Administrative Models

In our framework, the local RBAC policies need to be changed to facilitate the Policy Integration component. To support evolution of RBAC policies, administration of RBAC policies becomes more and more important. The use of role itself to manage RBAC has become an appealing idea recently. Sandhu *et al.* [15] have proposed an ARBAC97 (Administrative RBAC ‘97) model consisting of URA97 (User-Role Assignment ’97), PRA97 (Permission-Role Assignment ’97), and RRA97 (Role-Role Assignment ’97) model, which use RBAC to manage RBAC policies. They further extend this model to ARBAC99 [16] and ARBAC02 [17]. Crampton *et al.* [18] have proposed a SARBAC (Scoped Administration model for RBAC) model using the concept of administrative scope. SARBAC has been shown to be capable of addressing several shortcomings of ARBAC model and is better in terms of completeness, simplicity, practicality and versatility. Both ARBAC family of models and SARBAC assume that only standard role hierarchy is used. We briefly overview the SARBAC model next.

The basic idea of SARBAC is to use some roles to “administer” some other roles [18]. In this way, the administration can be decentralized. The central idea of SARBAC is the notion of *administrative scope*, which defines the range of roles that can be administered by the given role, as shown next.

Definition 2.2 [18] (Administrative Scope): Given a role a, its administrative scope, $S(a)$, is defined as:

$S(a) = \{r \in R: r \leq a\ ,\ \uparrow r \setminus \uparrow a \subseteq \downarrow a\}$

Where, $\uparrow r = \{x \in R: x \geq r\}$, $\downarrow r = \{x \in R: x \leq r\}$.

Informally, $r \in S(a)$ if every path upwards from r goes through a. This ensures that any change to r made by a will not have unexpected side effects due to inheritance elsewhere in the hierarchy. The strict administrative scope of r is defined as $S(r) \setminus \{r\}$, which we denote by $S^+(r)$. If $r \in S^+(a)$, a is referred to as the administrator of r [18]. The SARBAC model consists of three parts: Role Hierarchy Administration (RHA) model, User Role Assignment (URA) model, and Permission Role Assignment (PRA) model. SARBAC-RHA defines four administration operations: *AddRole*(a, r, $\triangle r$, ∇r), *DeleteRole*(a, r), *AddEdge*(a, c, p), and *DeleteEdge*(a, c, p), where $\triangle r$ is the set of the immediate juniors of the role r, and $\triangledown r$ is the set of the immediate seniors of the role r. Table 2.1 describes the conditions that are required for these operations to succeed. For example, the first rule in Table 2.1 specifies that an administrator role a is able to add a new role r (whose senior and junior roles are ∇r and Δr respectively), if and only if $\triangle r$ is within the strict administrative scope of a and $\triangledown r$ is within the administrative scope of a. The rationale here is that a can administrate both $\triangledown r$ and $\triangle r$ so it should also be able to add a new role between them. Similarly, the operations and their success conditions in SARBAC-URA are summarized in Table 2.2, where $\wedge C$ is a set of constraints needed to be satisfied by users or permissions and *ua-constraints* assign some constraints to each of the role r. Let $R' = \{r_1, \ldots, r_k\}$ be a subset of R and let $\wedge R'$ denote $r_1 \wedge \ldots \wedge r_k$, we have the following definition:

Definition 2.3 (*SARBAC Constraint*): An *SARBAC constraint has the form* $\wedge C$, *where* $C \subseteq R$. *A SARBAC constraint* $\wedge C$ *is satisfied by a user u if* $C \subseteq \downarrow R(u)$. *A SARBAC constraint* $\wedge C$ *is satisfied by a permission p if* $C \subseteq \uparrow R(p)$, *here for any* $Y \subseteq X$, $\uparrow Y = \{x \in X: \exists y \in Y$ *such that* $x \geq y\}$, *and* $\downarrow Y = \{x \in X: \exists y \in Y$ *such that* $x \leq y\}$.

According to definition 2.3, a user is said to satisfy a set of roles if she is assigned to any one of these roles, or the senior role of any of these roles. Intuitively, this constraint guarantees that if a user satisfies a set of roles, she is the member of all these roles. For example, the first row of Table 2.2 shows that if role *a* wants to assign user *u* to role *r*, *r* must be within the administrative scope of *a*; and *u* must satisfy the "pre-condition" associated with role *r*. SARBAC-PRA is very similar to SARBAC-URA by substituting "permissions" for "users".

Table 2. 1. Hierarchical Operations in SARBAC-RHA

Operation	**Conditions**
AddRole (a, r, $\triangle r$, $\triangledown r$)	$\triangle r \subseteq S^+(a), \triangledown r \subseteq S(a)$
DeleteRole (a, r)	$r \in S^+(a)$
AddEdge (a, c, p)	$c, p \in S(a)$
DeleteEdge (a, c, p)	$c, p \in S(a)$

Table 2. 2. User-Role operations in SARBAC-URA

Operation	**Conditions**
AssignUser(a, u, r)	$r \in S(a)$, u satisfies $\wedge C$, $(r, \wedge C) \in$ *ua-constraints*
RevokeUser(a, u, r)	$r \in S(a)$

2.2 SECURE INTEROPERATION IN MULTIDOMAIN ENVIRONMENTS

As indicated by the *principle of autonomy* and *principle of security*, the access control policies in the multidomain environment should be consistent with the access control policies in the individual domains. In the literature, there are two major research areas related to access control in multidomain environments: (1) global policy based approaches; and (2) trust management approaches. In this section, we describe these two areas in detail and also briefly review other related secure interoperation approaches.

2.2.1 Global Policy Based Approaches

A key approach to access control in multidomain environments involves mapping all the individual policies into one centralized global policy, based on which all the interoperation requests are authorized. We refer to such kind of approaches as "global policy based approach" in this thesis. Such work includes Gong *et al.*'s computational complexity analysis [1] based on the Access Control Matrix (ACM) model, Bonati *et al.*'s policy algebra [2], Dawson *et al.*'s approach [3] based on the Multi-Level Security (MLS) model, and Shafiq *et al.*'s secure interoperation framework [4] based on the RBAC model. A global policy should not violate the *principle of autonomy* and *principle of security*. Intuitively, *principle of autonomy* indicates that we may facilitate the interoperation by mapping different policies, but it should not result in the removal of any existing authorization relations in the local policies. *Principle of autonomy* is

typically implicitly guaranteed in the global policy by not changing any local policies when mapping. *Principle of security* indicates that during the interoperation no new authorization relation should be added within each individual domain. The use of global policy in multidomain environments where RBAC is employed could introduce two types of violations of *principle of security*. The first type of violation is referred to as *cyclic inheritance conflicts* [4]. In such a case, the cross-domain hierarchical relationship may introduce a cycle in the global policy enabling a subject lower (or junior) in the hierarchy to acquire the permissions of the subject higher (senior) in the hierarchy. Figure 2.5(a) shows such an example. In Figure 2.5(a), role r_2 of d_1 is made senior to r_3 of d_2 to facilitate some interoperation needs, and r_4 is also made senior to r_1 to facilitate another interoperation need. A cycle (r_2, r_3, r_4, r_1) is introduced in this case, and r_2 can now inherit the permissions of r_1, which violates the *principle of security*. The second type of violation is referred to as a *violation of Separation of Duty* (SoD) [4]. In such a case, the cross-domain hierarchical relationship may enable one subject to acquire permissions that violate the SoD constraint. Figure 2.5(b) shows such an example. In figure 2.5(b), r_1 and r_2 in domain d_1 are restricted by the SoD constraint. r_1 is made senior to r_3 in d_2 to facilitate an interoperation need, and r_3 is made senior to r_2 to facilitate another interoperation need. Now the users of r_1 can acquire the permissions of r_2, thus violating the SoD constraint. Several techniques have been proposed in the literature to detect and remove such violations introduced in the global policy. For example, Shafiq *et al.* propose an Integer Programming based approach to detect and remove *cyclic inheritance cycle* and *violation of SoD* in the global policy [4].

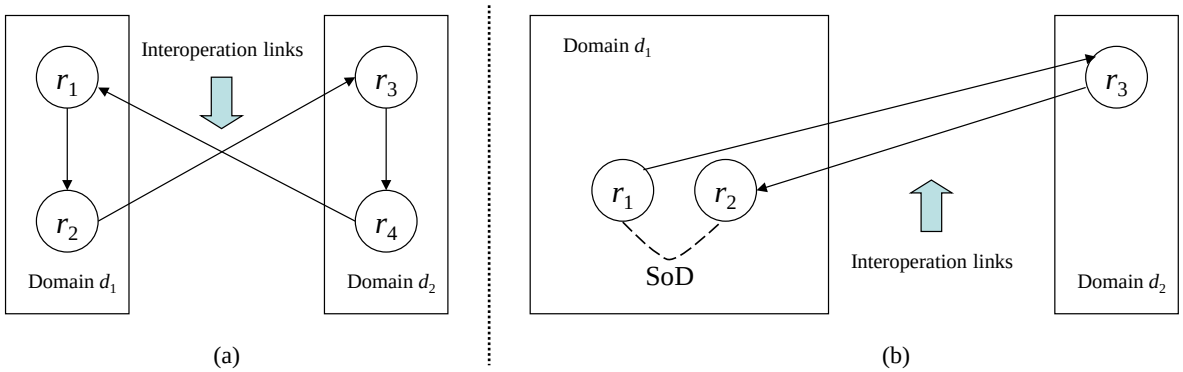


Figure 2. 5. Example of (a) *Cyclic inheritance conflict*; (b) *Violation of SoD*

2.2.2 Trust Management Approaches in Multidomain Environments

Global policy based approaches assume that the interoperation needs can be predefined in order to create the global policy. Another popular approach related to the cross-domain authorization is the trust management based approach [19, 20, 21, 22], which aims to make authorization decisions on dynamic interoperation requests involving domains previously unknown to each other. In such a case, each individual domain typically does not know the identity of the external users who issue the interoperation requests, and hence the authorization decisions are typically made based on the properties of the entities, and such properties are typically encoded in *credentials* in the literature [19, 20, 21, 22]. In a trust management system, each individual domain typically specifies several rules defining which credentials are needed to access its resources. External users requesting the resources need to prove that they have the required credentials for an access right over the resources as defined by the policy of the

resource-providing domain. Whether a user should be given a certain credential could in turn be defined by the policy of the domain issuing that credential, which may further require another set of credentials to be validated. For this reason, such trust management systems are sometimes referred to as *distributed proof systems* in the literature [23, 24, 25]. We refer to such a process of submitting and verifying the proof of credentials needed for requested resources according to the relevant policies distributed over the network as *distributed proof checking procedure*. In our framework, we propose to use a role based trust management language RT_0 [22] as the basis for the proposed Trust Management component.

RT_0 [22] is a role based trust management language that models all the resources using roles. For example, if *Alice* is given a credential certifying that she is a member of *IEEE*, *Alice* is said to be the member of role "IEEE.member". In RT_0, entities are made members of roles through four types of credentials, as shown below [22]:

- *Type* 1 (*Simple Membership*): $A.r \leftarrow D$
- *Type* 2 (*Simple Containment*): $A.r \leftarrow B.r_1$
- *Type* 3 (*Linked Roles*): $A.r \leftarrow A.r_1.r_2$
- *Type* 4 (*Role Intersections*): $A.r \leftarrow f_1 \cap f_2 \cap \ldots \cap f_n$, where f_i (i=1, 2, …, n) is a simple role, or a linked role

The four types of credentials in RT_0 are specified using logical rules. Therefore, we use the following phrases interchangeably in this proposal: ***RT*$_0$ policies, *RT*$_0$ rules,** and ***RT*$_0$ credentials**, and we will omit the prefix "RT_0" if the context is clear. Type 1 rule specifies that

entity D is made the member of role $A.r$; Type 2 rule specifies that any member of $B.r_1$ is also a member of $A.r$; Type 3 rule specifies that any member of r_2 defined by members of $A.r_1$ is also a member of $A.r$; and Type 4 rule specifies that members of all roles f_1 through f_n are also the members of $A.r$. Here, A, B, and D are entities. The body of the type 1 rule (i.e. D) can be the identifier of either a user or a domain. The entities in all other places (i.e. A, B) can only be the identifiers of domains. A role in RT_0 is defined by the domain defining it and a role identifier (i.e. $A.r$). Hereafter, we will use both domain identifier and role identifier to represent a role if we want to emphasize its domain. Otherwise, we will only use the role identifier to represent a role for simplicity.

Given a set of related RT_0 policies, the requesting user needs to prove that she is authorized to the requested resources (i.e. she is the member of the role associated to the requested resources) by showing a chain of credentials that link the requesting entity to the requested role. In a multidomain environment, the credentials may be distributed in different domains, making the discovery of such credential chains a great challenge. Li *et al.* have proposed a distributed credential chain discovery approach to efficiently find such credential chains if they exist [26]. In their approach, they use a graph model to represent all the credentials in the environment. Given an individual interoperation request (i.e. a user's request to assume a role), they check whether the requesting user is the member of the requested role in the graph. They propose a forward search, a backward search, and a bi-directional search to check the role

memberships in the graph. The time complexities of all the three search algorithms they have proposed are cubic to the number of credentials.

2.2.3 Tightly and Loosely-Coupled Environments

In the literature, a multidomain environment is sometimes characterized as "*tightly-coupled*" or "*loosely-coupled*". In [27], Joshi *et al.* describe the "*tightly-coupled* environment" as "there exists one master system and the master mediates accesses to individual systems through a global policy", and describe the "*loosely-coupled* environment" as one where "independent systems dynamically come together to share information for a period of time". It is clear that their characterization of "*tightly-coupled*" environment refers to the multidomain environment that has been studied extensively in the literature by using global policy based approaches. The *loosely-coupled* environment, on the other hand, is an area that has not been studied adequately in the literature. Our focus in this proposal is the access control challenges in *loosely-coupled* environments. We will give a clearer characterization of *tightly-coupled* and *loosely-coupled* environments in section 3. We do not claim that our characterization is the "only acceptable" one for these two terms, nor that our classification using these two terms are the only way to classify multidomain environments. Rather, our aim is to *provide a clearer distinction between these two types of environments so that it can help us to better understand the access control challenges in each.*

3.0 ACCESS CONTROL CHALLENGES IN LOOSELY-COUPLED ENVIRONMENTS

In this section, we first characterize the *tightly-coupled* and *loosely-coupled* multidomain environments. We then identify the specific access control challenges in loosely-coupled environments, which are the focus of this dissertation.

3.1 TIGHTLY-COUPLED ENVIRONMENTS

Characteristic: The domains in a *tightly-coupled* environment are typically closely related to each other and collaborate to pursue some specific common tasks. Such common tasks cannot be completed without proper interoperations, and *such interoperation needs are static and can be predefined*.

Example (from [4]): Consider the interoperation among various offices of a county for the collection and sale of real-estate tax (the common tasks) on property parcels located within the jurisdiction of a concerned county. The concerned county offices would include: County Clerk Office (CCO), County Treasure Office (CTO), and County Attorney Office (CAO). These offices interoperate and share information among each other for budget planning, tax billing and

collection, sale of delinquent taxes, auditing, and other legal purposes. Integration of these local databases is necessary to complete their tax processing tasks.

This is an example of a *tightly-coupled* environment, and the common tasks in this environment is the collection and sale of tax. The three domains have to interoperate with each other to complete their tasks. And the interoperation needs in such an environment are static. For example, the Delinquent Tax Clerk in the CTO *always* need to consult with the County Clerk in the CCO to collect the tax record, and the Redemption Cost Assessor in the CCO *always* need to consult with the Deputy County Attorney Tax Section in the CAO to estimate the tax redemption cost [4]. These interoperations are typically predefined before these organizations interoperate.

Challenges: There are two major access control challenges to ensure secure interoperations in *tightly-coupled* environments.

The first challenge is *how to make an access control decision for a particular interoperation request*. For example: should a user of DelinquentTaxClerk in the CTO be authorized to acquire the permissions of CountyClerk in the CCO? Since the interoperation needs are predefined in *tightly-coupled* environments, the administrators can create a global policy by mapping the local roles in order to facilitate those predefined interoperations. And all the interoperation requests can be checked against such a global policy. In this example, since it is predefined that Delinquent Tax Clerk in the CTO needs to access the records of County Clerk in the CCO to process the job, the administrators would make DelinquentTaxClerk senior to CountyClerk in the global policy to facilitate such an interoperation need. As a result, all the

users assigned to DelinquentTaxClerk are authorized to acquire the permissions of CountyClerk, and all other users in the CTO are not authorized to acquire permissions of CountyClerk in the CCO.

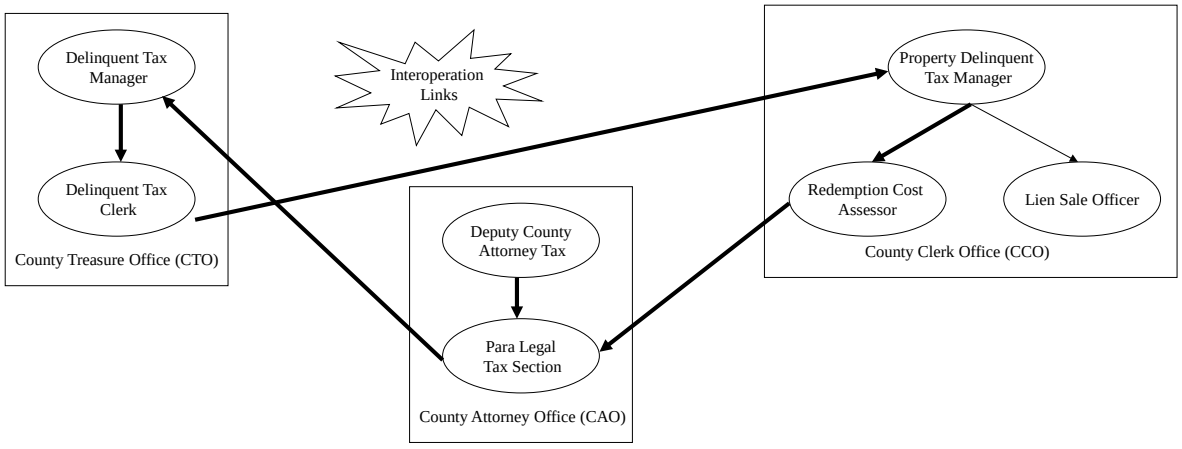


Figure 3. 1. An example of the *cyclic inheritance conflict* in a *tightly-coupled* environment

The second challenge is *how to preserve the principle of security in the global policy*. As discussed before, two types of violations of *principle of security* can be introduced in the global policy. Figure 3.1 shows an example of the *cyclic inheritance conflict* introduced in the global policy in a *tightly-coupled* environment. According to the predefined interoperation needs related to tax processing, Delinquent Tax Clerk in the CTO needs to access the records of Property Delinquent Tax Manager in the CCO to collect the tax record, Redemption Cost Assessor in the CCO needs to access the records of Deputy County Attorney Tax in the CAO to estimate the tax

redemption cost, and Para Legal Tax Section in the CAO needs to access the records of Delinquent Tax Manager in the CTO to collect the relevant records for preparing tax sales pleas [4]. These interoperation needs are facilitated by the three hierarchical relations shown in Figure 3.1. A *cyclic inheritance conflict* is introduced in this global policy (bold arrows in Figure 3.1) and the *principle of security* is violated. Figure 3.2 shows an example of the *violation of SoD* introduced in the global policy in a *tightly-coupled* environment. In domain CTO, Tax Assessment Clerk (TAC) and Tax Billing Clerk (TBC) of CTO are restricted by an SoD constraint, specifying that no single user should acquire the permissions of both the roles. According to the predefined interoperation needs related to tax processing, TBC in the CTO needs to consult the Property Tax Manager (PTM) in the CCO for the billing purpose, and PTM needs to consult TAC for estimating the tax [4]. These interoperation needs are facilitated by two hierarchical relations shown in Figure 3.2. In such a case, the user assigned to TBC can now acquire the permissions of TAC, violating the SoD constraint defined over TAC and TBC. Note that if we represent an SoD constraint using a bi-directional arrow (as shown in Figure 3.2), it can be detected in the way similar to detecting the *cyclic inheritance conflicts*, i.e., detecting the inheritance cycle in the global policy. To remove such violations, we need to break such inheritance cycles. If we remove some interoperation links in the cycle, the interoperation needs cannot be facilitated. If we remove some local link in the cycle, the *principle of autonomy* is violated. As a result, there is a trade-off between maximizing the interoperations and preserving the *principle of autonomy*.

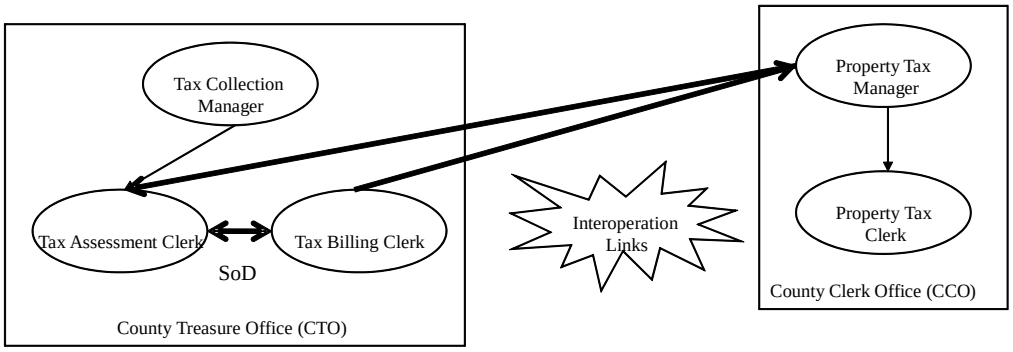


Figure 3. 2. An example of the *violation of SoD* in a *tightly-coupled* environment

3.2 LOOSELY-COUPLED ENVIRONMENTS

Characteristic: The domains in a *loosely-coupled* environment are typically independent of each other and are able to carry out their major functions without interoperating with each other. There are typically no specific common tasks that need to be done through interoperations of all participating domains. Rather, the interoperation needs are usually driven "on demand" to facilitate dynamic data sharing needs. Therefore, *the interoperation needs in loosely-coupled environments are dynamic and may not be predefined*.

Example: Consider a distributed health care system (e.g. HL7 [28]) consisting of different hospitals, clinic, healthcare stations, and other related organizations. Each domain operates on its own to carry out its daily healthcare services related functions. Moreover, they may interoperate and share their information such as Electronic Health Records (EHR) whenever needed, to facilitate dynamic information sharing needs that arise, for instance, when a registered patient is

taken cared of at another hospital while travelling away from home. Interoperation in such an environment is transient, and need based.

This is an example of a *loosely-coupled* environment since the interoperation needs are dynamic and cannot be predefined. For example, assume Bob travels outside his hometown and needs to go to an emergency unit. The local hospital (Hospital A) may need to access his health information from his home hospital (Hospital B) to provide him with a proper treatment. This particular interoperation need is driven by a *specific* event (Bob needs to go to the emergency ward), and we cannot predefine that Hospital A should *always* be authorized to access Bob's health information from Hospital B. Normally, there are no specific common tasks that require the interoperation of all these healthcare domains, and they are able to operate on their own to carry out their daily functions without interoperating with each other.

Challenges: A *loosely-coupled* environment has its specific access control challenges. Although the description of some of the challenges looks similar to those in *tightly-coupled* environments, different approaches are needed due to the unique features of *loosely-coupled* environments.

The first challenge is *how to model the access requests in loosely-coupled environments*. The access request in a single system or *tightly-coupled* environment configured with RBAC is straightforward. In a single system, users know the functional structure of the organization and know which roles they need to assume to perform specific tasks. In a *tightly-coupled* environment, the requesting users typically also know the functional structures of other domains and know which roles they need to assume to facilitate the interoperation. For example, users of

Delinquent Tax Clerk in the CTO know that Property Delinquent Tax Manager in the CCO has access to the records necessary to process the job. As a result, users of DelinquentTaxClerk can issue a request to assume the role of PropertyDelinquentTaxManager. In a *loosely-coupled* environment, however, the access request cannot be modeled by the requested roles since the domains usually do not know the policy structure of other domains. For example, when Bob goes to the emergency room in Hospital A, the healthcare workers there look up and find that Bob is registered in Hospital B (his home hospital). However, they do not know the policy of Hospital B. Therefore, they are not able to request to assume specific roles in Hospital B. Instead, they are only able to request to access Bob's health information. From an RBAC perspective, this example shows that it is more convenient to model the interoperation requests using the target permissions but not target roles in a *loosely-coupled* environment. And it should be the responsibility of the domains containing the requested permissions to identify some of their local roles for external users to assume. For example, Hospital B knows that role Nurse(patient=Bob) has the permissions to view Bob's health information, and may allow healthcare workers in Hospital A to acquire the permissions through it.

The second challenge is *how to make an access control decision for a particular interoperation request*. This challenge looks the same as the first challenge in *tightly-coupled* environments. However, unlike in *tightly-coupled* environments, we cannot let administrators define a global policy to facilitate interoperation because the interoperation needs in a *loosely-coupled* environment cannot be predefined. For example, at the time when both Hospital

A and Hospital B join the environment, one cannot predefine that Hospital A is authorized to access Bob's health information from Hospital B. This is because such a cross-domain access is only necessary when Bob needs to go to the emergency ward in Hospital A and this may never happen. In the literature, researchers have shown that trust management approaches are particularly useful to facilitate such distributed authorizations when the interoperation needs are dynamic and the requesting users are unknown. For example, when the healthcare workers in Hospital A request to access Bob's health information in Hospital B, Hospital B may require that only the users with valid healthcare licenses are allowed to access Bob's health information. Hence, Hospital B will need to ask healthcare workers in Hospital A to present their licenses in order to gain the desired accesses. Once the license has been verified, the interoperation request is authorized and the healthcare workers in Hospital A can access Bob's health information from Hospital B. *This challenge shows that a Trust Management component is necessary in a loosely-coupled environment.*

The third challenge is *how to preserve the principle of security during interoperations among various domains*. Recall that two types of violations of *principle of security* could be introduced in the global policy: i.e. *cyclic inheritance conflicts* and *violations of SoD*. As global policy based approaches cannot be applied to *loosely-coupled* environments, it seems that *loosely-coupled* environments will not suffer from such violations. However, if there are multiple interoperations generated within the same time period, the authorized interoperations could also introduce these two types of violations. Figure 3.3 shows an example of *cyclic inheritance*

conflicts in a *loosely-coupled* environment. Assume that Bob is registered and taken cared of at his home hospital (Hospital B), where both the doctor and resident are authorized to access his health care information. Typically, doctors have more privileges than residents, such as adding a new entry to his record, so Doctor can be made senior to Resident in Hospital B's local policy. In Hospital A located at another city, healthcare workers are responsible for maintaining patients' health information. There may be doctors who are specialists in cancer treatment and they need special privileges to maintain cancer-related information. Therefore, SpecialistDoctor can be made senior to HealthCareWorker in Hospital A. Now assume Bob needs to go to the emergency ward in Hospital A when he travels to that city. To take care of Bob, a healthcare worker in Hospital A needs to access Bob's healthcare records and also needs to add a new entry to Bob's records. If such an interoperation need is authorized, HealthCareWorker of Hospital A needs to be made senior to Doctor of Hospital B to facilitate it (interoperation 1 in Figure 3.3). Assume that at the same time, hospital B receives a cancer patient but is unable to make a proper treatment plan since they are not experts in cancer. The doctor in hospital B can ask the resident to get some help from the specialist doctors in Hospital A (e.g. by accessing some cancer-specific information in Hospital A to choose a proper treatment). If such an interoperation need is authorized, Resident of Hospital B needs to be made senior to SpecialistDoctor of Hospital A to facilitate it (interoperation 2 in Figure 3.3). At this time instant when both interoperations 1 and 2 in Figure 3.3 are authorized, an inheritance cycle is introduced in (shown by the 4 arrows). Figure 3.4 shows an example of the *violation of SoD* in a *loosely-coupled* environment. In a

hospital domain, an SoD constraint is defined over Doctor role and BillingClerk role specifying that no single user can take the Doctor and BillingClerk roles at the same time. Assume the Billing Clerk needs to acquire a patient's insurance information through the InsuranceAgent role in a insurance company. To facilitate this interoperation, BillingClerk is made senior to InsuranceAgent (interoperation 1 in Figure 3.4). At the same time, assume the Insurance Agent in the same insurance company needs to consult the Doctor for some patient's health information in order to estimate the insurance coverage. To facilitate such an interoperation, InsuranceAgent is made senior of Doctor (interoperation 2 in Figure 3.4). At this time instant when both interoperations 1 and 2 in Figure 3.4 are authorized, a *violation of SoD* occurs since Billing Clerk can now acquire the permissions of Doctor. Unlike in a *tightly-coupled* environment, there is no static global policy in *loosely-coupled* environments. Therefore, the existing violation detection and removal approaches employed in global policy based approaches in the literature cannot be applied here. *This challenge shows that a proper mechanism to ensure principle of security is necessary in a loosely-coupled environment.*

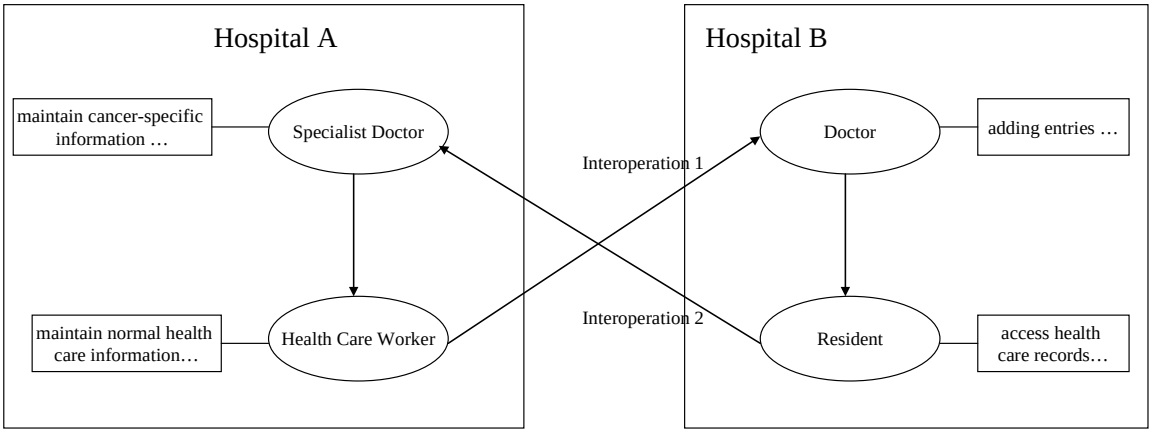


Figure 3. 3. An example of the *cyclic inheritance conflict* in a *loosely-coupled* environment

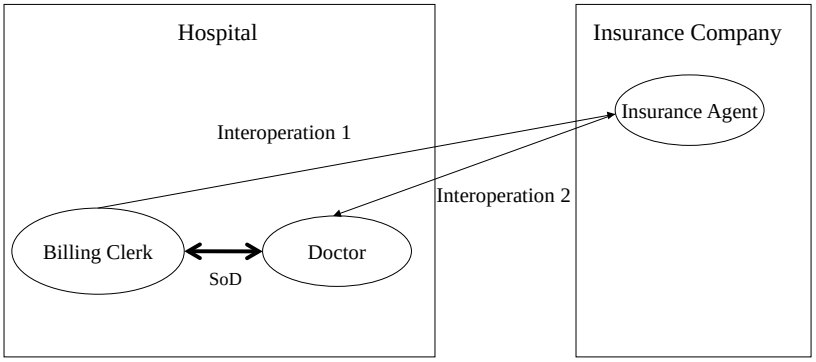


Figure 3. 4. An example of the *violation of SoD* in a *loosely-coupled* environment

4.0 THE PROPOSED ACCESS CONTROL AND TRUST MANAGEMENT FRAMEWORK

In this section, we propose our access control and trust management framework for *loosely-coupled* environments. We aim to address the key access control challenges in *loosely-coupled* environments listed in section 3.2.

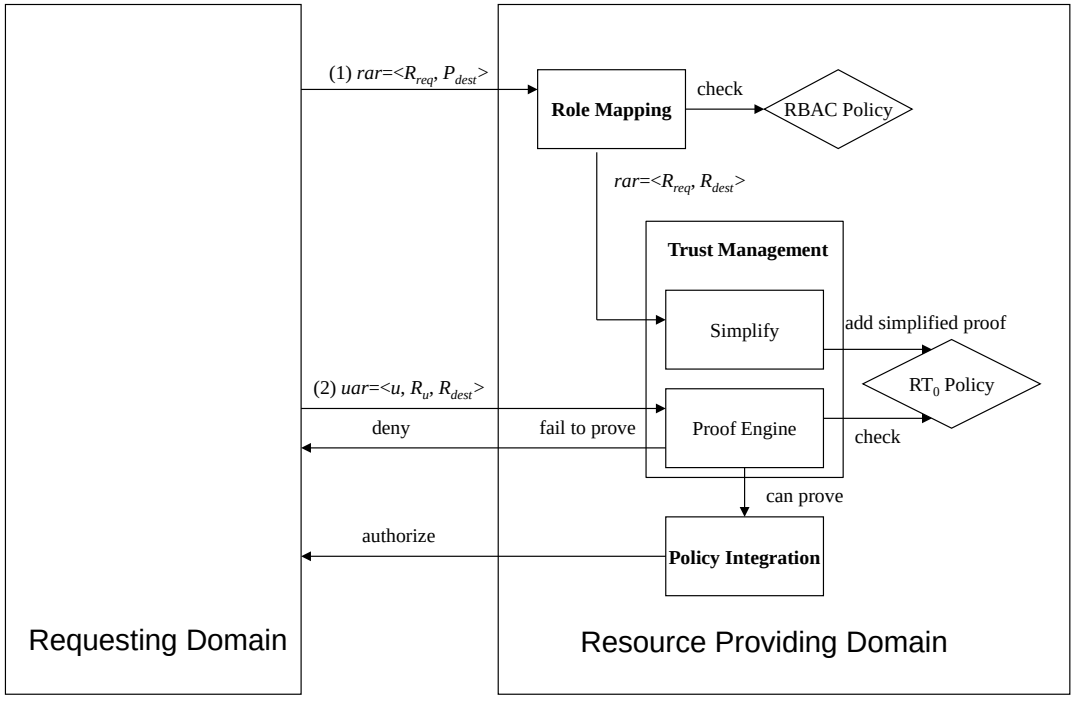


Figure 4. 1. Interaction and data flow among the components

4.1 OVERVIEW

The overall protocol and structure of our framework are shown in Figure 4.1. As mentioned in Section 1, it is more convenient for the requesting users to specify the requested permissions rather than the requested roles in *loosely-coupled* environments. Moreover, we have shown that the Role-based interoperation Access Request (*rar*) is convenient in loosely-coupled environments. Therefore, we first define *rar* formally in this section, as follows:

Definition 4.1(a) (Role-based interoperation Access Request): *A Role-based interoperation Access Request, rar, is defined as a tuple* $< d_{req}, R_{req}, d_{dest}, P_{dest}, T>$, *where* d_{req} *is the requesting domain,* R_{req} *is a set of roles in* d_{req} *such that the function of each role in* R_{req} *requires accessing the common requested external resources,* d_{dest} *is the resource providing domain,* P_{dest} *is a permission set in* d_{dest} *representing access to the requested resources, and T is the valid time period of the request.*

Since *rar* is defined by the requesting roles according to the role structure in the requesting domain, it should be issued on behalf of the requesting domain rather than the individual users. Note that the requesting domain needs to specify a valid time period *T* for the *rar* since the interoperation needs in *loosely-coupled* environments are *dynamic*. Since our focus is how to authorize P_{dest} to R_{req}, we will omit d_{req}, d_{dest}, and *T* from the expression of *rar* hereafter when the context is clear.

Upon receiving an *rar*, the resource providing domain needs to find a set of roles containing the requested permissions P_{dest}. This is done in our Role Mapping component and we refer to the resulting role set as R_{dest}. The new *rar* in terms of $<R_{req}, R_{dest}>$ is fed into the proposed **Simplify** algorithm replaces as many of external roles in the proof of R_{dest} with the local roles of the requesting domain as possible – thus, greatly simplifying the distributed proof procedure. The existing simplified proofs are then added into the RT_0 policy so that the individual users can use them to prove R_{dest}. Now, the individual users are allowed to issue his/her interoperation access requests, which are defined formally as follows:

Definition 4.1(b) (User interoperation Access Request): *A User interoperation Access Request, uar, is defined as a tuple* $< u, R_u, R_{dest}, T>$*, where u is the requesting user,* R_u *is the set of roles u is assigned to,* R_{dest} *is a role set in* d_{dest} *containing the requested permissions (returned by Role Mapping algorithm), and T is the valid time period of the request.*

Again, the most important part of an *uar* is R_u and R_{dest}, so we will omit *u* and *T* hereafter when the context is clear. The *uar* is verified in Proof Engine. If the user can prove R_{dest}, the Policy Integration component is called to authorize the *uar* without violating the *principle of security*; otherwise, the *uar* is denied.

Before we present the details of each component of our framework, we emphasize that the focus of our framework is the access control challenges in *loosely-coupled* environments. Several other issues in a multidomain environment, including the data management, authentication, and communication protocol, are beyond the scope of our work. Moreover, our

framework does not depend on any specific application area or implementation architecture. Different application domains (e.g. military applications, healthcare applications) may have different interoperation requirements and may specify different policies. Different architecture (e.g. Peer-to-Peer, Service Oriented Architecture) may implement our framework using different implementation techniques. However, our framework shall be able to solve the access control challenges in these different environments as long as they are *loosely-coupled* environments (i.e. satisfying the characteristics we described in section 3.2).

Heterogeneity has long been recognized as a fundamental problem in multidomain environments [29]. Especially in *loosely-coupled* environments, different autonomous domains are independent with each other and may represent their resources, organizational structures, and access control policies in different ways. Here, we emphasize that our framework makes two assumptions about the heterogeneity issue. First, we assume each individual domain employs GTRBAC with hybrid hierarchy. We believe this is a reasonable assumption since RBAC and hybrid hierarchy has been shown to be particularly useful in multidomain environments [4, 30], and GTRBAC has been shown as a valuable extension of RBAC [9]. Second, we assume that different domains share a uniform representation for the *essentially* same permission, so that each domain can understand the permissions that other domains request. This is because we use target permissions to model the access requests. Therefore, we need to make sure the permissions provided by the involved domains are exactly the permissions requested by the users. Note that a permission consists of an operation on an object, and is an abstract notion specific to RBAC.

During implementation, different architectures may implement the notion of permission in different ways. Therefore, how to ensure that different domains understand each other about the actual meaning of a permission is *architecture-dependent* and is beyond the scope of our work. For example, if the web service architecture [31, 32] is used, permissions would be implemented as services and many service discovery approaches [33, 34] have been proposed to identify a service according to its semantic representation.

4.2 THE ROLE MAPPING COMPONENT

In a traditional single domain RBAC system, a user's access request is typically modeled as a set of roles requested to be activated by the user. As discussed in section 3.2, it is desirable to model the access control request in terms of requested permissions in *loosely-coupled* environments because the users of a domain typically do not know the roles and hierarchical structures of the external domains. Even in a single RBAC system, allowing users to specify access control requests in terms of permissions would have some benefits. For example, a Windows user may directly request to execute an application (e.g. by double-clicking the shortcut) that only the administrators can execute. Even if she has an administrator account she may not be aware that she has to log-in as administrators to execute the application until prompted by Windows. In such a situation, we should allow the users to specify the requested permissions (i.e. executing a program) directly and let the system find out which roles are needed (i.e. administrator role) and

prompt the users to activate those roles (i.e. re-login as an administrator).

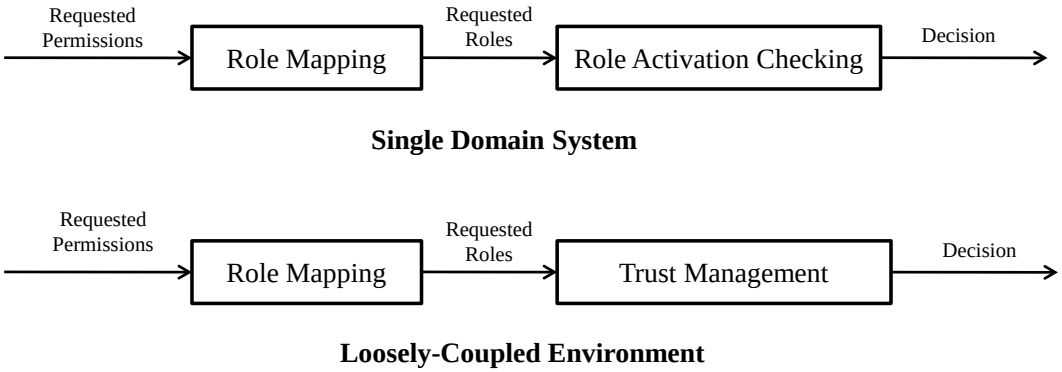


Figure 4. 2. User Authorization Query Model

Motivated by this, we propose a User Authorization Query (UAQ) model that allows the users to specify the requested permissions directly, as shown in Figure 4.2. The proposed UAQ model takes two steps to facilitate the authorization decision. In the first step, the resource providing domain runs a Role Mapping component to identify a set of roles that contains the requested permissions for the requesting user to activate. In the second step, the resource providing domain checks whether the requesting user is authorized to activate the roles returned by the Role Mapping component. If UAQ is used in a single domain RBAC system (i.e. the requesting user is from the resource providing domain), the domain needs to check the local policy to decide whether the user is authorized to activate the corresponding roles. We have proposed a UAQ model in the single domain in [35]. If UAQ is used in a *loosely-coupled*

environment, the resource providing domain does not necessarily know the identity of the requesting user. In this case, we use a trust management approach to check whether an external user is authorized to activate the corresponding roles. This is consistent with Figure 4.1 where we use Role Mapping first to get a set of requested roles then feed it into the Trust Management component.

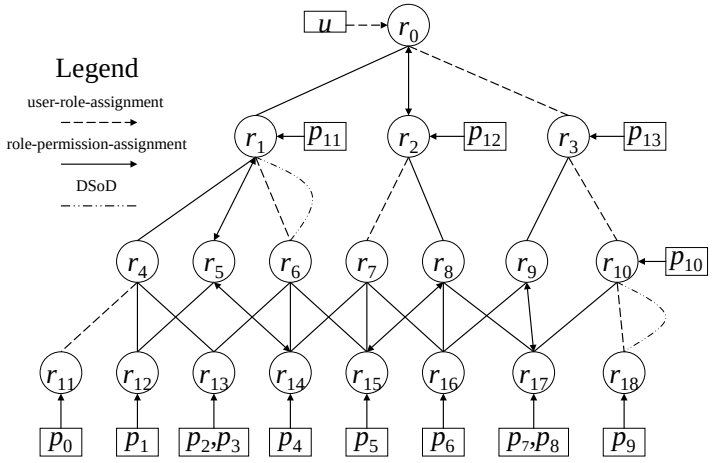


Figure 4. 3. An example RBAC policy to show the role mapping algorithms

Upon receiving the *rar*, the Role Mapping component is responsible for determining a set of roles in its local policy that contains a subset of the requested permissions. We formally define the Role Mapping problem as below:

Definition 4.2: *We define the permission set of a role* $r \in R$, $P_{au}(r)$, *and the permission set of a role set* $R_1 \subseteq R$, $P_{au}(R_1)$, *as follows:*

$P_{au}(r) = \{p \in P: p \text{ is assigned to } r_1, r \geq_i r_1\}$, $P_{au}(R_1) = \bigcup_{r \in R_1} P_{au}(r)$

where, $P_{au}(r)$ is the permission set that r can acquire, either through explicit assignments, or through the I-hierarchy. $P_{au}(R_1)$ is the union of the permission sets that can be acquired through each role in R_1.

Definition 4.3 (Role Mapping Problem): Given a request permission set P_{RQ},

1. If exactly matched role sets exist: find a minimum R_{RQ} such that $P_{au}(R_{RQ})=P_{RQ}$;
2. If no exact-matching role sets exist:

a) If *availability* is the major concern - find a minimum R_{RQ} such that $P_{au}(R_{RQ}) \supset P_{RQ}$;

b) If *least privilege* is the major concern - find a maximum R_{RQ} such that $P_{au}(R_{RQ}) \subset P_{RQ}$;

We note that if P_{RQ} is the least set of privileges, then at least P_{RQ} should be made available. In that case, we consider it as an *availability* concern. Next, we use the RBAC policy shown in Figure 4.3 to illustrate our role mapping algorithms for each of the 3 role mapping problems. And Example 4.1 shows $P_{au}(r)$ for each role in Figure 4.3 according to definition 4.2.

Example 4.1: Consider the RBAC policy in Figure 4.3, we have:

$P_{au}(r_0) = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_{11}, p_{12}\}$, $P_{au}(r_1) = \{p_1, p_2, p_3, p_4, p_{11}\}$,

$P_{au}(r_2) = \{p_5, p_6, p_7, p_8, p_{12}\}$, $P_{au}(r_3) = \{p_6, p_7, p_8, p_{13}\}$, $P_{au}(r_4) = \{p_1, p_2, p_3\}$, $P_{au}(r_5) = \{p_1, p_4\}$

$P_{au}(r_6) = \{p_2, p_3, p_4, p_5\}$, $P_{au}(r_7) = \{p_4, p_5, p_6\}$, $P_{au}(r_8) = \{p_5, p_6, p_7, p_8\}$, $P_{au}(r_9) = \{p_6, p_7, p_8\}$, $P_{au}(r_{10}) = \{p_7, p_8, p_{10}\}$, $P_{au}(r_{11}) = \{p_0\}$, $P_{au}(r_{12}) = \{p_1\}$, $P_{au}(r_{13}) = \{p_2, p_3\}$, $P_{au}(r_{14}) = \{p_4\}$,

$P_{au}(r_{15}) = \{p_5\}$, $P_{au}(r_{16}) = \{p_6\}$, $P_{au}(r_{17}) = \{p_7, p_8\}$, $P_{au}(r_{18}) = \{p_9\}$

In [36], Du *et al.* have shown that solving the role mapping problem 1 is NP-complete by reducing it to the well known *Minimal Set Cover* (MSC) problem. They also propose a greedy search algorithm to find the sub-optimal solution for the problem. However, their algorithm is based on the assumption that there always exists a role set that exactly matches the requested permission set. To accommodate inexact matches, we extend Du *et al.*'s algorithm to solve the role mapping problem 1 as shown in Figure 4.4. The time complexity of **Role-Mapping-1** is within $1 + ln\ |R|$ [36].

Role-Mapping-1(R, P_{RQ})
Input: R – a set of all roles; P_{RQ} – a set of requested permissions
Output: R_{RQ} – a set of roles, such that $P_{au}(R_{RQ})=P_{RQ}$ and $R_{RQ} \subseteq R$
1 $R_1 \leftarrow \varnothing$
2 **foreach** $r \in R$
3 **if** $P_{au}(r) \neq \varnothing$ and $P_{au}(r) \subseteq P_{RQ}$
4 $R_1 \leftarrow R_1 \cup \{r\}$
5 $R_{RQ} \leftarrow \varnothing$
6 **while** $P_{RQ} \neq \varnothing$ **do**
7 **if** $R_1 = R_{RQ}$ **return** $\varnothing$
8 Find role $v \in (R_1 \backslash R_{RQ})$ that maximize $P_{au}(v) \cap P_{RQ}$
9 $R_{RQ} \leftarrow R_{RQ} \cup \{v\}$
10 $P_{RQ} \leftarrow P_{RQ} \setminus P_{au}(v)$
11 **return** R_{RQ}

Figure 4. 4. The algorithm to solve role mapping problem 1

Example 4.2: The result of applying **Role-Mapping-1** to Figure 4.3 is shown in Table 4.1. In the last step, $P_{RQ} \neq \varnothing$, but $R_{RQ} = R_1$. The **Role-Mapping-1** terminates with $R_{RQ} = \varnothing$, which means we cannot find an exactly matched R_{RQ} for P_{RQ}.

Table 4. 1. Results of each step of **Role-Mapping-1**

Step 0	Step 1	Step 2	Step 3	Step 4 – Step 11
$R_{RQ} = \varnothing$ $R_1 = \{r_1, r_3, r_4, r_5, r_9, r_{10}, r_{12}, r_{13}, r_{14}, r_{16}, r_{17}\}$	$v = r_1$ $R_{RQ} = \{r_1\}$ $P_{RQ} = \{p_6, p_7, p_8, p_{10}, p_{12}, p_{13}\}$	$v = r_3$ $R_{RQ} = \{r_1, r_3\}$ $P_{RQ} = \{p_{10}, p_{12}\}$	$v = r_{10}$ $R_{RQ} = \{r_1, r_3, r_{10}\}$ $P_{RQ} = \{p_{12}\}$	$v = r_4, r_5, r_9, r_{12}, r_{13}, r_{14}, r_{16}, r_{17}$ $R_{RQ} = R_1$ $P_{RQ} = \{p_{12}\}$

In order to solve the role mapping problem 2(a), we modify the **Role-Mapping-1** to get **Role-Mapping-2a** that finds a minimal role set R_{RQ} whose permission set is the superset of P_{RQ}, as shown in Figure 4.5.

Here, $W(v) = |P_{au}(v)| / |P_{au}(v) \cap P_{RQ}|$ is the weight function for any role v. Note that $\forall v \in R$, $W(v) \geq 1$, and $W(v) = 1$ if $P_{au}(v) \subseteq P_{RQ}$. Therefore, this weight function favors those roles whose permission sets overlap the most with P_{RQ}. And if there are two roles v_1 and v_2 such that $W(v_1) = W(v_2)$, we select the one that covers more permissions of P_{RQ}, as shown in line 3. Note that R_{RQ} can always be found since at least R itself can satisfy the condition $P_{au}(R) \supseteq P_{RQ}$. Similar to **Role-Mapping-1**, the time complexity of **Role-Mapping-2a** is also within $1 + ln(|R|)$.

Role-Mapping-2a(R, P_{RQ})
Input: R – a set of all roles; P_{RQ} – a set of requested permissions.
Output: R_{RQ}– a set of roles, such that $P_{au}(R_{RQ}) \supset P_{RQ}$, $R_{RQ} \subseteq R$, and R_{RQ} is minimal
1 $R_{RQ} \leftarrow \varnothing$
2 **while** $P_{RQ} \neq \varnothing$ **do**
3 Find role $v \in (R \setminus R_{RQ})$ that minimize $W(v) / |P_{au}(v) \cap P_{RQ}|$
4 $R_{RQ} \leftarrow R_{RQ} \cup \{v\}$
5 $P_{RQ} \leftarrow P_{RQ} \setminus P_{au}(v)$
6 **return** R_{RQ}

Figure 4. 5. Algorithm for the role mapping problem 2(a)

Example 4.3: The result of applying **Role-Mapping-2a** to Figure 4.3 is shown in Table 4.2. In the last step, $P_{RQ} = \varnothing$, so **Role-Mapping-2a** terminates with $R_{RQ} = \{r_0, r_3, r_{10}\}$. Note that $P_{au}(R_{RQ})$ $= \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_{10}, p_{11}, p_{12}, p_{13}\} \supset P_{RQ}$ and $P_{au}(RR_Q) \setminus P_{RQ} = \{p_5\}$.

Table 4. 2. Results of each step of **Role-Mapping-2a**

Step 0	Step 1	Step 2	Step 3
$R_{RQ} = \varnothing$	$v = r_0$	$v = r_{10}$	$v = r_3$
	$W(v) = 10/9$	$W(v) = 3$	$W(v) = 4$
	$R_{RQ} = \{r_0\}$	$R_{RQ} = \{r_0, r_{10}\}$	$R_{RQ} = \{r_0, r3, r_{10}\}$
	$P_{RQ} = \{p_{10}, p_{13}\}$	$P_{RQ} = \{p_{13}\}$	$P_{RQ} = \varnothing$

The greedy algorithm to solve the role mapping problem 2b is shown in Figure 4.6. In **Role-Mapping-2b**, as we do not want to include any permission that is not in P_{RQ}, we first select $R_1 \subseteq R$ such that $P_{au}(R_1) \subseteq P_{RQ}$. Then we try to select a maximal set $R_{RQ} \subseteq R_1$ such that R_{RQ}

includes as many permissions in P_{RQ} as possible. Note that R_{RQ} can always be found because R_1 itself is a possible result. The algorithm terminates when no new permissions are added to $P_{au}(R_{RQ})$. Note that the time complexity of **Role-Mapping-2b** is also within 1+ $ln(|R|)$, which can be trivally proved since the while loop in **Role-Mapping-2b** ends in less steps compared to that of in **Role-Mapping-1**.

Role-Mapping-2b(R, P_{RQ})
Input: R – a set of all roles; P_{RQ} –a set of requested permissions.
Output: R_{RQ} – a set of roles, such that $P_{au}(R_{RQ}) \subset P_{RQ}$, $R_{RQ} \subseteq R$ and R_{RQ} is maximal
1 $R_1 \leftarrow \varnothing$
2 **foreach** r in R
3 **if** $P_{au}(r) \neq \varnothing$ and $P_{au}(r) \subseteq P_{RQ}$
4 $R_1 \leftarrow R_1 \cup \{r\}$
5 $R_{RQ} \leftarrow \varnothing$
6 $P_{old} \leftarrow \varnothing$
7 **while** $P_{RQ} \neq P_{old}$ **do**
8 **if** $R_1 = R_{RQ}$ **return** $\varnothing$
9 Find role $v \in (R_1 \setminus R_{RQ})$ that maximize $P_{au}(v) \cap P_{RQ}$
10 $P_{old} \leftarrow P_{RQ}$
11 $R_{RQ} \leftarrow R_{RQ} \cup \{v\}$
12 $P_{RQ} \leftarrow P_{RQ} \setminus P_{au}(v)$
13 **return** $R_{RQ} \setminus \{v\}$

Figure 4. 6. The algorithm to solve the role mapping problem 2(b)

Example 4.4: The result of applying **Role-Mapping-2b** to Figure 4.3 is shown in Table 4.3. In the last step, P_{old} = P_{RQ}. This means no new permissions can be added to $P_{au}(R_{RQ})$.

Role-Mapping-2b terminates with $R_{RQ}=R_{RQ} \setminus \{v\} = \{r_1, r_3\}$. Note that $P_{au}(R_{RQ}) = \{p_1, p_2, p_3, p_4, p_6, p_7, p_8, p_{11}, p_{13}\} \subset P_{RQ}$ and $P_{RQ} \setminus P_{au}(R_{RQ}) = \{p_{10}\}$.

We can see that **Role-Mapping-2a** and **Role-Mapping-2b** may not return the exactly matched role set R_{RQ} even if it exists. Therefore, we should first run **Role-Mapping-1** to try to find an exactly matched role set R_{RQ}. If **Role-Mapping-1** fails, we can then apply **Role-Mapping-2a** or **Role-Mapping-2b** based on whether *availability* or *least privilege* is more important. Here we assume the system knows the choice. For example, if the system trusts the user, the *availability* would be the main concern. Otherwise the *least privilege* may be the main concern. The issue of balancing the *availability* and *least privilege* concerns is beyond the scope of this thesis. Figure 4.7 shows the overall algorithm for the *role mapping* problem.

Table 4. 3. Results of each step of **Role-Mapping-2b**

Step 0	Step 1	Step 2	Step 3
$R_{RQ}=\varnothing$ $R_1 = \{r_1, r_3, r_4, r_5, r_9, r_{10}, r_{12}, r_{13}, r_{14}, r_{16}, r_{17}\}$ $P_{old}=\varnothing$	$v = r_1$ $R_{RQ} = \{r_1\}$ $P_{RQ} = \{p_6, p_7, p_8, p_{10}, p_{12}, p_{13}\}$ $P_{old} = \{p_1, p_2, p_3, p_4, p_6, p_7, p_8, p_{10}, p_{11}, p_{12}, p_{13}\}$	$v = r_3$ $R_{RQ} = \{r_1, r_3\}$ $P_{RQ}=\{p_{10}, p_{12}\}$ $P_{old} = \{p_6, p_7, p_8, p_{10}, p_{12}, p_{13}\}$	$v = r_4$ $R_{RQ}=\{r_1, r_3, r_4\}$ $P_{RQ} = \{p_{10}, p_{12}\}$ $P_{old} =\{p_{10}, p_{12}\}$

Example 4.5: As **Role-mapping-1** returns $\varnothing$, we call **Role-mapping-2a** if *availability* is more important, and get $R_{RQ} = \{r_0, r_3, r_{10}\}$. If *least privilege* is more important we call **Role-mapping-2b** and get $R_{RQ} = \{r_1, r_3\}$.

Role-Mapping(R, P_{RQ})
Input: R – a set of all roles; P_{RQ} –a set of requested permissions.
Output: R_{RQ} – a set of roles, and $R_{RQ} \subseteq R$
1 $R_{RQ} \leftarrow$ **Role-Mapping-1**(R, P_{RQ})
2 **if** $R_{RQ} \neq \varnothing$ **return** R_{RQ}
3 **if** *availability* is more important
4 $R_{RQ} \leftarrow$ **Role-Mapping-2**(R, P_{RQ})
5 **else if** *least privilege* is more important
6 $R_{RQ} \leftarrow$ **Role-Mapping-3**(R, P_{RQ})
7 **return** R_{RQ}

Figure 4. 7. Role-Mapping(R, P_{RQ})

In our framework, the providing domain needs to run the Role-Mapping algorithm to select a set of requested roles (R_{dest}) given the requested permissions (P_{dest}). Then, it replaces rar=<R_{req}, P_{dest}> with rar=<R_{req}, R_{dest}> and send the new request to the Trust Management component, where the authorization decision on whether or not R_{dest} can be made available to R_{req} is made. For example, when Hospital B receives an rar= <{HospitalA.HealthCareWorker}, {*add an entry to Bob's record, read Bob's record*} >, it runs **Role-Mapping** and determines that its Doctor role contains the requested permissions. Next, it replaces the requested permissions

with the requested roles and sends the new request *rar*= <{HospitalA.HealthCareWorker}, {HostpitalB.Doctor}> to the Trust Management Component. Such the new form of request indicates that Hospital B contains a proper local role Doctor for the requested permissions and the Trust Management component should decide whether or not HospitalB.Doctor can be authorized for HospitalA.HealthCareWorker. Note that in a *loosely-coupled* environment it is possible that no single domain contains all the requested permissions and multiple domains need to be involved to cover all the requested permissions.

Note that such role mapping results could be saved in the cache so that the same *rar* issued later will not require re-running the whole process. The rationale here is that we believe the local policy of each individual domain is relatively static.

4.3 THE TRUST MANAGEMENT COMPONENT

The Trust Management component is responsible for answering whether the requested roles can be authorized to the requesting roles. Although the requesting users' identities are also *available* from the *rar*, traditional identity-based access control approaches are not practical since the identities of the requesting users are usually not *known* to the providing domains in *loosely-coupled* environments. For example, the providing domain knows that a user named "Alice" is requesting some resources but doesn't really know who Alice is. In the literature, many distributed proof systems (e.g. [2, 3, 4]) have been proposed to make access control

decisions based on the properties of the entities, often encoded in *credentials* in the literature. Typically, a distributed proof system allows each domain to specify a logic-based policy to protect its own resources. Users requesting interdomain accesses need to prove the required credentials for an access right over a resource as defined by the policy of the resource-providing domain. Whether or not a user should be given a certain credential is defined by the policy of the credential issuing domain, which may require another set of credentials to be validated. As mentioned earlier, we refer to such a process as "distributed proof procedure" in this paper.

Although widely studied in the literature, we note that existing role-based distributed proof procedures are very expensive for the following two reasons: First, the distributed proof procedure typically requires proving the credentials issued by the external domain (referred to as external credentials, hereafter) which is much more expensive than proving the local credentials (i.e. the issuer and receiver of the credentials are from the same domain). For proving the access rights using local credentials, we assume that users either maintain a physical copy of the local credentials (e.g. University ID), or have direct access to the local database to obtain the local credentials very easily (e.g. log-in to the enterprise system). For proving the access rights using external credentials, if the credentials are stored at the receiver side (i.e. at the user side) proving them is as easy as proving local credentials. However, it has been shown that in many scenarios credentials are stored in the issuer side and users need to search the internet to prove that they have been issued the external credentials [26]. Moreover, such searching usually requires proving a chain of other external credentials, as shown by the DCCD algorithm [26]. Therefore,

we believe that the cost to prove a local credential is negligible or very small compared to the cost to prove an external credential in this paper.

Second, existing role-based distributed proof procedures are especially expensive when dealing with a specific interoperation request scenario that is very common in role-based multidomain environments. In role based multidomain environments, it is very common that several different users assigned to the same role (or a very small set of related roles) would request to acquire the same external resource several times within a given period. In this paper, as mentioned earlier, we refer to such interoperation requests as *role-based interoperation requests*, and refer to the role(s) that requesting users are assigned to as *requesting role*(*s*). In such a scenario, different users all request the same external resource because the functionality of the *requesting role* requires obtaining the external resource, and it is common that several users are assigned to the same role(s) (i.e. occupying the same position) in the same period. For example, assume Bob is travelling outside his city and needs to go to the emergency room in a local hospital. The assigned nurse there needs to obtain Bob's health information from his home hospital. Moreover, there might be several persons assigned to the nurse position (e.g. some during daytime, and some during night time) when taking care of Bob. They all need to acquire Bob's health information when they are on duty. From access control perspective, obviously it is not secure to allow the first nurse who has obtained Bob's health information to disclose it to the subsequent nurses. A more secure way is to require each nurse issuing a separate request so that each request is evaluated and authorized separately for each nurse. Here, we note the *role-based*

interoperation request scenario: different persons assigned to the same role (nurse) need to request the same external resource (Bob's health information) several times (when each person is taking the position) within a period (the time period when Bob is taken care of).

Based on the discussion above, if we could find a way to authorize the requested resource to the requesting role(s) directly, then all the requesting users need to do is to prove that they are assigned to the requesting roles - which can be expected to be much less expensive since requesting roles are local to the requesting users. Moreover, since all the requests are issued from the same requesting role(s), such an authorization needs to be done only once during the interoperation period. As a result, authorizing the requested resource directly to the requesting role(s) would remove both of the two causes of the expense in the existing distributed proof procedure. Motivated by this, we propose a formal framework for simplifying the distributed proof procedures for role-based interoperation requests. We assume that each domain uses RT_0 [22] to specify its trust management policy on how external users can prove the requested roles. We use RT_0 language as its semantics has shown to be easily captured by translation to negation-free Datalog rules which guarantees that the semantics is precise, monotonic and algorithmic [22]. The purpose of this thesis is not to introduce a new language with different expressivity but to build on the RT_0 framework, which has been well recognized as a framework that combines the strength of role and attribute based access control and trust management – which are important for secure interoperation. In this work, we do not deal with uncertainties and probability based access semantics related to cross domain accesses; hence, approaches such as

based on Bayesian techniques are not applicable here. Our approach is based on analyzing the similarities between the RT_0 policies defining the requesting roles (i.e. R_{req}) and the RT_0 policies defining the requested roles (i.e. R_{dest}). If any user that can prove R_{req} is guaranteed to be able to prove R_{dest}, then it is safe to authorize R_{dest} directly to R_{req}, i.e. allowing users of R_{req} to acquire permissions of R_{dest}. In this case, we say R_{req} is the *simplified proof* for users in d_{req} to acquire R_{dest}, compared to the expensive distributed proof procedures employed in existing approaches.

We first review the RT_0 language before we present our simplification framework. RT_0 uses 4 types of rules to define the membership of a role:

- *Type* 1 (*Simple Membership*): $A.r \leftarrow D$
- *Type* 2 (*Simple Containment*): $A.r \leftarrow B.r_1$
- *Type* 3 (*Linked Roles*): $A.r \leftarrow A.r_1.r_2$
- *Type* 4 (*Role Intersections*): $A.r \leftarrow B_1.R_1 \cap B_2.R_2 \cap \ldots \cap B_n.R_n$, where each $B_i.R_i$ can be defined by any of the above 4 types.

If the body of type 1 rule (i.e. D) is a user identifier, it is a special rule specifying that a credential has been directly issued to a user. We note that it is beneficial to distinguish it from other types of rules in the context of our framework. For example, "*UPMC.MD*←*Alice*" specifies that *UPMC* has issued *Alice* a credential certifying that *Alice* has a *MD* degree from there. If we consider it as part of the RT_0 policy, the size of the policy would increase significantly since such credentials could be issued to a huge number of users (e.g. all *MD* students who have graduated from *UPMC*). For this reason, we only consider other RT_0 rules (i.e.

rule type 1 where its body is a domain identifier, rule type 2, rule type 3, and rule type 4) as part of the policy, as defined below:

Definition 4.4 (Trust Management Policy of a Domain): *Given a domain A, we define its trust management policy, Pol*(*A*), *to be the set of* RT_0 *rules whose head role is defined by A and does not have a type 1 rule that has a user identifier in its body.*

We then define the local role set and external role set of a certain domain as follows:

Definition 4.5 (Local and External Roles): *Given a domain A, we define the set of its local roles, denoted as LocalRoles*(*A*), *to be the roles appearing in the head of at least one rule in Pol*(*A*); *we define the set of external roles of A, denoted as ExternalRoles*(*A*), *to be the roles with the domain identifier other than A.*

Note that we do not consider *A*'s roles that have no rule defining them to be *A*'s local roles. Such roles are usually internal roles that no external users can assume. Since our focus is simplifying the proof of the roles for external users, we do not consider such internal roles in our work. We require each of *A*'s local roles to be defined by at least one rule in *Pol*(*A*). We further assume that any user involved in the multidomain environment belongs to some domain. In this way, if *u* belongs to domain *A* and *A.r* also belongs to *LocalRoles*(*A*), we say *A.r* is a local role of *u*.

4.3.1 A Motivational Example

The following example illustrates how we can simplify the distributed proof procedure by analyzing the similarities of the RT_0 policies of the requesting roles and the requested roles.

Example 4.6:

Assume Bob has registered his health information in his home hospital (referred to as *HH* hereafter), and assume Bob is travelling outside and has to go to the emergency room in a local hospital (referred to as *LH* hereafter). To take care of Bob, the healthcare workers in *LH* (grouped by *HealthCare* role) need to access Bob's health information stored in *HH*. In *HH*, the primary doctor of Bob (grouped by *Doctor*(*patient=Bob*) role, and denoted as *Doctor* for short) has the permissions to access Bob's health information. From role based perspective, such interoperation need can be described as "the users assigned to *HealthCare* in *LH* requests to assume *Doctor* in *HH*".

HH allows outside users to assume its *Doctor* role for emergency needs by specifying the following policy using RT_0:

HH: *HH.Doctor* ← *HH.MD* rule 1

HH.MD ← *HH.MedicalSchool.MD* rule 2

HH.MedicalSchool ← *ABU.Accredited* rule 3

The policy specifies that users having Medical Doctor (*MD*) degree accepted by *HH* can assume *Doctor* in *HH* (rule 1). Furthermore, *HH* accepts *MD* degree issued by medical schools accepted by *HH* (rule 2). Finally, *HH* accepts all medical schools accredited by Accrediting Board for Universities (*ABU*) (rule 3).

Assume that UPMC is a medical school accredited by *ABU*, and one user of *LH.HealthCare*, Alice, has a *MD* degree from UPMC:

ABU: *ABU.Accredited* ← *UPMC* rule 4

UPMC: *UPMC.MD* ← *Alice* rule 5

It is easy to verify that Alice can prove *HH.Doctor* from the policies above. However, even in this very simple example constructing such proof for Alice involves the discovery of 5 rules from the policies of 3 domains. Moreover, other users assigned to *LH.HealthCare* needs to prove *HH.Doctor* separately when they are taking care of Bob.

Could such expensive distributed proof procedure be simplified for Alice, and all other users assigned to *LH.Healthcare*? If we examine the policy in *HH*, we can conclude that any user that has a *MD* degree issued by the medical school accredited by *ABU* can assume *HH.Doctor*:

HH: "*HH.Doctor* ← *ABU.Accredited.MD*" derived rule 1

Assume the policy in *LH* is defined as follows:

LH: *LH.HealthCare* ← *LH.MD* ∩ *LH.Licensed* rule 6

LH.MD ← *LH.MedicalSchool.MD* rule 7

LH.MedicalSchool ← *ABU.Accredited* rule 8

LH.Licensed ← *NMLS.Licensed* rule 9

The policy specifies that users having *MD* degree and Medical license accepted by *LH* can assume *LH.HealthCare* (rule 6). Similar to *HH*, *LH* also accepts the *MD* degree issued by medical schools accepted by *LH* (rule 7), and accepts all medical schools accredited by *ABU* (rule 8). In addition, *LH* accepts medical license issued by National Medical License Service,

NMLS (rule 9). From these rules, we can conclude that any user that has a *MD* degree issued by the medical school accredited by *ABU* and is licensed by *NMLS* can assume *LH.HealthCare*, i.e.

LH: "*LH.HealthCare* ←*ABU.Accredited.MD* ∩ *NMLS.Licensed* " derived rule 2

Comparing derived rule 1 and derived rule 2, we find some similarities. In particular, policy of *LH.HealthCare* is more restrictive than policy of *HH.Doctor*, implying that "any user who can prove *LH.HealthCare* can also prove *HH.Doctor*". Based on this, it is safe for *HH* to add a new rule in its policy specifying that any user who can prove *LH.HealthCare* can assume *HH.Doctor*:

HH: HH.Doctor ← *LH.HealthCare* new (simplified) rule

For users in *LH*, proving the membership of *LH.HealthCare* is much easier than proving *ABU.Accredited.MD*. This is because *LH.HealthCare* is a local role and is directly assigned to users in *LH*, while *ABU.Accredited.MD* is an external role and users in *LH* need to examine policy of *ABU* to find out whether or not their *MD* degree is issued by medical schools accredited by *ABU*. Moreover, such authorization (from *HH.Doctor* to *LH.HealthCare*) needs to be done only once. All subsequent requests from users assigned to *LH.HealthCare* need only to prove *LH.HealthCare* in order to assume *HH.Doctor*.In other words, the distributed proof procedure of *HH.Doctor* is simplified for users in *LH* by proving *LH.HealthCare*.

4.3.2 The Simplify() Algorithm

The goal of our work is to automatically find such simplified proofs given the policies related to the requested roles (e.g. *HH.Doctor*) and the requesting roles (e.g. *LH.Healthcare*). Example 4.6 illustrates that if we can represent the corresponding policies using credential sets (e.g. bodies of derived rule 1 and derived rule 2), we are able to find the existing simplified proof by analyzing the similarity of the two credential sets.

In order to find the derived rule as in Example 4.6, we can perform the following derivations over the raw RT_0 rules according to its semantic: two rules having the same head means that either body can be used to prove the head, thus the two bodies can be combined through OR relation. For example, $A.r \leftarrow expr_1$ and $A.r \leftarrow expr_2$ can be combined to $A.r \leftarrow expr_1$ OR $expr_2$. Similarly, the body of type 4 rule is actually the AND relation among simple roles or linked roles. Furthermore, any simple role $A.r$ in the body of a rule can be replaced by the body of another rule (say $rule_1$) if $rule_1$'s head is equal to $A.r$; any linked role $A.r_1.r_2$ in the body of a rule can be replaced by the body of another rule (say, $rule_1$) if $rule_1$'s head is equal to $A.r_1$. In this way, given a role $A.r$, we can start with the rules whose head is $A.r$ and replace the roles in the body using all the rules in A until no replacement can be made. The resulting expression is a propositional logic expression containing AND and OR relations and all its literals are roles that cannot be further replaced using *Pol*(*A*). *Such logic expression describes how to construct the proof of role A.R from A's perspective.*

Example 4.7:

Consider the policy of *LH* (rule 6 through rule 9) in Example 4.6. Assume we want to translate rules defining *LH.HealthCare* (which is rule 6 in this case). It is easy to see that rule 7 and rule 8 can be applied to replace the body of rule 6. The resulting expression is

LH.HealthCare←*LH.MedicalSchool.MD* AND *NMLS.Licensed*.

LH.MedicalSchool.MD can be further replaced by rule 9. Finally, we get

LH.HealthCare ← *ABU.Accredited.MD* AND *NMLS.Licensed*

No roles in its body can be further replaced by rules in *Pol*(*LH*). From *LH*'s perspective, the above expression indicates that *LH.HealthCare* can be proved by proving both *ABU.Accredited.MD* and *NMLS.Licensed*.

Note that in Example 4.7 "*ABU.Accredited.MD*" and "*NMLS. Licensed*" may be further replaced using policies in *ABU* and *NMLS* (e.g. *ABU.Accredited.MD* can be replaced using rule 4 in *ABU*). However, we do such derivation of rules using only the local policy for the following reasons: (*i*), it is not efficient to check the policies in other external domains since it could involve searching the entire network; (*ii*), there are no privacy violations in examining the local policy only; and (*iii*) Since only the local rules are involved, any changes in external domains will not affect the proof of the target role and there is no need to worry about policy changes in external domains which is expensive to detect. Next, we formally define a projection function $\prod$ to derive the proof of a role in the way described above.

Definition 4.6 (Projection Function ∏):

A projection function ∏ *takes A.r and Pol*(*A*) *as inputs, and outputs a proposition logic expression* ∏(*A.r, Pol*(*A*)) *such that:*

- *It contains literals, AND, OR and parenthesis only*
- *Each literal in the expression is a role appearing in Pol*(*A*).

And the logic expression is generated in the following way:

1. Combine all the bodies of rules whose head is A.r by OR relation with each body enclosed within parenthesis.

2. In the resulting logic expression, replace any role if it appears in the heads of other rules (for linked role $A.r_1.r_2$*, if* $A.r_1$ *appears in the heads of other rules) with the bodies of those rules connected by OR relation. Put a pair of parenthesis outside each replaced role.*

3. *Rewrite "∩" using AND in the resulting expression. Put a pair of parenthesis outside each element connected by "∩".*

Lemma 4.1:

∏(*A.r, Pol*(*A*)) *represents all the possible combinations of minimal roles* (*appear in Pol*(*A*) *and cannot be further replaced by rules in Pol*(*A*)) *that can prove A.r*

It is easy to prove Lemma 4.1 from the semantics of RT_0 and the construction steps of ∏ in Definition 4.6. Note that any propositional logic expression can be translated into Disjunctive Normal Form (*DNF*). We use $\prod_{DNF}$(*A.r, Pol*(*A*)) to denote the *DNF* representation of ∏(*A.r, Pol*(*A*)), and we define the proof of a role as follows:

Definition 4.7 (Proof of a Role):

Given a role A.r, Pol(*A*), *and* $\prod_{DNF}$(*A.r, Pol*(*A*)), *we define the proof of A.r, denoted as Proof*(*A.r*), *as follows: Proof*(*A.r*) = {R_i | *where* c_i *is a conjunction part in* $\prod_{DNF}$(*A.r, Pol*(*A*)) *and* R_i *is the set of roles in* c_i}

Proof(*A.r*) is a set of role sets. For any element *e* in *Proof*(*A.r*), we say a user proves *e* if she proves all the roles in *e*, we have the following theorem:

Theorem 4.1: *A user proves A.r if she proves at least one element in Proof*(*A.r*).

Proof:

According to Lemma 4.1, $\prod$(*A.r, Pol*(*A*)) represents all possible combinations of minimal roles that can prove *A.r*. As $\prod_{DNF}$(*A.r*, *Pol*(*A*)) is the logic translation of ($\prod$ *A.r*, *Pol*(*A*)). it also represents all possible combinations of minimal roles that can prove *A.r*. According to the logic semantic of a *DNF*, the whole *DNF* expression is evaluated to be *true* if at least one conjunction part is evaluated to be *true*. Recall that each *e* in *Proof*(*A.r*) is the collection of roles in one conjunction part of $\prod_{DNF}$(*A.r*, *Pol*(*A*)). Thus, proving at least one element in *Proof*(*A.r*) is enough to prove *A.r*. ■

Recall that each element in *Proof*(*A.r*) is a set of roles. Theorem 4.1 says that the user can prove ANY element in *Proof*(*A.r*) to prove *A.r* since the elements are connected by OR relation according to definition 4.6; and to prove an element a user needs to prove ALL the roles in it since they are connected by AND relation according to definition 4.6.

```
BuildAOT(Node node, Policy p)
Input: node: representing the role whose proof is to be built; p: the policy of that domain.
Output: an AND-OR tree starting from the root
1    if (node is a user identity U (i.e. type 1)) continue;
2    if (node is a simple role A.r (i.e. type 2))
3        node.type = OR;
4        foreach rule r in p defining A.r
5            add r.body to the children list of node;
6            BuildAOT(r.body, p);
7    if (node is a linked role A.r1.r2 (i.e. type 3))
8        node.type = OR;
9        foreach type 1 or type 2 rule r defining A.r1
10           add r.body.r2 to the children list of node;
11           BuildAOT(r.body, p);
12   if (node is an intersection role (i.e. type 4))
13       node.type = AND;
14       foreach role expression expr in the intersection role
15           add expr to the children list of node;
16           BuildAOT(expr, p);
Proof BuildProof(AOT root)
```

Figure 4. 8. **BuildAOT algorithm**

For a given role *A.r* and Policy(*A*), we build *Proof*(*A.r*) in two steps: (1) we build an and-or-tree such that each node is a role expression *expr* in the RT_0 policy and *expr* has children if and only if it can be derived according to definition 4.6. The algorithm to build such and-or-tree for a given role expression, **BuildAOT**(), is shown in Figure 4.8. **BuildAOT**() is a recursive function and should be called with the target role as the initial input; (2) we construct *Proof*(*A.r*) using the proof tree we have built according to definition 4.7. The corresponding

algorithm, **BuildProof**() is shown in Figure 4.9. **BuildProof**() is also a recursive function and should be called with the root of the proof tree as the initial input.

```
Input: root: the root of the corresponding proof tree
Output: the proof of the root node of the corresponding proof tree
1    if (root.childrenNo==0) return {{root}};
2    Proof result = {{}};
3    foreach child c in root's children list
4        Proof prf = BuildProof(c);
5        if (root.nodeType==OR)
6            result = Union(result, prf);
7        else if (root.nodeType==AND)
8            result = Combine(result, prf);
9    return result;
Note: Union(Proof p1, Proof p2) = {p1 ∪ p2}
      Combine(Proof p1, Proof p2) = {e1 ∪ e2 | e1∈p1, e2∈p2}
```

Figure 4. 9. the algorithm to build the proof for a single role

Now, we are ready to generalize the proof of a single role to the proof of a set of roles. For a role set *A.R*, we compute $\prod_{DNF}(A.r_i, Pol(A))$ for every r_i in *A.R*. We then connect their outputs by AND relation and translate the resulting logical expression (no longer in *DNF*) into *DNF*, denoted as $\prod_{DNF}(A.R, Pol(A))$.

Definition 4.8 (Proof of a Set of Roles): *Given a role set A.R, Pol*(*A*), *and* $\prod_{DNF}(A.R, Pol(A))$, *we define the proof of A.R, denoted as Proof*(*A.R*), *as follows: Proof*(*A.R*) = {R_i | *where* c_i *is a conjunction part in* $\prod_{DNF}(A.R, Pol(A))$ *and* R_i *is the set of roles* in c_i}

Proof(*A.R*) is also a set of role sets. For a target role set *A.R*, we say a user proves *A.R* if she proves all the roles in it, we have:

Theorem 4.2:

A user proves A.R if he proves at least one element in Proof(*A.R*).

Proof:

Proof is similar to the proof of theorem 4.1. ■

```
Proof BuildProof(Proof [] prfs)
Input: prfs: proofs of each single role
Output: proof of the role set
1    Proof result = {{}};
2    foreach proof prf in prfs
3        result=Combine(result, prf);
4    return result;
Note: Combine(Proof p1, Proof p2) = {e1 ∪ e2 | e1∈p1, e2∈p2}
```

Figure 4. 10. the algorithm to build the proof for a set of roles

Figure 4.10 shows the algorithm to build the proof of a set of roles from the proofs of each single role. This is simply a combination of the proofs of each single role according to definition 4.8. Note that in practice, the proof of a single role can always be constructed together with the domain's RT_0 policy. That is, once the corresponding RT_0 policy is created, the proof of each role protected by the policy can be calculated and saved for the future use. Therefore, we only need to

analyze the time complexity of calculating the proof of a set of roles, which is given by the following theorem:

Theorem 4.3:*Consider domain A, Pol*(*A*)*, and A.R. Let Proof*(*A.r*) *be proof of A.r in A.R, then the complexity of calculating Proof*(*A.R*) (*A.R*={$A.r_1$, …, $A.r_m$}) *is given by* $O(TN_{rule}^{|A.R|})$*, where* TN_{rule} *is the total number of rules defining roles in A.R*

Proof:

Proof(*A.R*) can be computed by the algorithm shown in Figure 4.10. The complexity of getting all the proofs for each single role is O(1) if *Proof*(*A.r*) is given for any $A.r \in A.R$. The complexity of combining all the single proofs is $|Proof(A.r_1)| \times |Proof(A.r_2)| \times \ldots \times |Proof(A.r_m)|$ according to the definition of **Combine**(). Since different elements in *Proof*(*A.r*) for any role *A.r* is connected by OR relation, the number of elements in *Proof*(*A.r*) is the number of OR relations plus 1. According to definition 4.6, the number of the OR relations connecting *Proof*(*A.r*) is the number of rules whose head is *A.r*. Therefore, the complexity of those combinations are $O(N_{rule}^{|A.R|})$ where N_{rule} is the average number of rules defining a single role $A.r \in A.R$. Hence, the overall complexity is given by $O(N_{rule}^{|A.R|})$. ■

Next, we need to find out whether the proof of the requesting roles is more restrictive than the proof of the requested roles as indicated in Example 4.6. Towards this, we define two important notions: *proof-dominate* and *partial-proof-dominate* relations as the foundation of our simplification approach. As discussed before, if the proof of r_{req} is more restrictive than the proof of R_{dest}, then proving r_{req} is enough to prove R_{dest}. But, we need to compare the "restrictiveness"

of the proofs of two roles. Intuitively, the proof of a role r_1 is more restrictive than the proof of another role r_2 if any combination of the roles that can prove r_1 is guaranteed to be able to prove r_2. In this case, we say "r_1 *proof-dominates* r_2" and define it formally as below:

Definition 4.9 (Proof-Dominates relation between Two Roles): *Given a pair of roles r_1 and r_2, we say r_1 proof-dominates r_2, written as $r_1 \gtrsim r_2$, iff.*

$$\forall e_1 \in Proof(r_1),\ \exists e_2 \in Proof(r_2), \text{ such that } e_1 \supseteq e_2$$

The following theorem shows the rationale of definition 4.9.

Theorem 4.4: $r_1 \gtrsim r_2 \rightarrow$ *any user who can prove r_1 can also prove r_2*

Proof:

According to theorem 4.1, any user who can prove r_1 must prove at least one element in $Proof(r_1)$. Without loss of generality, we assume an arbitrary user u proves an arbitrary element $e_1 \in Proof(r_1)$. Since $r_1 \geqslant r_2$, we have $\exists e_2 \in Proof(r_2)$, such that $e_1 \supseteq e_2$. And $e_1 \supseteq e_2$ indicates that u can also prove e_2. According to Theorem 4.1, u can prove r_2. ■

Definition 4.10 (Proof-Dominate between Two Role Sets): *Given a pair of role sets R_1 and R_2, we say R_1 proof-dominates R_2, written as $R_1 \gtrsim R_2$, iff.*

$$\forall e_1 \in Proof(R_1),\ \exists e_2 \in Proof(R_2), \text{ such that } e_1 \supseteq e_2$$

Theorem 4.5: $R_1 \gtrsim R_2 \rightarrow$ *any user who can prove R_1 can also prove R_2*

Proof: The proof is similar to the proof of Theorem 4.4. ■

For any target role set *A.R*, if we can find some role set *B.R* such that $B.R \gtrsim A.R$, *B.R* can be an alternative proof of *A.R*. Moreover, since *B.R* is the local role for users of *B* it is also a simplified

proof of *A.R*. However, in many cases we may not be able to find such a *B.R* that *proof-dominates A.R*. Nevertheless, it is still possible to simplify *A.R* if there exists *B.R* that *partial-proof-dominates A.R*, as shown below:

Example 4.8

Following Example 4.6, now we assume *HH* modifies its policy and requires that "any user having *MD* degree accepted by *HH* and is a member of American Medical Association (*AMA*) can assume its *Doctor* role, as shown below:

HH: $HH.Doctor \leftarrow HH.MD \cap AMA.Member$

In this case, *Proof*(*HH.Doctor*) becomes:

{{*ABU.Accredited.MD*, *AMA.Member*}}

According to definition 4.9, *LH.HealthCare* no longer *proof-dominates HH.Doctor* since there is no element of *Proof*(*HH. Doctor*) that is the subset of *Proof*(*LH.HealthCare*). However, any user of *LH.HealthCare* (who is able to prove *ABU.Accredited. MD*) is guaranteed to be able to prove *HH.Doctor* if the user can also prove *AMA.Member*. Therefore, an alternative way to prove *HH.Doctor* would be {{*LH.HealthCare*, *AMA.Member*}}. For users in *LH*, they only need to prove one external role (i.e. *AMA.Member*) in the new proof, while they need to prove two external roles (i.e. *ABU.Accredited.MD*, *AMA.Member*) in the original proof. In other words, such an alternative proof is a simplified proof for users in *LH*.

Motivated by this example, we define *partial-proof-dominate* relation between two roles as follows:

Definition 4.11 (Partial-Proof-Dominate between Two Roles): *Given a pair of roles r_1 and r_2, and a non-empty role set AUX, we say r_1 partial-proof-dominates r_2 with an auxiliary role set AUX, written as $r_1 \gtrsim_{AUX} r_2$, iff.*

$\forall e_1 \in Proof(R_1),\ \exists e_2 \in Proof(R_2)$, *such that* $e_1 \cup AUX \supseteq e_2$

Based on this, we have the following theorem:

Theorem 4.6: $r_1 \gtrsim_{AUX} r_2 \rightarrow$ *any user who can prove r_1 and AUX can also prove r_2*

Proof: According to theorem 4.1, any user who can prove r_1 must prove at least one element in $Proof(r_1)$. Without loss of generality, we assume an arbitrary user *u* proves an arbitrary element $e_1 \in Proof(r_1)$. Since $r_1 \gtrsim_{AUX} r_2$, we have $\exists e_2 \in Proof(r_2)$, such that $e_1 \cup AR \supseteq e_2$. Since the user can prove e_1 and *AUX*, she can prove e_2 also. In other words, *u* can prove an element of $Proof(r_2)$. According to theorem 4.1, *u* can prove r_2. ■

Definition 4.12 (Partial-Proof-Dominate between Two Role Sets): *Given a pair of roles R_1 and R_2, given a role set AUX, we say R_1 partial-proof-dominates R_2 with an auxiliary role set AUX, written as $R_1 \gtrsim_{AUX} R_2$, iff.*

$\forall e_1 \in Proof(R_1),\ \exists e_2 \in Proof(R_2)$, *such that* $e_1 \cup AUX \supseteq e_2$

Theorem 4.7: $R_1 \gtrsim_{AUX} R_2 \rightarrow$ *any user who can prove R_1 and AUX can also prove R_2*

Proof: The proof is similar to the proof of Theorem 4.6. ■

For any target role set *A.R*, if we can find some role set *B.R* such that *B.R* $\gtrsim$ *A.R*, *B.R* can be the simplified proof of *A.R*. If we are not able to find such role set but are able to find some *B.R* such that *B.R* $\gtrsim_{AUX}$ *A.R*, then *B.R* together with *AUX* can also be an alternative proof

according to theorem 4.7. Whether *B.R* together with *AUX* is a simplified proof is determined by the number of *B*'s external roles in it compared with the number of *B*'s external roles in the original proof. We define $N_D(R)$ as the number of *D*'s local roles in a role set *R*. The simplified proof of *A.R* for users in *B* is defined as below:

Definition 4.13 (Simplified Proof of *A.R* using *B.R*):

Given Proof(*A.R*) *such that* $\forall e \in Proof(A.R)$ *we have* $e \not\subseteq B.R$, *the simplified proof of A.R using B.R, denoted by SimplifiedProof*(*A.R, B.R*), *is given by* (*B.LR, AUX*) *where*:

(1) $\varnothing \neq B.LR \subseteq B.R$, *and* $B.LR \gtrsim_{AUX} A.R$

(2) *if* $AUX \neq \varnothing, \forall e \in Proof(A.R)$, $(|AUX| - N_B(AUX)) < (|e| - N_B(e))$

(3) $\forall r \in B.LR$, $(B.LR \setminus \{r\}, AUX)$ *does not satisfy condition (1) and (2)*

(4) $\forall r \in AUX$, $(B.LR, AUX \setminus \{r\})$ *does not satisfy condition (1) and (2).*

Given *Proof*(*A.R*) to be simplified, we require that none of its element is the subset of *B.R*. In other words, any possible proof (before simplification) of *A.R* must include at least one external roles of *B*. Otherwise, there is no benefits of simplifying the proof of *A.R* for users of *B*, since they can already prove *A.R* by proving local roles only before simplification. Given *Proof*(*A.R*), its simplified proof consists of two role sets, *B.LR* and *AUX*, that satisfy condition (1) through (4) respectively. Condition (1) ensures that we have a non-empty subset of *B.LR* that *proof-dominates A.R* (if $AUX = \varnothing$) or *partial-proof-dominates A.R* (if $AUX \neq \varnothing$). According to theorem 4.7, condition (1) actually ensures that users who can prove *SimplifiedProof*(*A.R, B.R*) can also prove *Proof*(*A.R*). In other words, *SimplifiedProof*(*A.R, B.R*) is an alternative way to

prove *A.R* if condition (1) holds. However, our goal is to simplify the distributed proof process for *A.R*. Therefore, we also need to make sure *SimplifiedProof*(*A.R*, *B.R*) is "simpler" than any element in *Proof*(*A.R*) (recall that any element in *Proof*(*A.R*) is one possible minimal role set that can prove *A.R*), and this is ensured by condition (2). The underlying assumption and the motivation of our approach is that proving a local role is much cheaper than proving an external role. As a result, given two sets of roles that can prove *A.R*, the one having less external roles is simpler than the one having more external roles. Therefore, condition (2) requires that the number of *B*'s external roles in *AUX* is smaller than the number of *B*'s external roles in any element of *Proof*(*A.R*). Condition (3) ensures that *B.LR* is the minimal role set that contributes to a simplified proof. If (*B.LR* \ {*r*}, *AUX*) satisfies (1) and (2), any user who can prove *B.LR* and *AUX* can also prove *B.LR* \ {*r*} and *AUX*, thus proving *A.R* in a "more simplified" way. In this case, it makes no sense to include *B.LR* in the simplified proof and we should include *B.LR* \ {*r*} as a simplified proof. Similarly, condition (4) ensures that *AUX* is the minimal role set that contributes to a simplified proof.

Given two sets of role sets RS_1 and RS_2, we define: (1) **Overlap**(RS_1, RS_2) = *true* iff. "$\exists e_1 \in RS_1,\ \exists e_2 \in RS_2,\ e_1 \cap e_2 \neq \varnothing$"; and (2) $RS_1 \cup^* RS_2 = \{e_1 \cup e_2 \mid \forall e_1 \in ES_1,\ \forall e_2 \in ES_2\}$. We present our proposed **Simplify**() algorithm in Figure 4.11. **Simplify()** constructs the simplified proof in three steps:

Algorithm: **Simplify**(*A.R*, Proof(*A.R*), *B.R*, *Proof*(*B.r*) *for all B.r*$\in$*B.R*)
Input: *A.R*: target role set, Proof(*A.R*): proof of *A.R* to be simplified
B.R: role set in *B* that is used to simplify *A.R*
Proof(*B.r*) for all *B.r*$\in$*B.R*: the corresponding proofs for all roles in *B.R*
Output: *SP*: A set of simplified proofs for *A.R* given *B.R*
/* *filter out roles not contributing to simplified proof* */
1 $T=\varnothing$; //*T stores candidate roles in B.R*
2 **foreach** role *B.r* **in** *B.R*
3 **if** (**Overlap**(*Proof*(*B.r*), *Proof*(*A.R*))==*true*) $T = T \cup \{B.r\}$;
4 **if** ($T==\varnothing$) **return** $\varnothing$; //*no simplified proof can be found*
/* *for each subset of roles in T, check whether it proof dominates or partial-proof-dominates A.R* */
5 **foreach** subset of roles $S \subseteq T$
6 *AUX*(*S*)=$\varnothing$; //a set of role sets
7 **foreach** e_i **in** *Proof*(*S*)
8 $AUX(e_i)=\varnothing$; //a set of role sets
9 **foreach** e_j in *Proof*(*A.R*)
10 **if** ($e_i \cap e_j \neq \varnothing$) $AUX(e_i) = AUX(e_i) \cup \{e_j \setminus e_i\}$
11 $AUX(S)=AUX(e_1) \cup *AUX(e_2) \cup *\ldots \cup *AUX(e_n)$, where $Proof(S)=\{e_1, e_2, \ldots, e_n\}$
/* *checks condition (2), (3), (4) in definition 4.13.* */
12 **foreach** *aux* **in** *AUX*(*S*)
13 **if** ($aux == \varnothing$) $SP=SP \cup \{(S, \varnothing)\}$;
14 **else if** ($\forall e_2 \in Proof(A.R)$, $|aux|-N_B(aux)<|e_2|-N_B(e_2)$) $SP=SP \cup \{(S, aux)\}$;
15 **foreach** sp_i *in SP*
16 **foreach** sp_j *in SP*
17 **if** ($sp_i.B.LR \supset sp_j.B.LR$ && $sp_i.AUX == Sp_j.AUX$) *remove* sp_i *from SP*;
18 **if** ($sp_i.AUX \supset sp_j.AUX$ && $sp_i.B.LR == sp_j.B.LR$) *remove* sp_i *from SP*;
19 **return** *SP*;

Figure 4. 11. Simplify() Algorithm

Step 1 (line 1-4): Selecting the candidate local roles of *B.R* to simplify *Proof*(*A.R*).

Here, we only consider those roles in *B.R* such that at least one role appearing in its proof also appears in *Proof*(*A.R*) (line 3). The rationale is given by the following theorem:

Theorem 4.8:

$\forall$*SimplifiedProof*(*A.R, B.R*)=(*B.LR, AUX*), $\forall r \in$*B.LR we have*:

Overlap(*Proof*(*r*), *Proof*(*A.R*)) = *true*

Proof:

We prove by contradiction and assume "**Overlap**(*Proof*(*r*), *Proof*(*A.R*)) = *false*", we then prove (*B.LR* \ {*r*}), *AR*) satisfies condition (1) and (2) in definition 4.13. This contradicts with (*B.LR, AUX*) is a simplified proof. To prove (*B.LR*\{*r*}), *AR*) satisfies condition (1) and (2), we need to prove the following:

(1) $B.LR \setminus \{r\} \neq \varnothing$

Sub-proof: Since *B.LR* is not empty it is equivalent to proving $B.LR \neq \{r\}$. In other words, prove ({*r*}, *AUX*) is not a simplified proof. We prove by contradiction and assume ({*r*}, *AUX*) is a simplified proof. According to definition 4.13, we have, $\{r\} \geq_{AUX} A.R$. According to definition 4.12, we have "$\forall e_1 \in Proof(r)$, $\exists e_2 \in Proof(A.R)$ such that $e_1 \cup AUX \supseteq e_2$". If $AUX=\varnothing$, we have $e_1 \supseteq e_2$ and then $e_1 \cap e_2 \neq \varnothing$. This contradicts the assumption that "**Overlap**(*Proof*(*r*), *Proof*(*A.R*)) = *false*". Otherwise if *AUX* is not empty, since *AUX* is the auxiliary roles in a simplified proof (*B.LR*, *AR*), the number of *B*'s external roles in *AUX* is less than the number of *B*'s external roles in e_2 according to condition (2) in definition 4.13. Without loss of generality, we assume *r** is *B*'s external roles that belongs to e_2 but does not belong to *AUX*. As a result, *r** must belong to e_1 to ensure $e_1 \cup AUX \supseteq e_2$. In other words, we have $e_1 \cap e_2 = \{r^*\} \neq \varnothing$. This contradicts with the assumption that **Overlap** (*Proof*(*r*), *Proof*(*A.R*)) = *false*

(2) $B.LR \setminus \{r\} \subseteq B.R$

Sub-proof: this is trivial since $B.LR \subseteq B.R$

(3) $B.LR \setminus \{r\} \geq_{AUX} A.R$

Sub-proof: Since (*B.LR*, *AUX*) is a simplified proof, we have $B.LR \geq_{AUX} A.R$. And we have "$\forall e_1 \in Proof(B.LR)$, $\exists e_2 \in Proof(A.R)$ such that $e_1 \cup AUX \supseteq e_2$". $\forall e_1' \in Proof(B.LR \setminus \{r\})$, it must be the subset of some element $e_1 \in Proof(B.LR)$ (according to the construction of the proof of a set of roles given by definition 4.8), i.e. $e_1 \supseteq e_1'$. Moreover, since $e_1 \setminus e_1'$ are roles appearing in *Proof*(*r*) it does not contain any role in e_2 (by **Overlap**(*Proof*(*r*), *Proof*(*A.R*)) = false). We conclude that $e_1' \cup AR$ is also the superset of e_2. In other words, we have $\forall e_1' \in Proof(B.LR \setminus \{r\})$, $\exists e_2 \in Proof(A.R)$ such that $e_1' \supseteq e_2$. According to definition 4.12, $B.LR \setminus \{r\} \geqslant_{AUX} A.R$.

(4) if $AR \neq \varnothing, \forall e \in Proof(A.R)$, $(|AUX| - N_B(AU)) < (|e| - N_B(e))$

Sub-Proof: this is trivial since *AUX* is the auxiliary set in a simplified proof ■

Theorem 4.8 says that if *B.LR* contributes to a simplified proof, the proof of any role in *B.LR* must overlap with *Proof*(*A.R*). As a result, we filter out those roles that do not satisfy this necessary condition in step 1 of the algorithm. Note that if the resulting *T* is empty, then we are not able to find any simplified proof using roles in *B.R* (line 4). Otherwise, we continue to check whether any subset of *T proof-dominates* or *partial-proof-dominates A.R.*

Step 2 (line 5-11): examine each subset of *T* and check whether it *proof-dominates* or *partial-proof-dominates A.R*.

Given a subset $S \subseteq T$, we examine each element of *Proof*(*S*) and *Proof*(*A.R*) and build a matrix *M* as follows. Each row corresponding to one element of *Proof*(*S*) and each column corresponding to one element of *Proof*(*A.R*). Each cell of *M*, *M*(*i*, *j*), is a set of roles that are in the corresponding element of *Proof*(*A.R*) (e_j) but not in the corresponding element of *Proof*(*S*) (e_i) if $e_i \cap e_j \neq \varnothing$. Otherwise, *M*(*i*, *j*) is null. In this way, *M*(*i*, *j*) (if not null) represents the set of auxiliary roles needed to prove e_j assuming e_i has been proved. We then record the union of this role sets into *AUX*(e_i) for each e_j (line 10). The semantic of *AUX*(e_i) is: if a user proves *S* using e_i, she can prove *A.R* by proving any element in *AUX*(e_i). Finally, we build a set of role sets *AUX*(*S*) where each element of it is the union of one element in each *AUX*(e_i) (defined by $\cup$*). The semantic of *AUX*(*S*) is: no matter which element of *Proof*(*S*) a user uses to prove *S*, that user can prove *A.R* by proving any element in *AUX*(*S*). For each element *aux* in *AUX*(*S*), we have $S \geq_{aux} A.R$ according to definition 4.13. Note that if some element *aux* in *AUX*(*S*) is empty, it means $S \geq A.R$.

Step 3 (line 12-19): check whether (*S*,*aux*) satisfies condition (2), (3) and (4) of definition 4.13.

Condition (2) is checked at line 12-14; Condition (3) is checked at line 17; and Condition (4) is checked at line 18.

Table 4. 4. Example of using **Simplify()**

(a): $AUX(e_i)$ when examine $S=\{B.r_1\}$

e_i \ e_j	$r_4, B.r_3, r_8$	$r_5, B.r_3, r_8$	r_4, r_9, r_{10}	r_5, r_9, r_{10}	$AUX(e_i)$
r_4	$B.r_3, r_8$	*null*	r_9, r_{10}	*null*	$\{\{B.r_3, r_8\},\{r_9, r_{10}\}\}$
r_5	*null*	$B.r_3, r_8$	*null*	r_9, r_{10}	$\{\{B.r_3, r_8\},\{r_9, r_{10}\}\}$

(b): $AUX(e_i)$ when examine $S=\{B.r_2\}$

e_i \ e_j	$r_4, B.r_3, r_8$	$r_5, B.r_3, r_8$	r_4, r_9, r_{10}	r_5, r_9, r_{10},	$AUX(e_i)$
r_8	$r_4, B.r_3$	$r_5, B.r_3$	*null*	*null*	$\{\{r_4, B.r_3\},\{r_5, B.r_3\}\}$
r_9	*null*	*null*	r_4, r_{10}	r_5, r_{10}	$\{\{r_4, r_{10}\},\{r_5, r_{10}\}\}$

(c): $AUX(e_i)$ when examine $S=\{B.r_1, B.r_2\}$

e_i \ e_j	$r_4, B.r_3, r_8$	$r_5, B.r_3, r_8$	r_4, r_9, r_{10}	r_5, r_9, r_{10}	$AUX(e_i)$
r_4, r_8	$B.r_3$	$r_5, B.r_3$	r_9, r_{10}	*null*	$\{\{\mathbf{B.r_3}\},\{r_5, B.r_3\},\{r_9, r_{10}\}\}$
r_4, r_9	$B.r_3, r_8$	*null*	r_{10}	r_5, r_{10}	$\{\{B.r_3, r_8\},\{\mathbf{r_{10}}\},\{r_5, r_{10}\}\}$
r_5, r_8	$r_4, B.r_3$	$B.r_3$	*null*	r_9, r_{10}	$\{\{r_4, B.r_3\},\{\mathbf{B.r_3}\},\{r_9, r_{10}\}\}$
r_5, r_9	*null*	$B.r_3, r_8$	r_4, r_{10}	r_{10}	$\{\{B.r_3, r_8\},\{r_4, r_{10}\},\{\mathbf{r_{10}}\}\}$

Example 4.9:

Assume that domain B wants to simplify the proof of a set of role $A.R$ in A, and assume $Proof(A.R)=\{\{r_4,B.r_3,r_8\},\{r_5,B.r_3,r_8\},\ \{r_4,r_9,r_{10}\},\{r_5,r_9,r_{10}\}\}$ (the focus of this example is the **simplify()** algorithm so we will not show how to calculate $Proof(A.R)$ from original RT_0 rules here). Assume after step 1, only two roles in B (i.e. $B.r_1$ and $B.r_2$) are included in T, and their proofs are:

$Proof(B.r_1)=\{\{r_4\},\{r_5\}\}$, and $Proof(B.r_2)=\{\{r_8\},\{r_9\}\}$

Now we start to examine all subsets of T. They are $\{B.r_1\}$, $\{B.r_2\}$, and $\{B.r_1, B.r_2\}$.

We first examine $S=\{B.r_1\}$ and the resulting $AUX(e_i)$ is shown in Table 4.4(a). We have $AUX(\{B.r_1\})=AUX(\{r_4\}) \cup *AUX(\{r_5\})=\{\{B.r_3, r_8\},\{B.r_3, r_8, r_9, r_{10}\},\{r_9, r_{10}\}\}$. Only one element in $AUX(\{B.r_1\})$ (i.e. $\{B.r_3, r_8\}$) has smaller number of B's external roles (i.e. 1) than the number

of *B*'s external roles in any element of *Proof*(*A.R*) (at least 2). As a result, we include ({$B.r_1$},{$B.r_3$, r_8}) in *SP*.

We then examine S={$B.r_2$}, and the resulting $AUX(e_i)$ is shown in Table 4.4(b). We have $AUX(\{B.r_2\})=AUX(\{r_8\}) \cup {}^*AUX(\{r_9\})=\{\{r_4,\ B.r_3,\ r_{10}\},\{r_4,\ B.r_3,\ r_5,\ r_{10}\},\{r_5,\ B.r_3,\ r_{10}\}\}$. No element in $AUC(\{B.r_2\})$ has smaller number of *B*'s external roles than the number of *B*'s external roles in any element of *Proof*(*A.R*) (at least 2). As a result, no element is added to *SP*.

We then examine $S=\{B.r_1,\ B.r_2\}$. We have $Proof(\{B.r_1,\ B.r_2\})=\{\{r_4,r_8\},\{r_4,r_9\},\{r_5,r_8\},\{r_5,r_9\}\}$ and the resulting $AUX(e_i)$ is shown in Table 4.4(c). We have $AUX(\{B.r_1,B.r_2\})= AUX(\{r_4,r_8\}) \cup {}^*AUX(\{r_4,r_9\}) \cup {}^*AUX(\{r_5,r_8\}) \cup {}^*AUX(\{r_5,r_9\})=\{\ldots\}$ (There are 81 elements so we omit the detailed result here). Only one element in $AUX(\{B.r_1,B.r_2\})$ (i.e. $\{B.r_3,\ r_{10}\}$, constructed by the bold elements in each $AUX(e_i)$) has smaller number of *B*'s external roles (i.e. 1) than the number of *B*'s external roles in any element of *Proof*(*A.R*) (at least 2). As a result, we include ({$B.r_1$, $B.r_2$},{$B.r_3$, r_{10}}) in *SP*.

Now we have *SP*={({$B.r_1$},{$B.r_3$, r_8}), ({$B.r_1$, $B.r_2$},{$B.r_3$, r_{10}})}. It is easy to verify that both of the two elements will pass the checking at line 17 and line 18. Therefore, *SP* is the output of our algorithm.

Next, we formally prove the correctness of **Simplify**() using the following two theorems:

Theorem 4.9 (Completeness of Simplify()):

Given A.R, B.R, and Proof(*B.r*) *for all B.r*$\in$*B.R, any Simplified Proof*(*A.R, B.R*) *will be included in the output of* ***Simplify***().

Proof:

For any *SimplifiedProof* (*A.R*, *B.R*) = (*B.LR*, *AUX*), it is easy to see from **Simplify()** that if the following sub-problems are proved, then it will be included in the output of the algorithm:

(1) $B.LR \subseteq T$ *at line 3*.

Sub-proof: To prove $B.LR \subseteq T$, we need to prove $\forall r \in B.LR,\ r \in T$. According to the construction of *T* in line 3, we need to prove $\forall r \in B.LR$, **Overlap**(*Proof*(*r*), *Proof*(*A.R*)) = *true*. This has been proved in theorem 4.8.

(2) *When examining* $S=B.LR \subseteq T$, $AUX \in AUX(B.LR)$ *in line 11*.

Sub-proof: We have $B.LR \geq_{AUX} A.R$. According to definition 4.12, we have "$\forall e_i \in Proof(B.LR)$, $\exists e_{j(i)} \in Proof(A.R)$ such that $e_i \cup AUX \supseteq e_{j(i)}$" (We use slightly different symbols compared to definition 4.12 but it is easy to verify that the semantic is the same). If $e_i \cap e_{j(i)} \neq \varnothing$, we have $e_{j(i)} \subseteq AUX$, which contradicts with "the number of *B*'s external roles in *AUX* is smaller than the number of *B*'s external roles in $e_{j(i)}$" (*AUX* is the auxiliary role set in a simplified proof). As a result, we have $e_i \cap e_{j(i)} \neq \varnothing$. According to line 10, the algorithm will mark $e_{j(i)} \setminus e_i$ in $M(i, j(i))$ and make $e_{j(i)} \setminus e_i \in AUX(e_i)$. We also have $e_{j(i)} \setminus e_i$ is the subset of *AUX*. Recall that here e_i is an arbitrary element in *Proof*(*B.LR*), so for any e_i, the above conclusions are true. Therefore, when we "$\cup$*" all $AUX(e_i)$ in line 11 to form *AUX*(*B.LR*), one element of *AUX*(*B.LR*) (say, *AUX'*) is the union of "$e_{j(i)} \setminus e_i$" for all e_i. Since each $e_{j(i)} \setminus e_i$ is the subset of *AUX*, *AUX'* is also a subset of *AUX*. Next, we prove that *AUX*' must equal to *AUX*. Otherwise (i.e. $AUX' \subset AUX$), *AIX*', as the element of *AUX*(*B.LR*), satisfies condition (1) and (2) of definition 4.13 (as will be proved in

sub-proof (1) and (2) in theorem 4.10). This contradicts with condition (4) in definition 4.13. Therefore, $AUX'=AUX$. In other words, $AUX \in AUX(B.LR)$.

(3) *If AUX is not empty, it will pass the check in line 14. That is,* $\forall e \in Proof(A.R)$, $|AUX|-N_B(AUX)<|e|-N_B(e)$

Sub-proof: This is trivial since (*B.LR, AUX*) is a simplified proof and condition (2) of definition 4.13 ensures this.

(4) B.LR will pass the check in line 17.

Sub-proof: This is trivial since (*B.LR, AUX*) is a simplified proof and condition (3) of definition 4.13 ensures this.

(5) AUX will pass the check in line 18.

Sub-proof: This is trivial since (*B.LR, AUX*) is a simplified proof and condition (4) in definition 4.13 ensures this.

Combining sub-proof (1) through (5), it is easy to verify that (*B.LR, AUX*) will be included in the output of **Simplify**(). ■

Theorem 4.10 (Soundness of Simplify()):

Any element in the output of **Simplify**() *is a simplified proof of A.R using B.R.*

Proof:

Without lose of generality, we assume ($S \subseteq T$, $aux \in AUX(S)$) is one arbitrary output of the algorithm. We need to prove (*S, aux*) is a simplified proof according to definition 4.13. We prove (*S, aux*) satisfies condition (1) through (4) of definition 4.13 as below:

(1) $\varnothing \neq S \subseteq B.R$ *and* $S \geq_{aux} A.R$.

Sub-proof: $\varnothing \neq S \subseteq B.R$ is trivial according to step 1. Next, we prove $S \geq_{aux} A.R$. According to the construction of *AUX*(*S*) at line 11, we have "*aux* is the union of one element in each *AUX*(e_i)" since $aux \in AUX(S)$. Without lose of generality, we assume *aux*(e_i) is the element of an arbitrary *AUX*(e_i) that forms *aux* with other elements through the union operation (i.e. $aux(e_i) \subseteq aux$). Besides, *AUX*(e_i) is constructed by $e_j \setminus e_i$ for some element e_j in *Proof*(*A.R*), therefore, $e_i \cup aux(e_i) \supseteq e_j$. In other words, we have "$\forall e_i \in Proof(S), \exists e_j \in Proof(A.R)$ such that $e_i \cup aux(e_i) \supseteq e_j$". Since $aux(e_i) \subseteq aux$ for all $e_i \in Proof(S)$, we have "$\forall e_i \in Proof(S), \exists e_j \in Proof(A.R)$ such that $e_i \cup aux \supseteq e_j$". According to definition 4.12, we have $S \geq_{aux} A.R$.

(2) if $aux \neq \varnothing, \forall e \in Proof(A.R), (|aux| - N_B(aux)) < (|e| - N_B(e))$

Sub-proof: This is trivial according to the checking at line 13 and 14.

(3) $\forall r \in S$, $(S \setminus \{r\}, aux)$ *does not satisfy condition (1) and (2) of definition 4.13*

Sub-proof: This is trivial since (*S*, *aux*) passes the checking at line 17.

(4) $\forall r \in aux$, $(S, aux \setminus \{r\})$ *does not satisfy condition (1) and (2) of definition 4.13*.

Sub-proof: This is trivial since (*S*, *aux*) passes the checking at line 18. ■

Next, we analyze the complexity of our approach. **Simplify**() requires *Proof*(*A.R*) as one of its inputs. The complexity of calculating the proof of a set of roles is given by Theorem 4.3. The complexity of **Simplify**() is given by the following theorem:

Theorem 4.11:

The worst case complexity of ***Simplify***() *is exponential to* |*A.R*| *and* |*T*|

Proof:

We need to use *Proof*(*A.R*), and the complexity of calculating *Proof*(*A.R*) is exponential to |*A.R*| according to theorem 4.3. Step 1 has complexity of O(|*B.R*|) since we need to examine every role in |*B.R*| to calculate *T*. The complexity of step 2 is given by $O(2^{|T|}\times(TN_{rule}{}^{|S|}+|Proof(S)|\times|Proof(A.R)|+|AUX(S))$. This is because we need to check any subset *S* of *T* in step 2. Checking one such *S* needs to calculate *Proof*(*S*) (the complexity is $TN_{rule}{}^{|S|}$ according to theorem 4.3), check any pair of elements in *Proof*(*S*) and *Proof*(*A.R*), and check any element in *AUX*(*S*). |*Proof*(*S*)| is further given by $ASPS^{|S|}$ where *ARPS* is the Average Size of *Proof*(*r*) for all $r \in S$ (by definition 4.8). |*Proof*(*A.R*)| is further given by $ASPAR^{|A.R|}$ where *ASPAR* is the Average Size of *Proof*(*r*) for all $r \in A.R$ (by definition 4.8). And O(|*AUX*(*S*)|) is given by $ASPS^{|S|}$ to the power of $ASPAR^{|A.R|}$ (by the construction of *AUX*(*S*) in step 2). In summary, the complexity of step 2 is exponential to |*A.R*| and |*T*| (Maximal *S* is *T*). The complexity of step 3 is $O(|AUX(S)|+|SP|^2)$. O(|*SP*|) is given by O(|*AUX*(*S*)×|*S*|) since any element in *AUX*(*S*) together with *S* could form an element of *SP*. Therefore, the complexity of step 3 is given by $O(|AUX(S)|^2)$ which is exponential to |*A.R*| and |*T*|. Combining all the three steps, the complexity of **Simplify**() is exponential to |*A.R*| and |*T*|. ■

In summary, the complexity of our simplification framework is exponential to the number of |*A.R*| since we need to examine *Proof*(*A.R*) which cannot be avoided in our approach. The complexity of our framework is also exponential to the number of |*T*| because we need to

examine every subset of candidate roles to find the simplified proof, and this expensive step cannot be avoided either, as shown in the following example:

Example 4.10:

Assume *Proof*(*A.R*)={{r_3,r_4}}. Consider two roles in *B* such that *Proof*(*B.*r_1)={{r_3}} and *Proof*(*B.*r_2)={{r_4}}. It is easy to verify that neither of these two roles *proof-dominates A.R*. However, if we consider the union of these two roles, i.e. {*B.*r_1, *B.*r_2}, we have *Proof*({*B.*r_1,*B.*r_2})={{r_3, r_4}}. Now we can verify that {*B.*r_1, *B.*r_2} $\geqslant$ *A.R*, and the corresponding simplified proof is ({*B.*r_1,*B.*r_2}, $\varnothing$).

Example 4.10 shows that a set of roles could contribute to a simplified proof without auxiliary set even if none of its subsets can. Therefore, in order to ensure that **Simplify()** can find all the existing simplified proofs, we need to examine every subset *S* of the candidate role set *T*.

Although the complexity of our algorithm is exponential to |*A.R*| and |*T*|, we argue that the performance of our algorithm is still acceptable for the following reasons:

|A.R| is likely to be very small in practice. Recall that *A.R* represents the roles that outside users want to assume in a single access request. Therefore, it is not likely to be a very large number in practice.

|T| is also likely to be small in practice. First, the size of *B.R* is likely to be small according to the discussion before. Second, we argue that usually only small portion of *B.R* will be included in *T* in practice. Intuitively, if a role is selected in *T*, its policy must share some common roles with the target role whose proof is to be simplified (**Overlap**(*Proof*(*r*), *Proof*(*A.R*)) = *true*). In practice,

we claim that usually only a small number of roles in two different domains will share some common roles (credentials) in their policies.

Example 4.11:

Consider two hospital (say, H_1 and H_2) domains both having *Doctor* role, *Nurse* role, *Finance* role, and *Admin* role (a simplified example). Consider the policy of H_2.*Doctor*. Usually the policy of H_1.*Doctor* would share some common credentials with it (e.g. both require *MD* degree and license of doctor), and the policy of H_1.*Nurse* may also share some common credentials with it (e.g. both require health-care license). On the other hand, the policies of H_1.*Finance* and H_1.*admin* usually do not share any common credentials with the policy of H_2.*doctor*. According to step 1 in **Simplify()**, H_1.*Finance* and H_1.*Admin* will be eliminated from T if we want to simplify the proof of H_2.*doctor*.

In case $|A.R|$ is relatively large (e.g. 3, where the worst case complexity of our approach becomes cubic), we propose several heuristics that are able to reduce the complexity of **Simplify**() from different aspects. During implementation, the developers can choose some of the heuristics according to their specific needs.

Heuristic1: Using "trimming" mechanism in calculating *AUX*(*S*). The most complex part in our algorithm is to examine each element in $AUX(S)$ which is given by $ASPS^{|S|}$ to the power of $ASPAR^{|A.R|}$. However, if we find some elements in $AUX(e_i)$ whose number of B's external roles is not smaller than the number of B's external roles in some element of $Proof(A.R)$, we do not need to include it in $AUX(e_i)$ since the auxiliary role set formed by it will not pass the checking at line

14. For example, in Table 4.4(a) $\{r_9, r_{10}\}$ is one element in $AUX(\{r_4\})$ that already contains two of B's external roles (equal to the number of B's external roles in some element of $Proof(A.R)$). The elements of $AUX(S)$ constructed from it (i.e. $\{B.r_3, r_8, r_9, r_{10}\}$ and $\{r_9, r_{10}\}$) will not pass the checking at line 18. As a result, we can remove $\{r_9, r_{10}\}$ from both $AUX(\{r_4\})$ and $AUX(\{r_5\})$. In this way, we can significantly reduce the complexity of calculating $AUX(S)$.

Heuristic 2: For each *S*, output one simplified proof whose auxiliary role set has smallest number of *B*'s external roles.

In **Simplify**(), for each S we output all simplified proofs with different auxiliary sets (elements in $AUX(S)$). Since proving external roles is expensive, we can choose to output one simplified proof for each S whose auxiliary role set has the smallest number of B's external roles. To ensure this, we need to slightly modify line 10 and line 11 in our algorithm. Instead of making the union of all $\{e_j \setminus e_i\}$ to form $AUX(e_i)$ (line 10), we choose $\{e_j \setminus e_i\}$ with the smallest number of B's external roles to be $AUX(e_i)$. And we use a standard union instead of "$\cup$*" in line 11 to form $AUX(S)$ (which becomes a single set) since each $AUX(e_i)$ is a single set now. In this way, the complexity of calculating $AUX(S)$ reduces to $|Proof(S)|\times|Proof(A.R)|$, which is no longer exponential.

Heuristic 3: Output one simplified proof with the smallest *B.LR*.

In our algorithm, we examine every subset S of T. If S contributes to a simplified proof, S itself becomes $B.LR$ in the simplified proof (recall that $B.LR$ is the first component in a simplified proof). During implementation, we can examine subsets of T with increasing size. Once we find a simplified proof, the algorithm terminates (instead of examining further subsets with larger size)

and output that simplified proof. In this way, we ensure only one simplified proof is returned and it has smallest *B.LR* among all possible simplified proofs. And the complexity of step 2 could be reduced significantly.

Heuristic 4: Restrict the size of S.

We can restrict the size of *S* when we examining the subsets of *T*. For example, we can restrict that only those subsets with size not greater than 3 will be examined. In this way, we do not want to simplify the proof by using more than 3 of *B*'s local roles. It is easy to see that the complexity of the algorithm is exponential to only $|A.R|$ in this case.

4.3.3 Proof Engine

After the simplified proof has been found for a particular *rar*, the relevant users (i.e. users assigned to some of the requesting roles and need to access the requested resource according to the functions of the requesting roles) can issue *uar* to actually request to access the requested resources. The Proof Engine is responsible for verifying whether a user can prove the requested roles using DCCD and the simplified proof if exist. Given an *uar* $<B.u, R_u, A.R>$, if there exists a simplified proof $<B.R, AUX>$ for *A.R* such that $R_u \supseteq B.R$, we say that *B.u* benefit from the simplified proof since *B.u* can prove *B.R*. The proving of *A.R* for *B.u* can be simplified by proving *B.R* and *AUX*. Note that *B.u* still needs to prove *AUX* using DCCD since *AUX* are not local roles of B. For those users not benefit from any simplified proofs, they need to prove *A.R* using DCCD.

4.3.4 Evaluation

The motivation of this work is to simplify the distributed proof procedure in *role-based interoperation scenario*. As discussed, Distributed Credential Chain Discover (DCCD) based on RT_0 is an extensively studied distributed proof procedure based on the notion of roles. Therefore, we evaluate the performance of our approach against DCCD approach. Here, we define the term "performance" as the time complexity of running the required algorithms to make an authorization decision on a set of *role-based interoperation requests*. Recall that a set of role-based interoperation requests are issued from different users from the same set of requesting roles for a set of requested roles. Assume m requesting users assigned to some roles in $B.R$ issues m requests for the requested role set $A.R$, and assume C is the total number of credentials in the environment. The complexity of authorizing all these m requests using traditional DCCD approach is given by:

$$T_D = m\,|A.R|\,O(C^3) \qquad (1)$$

This is because DCCD treats the m requests separately. That is, given a single requested role requested from one single user, DCCD approaches check whether the requesting user is able to prove that single role using the credential chains. Therefore, in our defined role-based interoperation requests, DCCD algorithms need to run for each single user and for each single requested role. Furthermore, the worst time complexity of DCCD algorithm is given by $O(C^3)$ [26]. Therefore, the total worst time complexity of DCCD approaches is given by (1)

The complexity of authorizing all these m requests using our simplification approach is:

$$T_S = O(N_{rule}^{|A.R|} 2^{|B.R|}) + O(1) + m_1 |AUX| O(C^3) + m_2 |A.R| O(C^3) \quad (2)$$

This is because we need to run our simplification algorithm once, the complexity of which is given by $O(N_{rule}^{|A.R|} 2^{|B.R|})$ from theorem 4.3 and 4.11. After the simplified proof is found, we need to ask those users assigned to *B.LR* (whose number is assumed to be m_1) to prove that they are assigned to *B.LR*. This can be done in O(1) time as discussed before. Then, in case of partial proof domination, (i.e. *AUX* is not empty). We need to ask those m_1 users to prove *AUX* using DCCD approach ($m_1 |AUX| O(C^3)$). Finally, we need to ask the remaining m-m_1=m_2 users (i.e. not assigned to *B.LR*, therefore cannot benefit from the simplified proof) to prove *A.R* using DCCD approach ($m_2 |A.R| O(C^3)$).

To simplify (2), we define $\alpha_1 = m_1 / m \in [0,1]$ as the ratio of users that can benefit from the simplified proof, and define $\alpha_2 = (1 - |AUX| / |A.R|) \in [1 / |A.R|, 1]$ as the simplification ratio indicating what percentage of B's external roles are simplified (i.e. replaced by local roles) after simplification. Given this, we have:

$$T_S = O(2^{|A.R||B.R|}) + O(1) + \alpha_1 (1- \alpha_2) m |A.R| O(C^3) + (1- \alpha_1) m |A.R| O(C^3) \quad (3)$$

We are interested to see whether our simplification work indeed simplifies the distributed proof procedure against DCCD approach. In other words, we want to see whether and in what conditions we have $T_S < T_D$, this can be translated as shown below according to (1) and (3):

$$O(2^{|A.R||B.R|}) + O(1) < \alpha_1 \alpha_2 m |A.R| O(C^3) \quad (4)$$

According to (4), it is easy to see that there are 6 variables that will affect whether $T_S < T_D$, that is: $|A.R|$, $|B.R|$, m, C, α_1 and α_2. One initial observation from (4) would be: in order to make $T_S < T_D$, $|A.R|$ and $|B.R|$ should be as small as possible, m, C, α_1 and α_2 should be as large as possible. We conduct several experiments to verify whether and when we have $T_S < T_D$.

SimulateEnvironment(N_d, N_a, N_u, n_r, n_p, n_u)
Input: N_d: total number of organization domains; N_a: total number of authority domains;
N_u: total number of users; n_r: average number of roles per domain;
n_p: average number of rules defining each role; n_u: average number of users per role.
Output: A multi-domain environment containing N_d domains; each domain has several roles defined by several RT_0 rules. Each role is assigned several users.

```
1   generate N_d organization domains
2   generate N_a authority domains
3   generate N_u users
4   foreach domain d in N_d domains
5       randomly assign some friend domains of d from N_d domains
6       randomly generate roles in d according to n_r
7   foreach domain a in N_a domains
8       generate one role in a
9   foreach domain a in N_a domains
10      foreach domain d in N_d domains
11          make d 50% chances to be accredited by a
12      if (a has not accredited any organization domain)
13          randomly select a domain d in N_d to be accredited by a
14  foreach domain d in N_d domains
15      foreach role r in domain d
16          randomly generate rules for this role according to n_p
17              foreach rule ru in np rules
18                  randomly assign the type of ru
19                  randomly generate the body of ru
20      randomly generate users for r according to n_u
```

Figure 4. 12. algorithm to simulate a multi-domain environment

Among the 5 variables that will affect whether $T_S < T_D$ according to (4), $|A.R|$, $|B.R|$, m, C are the attributes of the requests and policies and are not affected by the result of the simplification algorithm. On the other hands, α_1 and α_2 depend on the specific result of the simplification algorithm. Therefore, we choose to control $|A.R|$, $|B.R|$, m, C to simulate *role-based interoperation requests*, and α_1 and α_2 will be determined through the result of simulation. Specifically, we first simulate a multi-domain environment along with all the policies to generate the credential pool with the size C. The algorithm to simulate a multi-domain environment is shown by Figure 4.12.

In Figure 4.12, we try to simulate the multi-domain environment as close as the real interoperation scenario as possible. First, we define two types of domains: (1) Organization domains: normal domains that need to interoperate with each other; and (2) Authority domains: domains that "certify" some attributes of other organization domains. For example, Hospital A is a normal organization domain, and ABU is an authority domain certifying which domain is a valid university accredited by it (e.g. *ABU.accredited* ← *StateU*). Usually each authority domain is responsible for accrediting one type of the organization domains (e.g. ABU is responsible for accrediting the valid universities only). Therefore, we assign only one role to each authority domain as shown in line 8 in figure 4.12. Second, we define some "friend domains" that are trusted by each organization domain. We assume that each domain tends to define its role according to the roles in its friend domains. For example, University of Pittsburgh (UPitt) may collaborate with Carnegie Mellon University (CMU) and define that any student in

CMU is authorized to access the library resource in Pitt (i.e. *UPitt.library* ← *CMU.student*). In this case, we say CMU is the friend domain of UPitt. The notion of friend domains helps us to generate the body of rules (in line 19) in a more reasonable way. The remaining part of the algorithm is straightforward. We just randomly generate the elements of the environment (e.g. roles in each domain) according to some pre-defined parameters (e.g. average number of roles per domain). In order to make sure the generated environment has approximately C total credentials (rules), we make several experiments and learn the relations between C and the parameters of the algorithm. For example, we find that **SimulateEnvironment**(50, 5, 20, 5, 3, 5) will always generate approximately 2000 credentials. Next, we need to simulate role-based interoperation requests based on $|A.R|$, $|B.R|$, and m. This is straightforward and we just need to randomly pick two domains (A and B) first and pick several roles as $A.R$ and $B.R$ respectively. Since m different requests for $A.R$ are all issued from users assigned to $B.R$, we only need to generate $A.R$ and $B.R$ once for all m requests.

After the multidomain environment and interoperation requests are generated, we are ready to calculate T_S and T_D and compare them according to (3) and (4). We have one last tricky issue here: it is very difficult to guarantee that a simplified proof will be found using such randomly generated policies. In reality, since positions in organizations have real meanings and many of them are related to each other, we believe simplified proof could be found in many scenarios (As shown in Example 4.6). We left the analysis of how much percentage a simplified proof can be found as future work. In our simulation, although we try to simulate the policies as

close to the real life as possible, they are just randomly generated symbols with few real meanings so it is very difficult to guarantee a simplified proof could be found. Fortunately, for comparing T_S and T_D only α_1 and α_2 depends on whether a simplified proof exists or not according to their definitions. Therefore, in the case no simplified proof is found, we can simply make α_1 and α_2 to be uniformly distributed random variables within the range of [0, 1] and [*1* / |*A.R*|, *1*], respectively. The rationale of using such trick here is that we do not under-estimate the complexity of simplified approach by using randomly selected α_1 and α_2.

We simulate 9 pairs of |*B.R*| and |*A.R*| values, that is (1,1), (1,2), (1,3), (1,4), (2,1), (3,1), (4,1), (2,2), and (3,3) respectively. This is because (1): the number of |*A.R*| and |*B.R*| are likely to be very small as discussed above; (2) The memory size in our experimental machine does not allow us to simulate very large |*A.R*| and |*B.R*|. For each given pair of (|*B.R*|,|*A.R*|), we calculate T_S and T_D under different *m* and *C*. Specifically, we make *m*={1, 2, 5, 10, 20, 50, 100}, and make *C* = {100, 200, …, 1900, 2000}. For each combination of |*A.R*|, |*B.R*|, *m* and *C*, we simulate 100 times and use the average values as T_S and T_D.

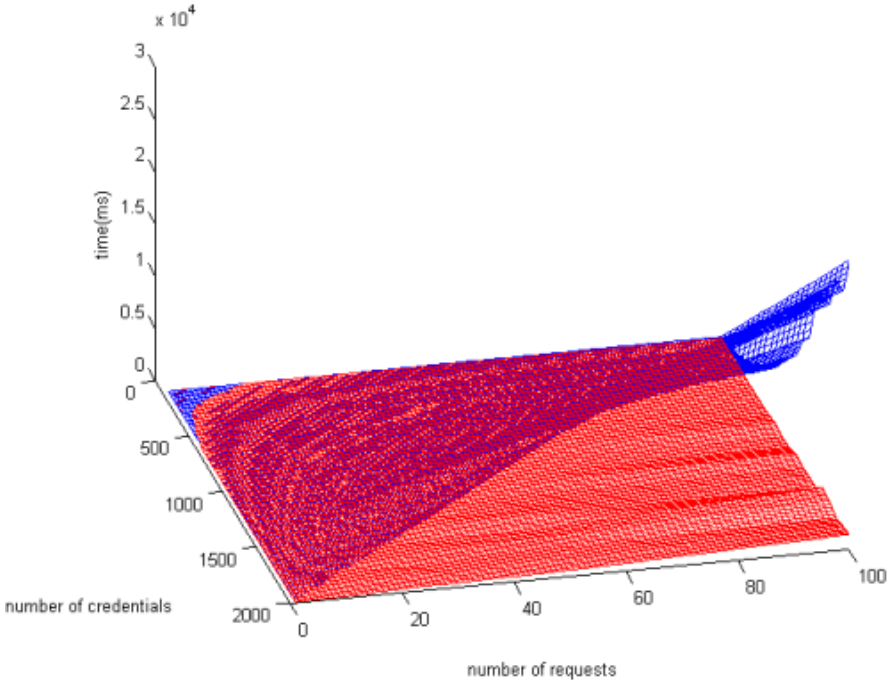


Figure 4.13 Effect of |*A.R*| when (|*B.R*|, |*A.R*|)=(1,1)

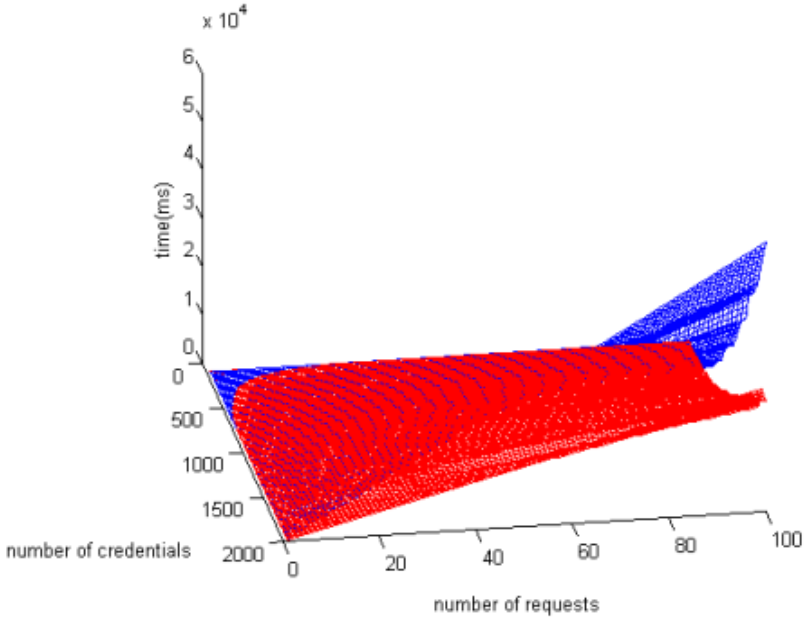


Figure 4.14 Effect of |*A.R*| when (|*B.R*|, |*A.R*|)=(1,2)

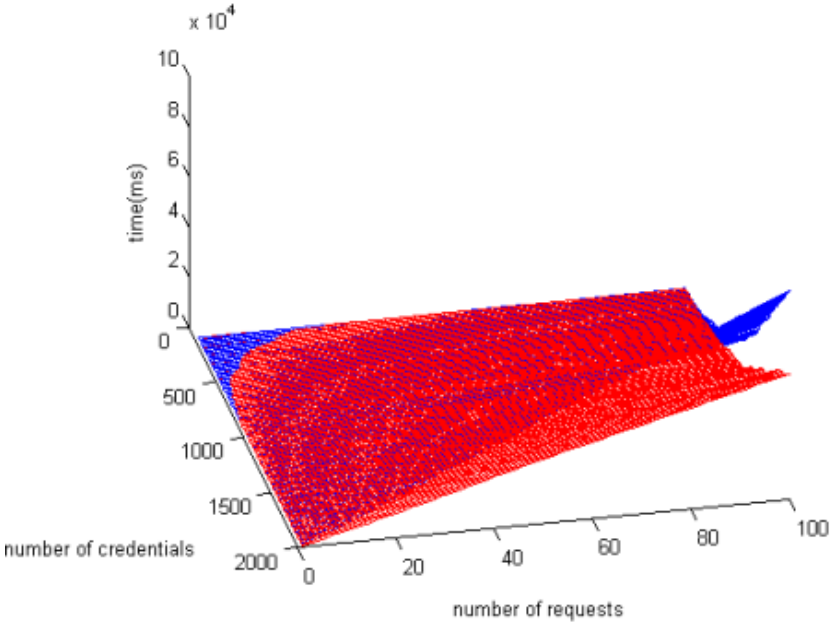


Figure 4.15 Effect of |*A.R*| when (|*B.R*|, |*A.R*|)=(1,3)

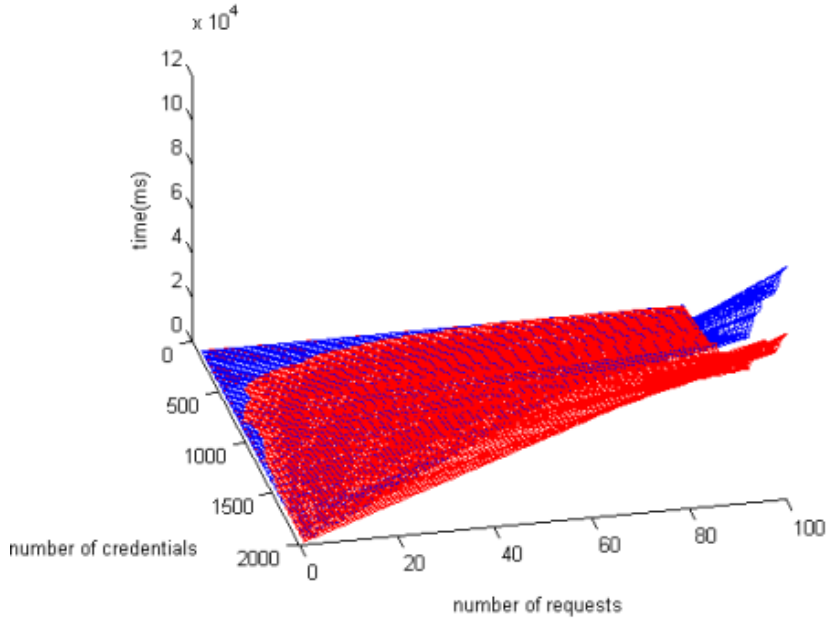


Figure 4.16 Effect of |*A.R*| when (|*B.R*|, |*A.R*|)=(1,4)

Figure 4.13-4.16 shows the comparison of T_S (red) and T_D (blue) with the increase of m and C when (|$B.R$|, |$A.R$|) is (1,1), (1,2), (1,3), and (1,4) respectively. In all of these four figures, T_D will become much larger than T_S when m and C are large enough. This is consistent with our analysis. We also note that the difference between T_D and T_S decrease with the increase of |$A.R$|. This is straightforward since T_S is exponential to |$A.R$| while T_D is only linear to |$A.R$|. Nevertheless, even when |$A.R$| is as large as 4, T_D is still larger than T_S as shown in Figure 4.16.

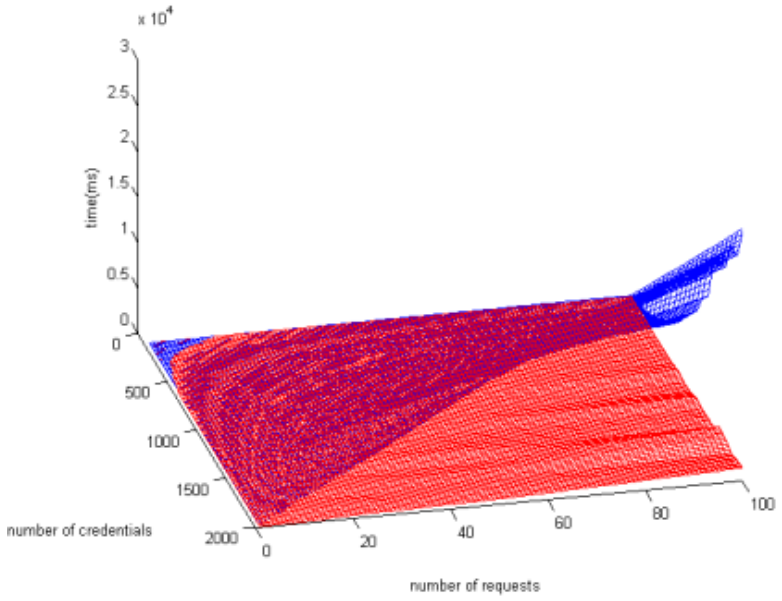


Figure 4.17 Effect of |$B.R$| when (|$B.R$|, |$A.R$|)=(1,1)

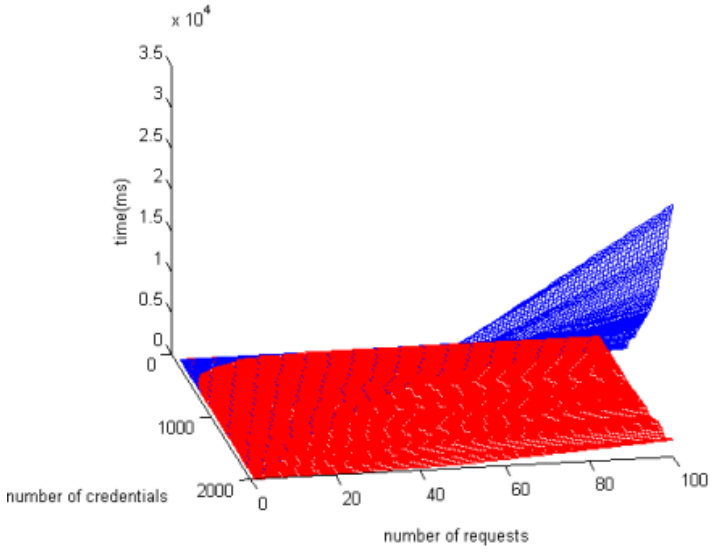


Figure 4.18 Effect of |*B.R*| when (|*B.R*|, |*A.R*|)=(2,1)

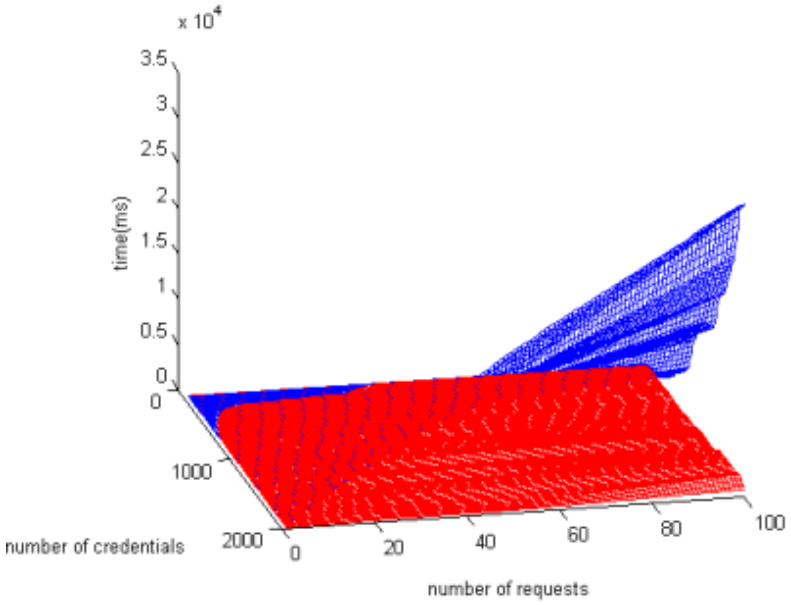


Figure 4.19 Effect of |*B.R*| when (|*B.R*|, |*A.R*|)=(3,1)

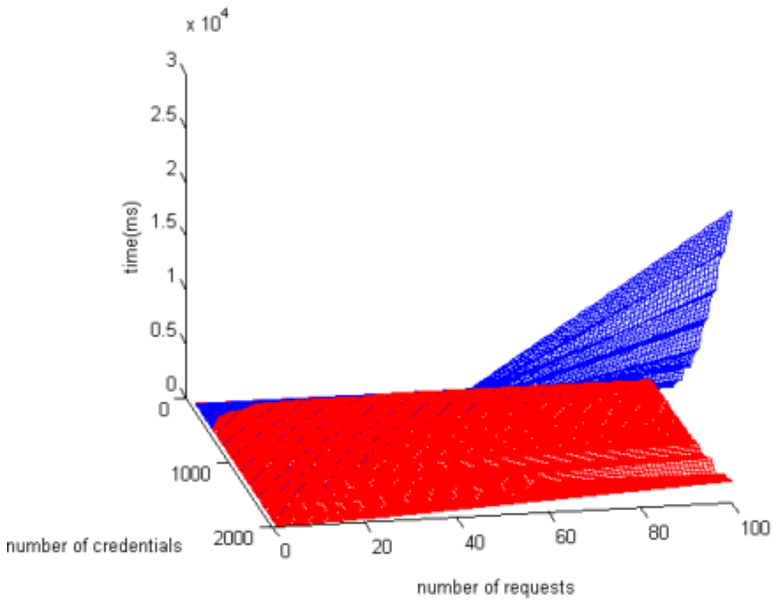


Figure 4.20 Effect of |*B.R*| when (|*B.R*|, |*A.R*|)=(4,1)

Figure 4.17-4.20 shows the comparison of T_S (red) and T_D (blue) with the increase of m and C when (|*B.R*|, |*A.R*|) is (1,1), (2,1), (3,1) and (4,1) respectively. Again, T_D will become much larger than T_S when m and C are large enough in all of these four figures. However, the difference of T_D and T_S does not vary much when we increase |*B.R*|. Since T_D does not depend on |*B.R*|, it shows that T_S also does not vary much with the increase of |*B.R*|. At first glance, it is not consistent with our analysis since T_S is expected to be exponential to |*B.R*|. The explanation of this is as follows: recall that our algorithm is exponential to |*B.R*| only in the worst case. Actually, not all of the roles in |*B.R*| will count for the complexity of our simplification algorithm. In the first step of the algorithm we eliminate some of the roles in |*B.R*| (the remaining role set is *T*) that cannot contribute to any simplified proofs. Therefore, T_S is only exponential to the size of $T \subseteq$

B.R, on average it is not exponential to |*B.R*|. And our simulation results show that the worst case rarely happens. In other words, T_S is not that sensitive with the increase of |*B.R*|.

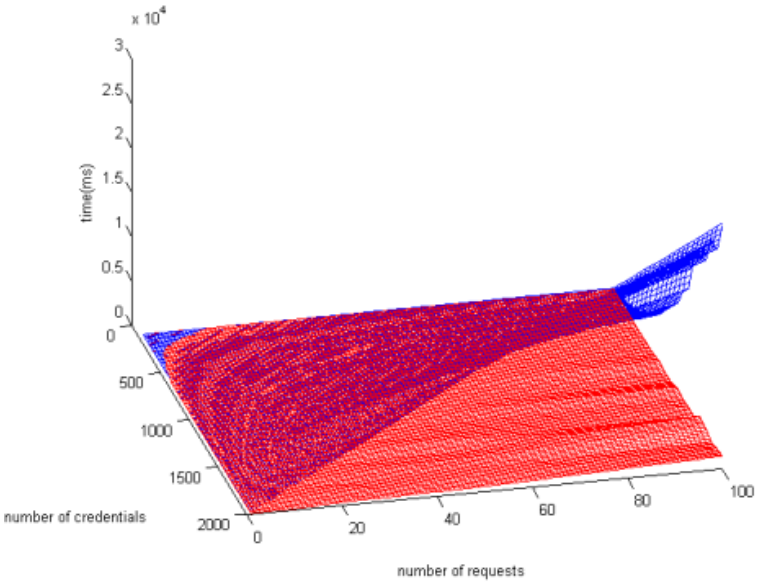


Figure 4.21 Effect of both when (|*B.R*|, |*A.R*|)=(1,1)

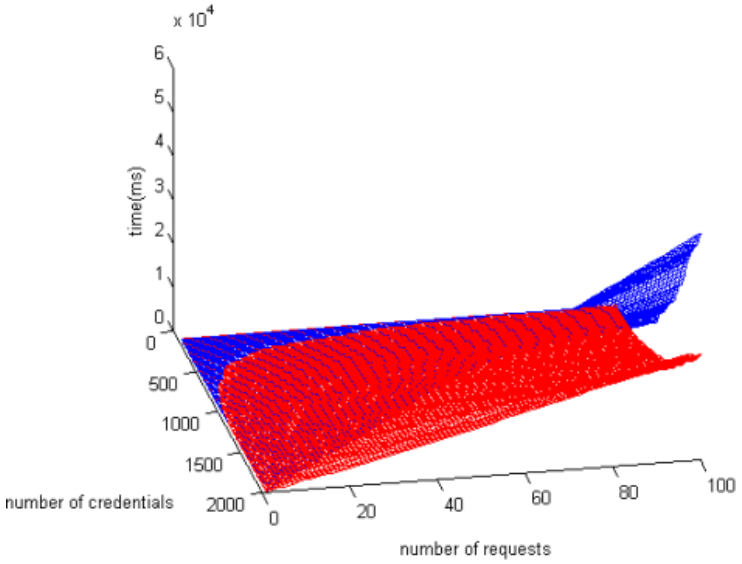


Figure 4.22 Effect of both when (|*B.R*|, |*A.R*|)=(2,2)

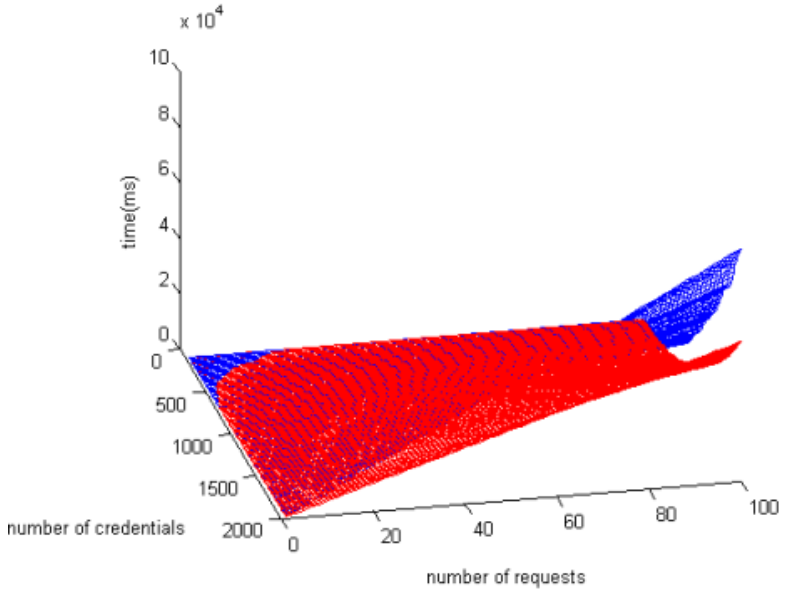


Figure 4.23 Effect of both when (|*B.R*|, |*A.R*|)=(3,3)

Figure 4.21-4.23 shows the comparison of T_S (red) and T_D (blue) with the increase of m and C when (|*B.R*|, |*A.R*|) is (1,1),(2,2),(3,3) respectively. The trend of the these three figures are similar to the trend of Figure 4.13-4.16, since only the increase of |*A.R*| will cause the exponential increase of T_S.

Figure 4.13-4.23 shows that (1) T_D will eventually become much larger than T_S when m and C is large enough (which is very common in practice); (2) |*B.R*| does not contribute to much to the comparison of T_D and T_S; (3) When |*A.R*| increases, the difference between T_D and T_S will decrease, but T_D is still larger than T_S. However, Figure 4.13-4.23 do not show clearly if m and C are small, whether T_S could be larger than T_D. The following set of figures show clearly when T_S is larger than T_D (i.e. in this case our simplification has no real benefits).

	100	200	300	400	500	600	700	800	900	1000	1100	1200	1300	1400	1500	1600	1700	1800	1900	2000
1	27	25	26	26	25	21	21	23	20	20	21	21	22	26	24	24	20	23	26	22
2	27	26	27	27	27	22	21	22	22	18	23	30	19	36	29	26	25	43	39	39
5	27	26	27	31	24	26	23	31	29	40	36	43	30	41	41	55	32	61	68	92
10	28	32	28	31	40	34	37	39	44	59	74	47	46	65	123	118	124	78	184	97
20	27	31	39	47	39	39	45	48	71	89	87	94	152	92	110	108	237	211	332	307
50	35	40	67	41	59	82	106	158	126	185	207	247	231	363	580	364	594	570	642	553
100	38	36	60	100	56	116	139	200	416	205	249	729	543	646	604	1627	1512	913	1197	1434

	100	200	300	400	500	600	700	800	900	1000	1100	1200	1300	1400	1500	1600	1700	1800	1900	2000
1	1	3	8	14	19	24	36	45	56	67	84	101	123	153	191	194	204	230	247	275
2	2	5	19	27	43	52	73	87	110	138	176	202	239	266	365	396	420	456	500	573
5	7	18	47	70	100	130	186	228	292	326	420	504	595	717	883	1003	1026	1115	1275	1404
10	15	44	84	147	213	261	354	446	562	700	884	1010	1144	1411	1696	1976	2027	2238	2559	2900
20	46	68	184	295	413	489	700	872	1147	1392	1717	1949	2335	2763	3559	3913	4074	4493	4837	5613
50	125	196	397	632	1076	1311	1807	2368	2931	3663	4264	5005	6084	6900	9388	10311	10317	11751	12696	14210
100	194	279	842	1425	2089	2867	3610	4615	5692	7268	8705	10076	12507	13862	18733	19972	20278	23687	23627	27565

Figure 4. 13. Comparison of |*A.R*| when (|*B.R*|, |*A.R*|)=(1,1)

	100	200	300	400	500	600	700	800	900	1000	1100	1200	1300	1400	1500	1600	1700	1800	1900	2000
1	47	56	55	54	62	66	72	73	92	101	134	128	130	168	182	201	226	215	253	292
2	50	53	56	65	67	77	99	103	148	164	197	237	237	282	327	348	420	403	516	501
5	49	62	75	85	104	136	176	210	286	349	445	457	527	661	754	795	1012	923	1198	1290
10	59	66	98	140	163	252	320	386	569	725	858	959	1109	1171	1350	1573	2149	1880	2100	2594
20	64	91	170	200	282	433	577	722	982	1230	1598	1894	2086	2505	2649	3415	4162	3088	4556	5071
50	74	152	284	434	687	1021	1297	1882	2420	3345	3879	4478	4843	5540	6869	8680	8085	9649	10694	12574
100	121	246	519	755	1269	1803	2579	3882	4738	6773	7673	7748	11399	12593	13285	17123	17383	18042	23585	23122

	100	200	300	400	500	600	700	800	900	1000	1100	1200	1300	1400	1500	1600	1700	1800	1900	2000
1	2	6	14	20	31	50	66	84	120	142	187	213	237	285	318	376	418	446	488	571
2	7	14	26	45	63	102	128	173	253	313	373	437	473	580	648	727	865	867	984	1102
5	14	37	76	111	152	240	315	426	601	771	866	1109	1176	1450	1598	1827	2093	2144	2548	2770
10	33	70	154	220	319	507	641	888	1164	1575	1788	2200	2496	2810	3113	3564	4753	4358	4766	5577
20	56	138	309	440	675	993	1280	1737	2278	3025	3720	4364	4906	5628	6207	7731	9315	8722	9858	11075
50	131	336	830	1106	1661	2454	3228	4507	5600	7744	9342	10070	11767	14186	15503	19108	19312	22393	25057	27828
100	266	580	1461	2239	3458	5121	6052	9089	11022	15166	18204	19776	24389	28437	30700	39218	39600	43735	50017	55640

Figure 4. 14. Comparison of |*A.R*| when (|*B.R*|, |*A.R*|)=(1,2)

	100	200	300	400	500	600	700	800	900	1000	1100	1200	1300	1400	1500	1600	1700	1800	1900	2000
1	87	91	89	106	109	134	140	157	178	215	240	230	271	309	338	380	415	464	530	546
2	93	98	101	106	138	175	214	233	318	312	365	403	476	547	598	656	775	867	1011	1102
5	99	131	159	166	223	316	410	460	609	634	774	894	1111	1244	1596	1661	1928	2191	2224	2579
10	99	142	199	269	375	530	751	931	1156	1285	1614	1651	2186	2294	2817	3123	3715	4430	4755	5034
20	103	150	318	407	630	993	1389	1768	2217	2374	2995	3369	4444	4683	5897	6462	7253	7973	9422	10013
50	140	289	583	996	1561	2478	3331	4136	5842	5894	7531	8939	10402	11704	14026	16253	19181	21451	24505	25786
100	198	563	1165	1805	3263	5058	6314	7895	11119	12411	14403	16272	21780	24898	30230	30986	37291	42962	45655	49919

	100	200	300	400	500	600	700	800	900	1000	1100	1200	1300	1400	1500	1600	1700	1800	1900	2000
1	4	9	22	30	53	78	114	142	195	215	256	290	350	397	478	528	633	674	778	821
2	7	18	43	61	116	168	226	274	397	403	514	583	715	796	931	1018	1236	1363	1575	1670
5	19	52	116	155	279	408	565	659	984	1023	1268	1412	1796	1996	2459	2657	3052	3503	3669	4099
10	34	105	237	331	551	773	1135	1371	1898	2051	2558	2913	3523	3902	4739	5181	6186	7149	7639	8231
20	96	195	478	637	998	1637	2212	2835	3832	4107	5161	5572	7434	7917	9581	10306	12228	13680	15237	16489
50	165	545	1032	1664	2643	4190	5836	7187	9948	10105	12416	14790	17547	19631	23266	26936	30992	35187	38352	42053
100	421	1014	2218	3316	5921	8978	10950	13774	19739	20713	25743	28425	35578	39625	48673	53685	61795	68827	75763	83039

Figure 4. 15. Comparison of |*A.R*| when (|*B.R*|, |*A.R*|)=(1,3)

	100	200	300	400	500	600	700	800	900	1000	1100	1200	1300	1400	1500	1600	1700	1800	1900	2000
1	113	230	277	198	278	256	336	319	756	479	532	555	594	642	641	748	770	776	892	1005
2	147	190	255	239	290	279	351	361	658	650	765	714	925	1057	1006	1209	1466	1330	1559	1729
5	164	237	266	321	408	506	759	750	1034	1224	1484	1515	2096	2280	2307	2696	3156	3104	3720	3913
10	145	335	383	398	660	842	1285	1342	1914	2231	2443	2842	3585	3946	4555	5227	6133	5666	7046	7476
20	175	300	623	788	1265	1359	2139	2353	3688	4057	4926	5247	6802	7284	8348	9656	12021	11989	14189	14976
50	273	476	1128	1496	2628	3296	5615	5978	7992	8676	11648	13151	13987	18806	21215	25288	28700	29376	34927	37098
100	363	891	2140	2974	4657	6033	10778	9033	16159	17688	23107	25961	26911	38192	41232	53791	56070	57392	67159	74550

	100	200	300	400	500	600	700	800	900	1000	1100	1200	1300	1400	1500	1600	1700	1800	1900	2000
1	5	12	31	47	82	98	151	190	269	264	342	393	479	568	645	716	818	839	1008	1118
2	13	21	62	87	183	189	298	374	503	542	674	753	951	1113	1184	1417	1736	1658	1969	2131
5	29	59	163	223	402	491	852	914	1207	1308	1682	1856	2442	2755	3027	3623	4415	4123	4945	5356
10	56	121	297	411	812	985	1544	1809	2385	2724	3361	3842	4917	5410	6309	7359	8623	8267	9879	10448
20	91	236	628	940	1619	1955	3105	3368	4982	5468	6889	7559	10003	10607	12088	13885	17271	16720	20134	21771
50	241	597	1609	2132	4101	4898	8286	8976	11514	13467	16698	19065	20684	26456	30914	37577	42321	41702	49401	53198
100	516	1269	3349	4499	7747	9655	16321	17881	24156	26355	33627	38709	40996	54964	62251	77767	84595	84049	97658	104648

Figure 4. 16. Comparison of |*A.R*| when (|*B.R*|, |*A.R*|)=(1,4)

Figure 4.24-4.27 shows the actual number of T_S (upper part) and T_D (lower part) with the increase of *m* and *C* when (|*B.R*|, |*A.R*|) is (1,1), (1,2), (1,3), and (1,4) respectively. The yellow shaded area shows the scenarios when T_S is larger than T_D. We can see that such area increases with the increase of |*A.R*|. This is straightforward since T_S increases much faster than T_D with the increase of |*A.R*|. However, even in figure 4.27 (|*A.R*| is 4), only 26% of the entire rectangular area is yellow. In other words, even in the worst case (from the perspective of favoring T_S) that we can simulate, T_D is larger than T_S in more than 70% cases. Furthermore, almost all those yellow shaded area reside in the area where *m* and *C* are very small. In practice, *m* and *C* are easily become very large. In those cases, T_D will be much larger than T_S

	100	200	300	400	500	600	700	800	900	1000	1100	1200	1300	1400	1500	1600	1700	1800	1900	2000
1	27	25	26	26	25	21	21	23	20	20	21	21	22	26	24	24	20	23	26	22
2	27	26	27	27	27	22	21	22	22	18	23	30	19	36	29	26	25	43	39	39
5	27	26	27	31	24	26	23	31	29	40	36	43	30	41	41	55	32	61	68	92
10	28	32	28	31	40	34	37	39	44	59	74	47	46	65	123	118	124	78	184	97
20	27	31	39	47	39	39	45	48	71	89	87	94	152	92	110	108	237	211	332	307
50	35	40	67	41	59	82	106	158	126	185	207	247	231	363	580	364	594	570	642	553
100	38	36	60	100	56	116	139	200	416	205	249	729	543	646	604	1627	1512	913	1197	1434

	100	200	300	400	500	600	700	800	900	1000	1100	1200	1300	1400	1500	1600	1700	1800	1900	2000
1	1	3	8	14	19	24	36	45	56	67	84	101	123	153	191	194	204	230	247	275
2	2	5	19	27	43	52	73	87	110	138	176	202	239	266	365	396	420	456	500	573
5	7	18	47	70	100	130	186	228	292	326	420	504	595	717	883	1003	1026	1115	1275	1404
10	15	44	84	147	213	261	354	446	562	700	884	1010	1144	1411	1696	1976	2027	2238	2559	2900
20	46	68	184	295	413	489	700	872	1147	1392	1717	1949	2335	2763	3559	3913	4074	4493	4837	5613
50	125	196	397	632	1076	1311	1807	2368	2931	3663	4264	5005	6084	6900	9388	10311	10317	11751	12696	14210
100	194	279	842	1425	2089	2867	3610	4615	5692	7268	8705	10076	12507	13862	18733	19972	20278	23687	23627	27565

Figure 4. 17. Comparison of |*B.R*| when (|*B.R*|, |*A.R*|)=(1,1)

	100	200	300	400	500	600	700	800	900	1000	1100	1200	1300	1400	1500	1600	1700	1800	1900	2000
1	49	43	37	37	28	28	29	31	28	26	30	26	27	28	32	29	27	30	35	28
2	58	49	32	34	32	29	33	29	27	32	31	30	31	37	31	34	38	36	43	59
5	39	53	46	37	38	44	38	37	36	35	38	47	33	64	53	45	95	58	102	96
10	51	54	43	37	45	37	40	35	44	55	43	58	74	45	106	90	174	71	176	149
20	55	49	43	54	73	65	63	65	101	82	112	109	100	83	122	113	213	319	259	195
50	62	59	55	74	61	91	164	96	45	231	161	271	326	382	273	481	554	319	743	469
100	52	75	94	86	132	193	183	88	173	275	473	464	615	336	694	1175	1102	602	1415	2031

	100	200	300	400	500	600	700	800	900	1000	1100	1200	1300	1400	1500	1600	1700	1800	1900	2000
1	1	3	8	14	20	26	36	44	54	66	83	97	119	134	145	186	208	237	262	300
2	3	9	17	26	42	56	80	85	111	139	169	196	241	271	302	346	414	460	535	626
5	10	26	45	69	102	140	189	209	277	357	423	508	569	688	775	871	1014	1149	1438	1570
10	23	48	70	123	199	280	384	407	573	683	853	956	1137	1360	1534	1729	2090	2368	3112	3062
20	46	96	156	219	427	562	716	825	1143	1383	1707	2062	2300	2737	3146	3623	4160	4657	5952	6020
50	85	155	367	616	1008	1360	1859	2126	2974	3292	4392	4891	6005	6551	7671	8820	10406	11618	14276	15306
100	204	485	797	1283	2063	2948	3872	3988	6144	6779	8666	10090	11644	13339	15298	17678	21186	23815	28134	30729

Figure 4. 18. Comparison of |*B.R*| when (|*B.R*|, |*A.R*|)=(2,1)

	100	200	300	400	500	600	700	800	900	1000	1100	1200	1300	1400	1500	1600	1700	1800	1900	2000
1	69	80	65	52	40	44	45	36	27	28	38	50	37	30	34	29	44	40	34	43
2	90	88	53	56	53	50	45	50	36	39	42	44	52	46	42	49	57	44	39	52
5	94	82	65	56	55	59	45	33	36	45	67	58	56	63	53	68	59	151	94	86
10	73	85	59	48	38	59	53	46	60	64	85	88	74	80	102	113	129	143	184	99
20	84	126	72	69	57	55	83	63	91	127	87	151	130	253	110	103	346	298	326	228
50	67	94	80	75	108	90	191	180	74	302	369	507	217	449	262	685	635	396	265	332
100	82	113	125	96	212	129	130	147	352	272	638	806	700	710	1349	1473	1507	1344	1801	909

	100	200	300	400	500	600	700	800	900	1000	1100	1200	1300	1400	1500	1600	1700	1800	1900	2000
1	2	4	10	13	21	29	38	47	61	77	101	125	159	146	185	200	233	262	272	339
2	3	10	17	26	39	57	80	97	132	145	222	259	328	323	344	411	447	501	551	667
5	7	23	49	65	107	136	188	236	299	377	537	581	858	919	911	970	1100	1223	1339	1690
10	20	32	90	121	225	251	392	437	591	765	1177	1359	1474	1721	1769	1970	2226	2396	3153	3432
20	37	68	169	256	394	547	792	958	1343	1794	2309	2767	3156	3410	3538	4055	4603	5081	5948	6559
50	87	157	384	652	956	1342	1881	2400	3321	4763	5835	6311	6830	7230	10238	9887	11445	12589	16773	15233
100	235	345	951	1409	2056	2689	4028	4689	5990	9632	11434	13244	13654	14175	18486	19385	22378	25159	31372	32287

Figure 4. 19. Comparison of |*B.R*| when (|*B.R*|, |*A.R*|)=(3,1)

	100	200	300	400	500	600	700	800	900	1000	1100	1200	1300	1400	1500	1600	1700	1800	1900	2000
1	59	39	34	31	16	25	22	14	18	18	14	23	14	15	13	24	18	18	16	28
2	36	40	34	28	17	16	21	25	39	16	17	22	24	32	11	18	39	15	32	29
5	48	43	30	26	21	32	21	29	24	35	27	33	35	41	34	51	38	60	79	30
10	27	46	29	33	35	44	13	35	27	49	60	40	64	90	112	49	155	50	129	126
20	40	28	33	28	41	32	55	60	94	114	87	55	139	181	96	194	276	168	175	257
50	45	52	61	39	41	65	75	85	133	151	160	268	143	218	338	422	331	327	536	915
100	50	55	76	147	147	102	228	243	288	327	403	634	307	578	1150	295	614	1431	1578	2009

	100	200	300	400	500	600	700	800	900	1000	1100	1200	1300	1400	1500	1600	1700	1800	1900	2000
1	1	3	9	12	19	25	36	44	56	73	86	98	121	137	157	179	208	224	261	264
2	1	5	15	21	35	52	73	86	113	144	173	199	243	274	315	352	410	442	503	546
5	11	18	32	62	89	126	170	216	289	364	439	508	617	638	787	898	1037	1107	1234	1379
10	12	37	86	103	185	265	344	452	602	726	854	1016	1248	1359	1593	1762	2106	2290	2551	2689
20	22	84	188	249	398	472	723	875	1156	1380	1735	1962	2482	2635	3211	3509	4118	4437	5144	5325
50	54	210	466	552	852	1240	1701	2310	3021	3443	4267	4839	5953	7124	8586	8830	10320	10875	12290	13712
100	201	281	747	1355	1984	2428	3527	4267	6452	6652	8468	9501	12269	13512	16257	17342	21176	22093	24875	27107

Figure 4. 20. Comparison of |*B.R*| when (|*B.R*|, |*A.R*|)=(4,1)

Figure 4.28-4.31 shows the actual number of T_S (upper part) and T_D (lower part) with the increase of *m* and *C* when (|*B.R*|, |*A.R*|) is (1,1), (2,1), (3,1), and (4,1) respectively. The yellow shaded area does not increase too much with the increase of |*B.R*|. And the yellow shaded area covers only a very small part of the entire rectangular area.

	100	200	300	400	500	600	700	800	900	1000	1100	1200	1300	1400	1500	1600	1700	1800	1900	2000
1	27	25	26	26	25	21	21	23	20	20	21	21	22	26	24	24	20	23	26	22
2	27	26	27	27	27	22	21	22	22	18	23	30	19	36	29	26	25	43	39	39
5	27	26	27	31	24	26	23	31	29	40	36	43	30	41	41	55	32	61	68	92
10	28	32	28	31	40	34	37	39	44	59	74	47	46	65	123	118	124	78	184	97
20	27	31	39	47	39	39	45	48	71	89	87	94	152	92	110	108	237	211	332	307
50	35	40	67	41	59	82	106	158	126	185	207	247	231	363	580	364	594	570	642	553
100	38	36	60	100	56	116	139	200	416	205	249	729	543	646	604	1627	1512	913	1197	1434

	100	200	300	400	500	600	700	800	900	1000	1100	1200	1300	1400	1500	1600	1700	1800	1900	2000
1	1	3	8	14	19	24	36	45	56	67	84	101	123	153	191	194	204	230	247	275
2	2	5	19	27	43	52	73	87	110	138	176	202	239	266	365	396	420	456	500	573
5	7	18	47	70	100	130	186	228	292	326	420	504	595	717	883	1003	1026	1115	1275	1404
10	15	44	84	147	213	261	354	446	562	700	884	1010	1144	1411	1696	1976	2027	2238	2559	2900
20	46	68	184	295	413	489	700	872	1147	1392	1717	1949	2335	2763	3559	3913	4074	4493	4837	5613
50	125	196	397	632	1076	1311	1807	2368	2931	3663	4264	5005	6084	6900	9388	10311	10317	11751	12696	14210
100	194	279	842	1425	2089	2867	3610	4615	5692	7268	8705	10076	12507	13862	18733	19972	20278	23687	23627	27565

Figure 4. 21. Comparison of |*A.R*|, |*B.R*| when (|*B.R*|, |*A.R*|)=(1,1)

	100	200	300	400	500	600	700	800	900	1000	1100	1200	1300	1400	1500	1600	1700	1800	1900	2000
1	175	192	181	146	138	136	127	149	172	171	184	166	207	198	223	223	243	267	283	326
2	186	166	196	166	160	153	175	185	205	208	235	228	300	324	358	392	445	424	531	524
5	181	183	196	211	202	271	266	296	341	352	492	452	614	664	741	722	987	1016	1053	1174
10	153	200	229	240	262	341	402	466	587	652	842	935	1098	1198	1349	1610	1745	1900	2078	2368
20	169	255	265	336	395	481	660	785	985	1066	1535	1714	2070	2115	2569	2882	3582	3785	4511	4164
50	252	294	448	676	849	1171	1601	2012	2413	3100	3826	4167	4508	6031	6790	7098	9334	9056	10875	11457
100	255	408	717	928	1343	1838	3129	3595	4876	5629	7150	7883	10009	10071	13367	14371	18306	20580	21122	24027

	100	200	300	400	500	600	700	800	900	1000	1100	1200	1300	1400	1500	1600	1700	1800	1900	2000
1	2	6	15	23	34	48	66	85	116	137	176	186	227	273	313	326	387	435	489	509
2	6	12	31	47	66	90	132	160	218	273	340	373	453	530	635	677	800	868	981	1013
5	15	27	72	125	162	252	338	441	576	652	867	901	1161	1297	1554	1621	1977	2130	2373	2520
10	28	59	145	218	366	486	714	829	1123	1294	1663	1868	2228	2588	3034	3375	3901	4280	4741	5158
20	60	129	301	465	741	965	1349	1720	2202	2497	3355	3714	4536	5131	5818	6939	7852	8589	9558	10211
50	180	382	740	1186	1757	2430	3534	4462	5853	6815	8153	9189	11444	12929	15161	16468	19251	21155	24054	25779
100	268	658	1547	2140	3646	4636	6841	9061	11659	12986	16905	19236	22620	25237	30286	32794	39739	43903	46928	51630

Figure 4. 22. Comparison of |*A.R*|, |*B.R*| when (|*B.R*|, |*A.R*|)=(2,2)

	100	200	300	400	500	600	700	800	900	1000	1100	1200	1300	1400	1500	1600	1700	1800	1900	2000
1	67	108	80	91	73	85	97	122	140	167	205	218	247	282	311	334	411	431	473	530
2	100	88	123	89	114	143	178	216	258	263	327	393	453	518	571	671	774	783	920	1005
5	100	82	159	159	215	251	352	415	511	646	760	890	1093	1171	1323	1655	1827	1934	2266	2441
10	72	120	166	223	336	449	560	709	1055	1112	1486	1874	2007	2453	2843	3096	3712	4182	4779	4950
20	92	132	276	374	684	817	1129	1420	2028	2455	2924	3456	4472	4883	5326	6137	7998	7686	8593	10149
50	137	236	587	878	1325	2166	2941	3739	4901	5992	7793	8921	10174	11619	13681	16146	18017	19045	22630	25175
100	215	522	1032	1578	2949	3497	5591	6987	9497	11971	14801	17179	21645	24161	28783	30557	35048	39532	44786	50224

	100	200	300	400	500	600	700	800	900	1000	1100	1200	1300	1400	1500	1600	1700	1800	1900	2000
1	3	8	22	34	54	67	100	126	162	194	243	296	350	402	468	538	598	642	772	820
2	6	18	43	66	105	146	206	248	348	402	502	590	686	812	949	1038	1237	1313	1494	1606
5	15	46	111	172	258	349	519	620	813	997	1254	1422	1734	1969	2371	2657	2956	3245	3853	4052
10	35	88	212	329	517	728	1022	1246	1683	1945	2446	3023	3435	4073	4732	5176	6289	6585	7628	7868
20	90	169	424	643	1093	1413	2021	2493	3383	3982	4981	5625	7125	8134	9183	10559	12488	12983	14796	16538
50	178	416	1056	1672	2537	3726	5133	6260	8236	9967	13172	14617	17468	19989	22936	27069	30424	32375	37177	41405
100	390	1031	2101	3088	5345	6671	10499	12738	16015	19536	25415	29578	36095	40799	46016	53221	61321	64017	75995	83832

Figure 4. 23. Comparison of |*A.R*|, |*B.R*| when (|*B.R*|, |*A.R*|)=(3,3)

Figure 4.32-4.34 shows the actual number of T_S (upper part) and T_D (lower part) with the increase of *m* and *C* when (|*B.R*|, |*A.R*|) is (1,1), (2,2), and (3,3), respectively. The trend of the yellow shaded area is similar to the trend of the yellow shaded area in Figures 12-15, since only |*A.R*| will affect the size of the yellow shaded area.

Conclusion:

(1) Only |*A.R*| will affect the complexity of our simplification approach.

(2) Even with the largest $|A.R|$ (which is 4) we simulated, T_D becomes larger than T_S when m is larger than 1 or C is larger than 1400, which is we believe is common in practice.

In summary, our simplification approach greatly simplifies the distributed proof procedure in practical role-based environment.

4.4 THE POLICY INTEGRATION COMPONENT

In this section, we propose our novel Policy Integration approach to facilitate the authorized *rar* while preserving the *principle of security*, as well as our novel administrative model to facilitate role-based administration of those operations required in the policy integration.

4.4.1 Policy Integration

After an *uar* has been authorized by the Trust Management component, we need to facilitate this interoperation such that the requesting user can actually acquire the requested resources. Recall that in section 3.2 we have shown that two types of violations, i.e. *cyclic inheritance conflicts* and *violations of SoD* could be introduced when there are multiple authorized interoperations. And no existing approaches focusing on removing these violations in the global policy can be applied here since there is no global policy in *loosely-coupled* environments.

Motivated by this, we propose a novel policy integration approach that uses the special semantics of hybrid hierarchy to preserve the principle of security. As discussed before, the *cyclic inheritance conflicts* occur when there exists cycles among authorized interoperations and local hierarchical relations. For *violation of SoD,* we have shown that if we represent an SoD constraint using bi-directional arrows then the *violation of SoD* is also formed by the cycle among authorized interoperations and local hierarchical relations. Hereafter, we refer to such a cycle as *inheritance cycle* and discuss how to detect and remove such cycles. Note that all the previous discussion about inheritance cycles assumes that the standard hierarchy is used to facilitate an authorized interoperation. That is, for an authorized $rar=<R_1, R_2>$, we make each role in R_1 senior to every role in R_2. In a standard hierarchy, the permissions are inherited upwards through all the hierarchical relations. This is the underlying reason why an inheritance cycle would cause those two types of violations. Therefore, we propose to use the specific semantics of the hybrid hierarchy to facilitate authorized interoperations and prevent such violations.

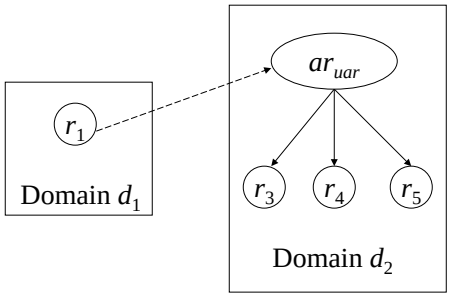
Definition 4.14 (Cycle and inheritance cycle in Hybrid Hierarchy): *In a hybrid hierarchy, a path* $P=(r_1, r_2, \ldots r_n, r_{n+1})$ *is a cycle iff* $r_1=r_{n+1}$*, and a cycle* $C=(r_1, r_2, \ldots r_n, r_{n+1})$ *is an inheritance cycle iff.* $\nexists i,j = 1,\ldots,n$ *such that* $(r_i, r_{i+1})=' \geq_i '$, $(r_j, r_{j+1})=' \geq_a '$, *and* $i>j$

According to Lemma 2.1, if a cycle contains an *I*-relation preceding an *A*-relation, the users of the roles before the *I*-relation cannot acquire the permissions of the roles after the *A*-relation. Therefore, we define the inheritance cycle in hybrid hierarchy as the cycle that does not contain

an *I*-relation preceding an *A*-relation. It is easy to see that the permissions of any role in the inheritance cycle can be inherited by any other role in the cycle, and this property does not hold for non-inheritance cycles in the hybrid hierarchy.

For each authorized *uar*, we create an *access role* in the resource providing domain for the requesting domain to access its resources, as defined below:

Definition 4.15 (Access Role): *Given an authorized* $uar=<R_u, R_{dest}>$*, the access role of this request,* ar_{uar}*, is a newly created role such that* $\forall\ r_1 \in R_u, r_2 \in R_2$*, we make* $r_1 \geq_a ar_{uar} \geq_i r_2$.



(*a*): $uar=<\{r_1\}, \{r_3, r_4, r_5\}>$

Figure 4. 24. the use of access role and hybrid hierarchy to facilitate interoperation

Figure 4.35 shows an example of an *access role*. We can easily verify that the users of R_u can acquire the permissions associated with $R_2=\{r_3, r_4, r_5\}$ by activating ar_{uar}, so the *uar* has been facilitated.

Next we show how such policy integration using *access role* preserve the *principle of security*. Figure 4.36(a) shows an example of how access role is used to prevent *cyclic inheritance conflicts*. If we directly link (r_2, r_3) and (r_4, r_1) using standard hierarchical relation, there is an inheritance cycle as shown in Figure 2.5(a). However, by using *access role* and hybrid hierarchy, we can see that the inheritance cycle does not exist even if there is a cycle in Figure 4.36(a). Consider the cycle of (r_1, r_2, ar_{uar1}, r_3, r_4, ar_{uar2}, r_1). In d_1 the users of r_2 cannot acquire the permissions of r_1 since there is an *I*-relation preceding an *A*-relation in the path. Figure 4.36(b) shows an example of how access role is used to prevent violations of SoD. If we directly link (r_1, r_3) and (r_3, r_2) using standard hierarchical relation, there is an inheritance cycle as shown in Figure 2.5(b). However, by using *access role* and hybrid hierarchy, we can see that the inheritance cycle does not exist even if there is a cycle in Figure 4.36(b). Consider the cycle of (r_1, ar_{uar1}, r_3, ar_{uar2}, r_2, r_1). In d_1 the users of r_1 cannot acquire the permissions of r_2 since there is an *I*-relation preceding an *A*-relation in the path. Therefore, the SoD constraint defined over r_1 and r_2 is not violated. More formally, we have:

Theorem 4.12: *Assume that each individual domain employs RBAC with hybrid hierarchy, and assume we facilitate the interoperation in the following way:*

An $uar = < R_u, R_{dest} >$ *is authorized* $\rightarrow \exists\, ar_{uar}$ *s.t.* $\forall\, r_1 \in R_u, r_2 \in R_{dest}, r_1 \geq_a ar_{uar}$ *and* $ar_{uar} \geq_i r_2$

Then, there exists no inheritance cycle in the environment

Proof:

For any cycle $C=(r_1, r_2, \ldots r_n, r_{n+1})$*, if all roles in the cycle are from the same domain*, C is not an inheritance cycle since each individual domain contains no inheritance cycle. If not all roles in the circle are from the same domain, there must be at least a pair of interoperation relations (one going out from the domain of r_1, and the other going back into the domain of r_1 to form a cycle), and such interoperation relations are constructed according to the description in Theorem 4.12. Without losing the generality, we assume $r_i \geq_a r_{i+1} \geq_i r_{i+2}$, and $r_j \geq_a r_{j+1} \geq_i r_{j+2}$, and $i+2>j$. We can easily see that there is an *I*-relation $r_{i+1} \geq_i r_{i+2}$ *precedes an A*-relation $r_j \geq_a r_{j+1}$ in the cycle *C*. Therefore, C is not an inheritance cycle according to Definition 4.14. ■

Theorem 4.12 proves that if we use the proposed policy integration approach by linking the *access role* through hybrid hierarchy, the *principle of security* will be implicitly preserved regardless of the specific interoperation needs.

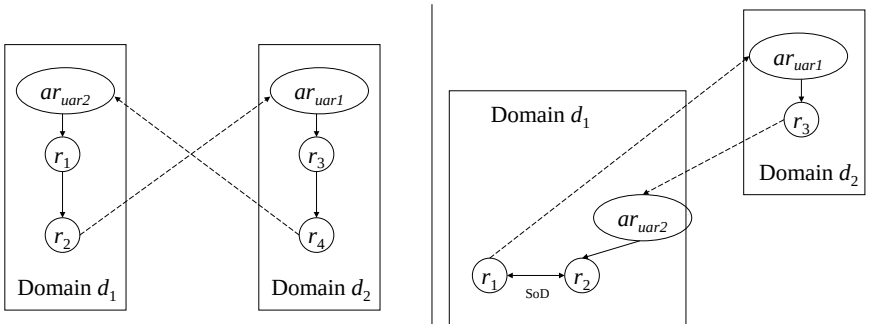


Figure 4. 25. Using access role to prevent (a) cyclic inheritance conflicts; (b) violations of SoD

According to our policy integration approach, after an *uar* is authorized, the requesting users need to activate the *access role* first and then acquire the permissions of the requested roles. For example, after *uar* = <Alice, {HospitalA.HealthCareWorker}, {HospitalB.Doctor}> has been authorized by the Trust Management component, Hospital B adds a new role ar_{uar}, as well as two hierarchical relations HospitalA.HealthCareWorker $\geq_a$ ar_{uar} and ar_{uar} $\geq_i$ HospitalB.Doctor to its local policy. Alice can then acquire permissions of HospitalB.Doctor through the hierarchical relations.

4.4.2 Administrative Model

Updating the interoperation policy given an authorized *rar* involves the following sequence of ` operations to the local RBAC policy: AddRole(ar_{rar}), AddEdge($rar.r_{req}$, ar_{rar}, *A*), $\forall r$ in $rar.R_{dest}$, AddEdge(ar_{rar}, *r*, *I*). To support evolution of RBAC policies, administration of RBAC becomes more and more important. The use of role itself to manage the RBAC policies has become an appealing idea recently. Sandhu *et al.* [15] have proposed an ARBAC97 (Administrate RBAC ‘97) model consisting of URA97 (User-Role Assignment ’97), PRA97 (Permission-Role Assignment ’97), and RRA97 (Role-Role Assignment ’97) model, which use RBAC to manage RBAC policies. They further extended this model to ARBAC99 [16] and ARBAC02 [17]. Crampton *et al.* [18] have developed a SARBAC (Scoped Administration model for RBAC) model using the concept of administrative scope. SARBAC has been known to be capable of

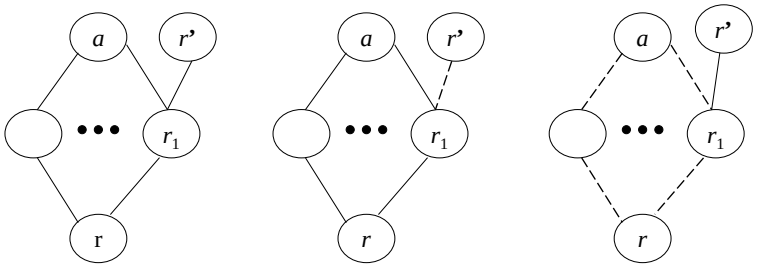
addressing several shortcomings of ARBAC model and is better in terms of completeness, simplicity, practicality and versatility. However, none of the existing role-based administration models can deal with hybrid role hierarchy. Here, we propose an extension of the popular SARBAC model, SARBAC-HH, which is able to support role-based administration in presence of hybrid hierarchy.

Since administrative scope is the core idea in SARBAC, we need to first extend it in presence of hybrid hierarchy in order to make SARBAC applicable to hybrid hierarchy. As discussed earlier, a role *r* can be administrated under another role *a* if and only if all path upwards from *r* go through *a*. On the contrary, suppose there is a path upwards from *r* that doesn't go through *a*, and instead, goes through role *r*'. Here *a* and *r*' have no relation between them, but both of them are related to *r*. If *a* makes some changes to *r*, then it would also affect *r*'. So *a* should not be allowed to administer *r*. Note that in a standard hierarchy, if there's a "path" between two different roles r_1 and r_2, then r_1 and r_2 must be hierarchically related, i.e. $r_1 \geq r_2$ or $r_2 \geq r_1$. Therefore, the definition of administrative scope closely relies on finding the direct and indirect relation in the path between r_1 and r_2. Based on the definition of derived relation $\geq_d$ earlier, we re-define the administrative scope as follows:

DEFINITION 4.18 (*Administrative Scope in Hybrid Hierarchy*): *The administrative scope for role a in hybrid hierarchy, $S_{HH}(a)$ is defined as follows:*

$S_{HH}(a) = \{r \in R: r \leq_d a\ ,\ \uparrow r \setminus \uparrow a \subseteq \downarrow a\}$**,** *Where,* $\uparrow r = \{x \in R: x \geq_d r\}$, $\downarrow r = \{x \in R: x \leq_d r\}$.

Similarly, the *strict* administrative scope would be $S_{HH}^{+}(r) = S_{HH}(r) / \{r\}$. If $r \in S_{HH}^{+}(a)$, we call *a* as an *administrator* of *r*. Figure 4.37 illustrates the difference between original administrative scope in SARBAC and our administrative scope in SARBAC-HH. Note that the structure of the three hierarchies is exactly the same and the only difference is the type of the hierarchy. Figure 4.37(a) is a standard hierarchy; Figures 4.37(b) and 4.37(c) are hybrid hierarchies. In Figure 4.37(a), role *a* cannot administer role *r* because *r'* is senior to *r* but is not junior to *a*. In figure 4.37(b), role *a* cannot administer role *r* either, since *r'* is "conditionally" senior to *r* but is not junior to *a*. In figure 4.37(c), however, role *a* can administer role *r* because there's no relation between *r* and *r'* even if there seems to be a "path" between them. Note that in Figure 4.37(c), *a* cannot administer r_1 because of *r'*. However, in the entire hierarchy, there may exist another role (e.g. the senior role of both *a* and *r'*) which can administer r_1. Next we will show that our definition of administrative scope keeps all the properties of the original one.



$r \notin S^{+}(a)$ (b) $r \notin S_{HH}^{+}(a)$ (c) $r \in S_{HH}^{+}(a)$

Figure 4. 26. Administrative Scope in SARBAC and SARBAC-HH

Flexibility: Our administrative scope is also determined by the role hierarchy itself, and changes dynamically as the hierarchy changes. This is similar to the original SARBAC model and is in contrast to Sandhu's ARBRAC97 model, where administration largely depends on the *can-modify* relation [15].

Decentralization and Autonomy: we illustrate this by proving the following proposition:

PROPOSITION 4.1 ***(Line Manager in Hybrid Hierarchy)****: In a hybrid hierarchy, if r has an administrator then the set of administrators of r has a unique minimal administrator, which we refer to as the line manager of r.*

The line manager can serve as a "local" administrator in the hybrid hierarchy. Therefore, our administrative scope keeps the decentralization and autonomy properties which is essential in large enterprise-wide RBAC systems. With this notion of extended administrative scope in presence of hybrid hierarchy, we present our extensions of RHA and URA next.

Table 4. 5. Hierarchical Operations in SARBAC-HH

Operation	**Success Conditions**
AddRole(*a*, *r*, $\Delta_a r$, $\triangledown_a r$, $\Delta_i r$, $\triangledown_i r$)	$\Delta_a r \subseteq S_{HH}^{+}(a)$, $\triangledown_a r \subseteq S_{HH}(a)$, $\Delta_i r \subseteq S_{HH}^{+}(a)$, $\triangledown_i r \subseteq S_{HH}(a)$
DeleteRole(*a*, *r*)	$r \in S_{HH}^{+}(a)$
PartitionRole(*a*, *r*)	$r \in S_{HH}^{+}(a)$
AddEdge(*a*, *c*, *p*, *type*)	$c, p \in S_{HH}(a)$
DeleteEdge(*a*, *c*, *p*)	$c, p \in S_{HH}(a)$
ChangeEdge(*a*, *c*, *p*, *type*)	$c, p \in S_{HH}(a)$

Besides the four operations defined in SARBAC-RHA as shown in Table 2.1, we further add two operations: *PartitionRole*() and *ChangeEdge*(), which we believe are necessary in hybrid hierarchy. The success conditions of each operation are shown in Table 4.5, where $\Delta_a r$ is set of immediate *A*-juniors of the role *r*, $\nabla_a r$ is the set of immediate A-seniors of role *r*, $\Delta_i r$ is the set of immediate *I*-juniors of role *r*, and $\nabla_i r$ is the set of immediate *I*-seniors of role *r*, as shown in Figure 4.38. The semantics of *ChangeEdge*(*a, c, p*) is straight forward since there are three types of edges in hybrid hierarchy. In fact, we can use *AddEdge*() and *DeleteEdge*() operation to perform *ChangeEdge*(). That is, first delete the old edge, and then add the edge with the new type. The semantic of *PartitionRole*() is complex. Specifically, we can partition a given role vertically, horizontally, or both [14].

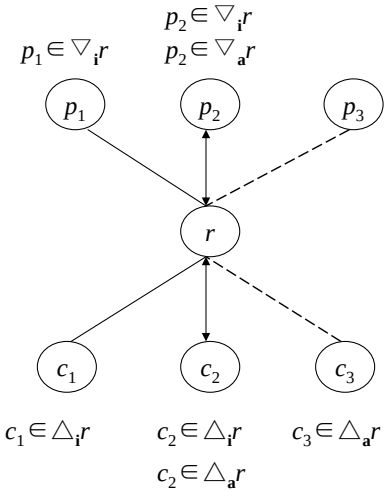


Figure 4. 27. Parameters in AddRole

Ideally, after each operation, we should keep the original semantics as much as possible. For example, when we want to delete a role *r*, which has an immediate senior *s* and an immediate junior *j*, we need to maintain the original relation between *s* and *j* after the operation. Moreover, to make sure same users can acquire same permissions after deleting the role, we need to reassign permissions of *r* to other roles and reassign users of *r* to other roles. This is a very challenging problem and is beyond the scope of this thesis. Interested readers are referred to [14], where Joshi *et al.* analyze these issues in greater detail. In the rest of this section, we will focus on maintaining the administrative scope during those operations. Specifically, two conditions need to be satisfied:

C_1: After AddRole() and PartitionRole() operations, the new role(s) should be within the administrative scope of a. C_2: After each operation, the original roles' administrators should not be changed.

It is obvious that C_1 is satisfied according to our definition. Since all the seniors of the new role should be administered by *a*, the new role itself is also administered by *a*. The condition C_2 is also satisfied for all operations. This conclusion is not obvious with *ChangeEdge*() operation, since the operation itself may change the relation between roles and thus affect the administrative scope, as shown in Figure 4.39. In Figure 4.39(a), $r \in S^+_{HH}(a)$. If we change the edge (r, r_1) to the *I*-type, as Figure 4.39(b) shows, $r \notin S^+_{HH}(a)$ now. However, in Figure 4.39(a), r_1 is not administered by *a*, so the *ChangeEdge*() operation fails. Therefore, if *ChangeEdg*e() operation succeeds, it is guaranteed that it will not affect the administrators of all the original roles.

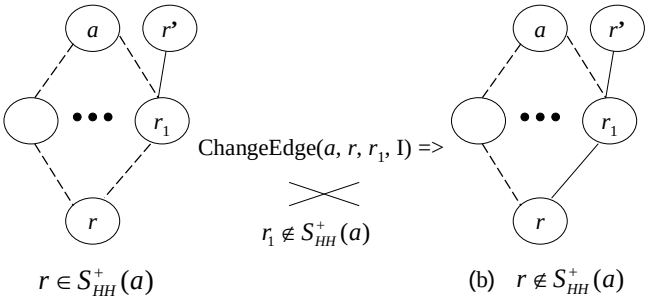


Figure 4. 28. The ChangeEdge operation won't succeed

The key operations in SARBAC-URA are shown in Table 4.6, and the permission-role assignment operations in SARBAC-PRA are similar. We first show that there is an ambiguity in the semantics of user-role assignment and permission-role assignment in the original SARBAC. We then show that our model can solve this ambiguity smoothly by redefining those operations in presence of hybrid hierarchy. To illustrate these, we first review an important concept in SARBAC, the SARBAC constraint, as follows: Let $R' = \{r_1, \ldots, r_k\}$ be a subset of R and let $\wedge R'$ denote $r_1 \wedge \ldots \wedge r_k$.

DEFINITION 4.19 (*SARBAC constraint*) *A SARBAC constraint has the form $\wedge C$, where $C \subseteq R$. A SARBAC constraint $\wedge C$ is satisfied by a user u if $C \subseteq \downarrow R(u)$. A SARBAC constraint $\wedge C$ is satisfied by a permission p if $C \subseteq \uparrow R(p)$, where for any $Y \subseteq X$, $\uparrow Y = \{x \in X: \exists y \in Y$ such that $x \geq y\}$, and $\downarrow Y = \{x \in X: \exists y \in Y$ such that $x \leq y\}$.*

Let's first analyze under what situation a user will satisfy a constraint. An example standard hierarchy is shown in Figure 4.40 (the same example used in the original SARBAC paper). According to definition 4.19, the constraint $PE_1 \wedge QE_1$ is satisfied by any user assigned to both

PE_1 and QE_1, and by any user assigned to either PL_1 or *DIR*. The semantics here is that any user assigned to either PL_1 or *DIR* is also a member of PE_1 and QE_1, thus satisfies the $PE_1 \wedge QE_1$ constraint. Obviously, the author of SARBAC implicitly assumes the hierarchy relation in any monotype hierarchy as "Is-a" relation [11]. That is, $x \geq y$ means any user assigned to x is also a member of y. For example, the leader of a team is also a member of the team. However, the semantics of monotype hierarchy have long been argued as ambiguous [10, 11, 13]. The hierarchical relation in a monotype hierarchy could be "Is-a", "Supervision", or "Activation" [11]. The use of hybrid hierarchy can solve this ambiguity accordingly by including three types of hierarchical relations. The above "Is-a" relation is essentially "IA" relation in the hybrid hierarchy, since x "is" y means any user assigned to x should be able to acquire all permissions assigned to y through x, and should also be able to activate y. Because whether a user satisfies a constraint depends on the definition of $\downarrow Y$ in Definition 2.3, we re-define it as:

$$\forall Y \subseteq X, \downarrow Y = \{x \in X: \exists y \in Y \text{ such that } x \leq y\} \qquad (1)$$

Note that the definition looks same as before, but here the symbol $\leq$ clearly means the IA-relation in hybrid hierarchy.

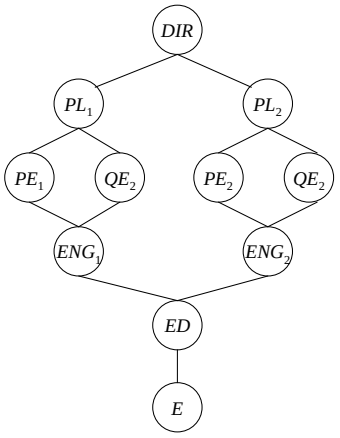


Figure 4. 29. An example standard hierarchy

Next let's analyze under what condition a permission will satisfy a constraint. In Figure 4.40, according to definition 4.19, the constraint $PE_1 \wedge QE_1$ is satisfied by any permission assigned to both PE_1 and QE_1, and by any permission assigned to either ENG_1 or ED or E. The semantics here is that any permission assigned to ENG_1 or ED or E is also in the permission set of PE_1 and QE_1, thus satisfies the $PE_1 \wedge QE_1$ constraint. In other words, $x \geq y$ means $P(y) \subseteq P(x)$, where $P(r)$ is the permission set available through r. Obviously, the author of SARBAC implicitly assumes the hierarchy relation in any monotype hierarchy as "Permission Inheritance" relation, which is in conflict with previous assumption of "Is-a" relation. We believe this ambiguity comes from the ambiguity of the monotype hierarchy, as claimed by many researchers [10, 11, 13]. Again, the use of hybrid hierarchy can solve this smoothly by using "I-relation". Specifically, since whether a permission will satisfy a constraint depends on the definition of $\uparrow Y$ in Definition 2.3, we re-define it as:

$$\forall Y \subseteq X, \uparrow Y = \{x \in X: \exists y \in Y \text{ such that } x \geq_i y\} \qquad (2)$$

Note that here we use the $\geq_i$ relation. Given the new definition of $\downarrow Y$ and $\uparrow Y$, we can define the SARBAC-HH constraint as follows:

DEFINITION 4.20 (*SARBAC-HH constraint*): *A SARBAC-HH constraint has the form $\wedge C$ for some $C \subseteq R$. A SARBAC07 constraint $\wedge C$ is satisfied by a user u if $C \subseteq \downarrow R(u)$. A SARBAC07 constraint $\wedge C$ is satisfied by a permission p if $C \subseteq \uparrow R(p)$, where the symbol $\uparrow$ and $\downarrow$ are defined by (1) and (2).*

The definition implies that the User-Role Assignment is determined by the *IA*-relation in the hybrid hierarchy, while the Permission-Role Assignment is determined by the *I*-relation in the hybrid hierarchy. The user-role assignment operations are the same with SARBAC, as shown in Table 2.2 (permission-role assignment operations are similar).

Now we are able to define the success conditions for the administrative operations required in the Policy Integration component, as shown in Table 4.6. As shown in Table 4.6, we require that all these operations to be done by only the administrators whose administrative scope includes all the requested roles (i.e. R_{dest}). This is straightforward since such operations would make R_{dest} available for external users to assume.

Table 4. 6. Success conditions for operations involved in the Policy Integration component.

Required Operations	**Success Condition**
AddRole(a, ar_{sar}, $\varnothing$, r_{req}, R_{dest}, $\varnothing$)	$R_{dest} \subseteq S_{HH}(a)$
AddEdge(a, r_{req}, ar_{sar}, A)	$R_{dest} \subseteq S_{HH}(a)$
$\forall r \in R_{dest}$, AddEdge(a, ar_{sar}, r, I)	$R_{dest} \subseteq S_{HH}(a)$

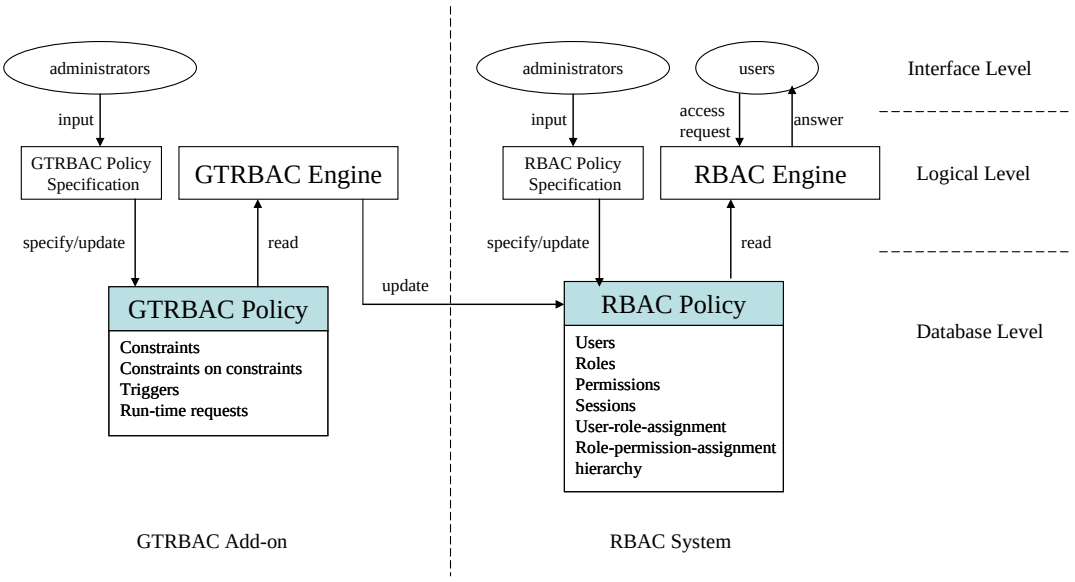


Figure 4. 30. Access control system in the individual domain

4.5 PROTOTYPE

In this section, we present our prototyping of our framework to validate our work. The prototyping of the components included in our framework is straightforward. For each of the component, we have proposed the corresponding algorithm in section 4, and we only need to prototype the algorithm accordingly. However, prototyping the access control system in the

individual domain (i.e. GTRBAC and Hybrid Hierarchy) is more challenging. GTRBAC is a fine-grained model that supports more than 50 temporal constraints. How to enforce such many constraints effectively in a conflict-free way is a big challenge. Unfortunately, the authors of GTRBAC have focused on the theoretical model and have not discussed the enforcement of GTRBAC. In the literature, the only GTRBAC implementation work we are aware of, X-GTRBAC [40], implements the policy specification module only. They mainly focus on how to specify GTRBAC policy using XML, and simply assume that there already exists a GTRBAC module to enforce those constraints without explaining how. To the best of our knowledge, no existing work has been proposed to address the policy enforcement of GTRBAC.

Motivated by this, we propose our novel GTRBAC enforcement engine in this section. The central idea of our work is to enforce all the different types of temporal constraints in a uniform way by generating a predefined set of system operations. The high-level architecture of our GTRBAC engine is shown in Figure 4.41. It describes the relationship among RBAC policy, RBAC engine and GTRBAC engine. Vertically, we divide the access control system in the individual domain into three levels: Interface Level, Logical Level, and Database Level. On interface level, we allow the administrators to specify/update the policies through policy specification modules; we also allow the user to issue access request to the RBAC engine and get the authorization decision from it. In the database level, the RBAC policy is stored as a set of tables in a relational database. In our current implementation, we use 7 tables to store RBAC policy. These tables actually represent the basic RBAC model and Hierarchical RBAC model in

RBAC standard [5]. In the future, we plan to add tables to support Separation of Duty (SoD) and thus implement the Constraint RBAC model as well. The GTRBAC policy is stored in 4 tables in the relational database. These 4 tables store all the constraints supported in GTRBAC model. On the logical level, we have policy specification modules that translate the user inputs to the data structures in the database. The GTRBAC engine and RBAC engine also reside on logical level. As shown before, our GTRBAC system does not need to affect the RBAC engine. The GTRBAC engine, on the other hand, is the most novel part of our system and will be discussed extensively next.

Figure 4.42 shows the working mechanism of our GTRBAC Engine. In general, the GTRBAC engine is responsible for checking all the GTRBAC constraints and updates the RBAC policy accordingly every time it runs. *The novelty of our engine is that we enforce all those different types of constraints in a uniform way by generating a predefined set of system operations.* By doing so, we are able to (1) enforce all those constraints by updating the RBAC policy according to the predefined system operations; and (2) solve the conflicts among original constraints by solving the conflicts among predefined system operations. According to the semantics of GTRBAC constraints, we define 4 pairs of system operations, and the two operations in each pair is the *inverse operation* of each other:

- *user-role assignment* and *user-role de-assignment*
- *role-permission assignment* and *role-permission de-assignment*
- *role enabling* and *role disabling*

- *role activation* and *role de-activation*

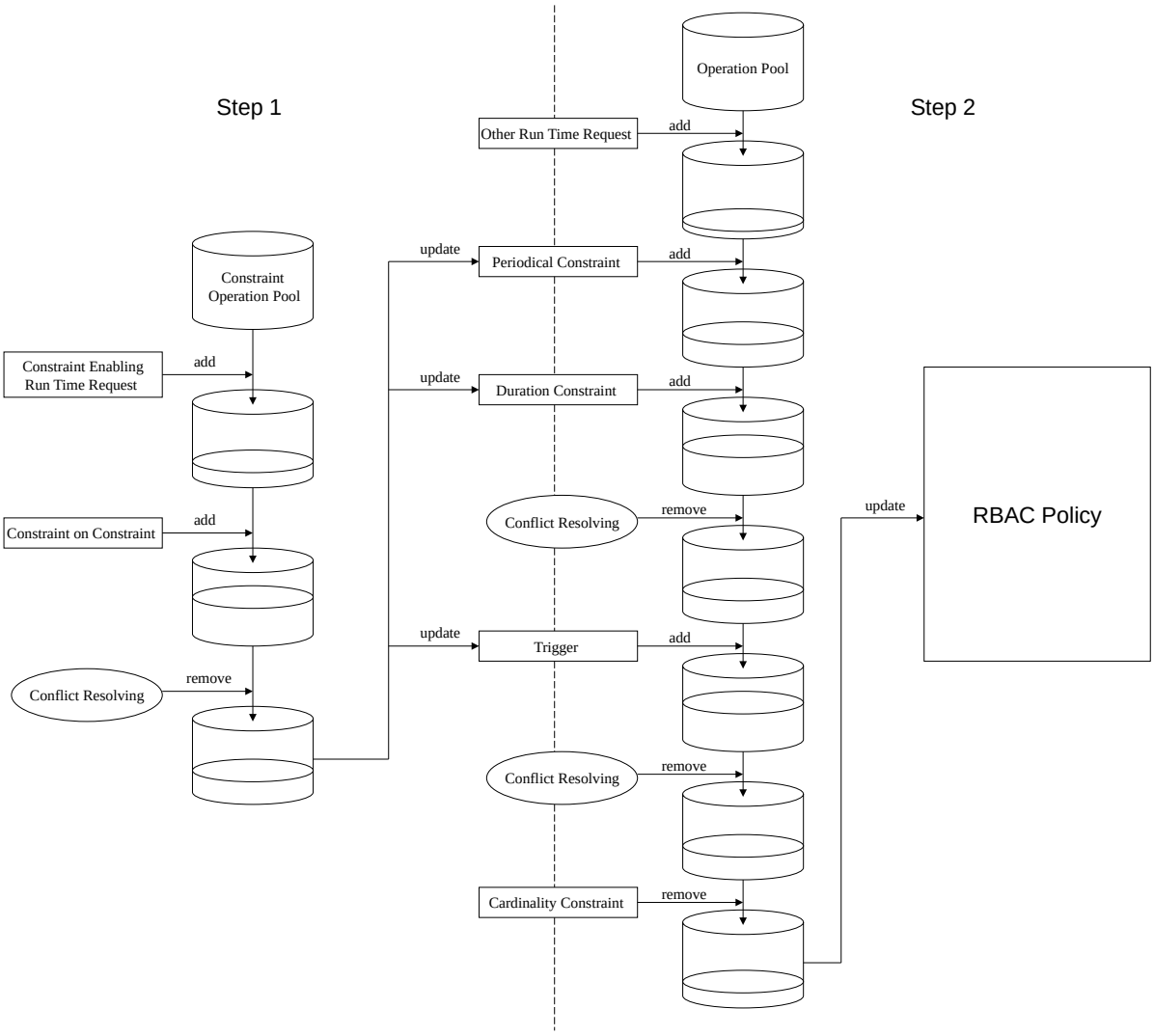


Figure 4. 31. The proposed GTRBAC engine

Next, we describe how we generate the above system operations from different types of constraints, and how we enforce those constraints by enforcing those system operations. We note

that only the enabled constraints should be enforced at the time GTRBAC engine runs. Determining which constraints are enabled is not straight forward because the enabling states of constraints change dynamically. For example, the administrator can issue a run-time request to explicitly enable or disable some constraints. Moreover, the constraint on constraint would also change the enabling state of the corresponding constraint. As a result, the first step of GTRBAC Engine is to check which constraints are enabled, as shown in Step 1 (left side) in Figure 4.42. In particular, we define a data structure called Constraint Operation Pool (COP). COP is a collection of constraint operations, and a constraint operation has the structure of <*mode, constraint name, priority*> (*priority* field will be omitted hereafter for simplicity), where *mode*$\in$ {*enable*, *disable*} and constraint name is a unique identifier of the constraint in the constraint. COP is initialized to be empty every time GTRBAC Engine runs. In this step, our engine checks the corresponding GTRBAC policy (i.e. constraint enabling/ disabling run-time requests, and constraints on constraint) to gradually add constraint operations to COP. For example, if there is a <enable, c_1> run-time request in the run-time requests table, we add <enable, c_1> into COP. After such checking we need to remove the conflicts existing in COP, which will be described later in conflict resolving part. Finally, we update the enabling states of constraints according to each constraint operation in COP. For example, if <enable, c_1> is in COP, we enable constraint c_1 in GTRBAC policy.

The next step (step 2, right side of Figure 4.42) is to check all enabled constraints and update RBAC policy accordingly. Similarly, we define a data structure called Operation Pool

(OP). OP is a collection of our predefined system operations, and an operation has the structure of <*mode*, *username*, *role_name*, *permission_name*>, where *mode* ∈ {activateRole, deactivateRole, assignUser, deassignUser, enableRole, disableRole, assignPermission, deassignPermission} and *username*, *role_name*, *permission_name* are unique identifiers of Users, Roles, and Permissions, respectively (again, *priority* field is omitted). Note that *username*, *role_name*, *permission*_name are all optional according to the specific *mode*. For example, if the *mode* of an operation is "enableRole" then only *role_name* is specified in the operation. OP is also initialized to be empty every time GTRBAC engine runs. In this step, we dynamically add operations into OP by checking run-time requests (except for constraint enabling run-time request which is checked in step 1) and those enabled constraints, or remove operations from it to remove the conflicts, as shown in Figure 4.42. Note that we must remove the conflicts before we check the triggers. This is because triggers will generate new operations according to the existing operations. It makes no sense to let conflicting operations (thus should be removed) to be the inputs of triggers. And we need to run conflict removing again after checking triggers because triggers may generate new operations which could conflict with the existing operations. We also emphasize that we should check the cardinality constraint at the very end of step 2. This is because cardinality constraints are used to remove operations rather than generating operations. For example, assume we had a cardinality constraint <3, activate Doctor> and Doctor has already been activated for 3 times. The operation <activateRole, *Bob*, Doctor> should be removed from OP after checking the cardinality constraint. Therefore, if we put cardinality checking earlier,

then the newly generated operations (e.g. by triggers) have no chance to be checked against them. Finally, we update the RBAC policy according to each operation in OP. For example, if <assignUser, *Bob*, Doctor> is in OP, we assign *Bob* to Doctor in RBAC policy.

Most of the checking shown in Figure 4.42 is straightforward. Next, we only describe how to check a periodicity constraint as an example. The checking rule is simple: if current time is within the periodical expression, we add the corresponding operation into OP. Otherwise we add the inverse operation to OP. For example, consider a periodical constraint < [9am, 9pm], enable Doctor>. If the engine runs at 10pm we add <disableRole, Doctor> into OP.

Now we discuss how we resolve the conflicts in each step shown in Figure 4.42. Generally speaking, two operations are conflicting with each other if they are the inverse operation and apply to the same user, role, or permission. For example, <enableRole, Doctor, priority: *high*> and <disableRole Doctor, priority: *medium*> is a pair of conflicting operations. We implement two rules to resolve the conflicts among operations, as defined in GTRBAC model [9]:

(1) *Higher priority overrides lower priority.* In the above example, Doctor should be enabled since <enableRole, Doctor, priority: *high*> has higher priority.

(2) *Negative* (*e.g. disable*) *overrides positive.* In the above example, if both operations have the same priority then Doctor should be disabled since “disable” is a negative operation and “enable” is positive.

The conflicts of constraint enabling operations in step 1 can be resolved in the same way using

these two rules.

Finally, we discuss when and how the GTRBAC engine should run. In our architecture, we need to run our GTRBAC engine repeatedly to update the RBAC policy dynamically. Consider a constraint <[9am, 9pm], assign *Bob* to Doctor>. Every time the engine runs, it will check the current time against this constraint, and assign *Bob* to Doctor or de-assign *Bob* from it accordingly. Obviously if we run the engine one time a week, then the effect of such constraint cannot be reflected in the system. Therefore, we choose to run the engine every 1 minute in the current implementation. We believe this frequency is high enough to capture all constraints and run-time requests in the system. On the other hand, running the engine in such a high frequency may be a waste of resource since at most time instants no changes will likely be made to the system. For example, assume <[9am, 9pm], assign *Bob* to Doctor> is the only constraint in the system and assume no any run-time requests will be generated. In this very simple case, theoretically we only need to run the engine at 9am and at 9pm to update the corresponding user-role assignment. However, it is very difficult (if not impossible) to predict perfectly when we should run the engine if there are hundreds of constraints and run-time requests (they could even be conflicting). We plan to study how to enhance the performance of our system by partly predicting when we should run the engine in the future work.

Table 4. 7. An example test case of GTRBAC Implementation

RBAC Policy	(assign$_U$ *Ami* to NurseInTraining)
	(*NightTime*, enable NightDoctor)
Periodical Constraints	c_1= (*DayTime*, enable DayDoctor)
	c_2= (*NightTime,* enable NightDoctor)
	c_3= ((*M, W, F*), assign$_U$ *Adams* to DayDoctor)
	c_4 = ((*T, Th, S, Su*), assign$_U$ *Bill* to DayDoctor)
	c_5=(*Everyday between* 10am-3pm,assign$_U$ *Carol* to DayDoctor)
Duration Constraints	c_6 = (2 *hours*, enable NurseInTraining)
	c_7 = (2 *hours*, active$_{R_total}$ NurseInTraining)
Constraints on Constraints	(6 *hours*, c_6)
Triggers	(enable DayNurse → enable c_1)
	(activate DayNurse for *Elizabeth* → enable NurseInTraining)
	(enable NightDoctor → enable NightNurse)
	(disable NightDoctor → disable NightNurse)
Cardinality Constraints	(10, active$_{R_n}$ DayNurse)
	(5, active$_{R_n}$ NightNurse)

5.0 CONCLUSION AND FUTURE WORK

Multidomain environments where multiple organizations interoperate with each other are becoming a reality as seen in the emerging Internet-based enterprise applications. In such an environment, it is a significant challenge to ensure that cross-domain accesses to facilitate information sharing are employed in a secure way. Role Based Access Control (RBAC) models have received much attention as a general approach to access control. A multidomain environment can be characterized into *tightly-coupled* environment and *loosely-coupled* environment. The access control challenges in *loosely-coupled* environments where each individual domain employs RBAC have not been studied adequately in the literature.

In this dissertation, we first show that it is desirable to allow users to issue the interoperation requests in terms of requested permissions rather than requested roles. And the resource-providing domains need to identify a set of its local roles containing the requested permissions for the external users to assume. We have propose three role mapping algorithms to identify a set of roles containing all the requested permissions. Our algorithms can handle the cases when (1) there is exactly matched role set; (2) there is no exactly matched role set but the

principle of least privilege is more important; (3) there is no exactly matched role set but the availability is more important.

Once the initial interoperation requests have been translated into a set of requested roles, the providing domain needs to make decisions on whether to authorize the requests or not based on their local policies and the interoperation requirements. We argue that in role based *loosely-coupled* environments, it is typical that several different users assigned to the same role (or a very small set of related roles) would request to acquire the same external resource several times in a period. Traditional role-based distributed proof approaches (e.g. DCCD) are inefficient in dealing with such type of requests since they all require individual users to prove the requested resource separately. We formally study how to simplify such distributed proof procedure and propose a **Simplify** algorithm based on the policies of the requesting role and the requested role. We formally prove the completeness and soundness of our algorithm. We conduct simulation and run several experiments to very our work. The experiment results show that our algorithm significantly outperforms DCCD when the total number of credentials is sufficiently large, which is very common in practical loosely-coupled environments.

Several researchers have shown that the introduction of global policy in *tightly-coupled* environments could violate the *principle of security*. Although there is typically no global policy in the *loosely-coupled* environment, the existence of multiple authorized interoperations could also violate the *principle of security*. We have proposed a policy integration approach to preserve the principle of security while facilitating the interoperations. Our approach makes use of the

special semantic of hybrid hierarchy to prevent unexpected permission inheritances. We also propose an administrative model for the RBAC model extended with hybrid hierarchy defining which administrators are authorized to make the policy changes required during policy integration.

Finally, we present the prototype of our framework to validate our research. The most challenging part of the prototyping is implementing the GTRBAC model. We have implemented a novel GTRBAC engine that generates a set of pre-defined system operations according to different temporal constraints. The conflicts among those temporal constraints are resolved within those system operations and the corresponding RBAC state is easily updated according to those system operations as well. We also implement the role mapping algorithms, the **Simplify** algorithm, and the Policy Integration module to make it a complete prototype of our proposed framework.

There are several future work related to the research presented in this thesis. First, the work presented in this research is theoretical in nature. Although we have implemented a prototype to validate it, we have not implemented it over real organizations. This requires a comprehensive work related to inter-domain collaborations. For example, how to discover which domain contains the requested permissions through service discovery. There are also a number of future work related to our simplification algorithm. First, if the requesting domain changes its policy within the valid period of an *rar*, the simplified proof of this *rar* may not be valid now. However, it is not straight forward to detect this unless the requesting domain “honestly” notifies

the providing domain. Second, in discussing heuristic 4 we find a trade-off between the completeness and complexity of our **Simplify** algorithm. That is, if we examine every subset of *T*, we can find every existing simplified proof but the algorithm becomes slow. On the other hand, if we only examine some subsets of *T*, we may miss some existing simplified proof but the algorithm will run much faster. How to balance between these two factors is a possible future work future work. Third, although we present our simplification framework in the context of RT_0 language, we believe the general idea of our approach does not rely on any specific policy language and should be applicable generally. Therefore, another future research direction is to apply the idea of our approach to other policy languages.

BIBLIOGRAPHY

[1] L. Gong and X. Qian, “Computational Issues in Secure Interoperation”, *IEEE Trans. Software Eng*., vol. 22, no. 1.

[2] P. Bonatti, S.D.C. Vimercati, and P. Samarati, “An Algebra for Composing Access Control Policies,” *ACM Trans. Information and System Security*, vol. 5, no. 1, Feb. 2002.

[3] S. Dawson, S. Qian, and P. Samarati, “Providing Security and Interoperation of Heterogeneous Systems,” *Distributed and Parallel Databases*, vol. 8, pp. 119-145, Aug. 2000.

[4] B. Shafiq, J. B. D. Joshi, E. Bertino, A. Ghafoor, “Secure Interoperation in a Multi-Domain Environment Employing RBAC Policies”, *IEEE Transactions on Knowledge & Data Eng*, vol. 17, no. 11, pp. 1557-1577, Nov. 2005.

[5] American National Standards Institute Inc. Role Based Access Control, ANSI-INCITS 359--2004, 2004.

[6] R. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, “Role-Based Access Control Models”, *IEEE Computer* 29(2): 38-47, IEEE Press, 1996

[7] D. Ferraiolo, R. Sandhu, S. Gavrila, D. Kuhn, and R. Chandramouli, “Proposed NIST standard for role-based access control,” ACM Transactions on Information and Systems Security, vol. 4, no. 3, pp. 224–274, August 2001.

[8] D. F. Ferraiolo, D. M. Gilbert, and N. Lynch, “An Examination of Federal and Commercial Access Control Policy Needs”, In Proceedings of NISTNCSC National Computer Security Conference, Baltimore, MD, Sep. 1993, pp. 107-116.

[9] J. B. D. Joshi, E. Bertino, U. Latif, and A. Ghafoor, "Generalized Temporal Role Based Access Control Model," *IEEE Transactions on Knowledge and Data Engineering*, Volume 7, Issue 1, Jan. 2005.

[10] R. Sandhu, “Role activation hierarchies”, Proceedings of the third ACM workshop on Role-based access control, Fairfax, Virginia, United States, 1998, pp. 33-40.

[11] J. D. Moffett and E. C. Lupu, “The uses of role hierarchies in access control”, Proceedings of the fourth ACM workshop on Role-based access control, Fairfax, Virginia, United States, 1999, pp. 153-160.

[12] J. B. D. Joshi, E. Bertino, and A. Ghafoor, “Temporal hierarchies and inheritance semantics for GTRBAC”, In Proceedings of the 7th ACM symposium on Access control models and technologies, ACM Press, New York, NY, USA, 74–83.

[13] N. Li, J. W. Byun, E. Bertino, "A Critique of the ANSI Standard on Role-Based Access Control," *IEEE Security and Privacy*, vol. 5, no. 6, pp. 41-49, Nov/Dec, 2007

[14] J. B. D. Joshi, E. Bertino, A. Ghafoor and Y. Zhang, “Formal Foundations for hybrid hierarchies in GTRBAC”, accepted by *ACM Transactions on Information and System Security*.

[15] R. Sandhu, V. Bhamidipati, and Q. Munawer, “The ARBAC97 Model for Role-Based Administration of Roles”, *ACM Transactions on Information and System Security (TISSEC)*, Volume 2, Issue 1, Feb. 1999, pp. 105-135.

[16] R. Sandhu and Q. Munawer, “The ARBAC99 Model for Administration of Roles (1999)”, In Proceedings of 15th Computer Security Applications Conference, Arizona, US, Feb 1999, pp. 229-238.

[17] S. Oh, and R. Sandhu, “A model for role administration using organization structure”, Proceedings of the 7th ACM symposium on Access control models and technologies, Monterey, CA 2002.

[18] J. Crampton, and G. Loizou, “Administrative scope: A foundation for role-based administrative models”, *ACM Transactions on Information and System Security (TISSEC)*, Volume 6, Issue 2, May. 2003, pp. 201-231.

[19] M. Blaze, J. Feigenbaum, and J. Lacy, “Decentralized Trust Management”, IEEE Symposium on Security and Privacy, May 1996.

[20] M. Y. Becker and P. Sewell, “Cassandra: distributed access control policies with tunable expressiveness”, 5th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY), 2004.

[21] A. J. Lee, M. Winslett, J. Basney, and V. Welch, “Traust: A Trust Negotiation-Based Authorization Service for Open Systems,” The Eleventh ACM Symposium on Access Control Models and Technologies (SACMAT 2006), June 2006.

[22] N. Li, J. C. Mitchell, and W. H. Winsborough, “Design of a Role-based Trust-Management Framework”, IEEE Symposium on Security and Privacy, May 2002.

[23] L. Bauer, S. Garriss, and M. K. Reiter, “Distributed proving in access-control systems”, in Proceedings of the 2005 IEEE Symposium on Security and Privacy, pages 81–95, May 2005.

[24] L. Bauer, S. Garriss, M. K. Reiter, “Efficient Proving for Practical Distributed Access-Control Systems”, ESORICS 2007, pp. 19-37.

[25] A. J. Lee, M. Winslett, “Towards an efficient and language-agnostic compliance checker for trust negotiation systems”, ASIACCS 2008: 228-239.

[26] N. Li, W. H. Winsborough, and J. C. Mitchell, “Distributed Credential Chain Discovery in Trust Management”, *Journal of Computer Security* 11(1):35-86, February 2003.

[27] J. B. D. Joshi, R. Bhatti, E. Bertino, and A. Ghafoor, “An Access Control Language for Multidomain Environments”, *IEEE Internet Computing*, Nov-Dec, 2004.

[28] http: //www.hl7.org/.

[29] A. Sheth and J. Larson, “Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases”, *ACM computing Surveys*, vol. 22, no. 3, pp. 183-236, Sep. 1990.

[30] S. Piromruen, J. B. D. Joshi, “An RBAC Framework for Time Constrained Secure Interoperation in Multi-domain Environment,” IEEE Workshop on Object-oriented Real-time Databases (WORDS-2005), 2005.

[31] M. Paolucci, K. P. Sycara, and T. Kawamura, “Delivering Semantic Web Services”, In Proceedings of the International World Wide Web Conference, 2003.

[32] E. Sirin and B. Parsia, “Planning for Semantic Web Services”, In Proceedings of International Semantic Web Conference, Workshop on Semantic Web Services, November 2004.

[33] N. Gooneratne, and Z. Tari, “Matching independent global constraints for composite web services”, In Proceeding of the 17th international Conference on World Wide Web (WWW '08), Beijing, China, Apr. 2008, pp. 765-774.

[34] A. Zisman, G. Spanoudakis, and J. Dooley, “A Framework for Dynamic Service Discovery”, In Proceedings of the 2008 23rd IEEE/ACM international Conference on Automated Software Engineering, Washington, DC, Sep. 2008, pp. 158-167.

[35] Y. Zhang and J. B. D. Joshi, “A Framework for User Authorization Query Processing in RBAC extended with Hybrid Hierarchy and Constraints”, ACM symposium on access control models and technologies (SACMAT-08), Jun. 2008, Estes Park, CO.

[36] S. Du, and J. B. D. Joshi, “Supporting Authorization Query and Inter-domain Role Mapping in Presence of Hybrid Role Hierarchy,” The 11th ACM Symposium on Access Control Models and Technologies, USA, June 2006.

[37] Y. Zhang and J. B.D. Joshi, “Role Based Domain Discovery in Decentralized Secure Interoperations”, 2010 Collaborative Technologies and Systems (CTS-10), May. 2010, Chicago, IL.

[38] H. Hu, and G. Ahn, “Enabling verification and conformance testing for access control model: In Proceedings of the 13th ACM Symposium on Access Control Models and Technologies (SACMAT '08), Estes Park, CO, USA, Jun. 2008, pp. 195-204.

[39] V. C. Hu, D. R. Kuhn, and T. Xie, “Property Verification for Generic Access Control Models”, 2008 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing, 2008, pp. 243-250.

[40] R. Bhatti, A. Ghafoor, E. Bertino, and J. B. D. Joshi, “X-GTRBAC: an XML-based policy specification framework and architecture for enterprise-wide access control”, *ACM Trans. Inf. Syst. Secur.* 8, 2 (May. 2005), 187-227.

[41] R. Sandhu, “Role Hierarchies and Constraints for Lattice-Based Access Controls”, Proceedings of Fourth European Symposium on Research in Computer Security (ESORIC’96), Rome, Italy.

[42] S. Piromruen, J. B. D. Joshi, “An RBAC Framework for Time Constrained Secure Interoperation in Multi-domain Environment,” IEEE Workshop on Object-oriented Real-time Databases (WORDS-2005), 2005.

[43] M. Shehab, E. Bertino, and A. Ghafoor, “SERAT: SEcure role mApping technique for decentralized secure interoperability”, In Proceedings of the Tenth ACM Symposium on Access Control Models and Technologies (SACMAT '05) Stockholm, Sweden, Jun. 2005, pp. 159-167.

[44] B. Lampson, “A note on the confinement problem”, *Communications of the ACM*, Vol. 16, No. 10, Oct. 1973, pp. 613-615.

[45] G. Graham, P. Denning, “Protection – Principles and Practice”, In Proceeding of Spring Joint Computer Conference, AFIPS Press, 1972.

[46] R. Sandhu, and P. Samarati, “Access Control: Principles and Practice”, *IEEE Computer*, Sep. 1994, pp. 40-48.

[47] D. Denning, “A Lattice Model of Security Information Flow”, Communications of the ACM, Vol. 19, 1976, pp. 236-243.

[48] J. Gray, “Toward a Mathematical Foundation for Information Flow Security”, In Proceedings of the 1991 IEEE Symposium on Research in Security and Privacy, IEEE Computer Society Press, 1991.

[49] R. Sandhu, “Lattice-based Enforcement of Chinese Walls”, Computer and Security, Vol. 11, No. 8, Dec. 1992, pp. 753-763.

[50] J. D. Mclean, “Security Models and Information Flow”, in Proceedings of 1990 IEEE Symposium on Security and Privacy, Oakland, CA, 1990, pp. 180-187.

[51] D. D. Clark, D.R. Wilson, “A Comparison of Commercial and Military Computer Security Policies”, IEEE Symposium on Security and Privacy, 1987, pp. 184-194.

[52] R. Sandhu, “Separation of Duties in Computerized Information Systems”, In Database Security IV: Status and Prospects. Elsevier North-Holland, Inc., New York, 1991, pp. 179-189.

[53] M. Nyanchama, S. L. Osborn, “Role-Based Security, Object-Oriented Databases and Separation of Duty”, SIGMOD Rec. 22, 4, Dec. 1993, pp. 45-51.

[54] M. Nyanchama, S. L. Osborn, “The Role Graph Model and Conflict of Interest”, *ACM Transactions on Information and System Security*, Vol. 2, No. 1, 1999, pp. 3-33.

[55] J. B. D. Joshi, W. G. Aref, A. Ghafoor, E. H. Spafford, "Security models for web-based applications" , Communications of the ACM, 44, 2 (Feb. 2001), Page 38-44.