Hasso–Plattner–Institut fuer Softwaresystemtechnik
an der Universitaet Potsdam

Access Control Model and Policies for Collaborative Environments

Dissertation

zur Erlangung des akademischen Grades
“Doctor rerum naturalium”
(Dr. rer. nat.)
am Fachgebiet Internet Technologien und Systeme

eingereicht an der
Mathematisch–Naturwissenschaftlichen Fakultaet
der Universitaet Potsdam

von
Wei Zhou

Potsdam 2008

Abstract

To effectively participate in modern collaborations, member organizations must be able to share specific data and functionality with the collaborative partners, while ensuring their resources are safe from inappropriate access. This requires access control models, policies, and enforcement mechanisms for collaborative systems. This thesis describes my research work that concerns two important issues in collaborative systems: access control model and policies.

Team and Task based RBAC (TT-RBAC) Model. TT-RBAC extends the Role-Based Access Control (RBAC) model through adding sets of two basic data elements called teams and tasks. TT-RBAC model as a whole is defined in terms of individual users being assigned to roles and teams, roles and tasks being assigned to teams, and permissions being assigned to roles and tasks. By virtue of team membership, users get access to team's resources specified by assigned tasks. However, for each user, the exact permissions he/she obtains to a team's resources will be determined by his/her roles and current activity of the team. The team defines a small and specific RBAC application zone, through which we can preserve the advantages of scalable security administration that RBAC-style models offer and yet offers the flexibility to specify fine-grained control on individual users in certain roles and on individual object instances. The integrated context constraints make the TT-RBAC be an active security model.

Function-Based Authorization Constraints. Authorization constraints are an important aspect of RBAC and its various extensions. They are often regarded as one of the principal motivations behind these access control models. There are two important issues relating to constraints: their specification and their enforcement. However, the existing approaches cannot comprehensively support both of them. We designed two novel authorization constraint schemes named prohibition constraint scheme and obligation constraint scheme respectively. They can be used for both expressing and enforcing authorization constraints. These schemes are strongly bound to authorization entity set functions and entity relation functions that could be directly mapped to the functions that need to be developed in application systems. Thus, these schemes provide the developers a clear view about which functions should be developed in an authorization constraint system. Based on these functions, various constraint schemes can be easily defined and then enforced. A constraint system could be scalable through defining new entity set and relation functions. This approach goes beyond the well known separation of duty constraints, and considers many aspects of entity relation constraints.

Label-Based Access Control Policy (LBACP). In collaborative systems, the participants and trust relationships may be dynamically changed and the authorization policies have to explicitly specify which users from which organizations can access which resources. When there are many organizations involved and these organizations need to be dynamically added or removed, it will bring considerable burden to the administration. So it would be better to move such dynamic information to some separated control policies. For this purpose, we developed the LBACP. Its basic principle is defining some labels that specify information flow constraints and then assigning them to other access control policies or their components. The usage of the labeled policy components must conform to the information flow constraints defined by the labels in order to avoid them being misused. The LBACP can reduce the burden of security

administration and avoid information leaks in collaborative environments. LBACP is a high level security policy layered on the top of normal access control policies.

Root Policy. In collaborative environments, there may involve several independent autonomous domains; each of them may have its own security mechanisms and policies; these mechanisms and policies should be combined together and enforced as one at runtime. Authorization in such a system needs to be flexible and scalable to support multiple security mechanisms and policies. We developed the root policy specification language that is used to specify multiple heterogeneous authorization policies' combination and the mechanism that is used to enforce these root policies. In a root policy, each involved authorization policy's storage, trust management and enforcement can be defined independently. Various policy combination algorithms and flexible context constraints enable the root policy to deal with complex authorization scenarios. On the other hand, multiple root policies can cooperate together to complete more complex authorization tasks in distributed fashion.

Zusammenfassung

Um eine effektive Teilnahme an modernen Kollabrationen zu ermöglichen, müssen beteiligte Organisationen fähig sein, spezifische Daten und Funktionalität mit ihren Partnern auszutauschen und dabei sicherstellen, dass die eigenen Ressourcen sicher vor unerlaubten Zugriff sind. Dies verlangt nach Zugriffskontrolle, Policies und Mechanismen für kollaborative Systeme. Diese Dissertation beschreibt meine Forschungsarbeit, welche zwei wichtige Belange in kollaborativen Systemen betrifft: Zugriffskontrolle und Policies.

Function-Based Authorization Constraints. Es existieren zwei wichtige Belange in Bezug auf Authorisierungseinschränkungen: Ihre Spezifizierung und ihre Durchsetzung. Jedoch können existierende Ansätze nicht beide Belange umfassend unterstützen. Im Rahmen meiner Arbeit habe ich neuartige Schemata für Authorisierungsaeinschränkungen entworfen, genannt „prohibition constraint scheme“ und „obligation constraint scheme“. Diese Schemata sind mit entity set functions und entity relation functions im Bereich der Authorisierung verwandt, die direkt auf die für Anwendungsszenarien zu entwickelden Funktionen abgebilded werden können. Basierend auf diesen Funktionen können verschiedene Schemata für Einschränkungen einfach definiert und durchgesetzt werden.

Team and Task based RBAC (TT-RBAC) Model. TT-RBAC erweitert das rollenbasierte Zugriffskontrollmodell durch die Hinzunahme von zwei grundlegenden Datenelementen, genannt Team und Aufgabe. Das TT-RBAC model ist im Wesentlichen definiert durch die Zuordnung von einzelnen Benutzern zu Rollen und Teams, die Beziehung zwischen Rollen und Aufgaben zu Teams und Berechtigungen, welche wiederum Rollen und Aufgaben zugewiesen sind. Aufgrund der Teamzugehörigkeit erhalten Benutzer Zugriff auf Ressourcen, die durch Aufgaben zugewiesen sind. Das Team definiert eine kleine und spezifische RBAC Anwendungszone, durch welche die Vorteile der skalierbaren Sicherheitsadministration erhalten werden können, die durch RBAC-Modelle geboten werden. Überdies wird eine Flexibilität erreicht durch die genaue Kontrolle von individuellen Benutzern in spezifischen.

Label-Based Access Control Policy (LBACP). Das grundlegende Prinzip von LBACP stellt die Definition von Auszeichnungen (labels) dar, die den Informationsfluss spezifizieren und deren Zuweisung zu anderen Authorisierungspolicies oder ihren Komponenten. Der Einsatz von solchen Policies oder Policy-Komponenten muss den Einschränkungen hinsichtlich des Informationsflusses folgen, welche durch die Label spezifieziert sind. LBACP kann zu ereiner Veringerung des Administrationsaufwandes führen und potentielle Sicherheitslücken ausschließen. Insgesamt stellt LBACP eine Abstraktionsschicht über herkömmliche Zugriffskontroll-Policies dar.

Root Policy. In kollaborativen Umgebungen wird eine flexibele und skalierbare Autorisierung benötigt, um verschiedene Sicherheitsmechanismen und Policies zu unterstützen. Dazu habe ich im Rahmen meiner Arbeit ein System genant Root-Policy entwickelt, welches verschiedene Sicherheitsmechanismen und Policies spezifizieren und durchstzen kann. In einer Root-Policy kann die Speicherung von Autoriesierungspolicies, das Trust-Management und deren Durchsetzung unabhängig voneinander definiert werden. Darüberhinas können verschiedene Root-Policies zusammen kooperieren, um verteilte Szenarien zu bedienen.

To my wife Dongyun Zhang and daughter Ziqi Zhou

To my parents Kuiwu Zhou and Shijie Wu

Preface

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration.

The pronouns ‘we’ and ‘our’ in the text, which have been used for stylistic reasons, should be taken to refer to the singular author.

This dissertation is not substantially the same as any that I have submitted for a degree or any other qualification at any other university.

No part of this dissertation has already been, or is being currently submitted for any such degree, diploma or other qualification.

This dissertation does not exceed 60,000 words including tables and footnotes, but excluding bibliography and diagrams.

Acknowledgements

This work would have not been possible without the continuous guidance, advice, support, and encouragement from my PhD supervisor, Prof. Dr. Christoph Meinel at University of Trier, and later at University of Potsdam. Throughout my pursuit of the PhD program, he has always been tireless in giving me invaluable comments and advice. He has been a source of great inspiration. I consider myself very fortunate and privileged to have benefited from his exceptional insight and experience in research. I would like to thank my colleagues in Institute for Telematics, University of Warwick and Hasso-Plattner-Institut for their friendly and valuable discussions.

I would like to express my sincere gratitude and appreciation to Professor Vinesh H. Raja at University of Warwick, for providing the opportunity to me to undertake my doctoral studies in conjunction with my work responsibilities.

I would like to express my appreciation to Yidong Xiang, Yang Shao and Xuewu Li for their cooperation and useful discussions when I try to test my research results in the real world.

I am particularly thankful to my parents and my wife's parents for their taking care of our daughter, their sacrifices, and their love to her when we pursued for doctoral degrees in foreign countries. I am also thankful to my sister and brother for their love, understand and support.

My deepest gratitude goes to my wife, Dr. Dongyun Zhang, for her constant love, support, encouragement and patience throughout the years of effort toward this degree. I owe a great debt to my lovely daughter, Ziqi Zhou. I am very sorry for not accompanying her in most of her childhood. It reminds me that I should work hard so that I could complete the study as soon as possible and live with my family together. This thesis is dedicated to them.

List of Publications

[ZMXS08b] W. Zhou, C. Meinel, Y. Xiang, Y. Shao. Authorization Constraints Specification and Enforcement. Journal of Information Assurance and Security (JIAS), Volume 3, Issue 1, March 2008, 38-50.

[ZM08a] W. Zhou, C. Meinel. Enforcing Information Flow Constraints in RBAC Environments. In Proceedings of the International Symposium on Electronic Commerce and Security (ISECS 2008), pp 159-164, Guangzhou, China, August 2008.

[ZM07c] W. Zhou, C. Meinel. A Policy Language for Integrating Heterogeneous Authorization Policies. In Proceedings of the 4th International Conference on Grid Service Engineering and Management (GSEM 2007), pp. 9-23, Leipzig, Germany, September 2007.

[ZM07b] W. Zhou, C. Meinel. Team and Task Based RBAC Access Control Model. In Proceedings of the 5th Latin American Network Operations and Management Symposium (LANOMS 2007), pp. 84-94, Petrópolis, Brazil, September 2007.

[ZM07a] W. Zhou, C. Meinel. Function-Based Authorization Constraints Specification and Enforcement. In the Proceedings of the Third International Symposium on Information Assurance and Security (IAS 2007), pp. 119-114, Manchester, United Kingdom, August 2007.

[ZRMA06] W. Zhou, V. H. Raja, C. Meinel, M. Ahmad. Label-Based Access Control Policy Enforcement and Management. In Proceedings of the Seventh International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD 2006), Vol. 00, pp. 395-400, Las Vegas, Nevada, June 2006.

[ZRMA05] W. Zhou, V. H. Raja, C. Meinel, M. Ahmad. A Framework for Cross-Institutional Authentication and Authorisation. In Proceedings of the eChallenges e-2005 Conference(e-2005), pp. 1259-1266, Ljubljana, Slovenia, October 2005.

[ZRM05] W. Zhou, V. H. Raja, C. Meinel. An Authentication and Authorization System for Virtual Organizations. In Proceedings of the 9th world multiconference on systemics, cybernetics and informatics (WMSCI 2005), Vol. VII, pp. 150-155, Orlando, Florida, U.S.A., July 2005.

[ZMR05] W. Zhou, C. Meinel, V. H. Raja. A Framework for Supporting Distributed Access Control Policies. In Proceedings of the 10th IEEE Symposium on Computers and Communications (ISCC 2005), pp. 442-447, La Manga del Mar Menor, Cartagena, Spain, June 2005.

[HZZM05] W. Huang, W. Zhou, X. Zhang, C. Meinel. A Dynamic, Secure and Multi-Solutions Supported Middleware System. In Proceedings of the 7th IEEE International Conference on Advanced Communication Technology (ICACT2005), pp. 724-729, Republic of Korea, February 2005.

[ZM04] W. Zhou, C. Meinel. Implement role based access control with attribute certificates. In Proceedings of the 6th IEEE International Conference on Advanced Communication Technology (ICACT2004), pp. 536-541, Republic of Korea, February 2004.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The objective of access control is to protect resources against unauthorized access, while ensuring authorized access.

Nearly all applications that deal with finance, privacy, safety or defense include some forms of access control. Access control is concerned with determining the allowed activities of legitimate users, mediating every attempt by a user to access a resource in the system. In some systems, complete access is granted after successful authentication of the user, but most systems require more sophisticated and complex control. In addition to the *authentication* mechanism (such as a password), access control is concerned with how *authorizations* are structured. Authorization is the process of ascertaining that an entity with a particular identity or set of attributes has the permission to perform a particular action on a particular resource. This step is preceded by authentication where the identity of the entity is established.

When planning an access control system, three abstractions of controls should be considered: *access control policies*, *access control models*, and *access control mechanisms*. Access control policies are high-level requirements that specify how access is managed and who may access information under what circumstances. For instance, policies may pertain to resource usage within or across organizational units or may be based on need-to-know, competence, authority, obligation, or conflict-of-interest factors. At a high level, access control policies are enforced through a mechanism that translates a user's access request, often in terms of a structure that a system provides. An access control list is a familiar example of an access control mechanism. Access control models bridge the gap in abstraction between policy and mechanism. Rather than attempting to evaluate and analyze access control systems exclusively at the mechanism level, security models are usually written to describe the security properties of an access control system. Security models are formal presentations of the security policy enforced by the system and are useful for proving theoretical limitations of a system. Discretionary access control, which allows the creator of a file to delegate access to others, is one of the simplest examples of an access control model.

With the advent of the information superhighway, governments, businesses and other organizations are finding that collaborations are increasingly crucial to their success. To effectively participate in modern collaborations, member organizations must be able to share specific data and functionality with collaborative partners, while ensuring their resources are safe from inappropriate access. Such collaborations may dynamically change participants and trust relationships during the life cycle. The dynamic and multi-institutional nature of these

environments introduces challenging security issues that demand new technical approaches for authorization policy management and enforcement. In this thesis I present my research on the topics of access control model, policies and mechanisms in collaborative environments.

This chapter describes the motivation and outlines the contribution of this work. It begins with the introduction of research motivation in Section 1.1. Section 1.2 outlines the contribution of this research. Finally, Section 1.3 describes the structure of this thesis.

1.1 Research motivation

In this section we investigate some research issues raised by the modern collaborations which we try to resolve in this thesis.

1.1.1 Access control model

Collaborative systems are becoming a popular means of providing efficient and scalable access to distributed computing capabilities. In collaborative systems a set of organizations share their computing resources, such as computer cycles, storage space, or online services, to establish virtual organizations aimed at achieving a particular task. Balancing the competing goals of collaboration and security is difficult because interaction in collaborative systems is targeted towards making people, information, and resources available to all who need it, whereas information security seeks to ensure the availability, confidentiality, and integrity of these elements while providing it only to those with proper authorization [TAPH05].

Traditionally, an access control decision is made based on subjects and objects. When there are many subjects and objects being involved, the subject-object model cannot provide satisfied security management. Role-Based Access Control (RBAC) [SCFY96, FSGK01] is an alternative to traditional discretionary and mandatory access controls. In RBAC, permissions are associated with roles, and users are made members of appropriate roles thereby acquiring the roles' permissions. RBAC is thus more scalable than user-based security specifications and greatly reduces the cost and administrative overhead. However, subsequent attempts to apply RBAC in collaborative environments revealed some RBAC's limitations that are described as follows:

1. RBAC lacks the ability to specify a fine-grained control on individual users in certain roles and on individual object instances. For collaborative environments, it is insufficient to have role permissions based on object types. Rather, it is often the case that a user in an instance of a role might need a specific permission on an instance of an object.
2. RBAC does not take into account the impact of context. Thus, RBAC belongs to a passive security model. In collaborative environment, it is highly demanded that an access control system needs to consider the overall context associated with any collaborative activity.
3. RBAC assumes that all permissions needed to perform a job function can be neatly encapsulated. In fact, role engineering has turned out to be a difficult task. The challenge of RBAC is the contention between strong security and easier administration. For stronger security, it is better for each role to be more granular, thus having multiple roles per user. For easier administration, it is better to have fewer roles to manage.
4. Authorization constraints are an important aspect of RBAC and a powerful mechanism for laying out higher-level organizational policy. However, the specification of such constraints has not been discussed in the RBAC model.

5. On the other hand, RBAC does not provide an abstraction to capture a set of collaborative users operating in different roles.

In order to overcome the shortcomings of RBAC, a variety of access control models have been developed over the years. These access control models incorporate additional contextual information and support higher-level policy abstractions that can simplify policy administration by reducing the semantic gap between enterprise-level policies and the policies that can be directly enforced within a system. Abstractions such as "team" and "task" are developed to model contextual information associated with organizational roles, responsibilities and collaborative activities. But these access control models proposed to date either have been only informally defined and, therefore, subject to ambiguous or incomplete specifications, and limited assurance, or only address a specific application scenario, as a consequence, could not be used for reference models.

Currently there is still no access control model that rigorously defines the relations among team, task and RBAC entities. Motivated by this requirement, we defined a Team and Task-based RBAC (TT-RBAC) access control model [ZM07b] that extends the NIST RBAC model [FSGK01] through adding sets of two basic data elements called teams and tasks. We also developed a mechanism that enables the TT-RBAC to be a context-aware access control model. This characteristic makes TT-RBAC be an active security model.

1.1.2 Authorization constraints

Authorization constraints are an important aspect of access control and are powerful mechanism for laying out higher-level organizational policy. The most important authorization constraint is the Separation of Duty (SoD) that requires if a sensitive task is comprised of two steps, then different users should each perform each step. A very simple example of this is that checks might require two different signatures. SoD is widely considered to be a foundational principle in computer security [SS75, CW87, FCK95, Bis03].

The opposite constraints of SoD are Binding of Duty (BoD). Examples of BoD constraints are that one user can be assigned to role *A* only when he/she is already a member of role *B* or one user is required to perform two different tasks in the same workflow instance. Some constraints are number related. These constraints are called cardinality constraints. Cardinality constraints require that one role can have a maximum number of members or certain tasks are performed a certain number of times. Authorization constraints have been part of most RBAC models of recent years and are often regarded as one of the principal motivations behind RBAC.

Despite its importance as a security principle and its well-understood application in business, industry, and government, few computer systems have well supported authorization constraints as a security policy to date. A more serious concern in using RBAC is the implementation of SoD controls. Existing RBAC products have only rudimentary SoD features [HFK06]. Static SoD may be supported, but very few provide dynamic SoD. There are two major reasons that are attributed by [GGF98]. First, SoD is an inherently application-oriented policy and, thus, has been perceived to yield limited payoff for operating systems and networks, since it cannot be used as a global, system-wide security policy. Second, when the SoD principle is interpreted within different applications, it may yield many different SoD policies and, thus, support of all policies is perceived to require both substantial system flexibility and recurrent administrative costs -- an unmistakable recipe for both system-vendor and market resistance in the absence of advanced policy-administration tools. As a consequence, both relationships among SoD properties and policy composability -- an important requirement for all application-oriented policies -- could not be easily established.

There are two important issues related to constraints: their specification and their enforcement. Although authorization constraints, such as SoD, are easy to understand, they are hard to express these principles in computer security systems. Early work in computer-supported SoD focused on mechanisms that were easy to implement and turned out to be rigid and unrealistic. Recently, mechanisms that support flexible policy-based SoD start to be examined in depth. But with a few exceptions they have always been specified using rule-based systems. Unfortunately, rule-based systems, while highly expressive, are harder to visualize and thus to use; thus far they have been avoided by practitioners. The existed work mainly addresses constraint expression rather than constraint enforcement. Currently there is still no useful approach for both expressing and enforcing constraints. On the other hand, the early research effort mainly concentrates on SoD. Other kinds of constraints, such as BoD constraints, have received less attention. We believe that existing constraint specification approaches are rather complicated and the few existing enforcement models are unlikely to scale well. We designed two novel authorization constraint schemes [ZM07a, ZMXS08a] that can be used for both expressing and enforcing authorization constraints.

1.1.3 Authorization policies and information flow

To effectively participate in modern collaborations, member organizations must be able to share specific data and functionality with collaborative partners, while ensuring their resources are safe from inappropriate access. Access control policies play an important role. In collaborative environments, access control policies may have to explicitly specify which users from which contexts (e.g. organizations) can perform what operation on which resources. However, it is not easy for some traditional authorization mechanisms, such as RBAC, to explicitly specify such kinds of constraints. If RBAC is used in a collaborative system, the only way to provide fine-grained access control is to define more roles that have different privileges and then assign these roles to the users who come from different contexts. When there are many contexts involved, especially these contexts need to be dynamically added or removed, considerable burden will be brought to the administration. So it would be better to move such kinds of control information to some separated control policies that specifically describe the dynamic control information, and then merge these control policies with traditional policies at runtime. When we need to add or remove some contexts or modify some exist relations, we only need to modify these control policies. This will reduce the security administrative burden. On the other hand, as authorization policies are becoming more complex, the possibility of information leaks caused by improperly designed authorization policies increases. If there are some extra constraints added to the traditional authorization policies, then some information leaks caused by these policies could be avoided. For example, some information may be prohibited to flow to a specified context even if it is permitted by the normal authorization policies. Next we discuss the information flow constraints in different application scenarios.

In a hospital, there is an authorization rule that specifies that only the persons who belong to the *department of epidemic disease* and have the role *doctor* can read patients' *HIV test* results. From the view of information flow, this rule specifies that the *HIV test* results can only flow to the *department of epidemic disease*. Now we remove the information flow related control information to a separated control rule, then this rule is split into two rules. One is the normal authorization rule that specifies that the persons who have the role *doctor* can read all kinds of patient medical records. Another is the information flow constraint rule that specifies that the *HIV test* results can only flow to the *department of epidemic disease*. At runtime the information flow constraint rule will be merged with the normal authorization rule to specify that only the

doctors belonging to the *department of epidemic disease* can read patients' *HIV test* results. There two major benefits of this approach. One is that we do not have to define a new role that is specifically defined for the access control to the *HIV test* results. The other is that the information flow constraints can avoid some intentional and unintentional information leakage.

In hosting environments, it will be ideal for a hosting company to set up and configure a single version of each application for which they provide hosted services. This can substantially reduce the costs associated with hosting. Because hardware, database, and applications instances can be shared, the costs associated with hardware, as well as installation and configuration of software, are lower than physically separated instances for each hosted customer. For this purpose, we need an access control mechanism that can be configured to keep data from different organizations separated within a single repository such as a database instance, so that organizations can share database tables but only see the data that pertains to them. From the view of information flow, we can understand that some data categories can only flow to some particular user groups.

Information flow control is also ideal for enforcing privacy concerns. For example, in health care environments, some kinds of patient's information are only accessible to the doctors who belong to the specified departments. In e-business applications, customers' personal information can be classified and then attached with some information flow constraints so that they can only be released to the desired marketing campaigns.

In [ZRMA06, ZM08b] we described an approach for specifying and enforcing information flow constraints to the normal authorization policies and their components.

1.1.4 Heterogeneous authorization policy composition

For the purpose of authorization, various authorization systems and mechanisms have been proposed and developed. Unfortunately, in many application scenarios one size does not fit all [FGHK05]. In collaborative environments, such as a grid system, there may involve several independent autonomous domains. Each of them may have its own policies, and these policies may need to be dynamically changed [LFSA06]. Authorization in such a system needs to be flexible and scalable to support multiple authorization mechanisms and security policies.

A further example is provided by laws concerning privacy issues. In a modern information system, the security policy of the organization should combine internally specified constraints with externally imposed privacy regulations.

Even if there is only one authorization mechanism being used, it is also possible to require an authorization system to combine multiple policies to complete some complex decision tasks. Consider now a large organization composed of different departments and divisions, each of which can independently specify security policies; the global policy of the organization results from the combination of all these components. Finally, as security policies become more sophisticated, even within a single system ruled by one administrator it may be desirable to formulate the policy incrementally by assembling small, manageable, and independently conceived modules [BVS02]. Some authorization policy languages already support multiple policy combination. However, some authorization policy languages do not support multiple policy combination at all.

Currently, there is still no dedicated work that comprehensively supports heterogeneous authorization policies' trust management, combination and enforcement in distributed authorization environments. The existing approaches are either too simple to support complex policy combination or tightly integrated into an operating system. Motivated by the requirement of managing and enforcing multiple heterogeneous authorization policies in distributed

authorization environments, we designed the root policy specification language and corresponding enforcing mechanism [ZM07c].

1.2 Thesis contribution

The aim of this thesis is to address the pending research issues described in Section 1.1. The main contributions of this thesis are:

Team and Task-based RBAC (TT-RBAC). TT-RBAC model extends the RBAC model through adding sets of two basic data elements called teams and tasks. The TT-RBAC model as a whole is fundamentally defined in terms of individual users being assigned to roles and teams, roles and tasks being assigned to teams, and permissions being assigned to roles and tasks. By virtue of team membership, users get access to team's resources that are specified by the assigned tasks. However, for each user, the exact privilege he/she obtains from a team is determined by his/her roles and the current activity of the team. The TT-RBAC model preserves the advantages of scalable security administration that RBAC-style models offer and yet offers the flexibility to activate permissions for individual users and to specific object instances. (Chapter 3)

Context-aware TT-RBAC enforcement. Designing and implementing a context-aware TT-RBAC system with object-oriented technology. This involves several aspects, ranging from the TT-RBAC core classes and their relations, flexible context constraint assignment and enforcement, as well as TT-RBAC access control decision maker and evaluation process. The design and implementation are focused on one key principle: practical applicability. (Chapter 4)

Function-based authorization constraints. We introduce two novel authorization constraint schemes named *prohibition constraint scheme* and *obligation constraint scheme* respectively. Both of them can be used for expressing and enforcing authorization constraints. These schemes are strongly bound to authorization entity set functions and authorization entity relation functions, so they can provide the system developers a clear view about which functions should be developed in an authorization constraint system. The security administrators can use these functions to create constraint schemes for their day-to-day operations. A constraint system can be scalable through defining new authorization entity set and relation functions. This approach goes beyond the well known separation of duty constraints, and considers many aspects of authorization entity relation constraints. (Chapter 5)

Label policy. In collaborative environments authorization policies may need to specify which users from which contexts can perform what actions on which resources. When there are many contexts involved, especially those contexts need to be dynamically added or removed, considerable burden will be brought to the administration. So it is better to remove such kinds of control information to a separated control policy. For this purpose, we propose the label-based access control policies that specify the information flow constraints. These label policies are assigned to other authorization policies or their components. The usage of the labeled policy components must conform to the information flow constraints defined by the labels in order to avoid them being misused. The label policies can also improve authorization policies' administration by grouping policy components together through annotating them with multiple labels. (Chapter 6)

Root policy. In order to manage and enforce multiple heterogeneous authorization policies in distributed authorization environments, we define the root policy specification language and corresponding enforcing mechanism. In a root policy, each involved authorization policy's storage, trust management and enforcement can be defined independently. These authorization policies can be enforced in distributed fashion. In root policies the policy schemas, policy sub-schemas, policy hierarchies and context constraints can be used to deal with various authorization scenarios. On the other hand, multiple root policies can cooperate to complete more complex authorization tasks. (Chapter 7)

Applications. We present several real application case studies that we use to test some of the ideas described in this thesis. (Chapter 8)

1.3 Dissertation outline

This thesis is organized as follows:

Chapter 2 reviews major work in the area of access control, with the focus on collaborative systems. It begins with a short introduction to earlier access control models. This is followed by examining existing access control models as applied to collaborative environments. The chapter ends with an introduction to a selection of existing authorization systems and mechanisms that are used in distributed systems.

Chapter 3 introduces the team and task-based RBAC model. It starts with a short introduction to the most important related work. This is followed by a description of NIST RBAC model. We then describe the TT-RBAC model that is the extension to the NIST RBAC model. Finally we give a comparison among TT-RBAC and other access control models against a set of assessment criteria for access control in collaborative systems.

Chapter 4 introduces how to implement a context-aware TT-RBAC system. We first define the TT-RBAC core classes and their relations. Next we add context constraints to the TT-RBAC entities and look at how the contextual entities affect the TT-RABC system. Finally, we investigate the TT-RBAC evaluation mechanism, and then give a short description to the implementation.

Chapter 5 introduces function-based authorization constraint specification and enforcement. After giving a review of related work, we define two constraint schemes and examine the functions required in TT-RBAC environment. We then describe these schemes' evaluation processes. Next we show these constraint schemes' expressive power. This is followed by the introduction of constraint schema. Finally, we introduce our implementation and summarize the results of this chapter.

Chapter 6 presents label-based access control policies. We first introduce the label model and label composition. This is followed by a description of label assignments and the effects to the labeled policy components. Next we extend this label model to include hierarchical contexts. We then show some application examples of label policies. Finally, we give the related work and summarize the results of this chapter.

Chapter 7 presents root policies that are used to manage and enforce multiple heterogeneous authorization policies in distributed access control environment. This chapter starts with the introduction of related work. This is followed by the description of root policy authorization model and language model. Next we describe the root policy evaluation and

combination. Finally, we focus on root policy enforcement mechanism.

Chapter 8 provides in-depth descriptions of three case studies that employ the authorization approaches described in this thesis. These applications aim to demonstrate these approaches' various specific features in practice. One case study shows how the group-based authorization system could be replaced by the TT-RBAC system. The other two case studies show how the function-based authorization constraints are integrated into some real application systems.

Chapter 9 concludes this thesis, with a summary of the main contributions and a brief discussion of potential future research and extensions.

Chapter 2

Background

This chapter examines fundamental technologies and major researches related to this thesis. We begin with a brief review of early access control models. This is followed by an investigation to access control in collaborative systems. Finally, we introduce some well known authorization mechanisms and systems.

2.1 Access control models

Since the early 1960s access control has been a major issue, mainly in operating systems and database management systems. Its objective was to protect system resources against undesired or inappropriate accesses, whilst permitting authorized accesses. Early research on access control was dominated by two approaches: Discretionary Access Control (DAC) and Mandatory Access Control (MAC). Recently, computer security texts often list Role-Based Access Control (RBAC) as one of the three primary access control policies. Some other important access control models are also introduced in this section.

2.1.1 Discretionary access control

Discretionary Access Control (DAC) leaves a certain amount of access control to the discretion of the object's owner or anyone else who is authorized to control the object's access [NCSC87]. For example, it is generally used to limit a user's access to a file; it is the owner of the file who controls other users' accesses to the file. Only those users specified by the owner may have some combination of read, write, execute, and other permissions to the file. DAC policy tends to be very flexible and is widely used in the commercial and government sectors. However, DAC is known to be inherently weak for two reasons. First, granting read access is transitive; for example, when Ann grants Bob read access to a file, nothing stops Bob from copying the file to an object that Bob controls. Bob may now grant any other user access to the copy of Ann's file without Ann's knowledge. Second, DAC policy is vulnerable to Trojan horse attacks. Because programs inherit the identity of the invoking user, Bob may, for example, write a program for Ann that, on the surface, performs some useful function, while at the same time destroys the contents of Ann's files. When investigating the problem, the audit files would indicate that Ann destroyed her own files. Formally, the drawbacks of DAC are as follows [HFK06]:

- Information can be copied from one object to another; therefore, there is no real assurance on the flow of information in a system.
- No restrictions apply to the usage of information when the user has received it.
- The privileges for accessing objects are decided by the owner of the object, rather than a system-wide policy that reflects the organization's security requirements.

Access Control Lists (ACLs) and owner/group/other access control mechanisms are by far the most common mechanism for implementing DAC policies [FCK03]. Other mechanisms, even though not designed with DAC in mind, may have the capabilities to implement a DAC policy.

2.1.2 Mandatory access control

Mandatory Access Control (MAC), supported by military research and civilian government, enforces access control by means of security labels. This model, first formalized by Bell and LaPadula [BL75], attaches labels or security classifications to every object and user. As the many variations of the Bell-LaPadula model led to confusion, Sandhu introduced a minimal model named BLP [San93], which encapsulates the essentials of the Bell-LaPadula model. In both of these, access is granted based on the subject's and the accessed object's security label. A typical sensitivity classification used by the military is: unmarked, unclassified, restricted, confidential, secret, and top secret.

We shall denote the security labels of these by $\lambda(s)$ and $\lambda(o)$ respectively. These security labels form a lattice with a partial order relation, $\leq$. As MAC was developed with a military environment in mind, confidentiality was a major motivating issue, which was achieved by information flow restrictions among entities (subjects or objects) of different security groups. These restrictions are expressed by the following two properties:

- The "simple security property", which is also known as "no read up", states that a subject can only read an object if $\lambda(o) \leq \lambda(s)$.
- The "*-property", or "no write down" property, allows a subject to write an object only if $\lambda(s) \leq \lambda(o)$. This property addresses information leakage by malicious programs. It does not allow programs to write information to objects that can be read by subjects with a less privileged security clearance. A variation on this property called the "strict *-property" requires that information can be written at, but not above, the subject's clearance level, formally $\lambda(s) = \lambda(o)$.

A similar model to BLP was published by Biba [Bib75]. Unlike BLP, Biba's model aims to achieve data integrity as opposed to confidentiality. It allows data flow only from high to low integrity data, which is exactly the inverse of permitted information flow in the BLP model. The need for a MAC mechanism arises when the security policy of a system dictates that:

- Protection decisions must not be decided by the object owner.
- The system must enforce the protection decisions (i.e., the system enforces the security policy over the wishes or intentions of the object owner).

Unfortunately MAC models are rather rigid. They support a fixed set of security classes, a fixed set of operations on objects, and allow only administrators to modify the access control policy. Nevertheless, the information flow restriction aspect of the MAC model greatly motivated parts of our work on label-based access control policy, described in chapter 6.

2.1.3 Clark and Wilson model

The primary objective of the Clark and Wilson model [CW87] is the prevention of fraud and error in commercial systems. Fraud is typically achieved by unauthorized modification of information, while error typically causes inconsistency of information. Both these concerns can be addressed by enforcing integrity policies. The Clark-Wilson model formalizes two basic principles for achieving information integrity: *well-formed transactions* and *separation of duty*.

The concept of well formed transactions is that manipulation of data by a principal must be constrained in such a way that its integrity is preserved. A very common and effective mechanism employed in accounting is double entry bookkeeping. The idea is to record every single transaction twice, once in a book for credit and once in a book for debit. A later balance check would reveal discrepancies if any entry were not recorded correctly. The intention of well-formed transactions is to ensure the internal consistency and accuracy of the data.

The principle of separation of duty attempts to ensure external consistency where the data in the system reflect the real-world entities they represent, e.g. when a payment is recorded on the account as the fulfillment of a purchase, and then there was indeed such a purchase, not a fraud. The correspondence to external entities is often abstract and hard to verify directly. The idea of separation of duty is to indirectly verify the correspondence to real-world entities by dividing a task among several principals. Provided these principals do not conspire, this mechanism should prevent both fraud and error.

The Clark-Wilson model partitions data into two sets: Constrained Data Items (CDI), whose integrity must be ensured, and Unconstrained Data Items (UDI), which are not under the control of the integrity policy. Two classes of procedures on these data items are defined to enforce the integrity policy: an Integrity Verification Procedure (IVP) verifies the integrity of all data items in the system, and a Transformation Procedure (TP) is a well-formed transaction that processes and changes a set of data items from one valid state to another.

The integrity policy can then be expressed in formalized rules, grouped into two types: *certification* and *enforcement*. Certification is an application-specific process that monitors the operations of a system with respect to a specific integrity policy. Enforcement rules are application-independent security functions that are automatically executed by the system. The rules in the Clark and Wilson model as formulated in [Amo94] are:

Certification

- C1 (IVP Certification) For any CDI, there must exist some IVP on the system that validates its integrity.
- C2 (Validity) All TPs must be certified to maintain the validity of CDIs they processed.
- C3 (Separation of Duty) All possible operations on CDIs by potential users must be certified to implement the principles of separation of duties and least privilege.
- C4 (Journal Certification) All TPs must be certified to ensure sufficient logging for their operations.
- C5 Special TPs that take UDIs must be certified to result in valid CDIs.

Enforcement

- E1 (Enforcement of Validity) Manipulation on a CDI must only be performed through a TP.
- E2 (Enforcement of Separation of Duty) Every user can only operate on a specific set of CDIs through a set of authorized TPs.

- E3 (User Authentication) Every user attempting to execute a TP must be properly authenticated by the system.
- E4 (Initiation) Only the administrator can specify authorizations to TPs and CDIs.

The Clark-Wilson model offers a distinctive view of, and a set of mechanisms for, access control problems in commercial environments. Their work laid the groundwork for research in commercial security, such as the Chinese Wall policy.

2.1.4 Chinese wall policy

Brewer and Nash identified the Chinese Wall policy [BN89] to address conflict-of-interest issues related to consulting activities within banking and other financial disciplines. The Chinese Wall policy is application-specific in which it applies to a narrow set of activities that are tied to specific business transactions. For example, consultants naturally are given access to proprietary information to provide a service for their clients. When a consultant gains knowledge amounting to insider information, and that knowledge can be used outside the company, thus undermining the competitive advantage of one or both institutions, or used for personal profit. The stated objective of the Chinese Wall policy is to prevent illicit flows of information that can result in conflicts of interest.

The Chinese Wall policy is a commercially inspired confidentiality policy. The access permissions change dynamically: as a subject accesses some objects, other objects that would previously have been accessible are now denied. For example, Chinese Wall policy is used where company-sensitive information is categorized into mutually disjoint conflict-of-interest categories (COI). Each company belongs to only one COI, and each COI has two or more member companies. The membership within a COI includes like companies, whereby a consultant obtaining sensitive information regarding one company would risk a conflict of interest if he or she were to obtain sensitive information concerning another company. Several COIs may coexist. For example, COI1 may pertain to banks, while COI2 may pertain to energy companies. The Chinese Wall policy aims to prevent a consultant from reading information for more than one company in any given COI.

Regarding this policy with respect to read operations, there are several observations can be made. First, as long as a consultant has not read information belonging to any institution, the consultant is not yet bound by the policy and is free to read any sensitive information of any institution. Note that although a consultant may be free to read sensitive information under the Chinese Wall policy, he or she may be restricted from reading sensitive information with respect to another policy, such as a MAC policy. Second, once a consultant has read sensitive information of bank A, the consultant is prohibited from reading sensitive information belonging to any other bank included in the COI of which bank A is a member. Third, all consultants are free to read all the public information of all institutions.

The Chinese Wall policy recognizes the importance of access history in protecting security and has made a seminal contribution to subsequent research on history-based access control and dynamic separation of duty in general [San88, San90, SZ97, GGF98].

2.1.5 Role-based access control

Because of the rigid nature of MAC, where users had little or no control over the access control policy, and the problems associated with policy changes in DAC, early access control models could not meet practical requirements of commercial organizations [CW87]. It was also realized

that in large organizations data is not owned by individual users, but by the organization itself, thus access to data should consider one's position in the organizational hierarchy. This inspired further work, a result of which was Role-Based Access Control (RBAC). Although RBAC is technically a form of non-discretionary access control [Ram02, Shi02], recent computer security texts often list RBAC as one of the three primary access control policies (the others are DAC and MAC).

Early work on role-based access control goes back to 1988, when Lochovsky and Woo defined roles and organized them into a hierarchy [LW88]. Over the years, many researchers have proposed models for RBAC [FK92, NO93, SCFY96, NO99, FBK99, FSGK01]. While the differences in these models are quite significant, the core concept remains fairly consistent between them. In RBAC, access decisions are based on the roles that individual users have as part of an organization. Users take on assigned roles (such as doctor, nurse, teller, or manager). Access rights are grouped by role name, and the use of resources is restricted to individuals authorized to assume the associated role. For example, within a hospital system, the role of doctor can include operations to perform a diagnosis, prescribe medication, and order laboratory tests; the role of researcher can be limited to gathering anonymous clinical information for studies. The use of roles to control access can be an effective means for developing and enforcing enterprise-specific security policies and for streamlining the security management process. A user establishes a session and activates some subset of roles assigned to him/her. The permissions available to the user in a session are those assigned to all the active roles in that session. The NIST RBAC model [FSGK01] is shown in Figure 2.1.

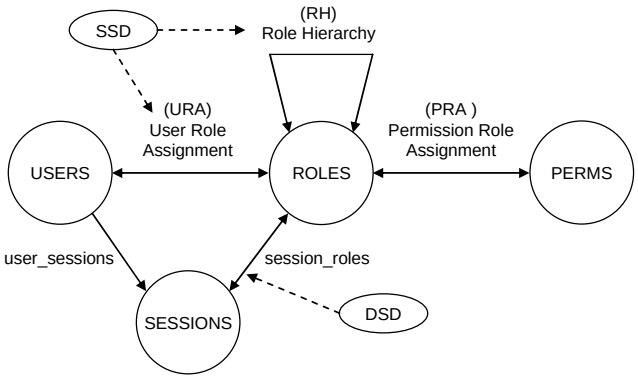


Figure 2.1: Role-based access control model.

Under RBAC, users are granted membership into roles based on their competencies and responsibilities in the organization. The operations that a user is permitted to perform are based on the user's role. User membership into roles can be revoked easily and new memberships established as job assignments dictate. Role associations can be established when new operations are instituted, and old operations can be deleted as organizational functions change and evolve. This simplifies the administration and management of privileges; roles can be updated without updating the privileges for every user on an individual basis.

When a user is associated with a role, the user can be given no more privilege than is necessary to perform the job; since many of the responsibilities overlap between job categories, maximum privilege for each job category could cause unauthorized access. This concept of least privilege requires identifying the user's job functions, determining the minimum set of privileges required to perform those functions, and restricting the user to a domain with those privileges and nothing more.

Role hierarchies can be established to provide for the natural structure of an enterprise. A role hierarchy defines roles that have unique attributes and that may contain other roles; that is, one role may implicitly include the operations that are associated with another role.

Separation of duty relations are used to enforce conflict of interest policies. Conflict of interest in a role-based system may arise as a result of a user gaining authorization for permissions associated with conflicting roles. One means of preventing this form of conflict of interest is though static separation of duty (SSD), that is, to enforce constraints on the assignment of users to roles. An example of such a static constraint is the requirement that two roles be mutually exclusive; for example, if one role requests expenditures and another approves them, the organization may prohibit the same user from being assigned to both roles. Dynamic separation of duty (DSD) relations limits the permissions that are available to a user by placing constraints on the roles that can be activated within or across a user's sessions. Although this separation of duty requirement could be achieved through the establishment of a static separation of duty relationship, DSD relationships generally provide the enterprise with greater efficiency and operational flexibility.

2.2 Access control in collaborative systems

To effectively participate in modern collaborations, member organizations must be able to share specific data and functionality with collaborative partners, while ensuring their resources are safe from inappropriate accesses. Such collaborations may dynamically change participants and trust relationships during the life cycle. This requires access control models, policies, and enforcement mechanisms for shared resources. In this section we first give the access control requirements for collaboration, and then examine existing access control models for collaborative environments.

2.2.1 Access control requirements for collaboration

Several groups [Edw96, JP96, FB97, Bul98] have studied the requirements for access control in collaborative environments. Tolone et al. [TAPH05] summarize these requirements as follows.

- Access control must be applied and enforced at a distributed platform level.
- Access control models should be generic and enable access rights to be configured to meet the needs of a wide variety of cooperative tasks and enterprise models. That is, such models should be expressive enough to specify access rights efficiently based on varied information (e.g., roles, context).
- Access control for collaboration requires greater scalability in terms of the quantity of operations than tradition single user models because the number of shared operations is much richer in collaborative environments compared to traditional single user systems.
- Access control models must be able to protect information and resources of any type and at varying levels of granularity. That is, they must have the ability to provide strong protection for shared environments and objects of various types as well as allow fine-grained control of access to individual objects and their attributes.
- Access control models must facilitate transparent access for authorized users and strong exclusion of unauthorized users in a flexible manner that does not constrain collaboration.

- Access control models must allow high level specification of access rights, thereby better managing the increased complexity that collaboration introduces.
- Access control models for collaboration must be dynamic, that is, it should be possible to specify and change policies at runtime depending on the environment or collaboration dynamics.
- Performance and resource costs should be kept within acceptable bounds.

Based on the work done by Tolone et al. [TAPH05] and our investigation, the merits and shortcomings of existing access control models in the context of collaborative environments are summarized in the following subsections.

2.2.2 Access matrix model

The access matrix [Lam71] is a model which specifies the rights that each subject possesses for each object. The access matrix provides a useful framework for describing resource protection in operating systems. This model defines three kinds of access-control entities: subjects, objects and access rights which associate the subject with the protected objects by specifying the operations that subjects are allowed to perform on objects. An access matrix *A*, with rows representing subjects, columns representing objects is used to define the protection state. *A*[*s*, *o*] denotes the access rights a subject *s* has over an object *o*. The access-checking rule of the model states that a request by subject *s* for accessing object *o* is granted only if *A*[*s*, *o*] contains the requisite right. An example of an access matrix is shown in Figure 2.2 (a).

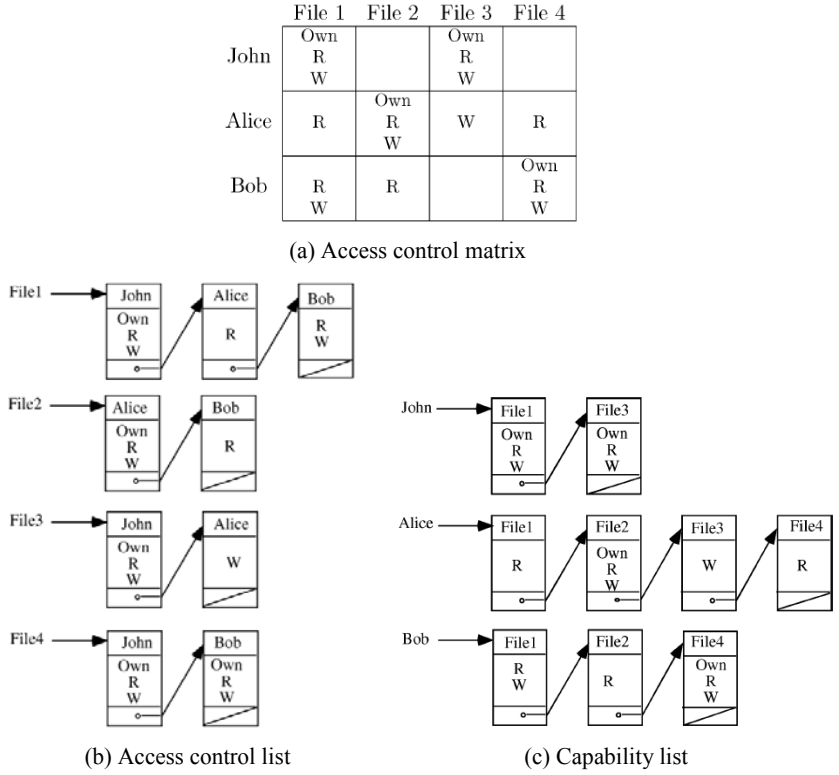


Figure 2.2: Access matrix model.

A common approach to implementing the access matrix is by means of Access Control Lists (ACLs). Each object is associated with an ACL, indicating for each subject in the system the accesses the subject is authorized to execute on the object. This approach corresponds to storing the matrix by columns. ACLs corresponding to the access matrix of Figure 2.2 (a) are shown in Figure 2.2 (b).

Capability Lists (C-lists) are a dual approach to ACLs. Each subject is associated with a list indicating for each object in the system, the accesses the subject is authorized to perform on the object. This approach corresponds to storing the access matrix by rows. Figure 2.2(c) shows capability lists for the files in Figure 2.2 (a).

There are several weaknesses to the access matrix model. Some are more general, while others are magnified due specifically to collaboration requirements. They are described as follows.

- More sophisticated access policies such as access based on competency, least privilege, or conflict-of-interest rules are difficult to provide without access rights that are associated with a subject's credentials when performing an operation.
- Users might change responsibilities at any point. ACL- and capability-based approaches lack the ability to support dynamic changes of access rights. For an object's ACL, it is easy to determine which modes of access subjects are authorized to have for that object. On the other hand, determining all the accesses that a subject has is difficult in an ACL-based system. It is necessary to examine the ACL of every object in the system to review access privileges with respect to a subject. Similarly, if all accesses of a subject need to be revoked, all ACLs must be checked one by one. In a capability list approach, it is easy to review all accesses that a subject is authorized to execute by reviewing the subject's capability list. But, determining all subjects who are allowed to access a particular object requires reviewing every capability list of each subject.
- In a collaborative or workflow setup, ownership might not be at the discretion of the user, that is, the system might own resources. ACLs and C-lists inadequately address this issue. Access rights may be related to content, attribute of resources, or other contextual information. Access matrix do not account for this situation.

2.2.3 Role-based access control

The essence of RBAC is that permissions are assigned to roles rather than to individual users. Roles are created for various job functions, and users are assigned to roles based on their qualifications and responsibilities. Users obtain the corresponding permissions through assigned appropriate roles. Users can be easily reassigned from one role to another without modifying the underlying access structure. RBAC is thus more scalable than user-based security specifications and greatly reduces the cost and administrative overhead. While very effective and popular for traditional and collaborative systems, RBAC has several weaknesses.

- Most early implementations of RBAC determined the set of roles in use as well as the role membership early in the lifetime of a session. Changes were not well supported. The nature of such roles could be called static. They lacked flexibility and responsiveness to the environment in which they were used.
- RBAC96 supports the notion of role activation within sessions, but it does not go far enough in encompassing the overall context associated with any collaborative activity. RBAC belongs to passive security systems that primarily serve the function of maintaining

permission assignments. An active security system, on the other hand, takes into account the impact of context as it emerges with progressing tasks and distinguishes task- and context-based permission activation from permission assignment.

- Traditional RBAC lacks the ability to specify a fine-grained control on individual users in certain roles and on individual object instances. For collaborative environments, it is insufficient to have role permissions based on object types. Rather, it is often the case that a user in an instance of a role might need a specific permission on an instance of an object.

2.2.4 Task-based access control

The Task-Based Access Control (TBAC) [TS97] model extends the traditional subject/object-based access control models by including domains that contain task-based contextual information. Access control in this model is granted in steps that are related to the progress of tasks. Each step is associated with a protection state containing a set of permissions. The contents of this set change based on the task. The model, thus, is an active model that allows for dynamic management of permissions as tasks progress to completion. Each step has a disjoint protection state as shown in Figure 2.3. TBAC also supports type, instance, and usage-based accesses. In addition, authorizations have a strict runtime usage, validity, and expiration characteristics. There are several weaknesses to this model. They are described as follows.

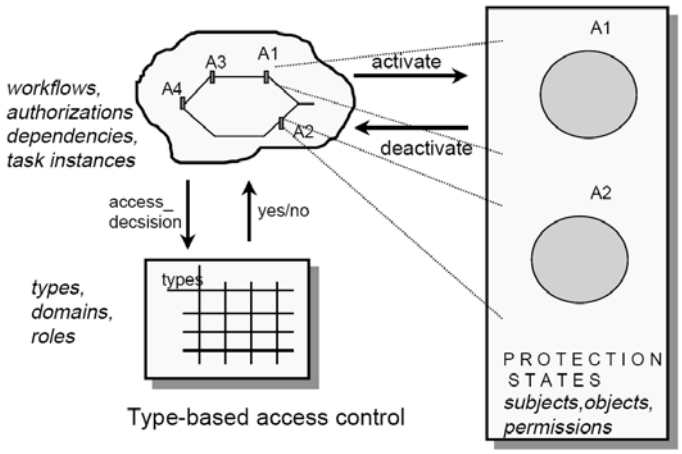


Figure 2.3: Task-based access control.

- TBAC systems recognize the need to incorporate contextual parameters into security considerations; however, it is limited to contexts in relation to activities, tasks, or workflow progress and is implemented mainly by keeping track of usage and validity of permissions. Collaborative systems require a much broader definition of context, and the nature of collaboration cannot always be easily partitioned into tasks with usage counts.
- Permissions are activated and deactivated in a just-in-time fashion. The draw-back of that is that if a central access control module manages permissions, it could introduce several constraints, such as race conditions, across workflows.
- Specification of complex policies and management and delegation and revocation of authorization privileges are very primitive. More fine grained components need to be defined to support dynamic environments motivated by TBAC.

- TBAC can be used effectively for security modeling and enforcement from an application or enterprise point of view and has its advantages over the system-centric approach in subject-object systems. But for most collaborative environments, TBAC is used within other access control models.

2.2.5 Task-role-based access control

The Task-Role-Based Access Control (T-RBAC) [OP03] as an improved model of RBAC putting the tasks between the roles and permissions. In this model permissions are assigned to tasks and tasks are assigned to roles. The T-RBAC model is shown in Figure 2.4. Tasks in the T–RBAC are classified into four classes. Tasks of class *W* and class *A* are used to compose the workflow schema. Tasks of class *S* and class *P* are used to compose the no workflow-oriented tasks. The access rights for the tasks in class *W* and *P* are not inherited to ancestor job positions or business roles. On the contrary the access rights for tasks in class *A* and class *S* are inherited to ancestor job positions or business roles.

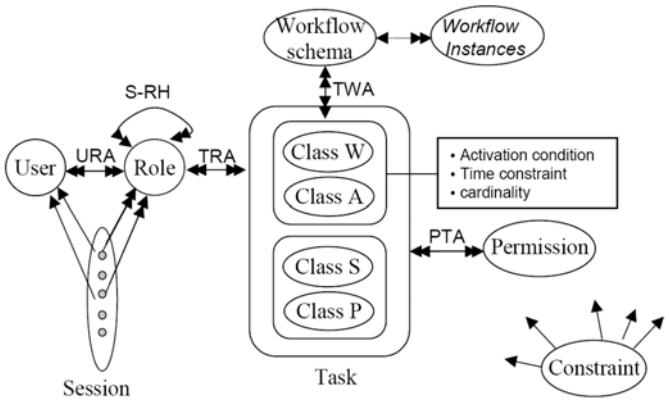


Figure 2.4: Task-role-based access control.

Fundamentally, the T-RBAC still belongs to the RBAC model except it enhances the permission management so that the traditional RBAC can be used to support workflow that is the typical requirements of the enterprise environment. From the view of collaboration, T-RBAC model does not resolve the weaknesses that RBAC has. On the other hand the access right classification and inheritance are complicated.

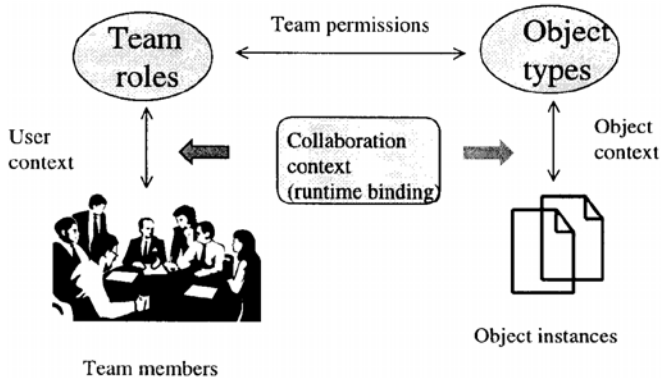


Figure 2.5: Team-based access control.

2.2.6 Team-based access control

The Team-Based Access Control (TMAC) [Tho97] model is shown in Figure 2.5. It defines two important collaboration contexts: user context and object context. User context provides a way of identifying specific users playing a role on a team at any given moment, and object context identifies specific objects required for collaboration purposes. TMAC offers the advantages of RBAC along with provisions to specify fine grained control on individual users in certain roles and on individual object instances.

As a further extension to this approach, Context-based Team Access Control (C-TMAC) [GMPT01] integrates RBAC and TMAC by incorporating context as an entity in the architecture. C-TMAC seeks to include contextual information other than user and object contexts such as time and place. The C-TMAC model is shown in Figure 2.6.

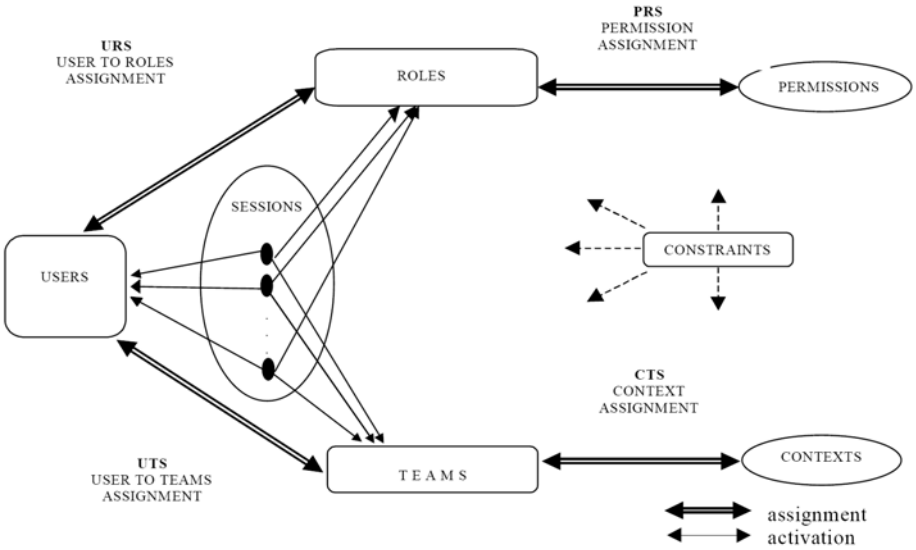


Figure 2.6: Context-based team access control.

TMAC and C-TMAC address to support contextual information and the dynamic nature of team-based environments. However, there are several weaknesses to these models. They are described as follows.

- TMAC extend RBAC with the notion of a team. However, this model has not yet been fully developed, and it is not clear how to incorporate the team concept into a general RBAC framework.
- TMAC and C-TMAC lack the self administration of assignment relations between entities. They also need to reflect the multidimensional definition of rich collaborative contexts, such as organizational entities, workflow tasks and so on. The fine-grained administration of TMAC and C-TMAC entities and relations is necessary to demonstrate applicability and usability of these models.
- Wang [Wan99] suggests the use of both Team- and Role-Based Access Control for Hypermedia environments. The model is general applicability, however, requires additional testing in more realistic settings. The specification, selection, and application of policies in multiple settings also require further investigation.

2.2.7 Spatial access control

Bullock and Benford [BB99] propose a spatial access control for collaborative virtual environments (SPACE). It takes into account the environment of collaboration and exploits it to hide explicit security mechanisms from the end users. The model consists of two components: boundary and access graph. The boundary divides the collaborative environment into regions and accounts for both traversal and awareness within regions. It uses the notion of credentials to allow access within regions. The access graph is used to specify the constraints on movement in the collaboration space and the management of the access requirements in the environment.

This model is concerned only with navigational access requirements in collaborative environments and does not provide for fine-grained control. The SPACE model is not provably secure. It is possible for users to apply the SPACE model and create insecure regions. This model lacks the complexity needed for systems where the level of security provided is important. This model cannot be used unless it is possible to represent an application in terms of regions and boundaries.

2.2.8 Context-aware access control

Covington et al. [CLSD01] extended RBAC with the notion of environment roles for security in context-aware applications. They use roles, called environment roles, to capture environmental state. These roles use role hierarchy, activation, separation and so on, similar to traditional RBAC, to manage the policies and constraints that depend on the context of collaboration. These roles are activated based on environment conditions at runtime. The environment RBAC has been shown to be of use in ubiquitous computing where environment-sensitive information is pervasive. This approach, however, requires further testing within the collaborative systems domain.

2.3 Authorization mechanisms and systems

In many application scenarios one authorization mechanism does not fit all, especially in the collaborative environments such as grid computing systems. A grid system is a virtual organization that is composed of several autonomous domains. Authorization in such a system needs to be flexible and scalable to support multiple authorization mechanisms or security policies. It is one of the major motivations for us to develope the root policy system that is described in Chapter 7. This section introduces some well known authorization mechanisms and systems that are mainly used for providing authorization functionality in distributed systems.

2.3.1 Akenti

The Akenti authorization model [TEM03] is shown in Figure 2.7. In order to access the resources that are protected by an Akenti system, the users connect to the resource gateway to present authenticated X.509 certificates. Akenti provides both attribute and policy assertion functions. It allows the stakeholders to create signed XML certificates containing attribute assertions and policy assertions in the form of policy certificates and use conditions. These certificates express the attributes what a user must have in order to get specific rights to a resource, who is trusted to create use-condition statements, and who can attest to a user's attributes. At the time of access, the resource gateway asks a trusted Akenti server, what access the user has to the resource. The Akenti server finds all the relevant certificates, verifies that

each one is signed by an acceptable issuer, evaluates them, and returns the allowed access.

There are several models for the authorization systems. One is the pull model where the user presents only his authenticated identity to the gatekeeper who finds the policy information for the resource and evaluates the user's access. The other is the push model where the user presents one or more tokens or assertions that grant the holder specific rights to the resource. In this model, the gatekeeper has to verify that the user has the rights to use the tokens and then to interpret the rights that have been presented. There are also hybrids of the two models, such as when a user presents identity information that includes restrictions on his full set of rights, or presents a handle to an authentication/authorization server from which the gatekeeper may pull information about the user and his rights.

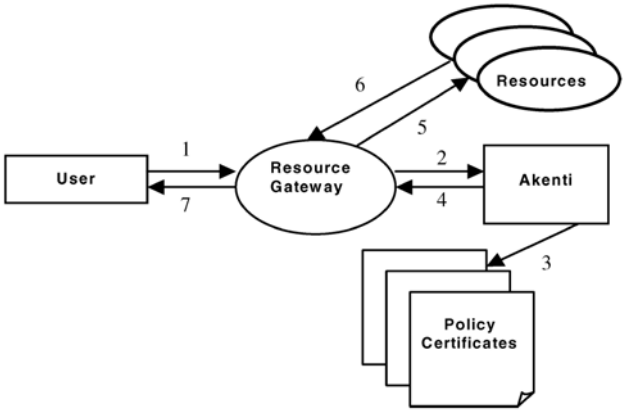


Figure 2.7: Akenti authorization model.

2.3.2 Cardea

Cardea [Lep03] is a distributed authorization system, developed as part of the NASA Information Power Grid, which dynamically evaluates authorization requests according to a set of relevant characteristics of the resource and requester rather than considering specific local identities. Potentially accessed resources within an administrative domain are protected by local access control policies, specified with the XACML syntax, in terms of requester and resource characteristics. The information needed to complete an authorization decision is assessed and collected during the decision process itself. This information is assembled appropriately, either by the requester, an agent, a policy enforcement point (PEP), or a SAML policy decision point (PDP) and presented to an XACML PDP for evaluation. Once obtained, the SAML PDP then returns the final authorization decision for the access request together with any relevant details to the initial requester.

Cardea leverages the XACML model for authorization evaluation and SAML for obtaining assertion data used during the evaluation process. Cardea assumes that the SAML PDP that accepts the initial request is responsible for providing the final authorization decision details to the PEP. The SAML PDP depends upon the content of the initial request to determine the correct XACML PDP to evaluate the request.

2.3.3 CAS

The Community Authorization Service (CAS) [PKWF03] allows for a separation of concerns between site policies and virtual organization (VO) policies. Sites can delegate management of a

subset of their policy space to the VO. CAS functions as a push-model authorization service is shown in Figure 2.8. The steps in the Figure are:

(1) The client sends a signed SAML request to the CAS server to indicate which resources they wish to access and which actions they desire to invoke.

(2) The CAS server establishes the user's identity. Using the identity it determines the rights as established by the VO's policy. It then returns a signed SAML assertion that contains the user's identity and some subset of the user's requested actions

(3) The user presents the SAML assertion to a resource along with an authenticated request. The resource uses the SAML assertion, subject to local policy regarding how much authority was delegated to the CAS service, to authorize the request.

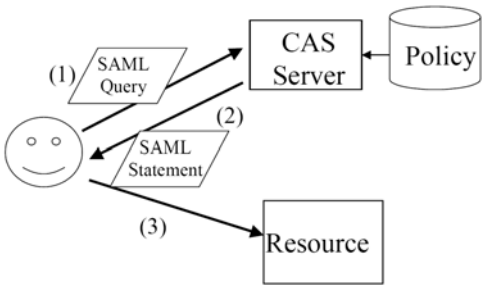


Figure 2.8: CAS architecture.

It is common for a client to ask for an assertion containing a complete set of rights they may have on a given resource, set of resources, or even on all resources for which a CAS server has authority. Since the SAML statement returned is typically valid for a number of hours, an assertion with multiple rights allows the user to undertake a number of different actions without having to re-contact the CAS server.

2.3.4 PRIMA

PRIMA [LAKK03] is a system for privilege management and access control. PRIMA leverages X.509 Attribute Certificates (ACs) to carry privilege and policy statements. An access control decision function authorizes requests based on the combination of subject attributes with resource policies and provisions low-level access control enforcement functions with decision qualifications. These enforcement mechanisms assign and configure local user accounts dynamically and leverage POSIX file system ACLs and the XML based grid ACLs to assign and manage fine-grained access rights.

PRIMA uses a hybrid authorization sequence. When a subject requests a service to a resource, an Authorization Enforcement Function (AEF) queries an Authorization Decision Function (ADF) for a coarse access decision (yes/no) and a handle to an execution environment within which the service should be executed. This conforms to the flow outlined in the pull scenario. However, the ADF also configures an execution environment for the service with the least amount of fine-grained rights required by the service. This second step in which decision qualifiers are provisioned to an AEF follows the information flow described by the push sequence.

PRIMA's ADF makes access decisions based on subject attributes and privilege management policies. The subject can specify what attributes will be considered by the ADF.

The PRIMA approach constitutes an additive, capability-based security model where missing or deliberately omitted attributes will result in fewer access permissions. The ADF is co-located with the resource and communicates with the gatekeeper's authorization module via XACML requests and responses. Subject privileges are supplied to the ADF in the form of a dynamically created policy that is specific to the request and encompasses all the privileges that the subject presented.

2.3.5 PERMIS

PERMIS [CO03] is an attribute based authorization infrastructure comprising the following components: an authorization policy written in XML and put in an X.509 attribute certificate (AC), user authorization tokens are also put in the X.509 ACs, one or more LDAP directory servers that are used to store the ACs, an Access Decision Function (ADF), the API to the ADF and the tools for creating ACs and policies. The PERMIS privilege verification system is shown in Figure 2.9.

PERMIS currently works in the authorization pull model. The user contacts the resource protected by an AEF, which in turn contacts the PERMIS ADF. The PERMIS ADF makes the decision, based on the user's attributes obtained from the user's ACs and the policy for the resource, and returns this to the AEF. The AEF then enforces this decision on behalf of the resource. The PERMIS decision-making is two stages. The first stage, which typically takes place at user log on, is to evaluate the user's ACs according to the policy, and to reject untrustworthy ones. Valid attributes are kept and returned to the AEF. The second stage, which typically takes place when the user attempts to perform some action, is to make a grant or deny decision based on the user's validated attributes and the policy.

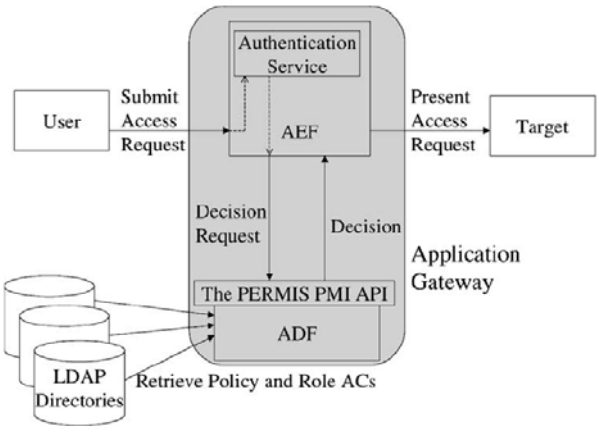


Figure 2.9: PERMIS privilege verification subsystem.

2.3.6 VOMS

The Virtual Organization Membership Service (VOMS) [ACCA05] is developed in the framework of the European DataGrid and DataTAG projects. VOMS allows a fine grained control of the use of the resources both to the users' organizations and to the resource owners. VOMS supports both the push and pull models. The VOMS system is shown in Figure 2.10.

In the pull model, the VOMS server is periodically contacted by each resource using HTTPS, and requests are made for listings of members' certificate subject names matching specified criteria, for example, all the members of a given group. Since the VOMS HTTPS

server identifies itself by its own certificate at the start of the connection, no additional signing of this information is used.

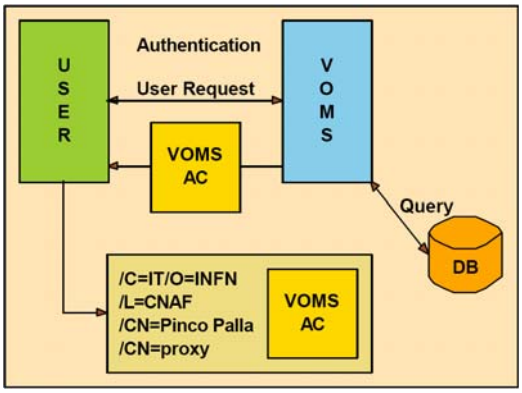


Figure 2.10: VOMS system.

In the push model, subjects contact the VOMS server. Client tools identify the subject to the VOMS using a GSI proxy certificate [TEFW02], and request the desired set of attribute assertions for the subsequent session. The VOMS server issues a signed text block of name-value pairs containing those requested attributes that the subject is entitled to. The text block starts with the subject and VOMS certificate names, VO name, assertion valid time. One or more Group, Role, Capability triplets then follow is being asserted. This signed assertion is then included as an extension in a new GSI proxy certificate for the subject, generated by the subject's client software.

Chapter 3

TT-RBAC Model

3.1 Introduction

Collaborative systems are becoming a popular means of providing efficient and scalable access to distributed computing capabilities. In collaborative systems, a set of participants (organizations or users) share the computing resources, such as data, compute cycles, storage space, or online services, aimed at achieving common tasks. Balancing the competing goals of collaboration and security is difficult because interaction in collaborative systems is targeted towards making people, information, and resources available to all who need it, whereas information security seeks to ensure the availability, confidentiality, and integrity of these elements while providing it only to those with proper authorization.

Traditionally, an access control decision is made based on subjects and objects. When there are many subjects and objects being involved, the subject-object model cannot provide satisfied security management. A variety of access control models have been developed over the years in response to system and security administration requirements. These access control models have been motivated by the need to reduce the security administration overhead commonly associated with low-level subject-object permissions. Models that incorporate additional contextual information and support higher-level policy abstractions can simplify policy administration by reducing the semantic gap between enterprise-level policies and policies that can be directly enforced within a system. Abstractions, such as “role”, “team” and “task” are developed to model contextual information associated with organizational roles, responsibilities and collaborative activities.

Currently there is still no access control model that rigorously defines the relations among team, task and Role-Based Access Control (RBAC) [SCFY96, FSGK01] entities. Motivated by this requirement, we defined a Team and Task based RBAC (TT-RBAC) access control model [ZRMA05, ZM07b] that extends the NIST RBAC model [FSGK01] through adding sets of two basic data elements called teams and tasks. The TT-RBAC model defines four model components through which entity relations under different situations are defined. The functional requirements for these model components are also specified.

The rest of this chapter is organized as follows. Section 3.2 introduces the major related work. Section 3.3 gives an overview of the NIST RBAC reference model. Section 3.4 introduces the Core TT-RBAC model. Section 3.5 investigates the Hierarchal TT-RBAC model. Section 3.6 investigates the Constrained TT-RBAC model. Section 3.7 provides an overview of

the TT-RBAC functional specification in three areas: administrative operations, administrative review capabilities and system functions. Finally, Section 8 summarizes the results of this chapter.

3.2 Related work

In Section 2.2 we systemically introduced the access control requirements for collaboration, examined existing access control models as applied to collaborative environments in light of these requirements, and especially highlighted the weaknesses of these models. In this section we summarize the work that directly related to the motivation of developing the TT-RBAC model.

The essence of RBAC [SCFY96] is that permissions are associated with roles, and users are made members of appropriate roles thereby acquiring the roles' permissions. Permissions can be granted to roles or revoked from the roles as needed. Users can be easily reassigned from one role to another without modifying the underlying access structure. RBAC is thus more scalable than user-based security specifications and greatly reduces the cost and administrative overhead associated with fine-grained security administration at the level of individual users, objects or permissions. But subsequent attempts to apply RBAC in collaborative environments revealed some of RBAC's limitations. (1) RBAC lacks the ability to specify a fine-grained control on individual users in certain roles and on individual object instances. For collaborative environments, it is insufficient to have role permissions based on object types. Rather, it is often the case that a user in an instance of a role might need a specific permission on an instance of an object [TAP05]. (2) As a passive security model, RBAC does not take into account the impact of context. In collaborative environment, it is highly demanded that an access control system needs to consider the overall context associated with any collaborative activity. (3) Authorization constraints are an important aspect of RBAC and a powerful mechanism for laying out higher-level organizational policy. However, the specification of such constraints has not been discussed in the RBAC model. (4) On the other hand, RBAC does not provide an abstraction to capture a set of collaborative users operating in different roles.

In RBAC models, groups are defined on the basis of users performing the same role. While the strengths of this approach have been discussed, there is a limitation when one considers instances of roles collaborating on group work. Teams, on the other hand, appear to be a natural way of grouping users in an enterprise or organization and associating a collaborative context with the activity to be performed. TeaM-based Access Control (TMAC) proposed by Thomas [Tho97] defines two important aspects of the collaborative context: user context and object context. User context provides a way of identifying specific users playing a role in a team at any given moment, and object context identifies specific objects required for collaborative purposes. TMAC offers the advantages of RBAC along with provisions to specify fine grained control on individual users in certain roles and on individual object instances. However, this model has not yet been fully developed, and it is not clear how to incorporate the team concept into a general RBAC framework.

The Context-based TeaM Access Control (C-TMAC) [GMPT01] as an extension of the TMAC approach addresses the issue of incorporating broader contextual information such as the time and location of access into the TMAC model. This model extends the TMAC proposal in two directions. First, it gives a framework to integrate TMAC concepts with RBAC. Second, it extends TMAC to use general contextual information, such as time, place, and so on, other than user and object contexts. But this model has two obvious flaws. The first is that the team

defined in C-TMAC does not have any other constraint except the associated context constraints. So it is impossible to strictly define a team's permissions. One person joins or leaves a team may dramatically affect the team's permissions. The second is that a team member will automatically obtain all the permissions of the team. In many application scenarios this should be unacceptable. The notion of integrating the concept of team with RBAC was also pursued by Wang in [Wang99], where the focus was cooperative hypermedia environments. The model's general applicability, however, requires additional testing in more realistic settings. The specification, selection and application of policies in multiple settings also require further investigation.

Access control models have also been developed to synchronize access permissions with ongoing tasks and workflow instances. Task-Based Authorization Control (TBAC) [TS97] introduces a family of models that support the specification of active security models. In TBAC permissions are activated and deactivated based on current task state. This allows permissions to track the overall progress of a task and supports secure workflow management. TBAC can be used effectively for security modeling and enforcement from an application or enterprise point of view and has its advantages over the system-centric approach in subject-object systems. But for most collaborative environments, TBAC is used within other access control models. Kang et al. [KPF01] present access control mechanisms for inter-organizational workflow to support the autonomy of the participating organizations. At runtime a workflow is split into multiple autonomous workflows, one for each participating organization. This approach permits the grouping of related tasks into a more abstract, higher-level task, while access is controlled at the role level.

The Task-Role-Based Access Control (T-RBAC) [OP2003] puts the tasks between the roles and permissions. In this model permissions are assigned to tasks, and tasks are assigned to roles. Tasks in the T–RBAC model are classified into four classes that are used to model workflow related and no workflow related tasks. The workflow related access rights are bound and activated during the execution of the corresponding task instances. Fundamentally, the T-RBAC still belongs to the RBAC model except it enhances the permission management so that the traditional RBAC can be used to support workflow that is the typical requirement in enterprise environments. From the view of collaboration, the T-RBAC model does not overcome the weakness of RBAC.

Coalition-Based Access Control (CBAC) [CTWS02] addresses the issue of information sharing and security in dynamic coalition. A coalition consists of two or more different organizations that temporarily work together to achieve a common goal. Coalitions can be formed dynamically, and each coalition member may share a number of information resources with other coalition members. Each party must be able to individually tailor the access rules for their content according to the status of other coalition members. The CBAC model incorporates aspects of RBAC, TMAC and TBAC, along with coalition-targeted abstractions and context information. However, the implementation of CBAC model may prove challenging. In particular, system context information, including user sessions and the activation states of missions, functions, tasks, etc., may be complex to implement.

From previous introduction, we know user, role, team, task and permission are the five basic entities in collaborative environments. But none of these related research efforts systemically discusses the relations among these entities. Motivated by this requirement, we developed the Team and Task based RBAC (TT-RBAC) access control model that integrates RBAC with teams and tasks. Its primary definition was originally presented in [ZRMA05] and then improved in [ZM07b]. Contextual information is another important factor in access control systems. It decides that an access control model belongs to either active or passive security

model. Contextual information affects an access control system through activating/deactivating its basic entities. We will investigate context-aware TT-RBAC in Chapter 4.

3.3 NIST RBAC model overview

The NIST RBAC model [FSGK01] is defined in terms of four model components: *Core RBAC*, *Hierarchical RBAC*, *Static Separation of Duty Relations*, and *Dynamic Separation of Duty Relations*. Core RBAC defines a minimum collection of RBAC elements, element sets, and relations in order to completely achieve a role-based access control system. Hierarchical RBAC component adds relations for supporting role hierarchies. Static Separation of Duty Relations adds exclusivity relations among roles with respect to user assignments. Dynamic Separation of Duty Relations defines exclusivity relations with respect to roles that are activated as part of a user's session. The NIST RBAC model is defined in Figure 3.1.

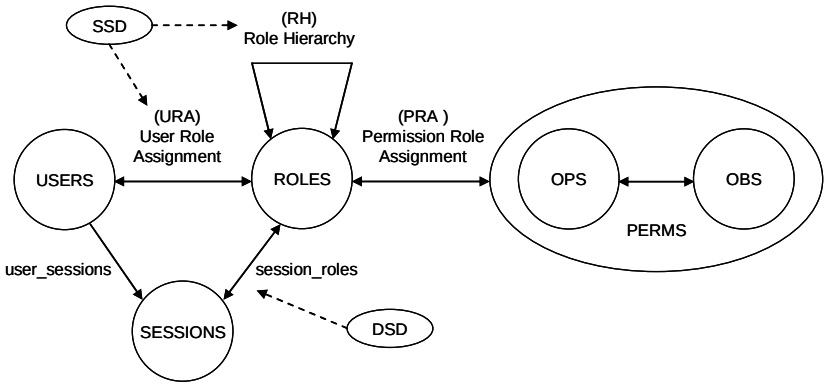


Figure 3.1: NIST RBAC.

3.3.1 Core RBAC

Core RBAC model element sets and relations are defined in Figure 1 (without RH, SSD, DSD). Core RBAC includes sets of five basic data elements called users (USERS), roles (ROLES), objects (OBS), operations (OPS) and permissions (PERMS). The RBAC model as a whole is fundamentally defined in terms of individual users being assigned to roles and permissions being assigned to roles. As such, a role is a means for naming many-to-many relationships among individual users and permissions. In addition, the Core RBAC model includes a set of sessions (SESSIONS) where each session is a mapping between a user and an activated subset of roles that are assigned to the user.

A *user* is defined as a human being or an autonomous agent. A *role* is a job function within the context of an organization with some associated semantics regarding the authority and responsibility conferred on the user assigned to the role. *Permission* is an approval to perform an operation on one or more objects. An *operation* is an executable image of a program, which upon invocation executes some function for the user. For example, within a file system, operations might include read, write, and execute. An *object* is an entity that contains or receives information. The objects can represent information containers, such as files and directories in an operating system or contents within a database management system. Objects can also represent exhaustible system resource, such as printer, disk space, and CPU cycle. The set of objects covered by RBAC includes all of the objects listed in the permissions that are

assigned to roles. The types of operations and objects that RBAC controls are dependent on the type of system in which they will be implemented.

Central to RBAC is the concept of role relations, around which a role is a semantic construct for formulating policy. Figure 3.1 illustrates *user role assignment* (*URA*) and *permission role assignment* (*PRA*) relations. The arrows indicate a many-to-many relationship. For example, a user can be assigned to one or more roles, and a role can be assigned to one or more users. Each *session* is a mapping of one user to possibly many roles that he/she is assigned and each user is associated with one or more sessions. The permissions available to the user are the permissions assigned to the roles that are activated across all the user's sessions. The following is the Core RBAC definition.

- *USERS*, *ROLES*, *OPS*, and *OBS* (users, roles, operations, and objects, respectively).
- $URA \subseteq USERS \times ROLES$, a many-to-many mapping user-to-role assignment relation.
- $assigned\_users(r : ROLES) \rightarrow 2^{USERS}$, the mapping of role *r* onto a set of users. Formally: $assigned\_users(r) = \{u \in USERS \mid (u, r) \in UA\}$.
- $PERMS = 2^{(OPS \times OBS)}$, the set of permissions.
- $PRA \subseteq PERMS \times ROLES$, a many-to-many mapping permission-to-role assignment relation.
- $assigned\_permissions(r : ROLES) \rightarrow 2^{PERMS}$, the mapping of role *r* onto a set of permissions. Formally: $assigned\_permissions(r) = \{p \in PERMS \mid (p, r) \in PA\}$.
- $op(p : PERMS) \rightarrow \{op \subseteq OPS\}$, the permission-to-operation mapping, which gives the set of operations associated with permission *p*.
- $ob(p : PERMS) \rightarrow \{ob \subseteq OBS\}$, the permission-to-object mapping, which gives the set of operations associated with permission *p*.
- *SESSIONS*, the set of sessions.
- $user\_sessions(u : USERS) \rightarrow 2^{SESSIONS}$, the mapping of user *u* onto a set of sessions.
- $session\_roles(s : SESSIONS) \rightarrow 2^{ROLES}$, the mapping of session *s* onto a set of roles. Formally: $session\_roles(s_i) \subseteq \{r \in ROLES \mid (session\_user(s_i), r) \in UA\}$.
- $avail\_session\_perms(s : SESSIONS) \rightarrow 2^{PERMS}$, the permissions available to a user in a session, $\bigcup_{r \in session\_roles(s)} assigned\_permissions(r)$.

3.3.2 Hierarchal RBAC

This model component introduces Role Hierarchies (RH) as shown in Figure 3.1 (without SSD, DSD). A hierarchy is mathematically a partial order defining a seniority relation between roles, whereby senior roles acquire the permissions of their juniors and junior roles acquire users of their seniors. Role hierarchies define an inheritance relation among roles. Inheritance has been described in terms of permissions; that is, r_1 inherits role r_2 if all privileges of r_2 are also privileges of r_1. This standard recognizes both general and limited role hierarchies.

General role hierarchies support the concept of *multiple inheritances*, which provide the ability to inherit permission from two or more role sources and to inherit user membership from two or more role sources. The following is the general role hierarchy definition.

- $RH \subseteq ROLES \times ROLES$ is a partial order on ROLES called the inheritance relation, written as $\geq$, where $r_1 \geq r_2$ only if all permissions of $r2$ are also permissions of $r1$, and all users of $r1$ are also users of $r2$. Formally:

  $r_1 \geq r_2 \Rightarrow authorized\_permissions(r_2) \subseteq authorized\_permissions(r_1) \wedge authorized\_users(r_1) \subseteq authorized\_users(r_2)$.

- $authorized\_users(r : ROLES) \rightarrow 2^{USERS}$, the mapping of role r onto a set of users in the presence of a role hierarchy. Formally:

  $authorized\_users(r) = \{u \in USERS \mid r' \geq r\ (u, r') \in UA\}$.

- $authorized\_permissions(r : ROLES) \rightarrow 2^{PERMS}$, the mapping of role r onto a set of permissions in the presence of a role hierarchy. Formally:

  $authorized\_permissions(r) = \{p \in PERMS \mid r \geq r'\ (p, r') \in PA\}$.

Roles in a limited role hierarchy are restricted to a single *immediate descendant*. Although limited role hierarchies do not support multiple inheritances, they nonetheless provide clear administrative advantages over Core RBAC. r_1 is an immediate descendent of r_2 by $r_1 >> r_2$, if $r_1 \geq r_2$, but no role in the role hierarchy lies between r_1 and r_2. That is, there exists no role of r_3 in the role hierarchy such that $r_1 \geq r_3 \geq r_2$, where $r_1 \neq r_2$ and $r_2 \neq r_3$. The limited role hierarchies can be defined as a restriction on the immediate descendants of the general role hierarchy. The restriction defined for the limited role hierarchies is given below.

$\forall r, r_1, r_2 \in ROLES, r \geq r_1 \wedge r \geq r_2 \Rightarrow r_1 = r_2$.

3.3.3 Constrained RBAC

Constrained RBAC model element sets and relations are shown in Figure 3.1. Constrained RBAC adds separation of duty relations to the RBAC model. Separation of duty relations are used to enforce conflict of interest policies that organizations may employ to prevent users from exceeding a reasonable level of authority for their positions. The NIST RBAC allows for both static and dynamic separation.

Static Separation of Duty (SSD) relations provide the capability to address potential conflict-of-interest issues at the time a user is assigned to a role. The static constraints defined in this model are limited to those relations that place restrictions on sets of roles and in particular on their ability to form URA relations. This means that if a user is assigned to one role, the user is prohibited from being a member of a second role. The following is the static separation of duty definition.

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

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

- In the presence of a role hierarchy *static separation of duty* is redefined based on authorized users rather than assigned users as follows.

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

Dynamic Separation of Duty (DSD) relations limit the availability of the permissions over a user's permission space by placing constraints on the role that can be activated within or across a user's sessions. DSD allows a user to be authorized for two or more roles that do not create a conflict of interest when acted on independently, but produce policy concerns when activated simultaneously. The following is the dynamic separation of duty definition.

- $DSD\subseteq\left(2^{ROLES}\times N\right)$ is a collection of pairs (*rs*, *n*) in *Dynamic Separation of Duty*, where each *rs* is a role set and *n* is a natural number ≥2, with the property that no subject may activate *n* or more roles from the set *rs* in each $dsd\in DSD$. Formally:

$$\forall rs\in 2^{ROLES}, n\in N, (rs,n)\in DSD\Rightarrow n\geq 2\wedge |rs|\geq n, \text{and}$$
$$\forall s\in SESSIONS, \forall rs\in 2^{ROLES}, \forall role\_subset\in 2^{ROLES}, \forall n\in N, (rs,n)\in DSD,.$$
$$role\_subset\subseteq rs, role\_subset\subseteq session\_roles(s)\Rightarrow |role\_subset|<n$$

3.4 Core TT-RBAC

In this section we first give an application case, observe the basic requirements from the standpoint of access control, and then use several existing relevant access control models to analyze this case. Next we give the general description of TT-RBAC.

3.4.1 Case analysis

A typical clinical workflow scenario may contain the following steps: admission, diagnosis, treatment and discharge. A number of clinical staffs are involved in various roles (job functions) in providing care at different steps in the workflow. These staffs are often organized into various administration and care teams (groups). In a large hospital, at each step there may be several parallel groups that perform similar tasks in one department. The only difference among of these groups is that they care different patients. Here we focus on the residency division in which we assume there are several parallel care groups. Each group consists of doctors and nurses. From the standpoint of access control, we observe that the following basic security requirements should be considered.

1. The permissions assigned to a clinical staff should be based on their qualifications and responsibilities. For example, only the doctors, not nurses, may order lab test.
2. The team members can only access the medical records of patients who the team is caring. Thus, although a doctor may have the right to order a lab test according to his/her qualifications and responsibilities, he/she should do so for a patient only when the patient is being cared by his/her team.
3. For the purpose of fine-grained access control, in each group the doctors have different

responsibilities based on their qualifications such as *consultant*, *associate consultant*, *principal doctor* and *residency doctor*. The doctors who have higher level position can modify the medical treatment given by the doctors who have low level position, but the doctors who have lower level position cannot modify the medical treatment given by the doctors who have higher level position.

Now we use access matrix, role-based access control and team-based access control to analyze this application scenario.

- *Access matrix*. There are two approaches to implement the access matrix. One is the access control lists (ACLs); the other is the capability lists (C-lists). In above application scenario, the operations to patient data are mainly related to database queries. There are no well-defined persistent objects to put an ACL, so ACLs model can be harder to apply in this scenario.

  C-lists are dual approach to ACLs. Each subject is associated with a list indicating what operations the subject are authorized to perform on objects. When there many users are involved, user-groups could be used to simply the C-lists administration. Users who have the same privileges are organized into a group, and the permissions are granted to the group. In above scenario, the users are organized into groups, and the users in a group are further organized into some subgroups according to their qualifications and responsibilities, e.g. consultant group, nurse group and so on. The permissions are assigned to the subgroups, so the requirements 1 and 3 are satisfied. Patients are assigned to the group, and the group users can only access the data of patients who are assigned to the group, then the requirement 2 is satisfied. The major drawbacks of this approach are:

  - To accurately determine what permissions have been granted to a user or withdraw all permissions assigned to a user, administrators have to examine all the subsystems and possible all the groups in each subsystem. Since there are too many objects to query, the state of access control with respect to a particular user is sometimes rarely verified.
  - As group collections grow, the number of places where administrators need to manage permissions grows. It requires consistent practice and coordination among administrators and precise definitions of user groups. For example, one administrator *A* assigned user *Peter* to the group *G*, but he/she did not tell this assignment to another administer *B*, then administrator *B* does not know *Peter* has some new privileges. Because one administrator normally does not check all the user groups to find what privileges have been assigned to a given user. These processes slow down the administrative process.

- *Role-based access control*. We assume there are five subgroups defined in each group; they are *consultant*, *associate consultant*, *principal doctor* and *residency doctor* and *nurse* in above scenario. If RBAC is used, then we have to define five roles for each group, assign the users to these roles, and assign the permissions to these roles in the granularity of patient, so RBAC is not suitable in this scenario.

  Giuri and Iglio [GI97] proposed Role Template (RT) that is used to define *parameterized roles*. This work is based on extending privileges to include parameterized constraints that are evaluated against the content of the requesting subject and the requested object at access time. A role is regarded as a set of privileges, so the parameters of a role template correspond to those of its privileges. For example, instead of defining an own *Constant* role for each group, a generic parameterized role may be defined. Users are assigned to a parameterized role instance such as *Consultant*<*group*1>, and the name of the

group is used as a parameter for the operation assignments. Hence, according to the concrete parameter value, a user may only access patient records that are allocated to his/her group. The major drawbacks of the RT approach are:

- The RT approach does not reduce the number of roles compared to the pure RBAC approach, but it improves the role management and simplifies permission-assignments. Because roles are created from role templates, and the permissions are assigned to role templates rather than the role instances. Defining too many roles will certainly bring the administration problems.
- In this application scenario group users and roles are not defined in intuitive way. In order to decide who are in a given group, the administrators have to examine all the parameterized roles to find out the roles that have the given parameter and then find out who have been assigned to these roles.

- *Team-based access control*. The TMAC proposed by Thomas [Tho97] has not yet been fully developed, and it is not clear how to incorporate the team concept into a general RBAC framework, so it cannot be used to analyze a really application scenario.

  C-TMAC [GMPT01] as an extension of the TMAC gives a framework to integrate TMAC concepts with RBAC. In C-TMAC users are organized in teams, the permissions that are available to a user is the combination of his/her session roles and the team-roles that are other team members' session roles, teams are assigned with context constraints that are used to filter the permissions that are available to a team member. Through the context constraints the requirement 2 can be achieved. But as all the team members have the same permissions in a team, so the requirements 1 and 3 are violated.

From above analysis we know C-lists and RT approaches are suitable to the given application scenario. But each of them also has some obvious drawbacks. In the rest of this chapter we systemically introduce the TT-RBAC model that is specially developed for the access control in collaborative environments. It can overcome the shortcomings caused by the existing access control models when they are used in collaborative environments.

3.4.2 Core TT-RBAC definition

TT-RBAC model is defined through adding two basic data element sets called teams (TEAMS) and tasks (TASKS) to the NIST RBAC model. Similar to the NIST RBAC Model, TT-RBAC model is defined in terms of four model components: *Core TT-RBAC*, *Hierarchical TT-RBAC*, *Static Separation of Duty Relations*, and *Dynamic Separation of Duty Relations*. Core TT-RBAC model element sets and relations are defined in Figure 3.2.

Besides the relations defined in the Core RBAC, the TT-RBAC model as a whole is fundamentally defined in terms of individual users being assigned to teams, roles and tasks being assigned to teams, and permissions being assigned to tasks. The relations among of them are many-to-many. In addition, the Core TT-RBAC model includes a set of sessions (SESSIONS) where each session is a mapping onto an activated subset of roles and an activated subset of teams that are assigned to the user.

A *task* is a fundamental unit of business work or activity. Business process (workflow) is defined as a set of tasks that are connected to achieve a common goal. A task may belong to business process or not belong to business process. For example, an online selling process may consist of *receive customer order*, *check customer order*, *check product stock*, *approval order* and *ship product* five steps. Each step can be modeled to one task that encapsulates a set of

permissions needed to complete it. The whole selling process could also be modeled into one task *online selling* that comprises the subtasks defined previous. *Monitor sales* may not belong to any process, and can also be modeled to one task.

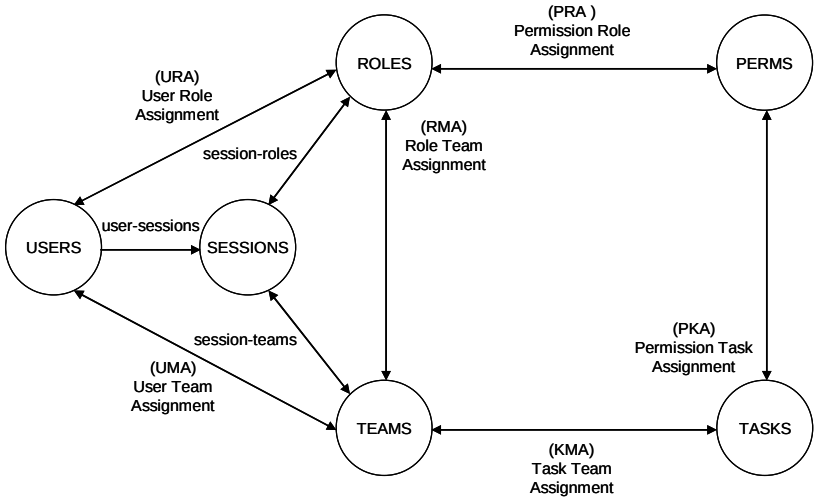


Figure 3.2: Core TT-RBAC.

In TT-RBAC each task associates with a set of permissions that are needed to execute it. From the view of permission assignment, the task has the similar functionality with the role. But there are three major differences between roles and tasks. The first difference is that their assignment targets are different. Roles are assigned to users and tasks are assigned to teams. The second difference is that each task instance always relates to some states such as active and inactive. A task instance may have a lifetime and a life-cycle associated with it, and it may be in various states during its life-cycle. In TT-RBAC we specify that each task instance in its life-cycle must include the "active" state, and only in this state the permissions encapsulated in the task can be used for making access control decisions. For roles there is no such restriction. The third difference is related to role and task managements. We can define as many tasks as we want and these tasks can be in any granularity. On the contrary, we cannot define too many roles in a system. It will bring privilege administration problem. The permissions assigned to a role should be type-based. The permissions assigned to a task can be type-based or instance-based. For example, we can define a task for approving all the customer orders or define subtasks for approving each individual customer orders. If some of these subtasks are assigned to a team, then the team gets the privileges to approve these selected customer orders. But for the role definition, we should not define roles for approving each individual customer orders.

A *team* encapsulates a collection of users in various roles and a set of roles with the objective of accomplishing specific tasks. A team member can perform what tasks is decided by what roles he/she can activate in the team. A team member can only activate the roles that are assigned to both him/her and the team. The team roles decide the maximum permissions that a team can perform, and the team tasks decide the maximum permissions assigned to a team. The team members decide who can active his/her roles and perform the team tasks in a team. So the team defines a small RBAC application zone, through which we can preserve the advantages of scalable security administration that RBAC-style models offer and yet offers the flexibility to activate permissions for individual users on specific object instance.

Central to TT-RBAC is the concept of team relations, around which users, roles and tasks

are connected together. Through roles and tasks, the permissions are introduced into the teams. Besides the entity relations defined in the NIST RBAC model, Figure 3.2 illustrates *user-team assignment* (*UMA*), *role-team assignment* (*RMA*), *task-team assignment* (*KMA*) and *permission-task assignment* (*PKA*) relations defined in TT-RBAC model. The arrows indicate many-to-many relationships. For example, a user can be assigned to one or more teams, and a team can be assigned to many users. These arrangements provide great flexibility and granularity for assigning permissions to users. For a team, the user-team relations decide which users can be the team members, the role-team relations decide which roles can be activated by the team members, and the task-team relations decide which permissions can be performed by the team members. For a team member, he/she can only activate the roles that are assigned to both him/her and the team, so the team members may have different privileges. These characteristics enable a team to organize a collection of users with different privileges collaborating together to complete some specific tasks assigned to the team. Any increase in the flexibility of controlling access to resources also strengthens the application of the principle of least privilege.

Each session is a mapping of one user to possible many roles and teams, that is, a user establishes a session during which the user activates some subset of roles and some subset of teams that he/she is assigned. Each session is associated with a single user and each user is associated with one or more sessions. In TT-RBAC, besides the functions defined in the NIST RBAC, a set of team and task related functions are defined. For example, the function *session_teams* gets the teams activated by a session, the function *session_team_roles* gets the roles activated in a session team, and the function *session_team_task*s gets the tasks activated in a session team. The permissions available to the user through a session team are the intersection of the permissions gotten through the session team roles and the permissions gotten through session team tasks. In a session, the permissions available to the user are the union of the permissions gotten through session roles and the permissions gotten through session teams. The permissions available to the user are the union of the permissions available across all the user's sessions.

We summarize the above in the following definition.

Definition 1. Core TT-RBAC.

- *USERS*, *ROLES*, *PERMS*, *TEAMS*, and *TASKS* are users, roles, permissions, teams, and tasks, respectively.
- $UMA \subseteq USERS \times TEAMS$, a many-to-many mapping user-to-team assignment relation.
- $assigned\_team\_users(m : TEAMS) \rightarrow 2^{USERS}$, the mapping of team *m* onto a set of users. Formally: $assigned\_team\_users(m) = \{u \in USERS \mid (u, m) \in UMA\}$.
- $RMA \subseteq ROLES \times TEAMS$, a many-to-many mapping role-to-team assignment relation.
- $assigned\_team\_roles(m : TEAMS) \rightarrow 2^{ROLES}$, the mapping of team *m* onto a set of roles. Formally: $assigned\_team\_roles(m) = \{r \in ROLES \mid (r, m) \in RMA\}$.
- $KMA \subseteq TASKS \times TEAMS$, a many-to-many mapping task-to-team assignment relation.
- $assigned\_team\_tasks(m : TEAMS) \rightarrow 2^{TASKS}$, the mapping of team *m* onto a set of tasks. Formally: $assigned\_team\_tasks(m) = \{k \in TASKS \mid (k, m) \in KMA\}$.
- $PKA \subseteq PERMS \times TASKS$, a many-to-many mapping permission-to-task assignment

relation.

- $assigned\_task\_permissions(k : TASKS) \rightarrow 2^{PERMS}$, the mapping of task *k* onto a set of permissions. Formally:

  $assigned\_task\_permissions(k) = \{p \in PERMS \mid (p,k) \in PKA\}$.

- *SESSIONS*, the set of sessions.

- $session\_teams(s : SESSIONS) \rightarrow 2^{TEAMS}$, the mapping of session *s* onto a set of teams. Formally:

  $session\_teams(s) \subseteq \{m \in TEAMS \mid (session\_user(s), m) \in UMA\}$.

- $session\_team\_roles(s : SESSIONS, m : TEAMS) \rightarrow 2^{ROLES}$, the mapping of team *m* onto a set of roles in a session *s*. Formally:

  $$session\_team\_roles(s,m) \subseteq \begin{Bmatrix} r \in ROLES \mid (session\_user(s), r) \in URA \wedge \\ (session\_user(s), m) \in UMA \wedge (r,m) \in RMA \end{Bmatrix}.$$

- $session\_team\_tasks(s : SESSIONS, k : TASKS) \rightarrow 2^{TASKS}$, the mapping of team *m* onto a set of tasks in a session *s*. Formally:

  $$session\_team\_tasks(s,m) \subseteq \begin{Bmatrix} k \in TASKS \mid (session\_user(s), m) \in UMA \wedge \\ (m,k) \in KMA \end{Bmatrix}.$$

- $session\_team\_permissions(s : SESSIONS, m : TEAM) \rightarrow 2^{PERMS}$, the permissions available to a team in a session,

  $$\bigcup_{r \in session\_team\_roles(s,m)} assigned\_role\_permissions(r) \cap$$
  $$\bigcup_{k \in session\_team\_tasks(s,m)} assigned\_task\_permission(k)$$
  .

- $session\_permissions(s : SESSIONS) \rightarrow 2^{PERMS}$, the permissions available to a user in a session,

  $$\bigcup_{r \in session\_roles(s)} assigned\_role\_permissions(r) \cup$$
  $$\bigcup_{m \in session\_teams(s)} session\_team\_permissions(s,m)$$
  .

3.5 Hierarchical TT-RBAC

This model component introduces team hierarchies (MH) and task hierarchies (KH) as shown in Figure 3.3. Similar to role hierarchies, team and task hierarchies are also natural means of reflecting organization's lines of authority and responsibility.

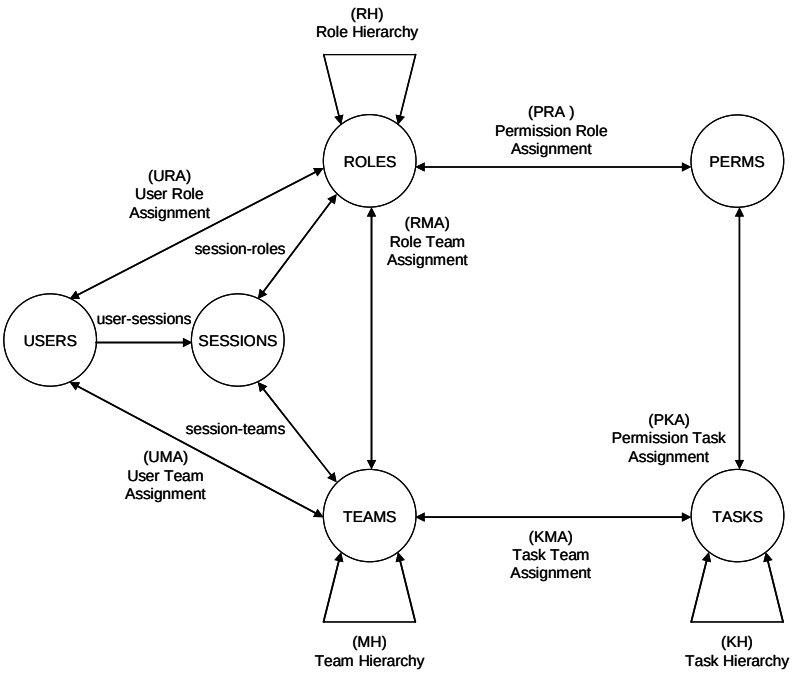


Figure 3.3: Hierarchical TT-RBAC.

3.5.1 Team hierarchies

Team hierarchies define inheritance relations among teams. Team inheritance is described in terms of users; that is, team m_2 inherits team m_1 if all team members of m_2 are also team members of m_1. In TT-RBAC team roles and tasks cannot be inherited through team hierarchies, i.e. the application scope of the team roles and tasks are confined to the team to which they are assigned. So the team hierarchy is only used to describe the user relationships among the teams. For example, there is a team hierarchy in which m_2 inherits m_1. The m_1 has team role permissions $\{p_1, p_2, p_3\}$ and team task permissions $\{p_2, p_3, p_4\}$, so the available permissions in m_1 are $\{p_2, p_3\}$. The m_2 has team role permissions $\{p_4, p_5, p_6\}$, and team task permissions are $\{p_5, p_6, p_7\}$, so the available permissions in m_2 are $\{p_5, p_6\}$. A user possesses the permissions $\{p_3, p_4, p_5\}$ through the assigned roles, and is assigned to m_2. According to the team hierarchy, he/she is also a member of m_1. Because team roles and tasks cannot be inherited through team hierarchy, team task permission p_4 of m_1 cannot be merged to the team task permissions of m_2. So the permission p_4 cannot be available to the m_2, and this user can get permissions $\{p_3\}$ in m_1 and permissions $\{p_5\}$ in m_2. The reason that the team roles and tasks cannot be inherited is that each team already defines a small and stable RBAC system through its team roles and tasks. If team roles and tasks can be inherited through team hierarchies, then the stable RBAC system established by the team will be broken. For example, if team roles and tasks can be inherited, then the user in above example will obtain the permission p_4 in m_2, but neither m_1 nor m_2 expects any user has this permission.

This TT-RBAC model recognizes both general and limited team hierarchies. General team hierarchies support the concept of *multiple inheritances*, which provides the ability to inherit user team membership from two or more team sources. Limited team hierarchies impose restrictions resulting in a simpler tree structure, i.e. a team may have one or more immediate descendants, but is restricted to a single immediate ascendant. Examples of possible team

hierarchical structures are shown in Figure 3.4. Note that team is inherited bottom-up.

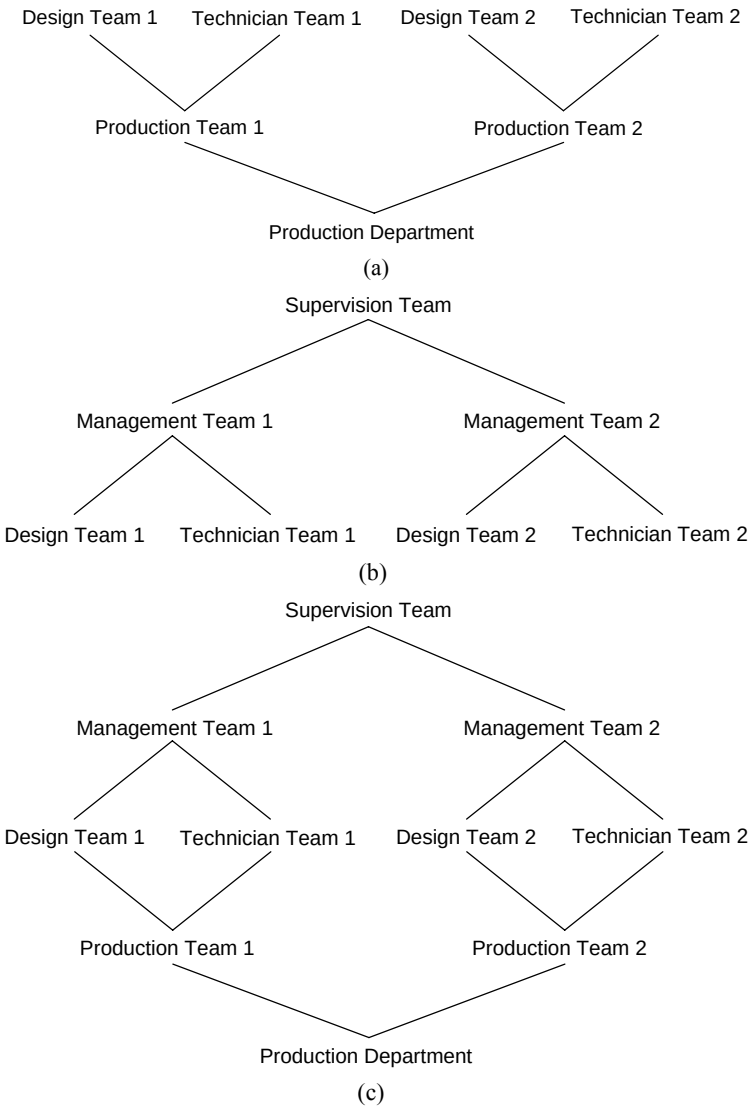


Figure 3.4: Example team hierarchies: (a) inverted tree; (b) tree; (c) lattice.

The general team hierarchies are formally defined as follows.

Definition 2a. General Team Hierarchies.

- $MH \subseteq TEAMS \times TEAMS$ is a partial order on *TEAMS* called team inheritance relation, written as $\geq$, where $m_2 \geq m_1$ only if all members of m_2 are also members of m_1. Formally:
  $m_2 \geq m_1 \Rightarrow authorized\_team\_users(m_2) \subseteq authorized\_team\_users(m_1)$.
- $authorized\_team\_users(m : TEAMS) \rightarrow 2^{USERS}$, the mapping of team *m* onto a set of users in the presence of a team hierarchy. Formally:
  $authorized\_team\_users(m) = \{u \in USERS \mid m' \geq m, (u, m') \in UMA\}$.

Team hierarchies provide the ability to compose a team from multiple subordinate teams. The superior team membership can be managed in subordinate teams. It reflects the organization management style in real world.

General team hierarchies support the concept of multiple inheritances that provide important hierarchy properties. The first is that one subordinate team can take part in multiple superior teams. The second is that multiple inheritances provide uniform treatment of user/team assignment relations and team/team inheritance relations.

Although limited team hierarchies do not support multiple inheritances, they nonetheless provide clear administrative advantages over Core TT-RBAC. We represent m_1 as an immediate ascendant of m_2 by $m_2 >> m_1$, if $m_2 \geq m_1$, but no team in the team hierarchy lies between m_1 and m_2. That is, there exists no team m_3 in the team hierarchy such that $m_2 \geq m_3 \geq m_1$, where $m_2 \neq m_3$ and $m_1 \neq m_3$. We now define limited team hierarchies as a restriction on the immediate ascendants of the general team hierarchy.

Definition 2b. Limited Team Hierarchies.

- Definition 2a with the limitation:

$$\forall m, m_1, m_2 \in TEAMS, m >> m_1 \wedge m >> m_2 \Rightarrow m_1 = m_2$$

3.5.2 Task hierarchies

Task hierarchies define inheritance relations among tasks. Task inheritance is described in terms of permissions; that is, task t_2 inherits task t_1 if all permissions of t_1 are also permissions of t_2.

This TT-RBAC model recognizes both general and limited task hierarchies. General task hierarchies support the concept of multiple inheritances, which provides the ability to inherit permission from two or more task sources. Limited task hierarchies impose restrictions resulting in a simpler tree structure, i.e., a task may have one or more immediate descendants, but is restricted to a single immediate ascendant. Examples of possible task hierarchical structures are shown in Figure 3.5. Note that task is inherited bottom-up.

The general task hierarchies are formally defined as follows.

Definition 3a. General Task Hierarchies.

- $KH \subseteq TASKS \times TASKS$ is a partial order on *TASKS* called task inheritance relation, written as $\geq$, where $k_2 \geq k_1$ only if all permissions of k_1 are also permissions of k_2. Formally:

$$k_2 \geq k_1 \Rightarrow authorized\_task\_permissions(k_1) \subseteq authorized\_task\_permissions(k_2)$$

- $authorized\_task\_permissions(k : TASKS) \rightarrow 2^{PERMS}$, the mapping of team k onto a set of permissions in the presence of a task hierarchy. Formally:

$$authorized\_task\_permissions(k) = \{p \in PERMS \mid k \geq k', (p, k') \in PKA\}.$$

General task hierarchies support the concept of multiple inheritances that provide the ability to compose a task from multiple subordinate tasks (with fewer permissions) in defining tasks and relations that are characteristic of the organization business structures, which these tasks are intended to represent.

Tasks in a limited task hierarchy are restricted to a single immediate ascendant. Although limited task hierarchies do not support multiple inheritances, they nonetheless provide clear administrative advantages over Core TT-RBAC. We represent k_1 as an immediate ascendant of

k_2 by $k_2 >> k_1$, if $k_2 \geq k_1$, but no task in the task hierarchy lies between k_1 and k_2. That is, there exists no task k_3 in the task hierarchy such that $k_2 \geq k_3 \geq k_1$, where $k_2 \neq k_3$ and $k_1 \neq k_3$. We now define limited task hierarchies as a restriction on the immediate ascendants of the general task hierarchy.

Definition 3b. Limited Task Hierarchies.

- Definition 3a with the limitation:

$$\forall k, k_1, k_2 \in TASKS, k >> k_1 \wedge k >> k_2 \Rightarrow k_1 = k_2$$

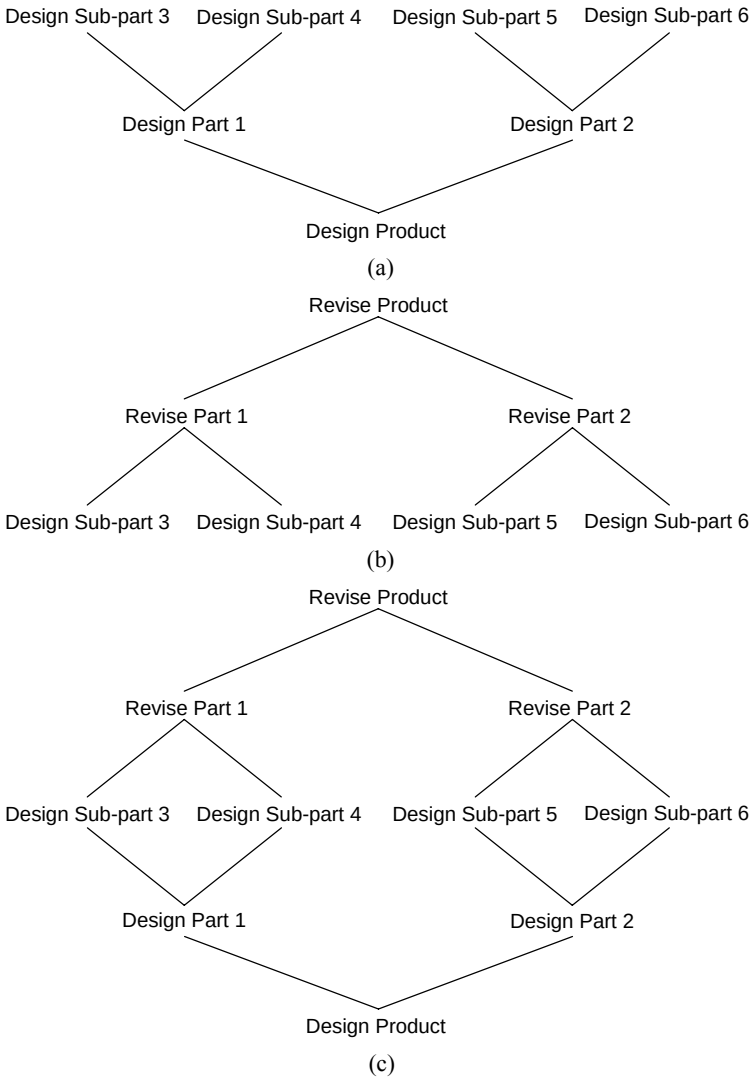


Figure 3.5: Example task hierarchies: (a) inverted tree; (b) tree; (c) lattice.

3.6 Constrained TT-RBAC

Constrained TT-RBAC adds separation of duty relations to the TT-RBAC model. The

constraints defined in this chapter are limited to those relations that place restrictions on sets of teams and in particular on their ability to form user-team, role-team and task-team assignment relations. This component allows both static and dynamic separation of duty as defined within the next two subsections.

3.6.1 Static separation of duty relations

Static constraints generally place restrictions on administrative operations that have the potential to undermine higher-level organizational SoD policies. In TT-RBAC, besides the conflict of interest in RBAC, conflict of interest may also arise as results of a user gaining authorization for permissions associated with conflicting teams and a team gaining authorization for permissions associated with conflicting roles or conflicting tasks. So in TT-RBAC, the Static Separation of Duty (SSD) relations, besides the user-role separation of duty relations defined in RBAC, are extended to include user-team, role-team and task-team separation of duty relations. The SSD constraints defined in the TT-RBAC are shown in Figure 3.6.

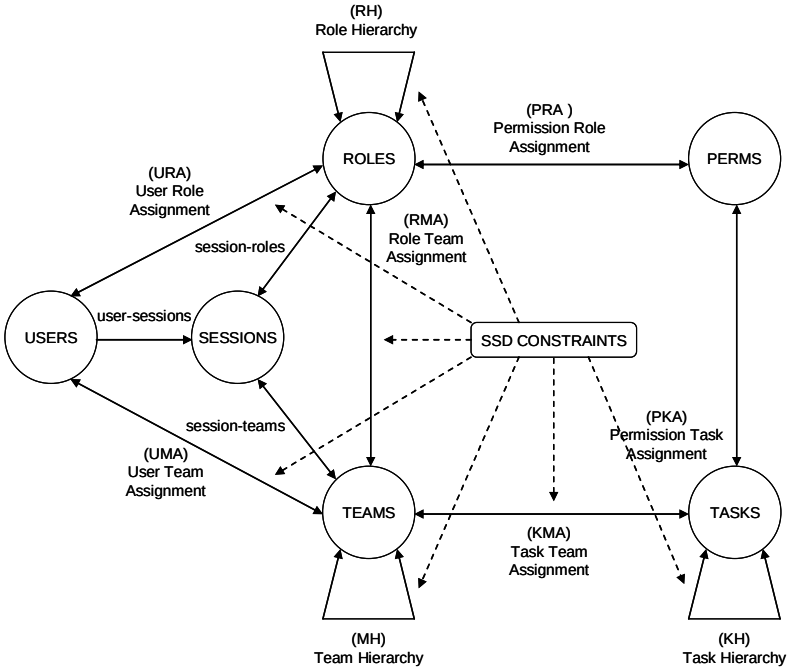


Figure 3.6: SSD within Hierarchical TT-RBAC.

The user-team static separation of duty enforces constraints on the assignment of users to teams. The user-team SSD constraints defined in TT-RBAC model are limited to those relations that place restrictions on sets of teams and in particular on their ability to form UMA relations. This means that if a user is assigned to one team, the user is prohibited from being a member of a second team. For example, if the team *Account* and team *Audit* are mutually exclusive teams, then a user who is assigned to the team *Account* cannot be assigned to the team *Audit*.

In TT-RBAC model we define user-team SSD with two arguments: a team set that includes two or more teams, and cardinality greater than one indicating a combination of teams that would constitute a violation of the user-team SSD policy. For example, an organization may require that no user may be assigned to more than two of the teams. When applying user-team

SSD relations in the presence of a team hierarchy, special care must be applied to ensure that user inheritance does not undermine user-team SSD policies. To address this potential inconsistency we define user-team SSD relations as a constraint on the authorized users of the teams that have an SSD relation. The formal definition of user-team SSD is given below.

Definition 4a. User-Team Static Separation of Duty.

- $UMSSD \subseteq \left(2^{TEAMS} \times N\right)$ is a collection of pairs (*ms*, *n*) in *User-Team Static Separation of Duty* (*UMSSD*), where each *ms* is a team set, *t* is a subset of teams in *ms*, and *n* is a natural number ≥ 2, with the property that no user is assigned to *n* or more teams from the set *ms* in each $(ms,n) \in UMSSD$. Formally:

$$\forall (ms,n) \in UMSSD, \forall t \subseteq ms : |t| \geq n \Rightarrow \bigcap_{m \in t} assigned\_team\_users(m) = \varnothing .$$

Definition 4b. User-Team Static Separation of Duty in the Presence of a Team Hierarchy.

- In the presence of a team hierarchy user-team static separation of duty is redefined based on authorized team users rather than assigned team users as follows:

$$\forall (ms,n) \in UMSSD, \forall t \subseteq ms : |t| \geq n \Rightarrow \bigcap_{m \in t} authorized\_team\_users(m) = \varnothing .$$

The role-team static separation of duty and task-team static separation of duty enforce constraints on the assignments of roles to teams and the assignments of tasks to teams, respectively. Similar to the definition of user-team SSD constraints, we can define the role-team SSD constraints and task-team SSD constraints. They are defined as follows.

Definition 5a. Role-Team Static Separation of Duty.

- $RMSSD \subseteq \left(2^{ROLES} \times N\right)$ is a collection of pairs (*rs*, *n*) in *Role-Team Static Separation of Duty* (*RMSSD*), where each *rs*, is a role set, *t* is a subset of roles in *rs*, and *n* is a natural number ≥ 2, with the property that no team is assigned to *n* or more roles from the set *rs* in each $(rs,n) \in RMSSD$. Formally:

$$\forall (rs,n) \in RMSSD, \forall t \subseteq rs : |t| \geq n \Rightarrow \bigcap_{r \in t} assigned\_role\_teams(r) = \varnothing .$$

Definition 5b. Role-Team Static Separation of Duty in the Presence of a Role Hierarchy.

- In the presence of a team hierarchy the constraint is redefined based on authorized role teams rather than assigned role teams as follows:

$$\forall (rs,n) \in RMSSD, \forall t \subseteq rs : |t| \geq n \Rightarrow \bigcap_{r \in t} authorized\_role\_teams(r) = \varnothing .$$

Definition 6a. Task-Team Static Separation of Duty.

- $KMSSD \subseteq \left(2^{TASKS} \times N\right)$ is a collection of pairs (*ks*, *n*) in *Task-Team Static Separation of Duty* (*KMSSD*), where each *ks* is a task set, *t* is a subset of tasks in *ks*, and *n* is a natural number ≥ 2, with the property that no team is assigned to *n* or more tasks from the set *ks* in each $(ks,n) \in KMSSD$. Formally:

$$\forall (ks,n) \in KMSSD, \forall t \subseteq ks : |t| \geq n \Rightarrow \bigcap_{k \in t} assigned\_task\_teams(k) = \varnothing .$$

Definition 6b. Task-Team Static Separation of Duty in the Presence of a Task Hierarchy.

- In the presence of a task hierarchy the constraint is redefined based on authorized task teams rather than assigned task teams as follows:

$$\forall(tks, n) \in KMSSD, \forall t \subseteq ks :| t |\geq n \Rightarrow \bigcap_{k \in t} authorized\_task\_teams(k) = \varnothing .$$

3.6.2 Dynamic separation of duty relations

Dynamic Separation of Duty (DSD) relations, like SSD relations, are intended to limit the permissions that are available to a user. However DSD relations differ from SSD relations by the context in which these limitations are imposed. SSD relations define and place constraints in the TT-RBAC element relations creation stage. DSD relations define and place constraints in the TT-RBAC elements execution stage. This TT-RBAC model component defines DSD properties that limit the availability of the permissions over a user's permissions space by placing constraints on the roles and teams that can be activated within or across a user's sessions. DSD properties provide extended support for the principle of least privilege in that each user has different levels of permission at different times, depending on the role or team being performed. These properties ensure that permissions do not persist beyond the time that they are required for performance of duty. As shown in Figure 3.7 the TT-RBAC DSD model extends the RBAC DSD through adding team activation constraints within or across users' sessions. For example a user cannot simultaneously activate two exclusive teams within a session.

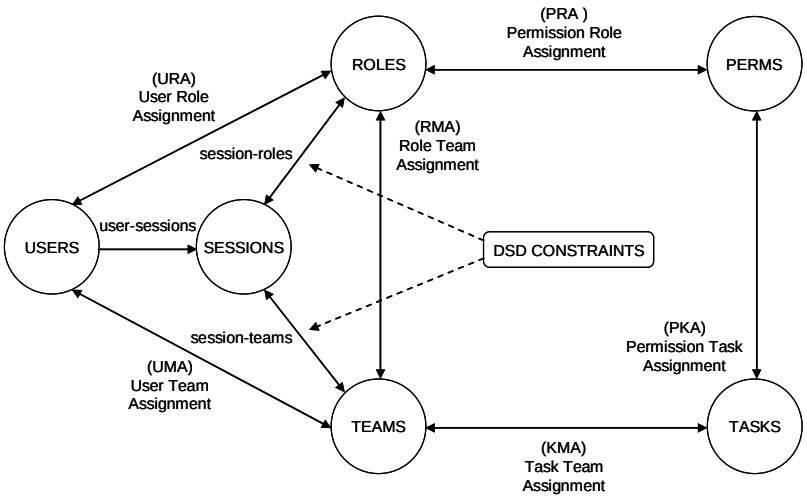


Figure 3.7: Dynamic separation of duty relations.

The user-team dynamic separation of duty enforces constraints on the activation of users to teams. The user-team DSD allows a user to be authorized for two or more teams that do not created a conflict of interest when acted on independently, but produce policy concerns when activated simultaneously. For example, a user may be authorized for both the teams of *Account* and *Audit*. If the individual acting some roles in team *Account* attempted to switch to the team *Audit*, DSD would require the user to inactive team *Account* firstly. As long as the same user is not allowed to active both of these teams at the same time, a conflict of interest situation will not arise. We define team DSD relations as a constraint on the teams that are activated in a

user's session. The formal definition of team DSD is given below.

Definition 7. User-Team Dynamic Separation of Duty.

- $UMDSD \subseteq \left(2^{TEAMS} \times N\right)$ is a collection of pairs (*ms*, *n*) in *User-Team Dynamic Separation of Duty* (*UMDSD*), where each *ms* is a team set and *n* is a natural number ≥ 2, with the property that no user may activate *n* or more teams from the set *ms* in each $(ms,n) \in UMDSD$ in a session. Formally:

$$\forall ms \in 2^{TEAMS}, n \in N, (ms,n) \in UMDSD \Rightarrow n \geq 2 \wedge |\, ms \,| \geq n, and$$
$$\forall s \in SESSIONS, \forall ms \in 2^{TEAMS}, \forall team\_subset \in 2^{TEAMS}, \forall n \in N,$$
$$(ms,n) \in UMDSD,$$
$$team\_subset \subseteq ms, team\_subset \subseteq session\_teams(s) \Rightarrow |\, team\_subset \,| < n$$

3.7 Functional specification overview

In this section, we provide an overview of the functionality involved in meeting the requirements of administrative operations, session management, and administrative review for the components Core TT-RBAC and Hierarchical TT-RBAC. The topic of TT-RBAC constraints will be further investigated in Chapter 5. There are three categories of functions in the TT-RBAC functional specification. Their purposes are:

- *Administrative functions*: creation and maintenance of element sets and relations for building the various TT-RBAC model components;
- *Supporting system functions*: functions that are required by the TT-RBAC implementation to support the TT-RBAC model constructs (e.g., TT-RBAC session attributes and access decision logic) during user interaction with an IT system; and
- *Review functions*: review the results of the actions created by administrative functions.

The complete administrative functions, supporting system functions and review functions required for Core TT-RBAC and Hierarchical TT-RBAC are given in the Appendix A

3.7.1 Functional specification for Core TT-RBAC

- *Administrative functions*. There are two kinds of administrative functions. One is used to create and maintain *element sets*. The other is used to create and maintain *element relations*. The basic element sets in Core TT-RBAC are USERS, ROLES, TEAMS, TASKS, OPS and OBS. Administrators create and delete these entities, and establish relationships among them. For example, the required administrative functions for USERS are *AddUser* and *DeleteUser*, for user-to-role assignment relations are *AssignUser* and *DeassigneUser*, and for user-to-team assignment relations are *AssignTeamUser* and *DeassignTeamUser*.
- *Supporting system functions*. Supporting system functions are required for session management and making access control decisions. The function *CreateSession* creates a session establishes a default set of activate roles and a default set of activate teams in which there may activate a default set of team-roles and a default set of team-tasks for the user at the start of the session. These default sets can then be altered by the user during the session

- through some functions, such as *AddActiveRole* and *DropActiveRole* that are defined for session-roles, and *AddActiveTeamRole* and *DropActiveTeamRole* that are defined for session-team-roles. The function *CheckAccess* determines if the session subject has permission to perform the requested operation on an object according to the permissions gotten through the session-roles and session-teams.
- *Review functions*. When entity relation instances have been created, it should be possible to view the contents of those relations from different perspectives. For example, the function *AssignedTeamUsers* is used to get all the users who are assigned to a given team. In addition, it should be possible to view the results of the supporting system functions to determine some session attributes such as the active roles and teams in a given session or the total permissions of a given session. For example, the function *SessionTeams* is used to get all the active teams associated with a session.

3.7.2 Functional specification for Hierarchical TT-RBAC

- *Administrative functions*. The administrative functions required for Hierarchical TT-RBAC include all the administrative functions that were required for Core TT-RBAC. The additional administrative functions required for the Hierarchical TT-RBAC model pertain to creation and maintenance of the partial order relation (RH) among roles, the partial order relation (MH) among teams and the partial order relation (KH) among tasks. For example, the function *AddTeamInheritance* establishes a new immediate inheritance relationship between two existing teams. A general hierarchy allows multiple inheritances, while a limited hierarchy is essentially a tree structure. For a limited team hierarchy, the *AddTeamInheritance* function is constrained to a single ascendant team.

  The outcome of the *DeleteInheritance* function may result in multiple scenarios. When *DeleteInheritance* is invoked with two given teams, say Team *A* and Team *B*, the implementation system is required to do one of the following. The system may preserve the implicit inheritance relationships that team *A* and team *B* have with other teams in the hierarchy. That is, if team *A* inherits other teams, say *C* and *D*, through team *B*, team *A* will maintain permissions for *C* and *D* after the relationship with team *B* is deleted. Another option is to break those relationships because an inheritance relationship no longer exists between them.
- *Supporting system functions*. The supporting system functions for hierarchical TT-RBAC are the same as for Core TT-RBAC and provide the same functionality. However because of the presence of a team and task hierarchy, the functions *AddActiveTeam* and *AddActiveTask* have to be redefined. In a team hierarchy, a given team may belong to one or more of the teams obtained through team hierarchies. When that given team is activated by a user, the question of whether the other teams obtained through team hierarchies are automatically activated or must be explicitly activated is left as an implementation issue and no one course of actions prescribed as part of this specification. However, when the latter scenario is implemented (i.e. explicit activation) the corresponding supporting functionality shall be provided in the supporting system functions.
- *Review functions*. All the review functions specified for Core TT-RBAC remain valid for Hierarchical TT-RBAC. In addition, the entities' membership should be considered in the presence of role, team and task hierarchies to capture these expanded memberships for different TT-RBAC entities. Because of the presence of partial order among the roles, teams and tasks, the functions relating to permission set should be redefined.

3.8 Evaluation of TT-RBAC

In this section, we evaluate the TT-RBAC through comparing different access control models in collaborative systems. We first describe the assessment criteria for access control in collaborative systems, and then provide a comparison result in a table.

3.8.1 Assessment criteria introduction

Tolone et al. [TAP05] systematically examines existing access control models as applied to collaboration, highlighting not only the benefits, but also the weaknesses of these models. Here we adopt the assessment criteria used in their work. The criteria used to characterize the access control models are as follows.

- *Complexity* defines the nature of the access control model. It is considered an important aspect of consideration because an overly complex model can lead to unforeseen problems and implementation can become difficult. There is a trade off between functionality and complexity.
- *Understandability* defines the transparency of the model and its underlying principles. The consequences of manipulation and changes of access rights should be obvious for the proper use of the system.
- *Ease of Use* indicates how simple the system is from the end user's point of view in terms of its usage in a collaborative environment. If it is very cumbersome to use, then there is a chance that users will not favor it. Security systems always bring a degree of complexity into the system, and users need to be reassured of the ease of use of any system. The simpler the model is, the more popular it will be.
- *Applicability* of an access control model is an indication of its practicality. A good, but solely theoretical, model may provide few benefits. An infrastructure should exist where the model can be deployed.
- *Support for Collaboration* is the most important aspect of consideration for access control models devised for collaborative environments. There are several aspects of a collaborative environment that determine the ultimate usability of any access control model. These factors are discussed individually.
    - *Groups of Users*. A collaborative environment in its most basic form implies a common task undertaken by a group of people. The access control model should represent support for changes, manipulation, and specifications made for groups of users in addition to individual users.
    - *Policy Specifications*. Access control models are based on the specification and representation of policies that govern a collaborative environment. The model should support ways of specifying policies and an appropriate syntax, pattern, or language that allows extensions or modifications in a simple and transparent manner. This helps to ensure the scalability of the system.
    - *Policy Enforcement*. It is important for the access control model to provide means to ensure that the policies or constraints specified are enforced correctly.
    - *Fine-Grained Control*. Collaborative environments are characterized by situations where it is not sufficient to have access rules only for groups of users on clusters of

- objects. Often, a user in an instance of a role might need a specific permission on an instance of an object at a particular point in the collaboration instance. A level of fine-grained control is required for such situations, without introducing compromises or complexities into the system.
- *Active/Passive*. It is desirable for the access control model to be active so as to handle the dynamism of a collaborative system.
- *Contextual Information*. Context is one of the most important characteristics of any collaboration, and it is important to know the degree to which contextual information is utilized by the access control model in order to secure the system.

3.8.2 Comparison of access control models

Our comparison of access control models is based on the work done by Tolone et al. [TAP05]. We use the same assessment criteria to analysis the TT-RBAC model. We also analyze the T-RBAC [OP2003] model that is omitted by [TAP05] against the criteria. These access control models have been introduced in the Section 2.2. The assessment result is shown in Table 3.1.

Criteria	Matrix	RBAC	TBAC	TMAC	C-TMAC	SAC	Context-A	T-RBAC	TT-RBAC
Complexity	Low	Medium	Medium	Medium	Medium	Low	High	High	Medium
Understandability	Simple	Simple	Simple	Simple	Simple	Simple	Simple	Simple	Simple
Ease of Use	Medium	High	Medium	High	High	Low	High	Medium	High
Applicability	Medium	High	Medium	Medium	High	Low	High	Medium	High
Collaboration Support:									
Groups of Users	Low	Y	Y	Y	Y	Y	Y	Y	Y
Policy Specification	Low	Y	Low	Y	Y	Y	Y	Y	Y
Policy Enforcement	Low	Y	Low	Y	Y	Low	Y	Y	Y
Fine Grained Control	N	Low	Low	Y	Y	N	Y	Low	Y
Active/Passive	Passive	Passive	Active	Active	Active	Active	Active	Active	Active
Contextual Information	N	Low	Medium	Medium	Medium*	Medium	Medium*	Medium	High

Table 3.1: Characterization of access control models for collaborative systems.

The table makes use of comparative terminology such as Low, Medium, and High, descriptive terminology such as Simple, Active, and Passive, and the standard Yes (Y) and No (N) terminology for characterization against the criteria.

For the contextual information criteria, Medium* is used to identify those models that appear to support the strongest notion of context among those in the Medium category. Use of Low, Medium, and High for criteria such as Complexity describes the degree. Low Complexity indicates that the model is fairly simple in nature.

Low has also been used to describe criteria such as groups of users when it is not convenient to use a simple Yes or No means of description. For example, ACLs provide ways of specifying access rights for groups of users, but it is very primitive and cumbersome from the point of view of supporting groups in a collaborative environment.

Yes and No have been used whenever it is possible to indicate the facilitation or lack of facilitation of the concerned criteria by the access control model. Wherever it is insufficient to simply indicate the presence of support for a feature, and it is also important to indicate the degree to which a feature is supported, Low, High and Medium have been used. For example,

ACLs and traditional RBAC do not support consideration of contextual information in decision-making, whereas other models support varying degrees of contextual information consideration.

In TT-RBAC, each team defines a small RBAC application zone, in which it strictly conforms to the rules of RBAC. So basically, the *complexity*, *understandability*, *ease of use* and *applicability* are the same as the RBAC. *Group of users* is the nature of TT-RBAC. *Policy specification* and *policy enforcement* are the same as the RBAC. Because the tasks assigned to a team can be in any granularity, it is possible to assign a specific object to a given team. On the other hand, one team can contain only one team member, or the team members have different roles, so it is possible for TT-RBAC to assign the permissions based on individual users and objects. Thus, *Fine-grained control* is well supported by TT-RBAC. *Context information* is the key factor that decides if a security system is a passive security system or an active security system. To our understanding an access control model supporting context information or not strongly depends on its enforcing mechanism. We have developed a TT-RBAC enforcing mechanism that permits any TT-RBAC entity can be associated with context information. This characteristic makes TT-RBAC be an *active* security model. The TT-RBAC enforcing mechanism is described in Chapter 4.

3.9 Summary

We have presented an access control model that integrates the teams and tasks with RBAC. In TT-RBAC a team encapsulates a collection of users in specific roles and a set of roles with the objective of accomplishing specific tasks. The team tasks decide the maximum permissions assigned to a team, the team roles decide the maximum permissions that a team can perform, and the team members decide who can active his/her roles and perform the team tasks in a team. A team member can perform what tasks is decided by what roles he/she can activate in the team. Thus, a team defines a small and specific RBAC application zone, through which we can preserve the advantages of scalable security administration that RBAC-style models offer and yet offers the flexibility to activate permissions for individual users and on specific object instances. The TT-RBAC model overcomes the shortcomings of traditional RBAC that lacks the ability to specify a fine-grained control on individual users in certain roles and on individual object instances. The integrated context constraint component makes the TT-RBAC be an active security model.

We have used the TT-RBAC model to analyze a real hospital authorization system. Details bout this case study can be found in Section 8.1.

Chapter 4

TT-RBAC Enforcement

4.1 Introduction

In Chapter 3 we systemically introduce the TT-RBAC reference model that defines a collection of model components and the functionalities involved in meeting the requirements for each of these components. As adding two new data elements teams and tasks, the entity relations in TT-RBAC are more complicated then in RBAC. This makes implementing a TT-RBAC system be a challenge task. In this chapter we introduce a software structure for TT-RBAC implementation.

The major contribution of this work is using object-oriented technology to implement fine-grained context-aware TT-RBAC systems. We will introduce what objects should be defined in a TT-RBAC system, how the functionalities defined in TT-RBAC are arranged into these objects, and how these objects work together to make access control decisions. We also introduce a mechanism for adding context constraints to any TT-RBAC entities.

The rest of this chapter is organized as follows. Section 4.2 defines the core classes and their relations in our TT-RBAC implementation. Section 4.3 introduces how the context constraints are added to TT-RBAC entities. Section 4.4 investigates the TT-RBAC authorization evaluation process.

4.2 TT-RBAC core classes

Our TT-RBAC system is developed with object-oriented technology. We define eight classes that represent *user*, *role*, *permission*, *object*, *operation*, *team*, *task* and *session*, respectively. All of them inherit from a superclass named *TTRObject* that defines the common variables and methods needed by its subclasses. These classes are called TT-RBAC core classes. The major variables and methods defined for these classes and their inheritance relations are shown in Figure 4.1.

- *TTRObject* is the superclass of all other TT-RBAC core classes. In *TTRObject* there define three variables; they are *id*, *type* and *state* that are used to hold an entity's identifier, type and state, respectively. Entity type could be USER, ROLE and so on. An entity could be in *active* state or *inactive* state. Entity state decides if an entity can be used for access control. There are various factors, such as context, that can affect entity state.

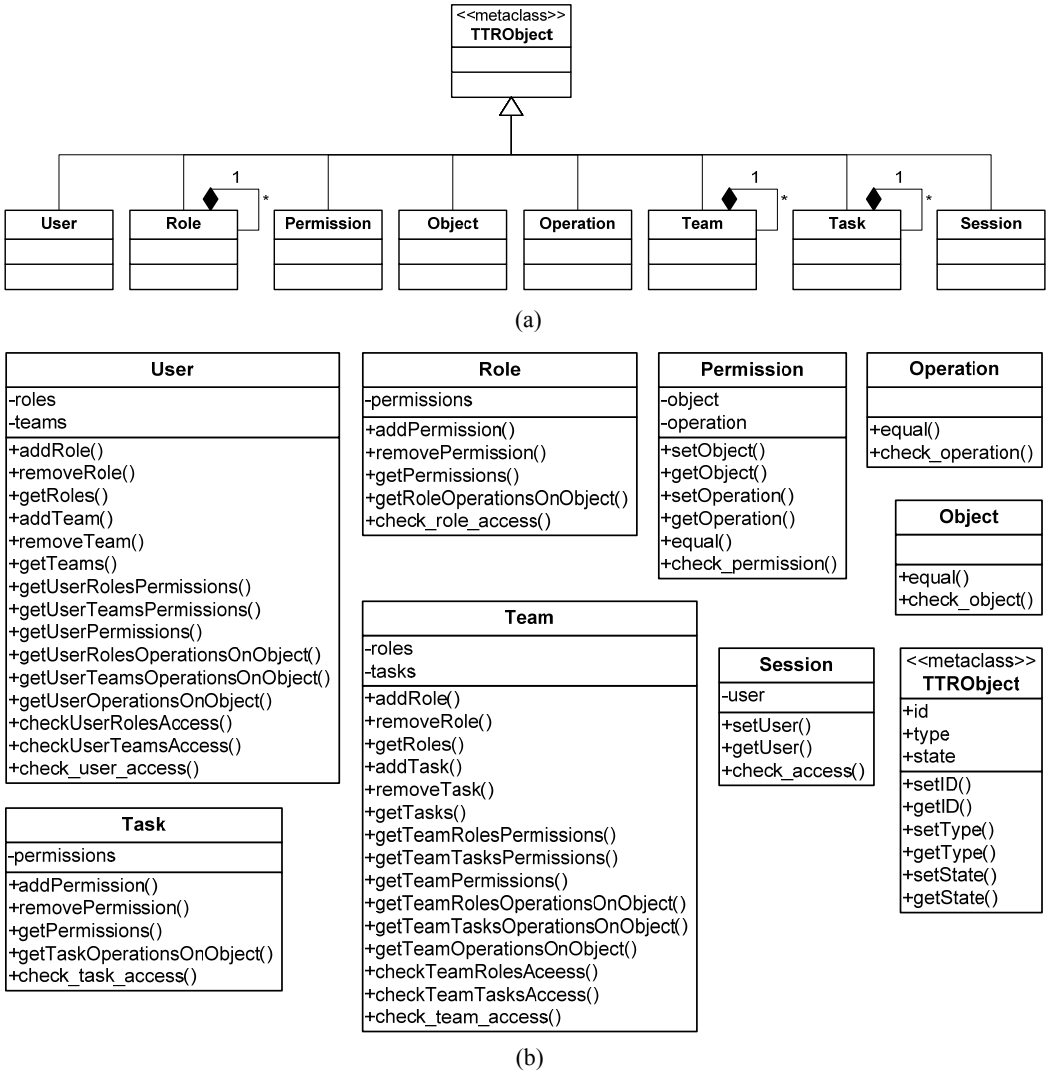


Figure 4.1: TT-RBAC core classes: (a) inheritances; (b) definitions.

- *User* is the class used to describe users. The variables *roles* and *teams* are used to hold *Role* objects and *Team* objects, respectively. The methods *checkUserRolesAccess*, *checkUserTeamsAccess* and *check_user_access* are used for checking user access.
- *Role* is the class used to describe roles. The variable *permissions* is used to hold *Permission* objects. The method *check_role_access* is used for checking role access.
- *Permission* is the class used to describe permissions. The variables *object* and *operation* are used to hold *Object* and *Operation* objects, respectively. The method *equal* is used for comparing with an input *Permission* object. The method *check_permission* is used for checking permission access.
- *Object* is the class used to describe resources. The method *equal* is used for comparing with an input object name. The method *check_object* does the similar function to *equal*, except it also considers object's state.
- *Operation* is the class used to describe operations. The method *equal* is used for comparing

with an input operation name. The method *check_operation* does the similar function to *equal*, except it also considers object's state.

- *Team* is the class used to describe teams. The variables *roles* and *tasks* are used to hold the *Role* objects and *Task* objects, respectively. The methods *checkTeamRolesAccess*, *checkTeamTasksAccess* and *check_team_access* are used for checking team access.
- *Task* is the class used to describe tasks. The variable *permissions* is used to hold the *Permission* objects. The method *check_task_access* is used for checking task access.
- *Session* is the class used to describe sessions at runtime. The variable *user* is used to hold a *User* object. The method *check_access* is used for checking session access.

From the above introduction, we know these classes are not independent. Their relations are shown in Figure 4.2.

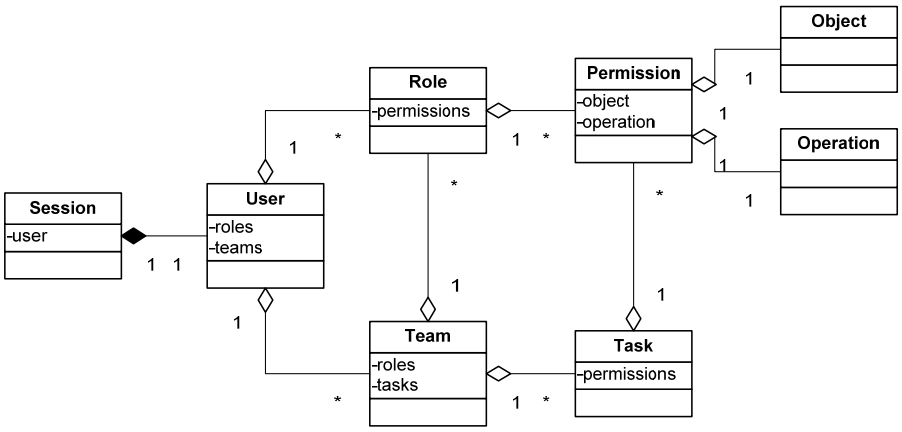


Figure 4.2: TT-RBAC core class relations.

4.3 Context constraints

In TT-RBAC the entity state may be affected by the external facts that normally called *context* information in literature. Context is an elusive concept, which has many different meanings to different people and communities. In the area of ubiquitous and pervasive computing context can be defined as: "any information that can be used to characterize the situations of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves" [Dey2001]. Context may consist of almost all information describing a specific situation. The context information may be static like a person's nationality or dynamic like time.

Our context constraint mechanism is closely related to the work proposed by Strembeck and Neumann [SN2004], where a context constraint is defined as a dynamic constraint that checks the actual values of one or more contextual attributes for predefined conditions. If these conditions are satisfied, the corresponding access request can be permitted. Accordingly, a conditional permission is constrained by one or more context constraints. Their work aims to preserve the advantages of RBAC and offer an additional means for the definition and enforcement of context-dependent access control policies.

But their work also has some obvious weaknesses. Firstly, their context constraints are only applied to permissions. Secondly, their context constraint check is tightly connected to their

RBAC system and only performed at stage of permission check. Lastly, their implementation is not based on object-oriented technology and difficult to be understood. We have developed our own context constraint mechanism that overcomes some weaknesses of [SN2004]. For example, our context constraints can be assigned to any TT-RBAC entities and the implementation is easy to be understood and used. Our context constraint mechanism is described in the following subsections.

4.3.1 Context constraint definition

Context constraint is an abstract concept on the modeling level. The context constraint specifies that certain context attributes must meet certain conditions to permit a specific operation. For example, a context role can be activated only when all its associated context constraints evaluate to true. Figure 4.3 shows that RBAC roles are associated with context constraints. A *context constraint* is defined through the terms *context attribute*, *context function*, and *context condition*. They are described as follows.

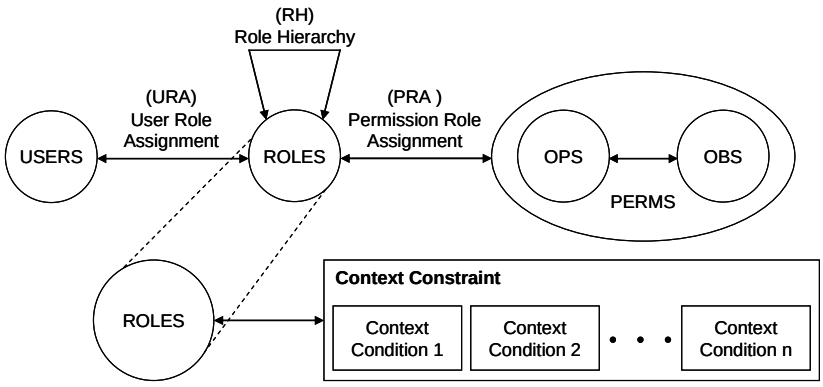


Figure 4.3: RBAC roles with context constraint.

- *Context attribute* represents a certain property of the context whose actual value may change dynamically (e.g. time, date or session-data) or which varies for different instances of the same abstract entity (e.g. location, birthday, or nationality). Thus, context attributes are a means to make context information explicit.
- *Context function* is a mechanism to obtain the current value of a specific context attribute. For example, the function *getDate*() returns the current date. One or more context functions are encapsulated into a *context observer* that corresponds to a library or a package in the implementation. For example, the functions *getDate*(), *getTime*() and *getIP*() may be organized into *LocalHostObserver*.
- *Context condition* is a predicate that consists of an operator and two or more operands. At least one operand represents a certain context attribute, while the other operands may be either context attributes or constant values. Context attributes are gotten by using corresponding context functions. The operator is either a prefix operator that accepts two or more input parameters or a binary infix operator that compares two values.
- *Context constraint* is a clause containing one or more context conditions. It is satisfied if and only if all its context conditions are satisfied.

  Context constraints are used to define conditional entities such as conditional roles. A

conditional entity is associated with one or more context constraints and grants for making access control decisions if and only if each corresponding context constraint evaluates to true. The relation between context constraints and entities is a many-to-many relation. Thereby, a number of entities can be associated with the same context constraint if necessary. Similarly, one entity can be associated with many context constraints.

Now we give a context constraint example about working time. Here we assume that roles *R*1 and *R*2 can be activated between 09:00 and 17:00, and role *R*2 can only be activated from Monday to Friday. The definitions of context attributes, conditions, constraints and constraint assignment are shown in Figure 4.4.

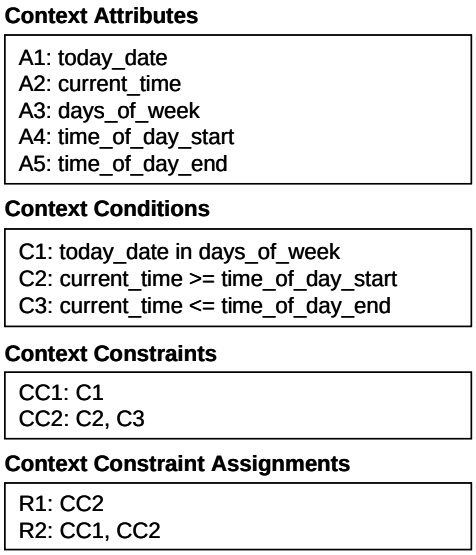


Figure 4.4: Example of context constraints and their assignments.

In this example we define five context attributes, three context conditions and two context constraints which are assigned to roles *R*1 and *R*2. At runtime, the role *R*1 can be activated between 09:00 and 17:00 everyday, and the *R*2 can only be activated between 09:00 and 17:00 from Monday to Friday.

4.3.2 Context constraint component

In TT-RBAC implementation the context constraints are handled by the *Context Constraint Component* that is integrated into the class *TTRObject*. Thus, any class that inherits from this class will automatically obtain the functionality of context constraint. As shown in Figure 4.5, the context constraint component is formed by the classes *ContextConstraint*, *ContextCondition* and *ContextObserver*. In order to handle the associated the *ContextConstraint* objects, some new variables and methods have been added to *TTRObject*. The method *check_state* invokes all the *ContextConstraint* instances to get their evaluation results. It returns the value true only when all the context constraints evaluate to true. Details about the classes that compose the context constraint component are described as follows.

- *ContextConstraint* is the class used to unite multiple *ContextCondition* objects. The method *check_context_constraint* invokes all the *ContextCondition* objects to get their evaluation results. Only when all the context conditions evaluate to true, it returns the value true.

- *ContextCondition* is the class used to hold and evaluate context conditions. A series of class variables are defined to specify a context condition. *dataType* holds the data type of operands. *operator* specifies how to compare the operands. *leftObserver*, *leftFunction* and *leftParameter* specify that the left operand is gotten through which context function in which context observer with what parameter. *rightObserver*, *rightFunction* and *rightParameter* specify that the right operand is gotten through which context function in which context observer with what parameter. These operands are gotten through the method *get_context_attribute* that is defined in the class *ContextObserver*. The method *check_context_condition* checks if the *predicate* defined by the given condition is satisfied.
- *ContextObserver* is an interface through which different context observers can be implemented separately and independently. Each context observer has the same interface so that the observer invoker can treat them in a unified way. The Java interface is defined as:

```
public interface ContextObserver {
    public String getContextAttribute(String function, String parameter);
}
```

In this Java interface there defines one method *getContextAttribute* that has two input arguments. One is *function* that indicates which function should be used in a context observer. Another is *parameter* that is used to provide extra information needed by the function. For example, the function *getPersonalNationality* needs to use *user identity* as its input parameter.

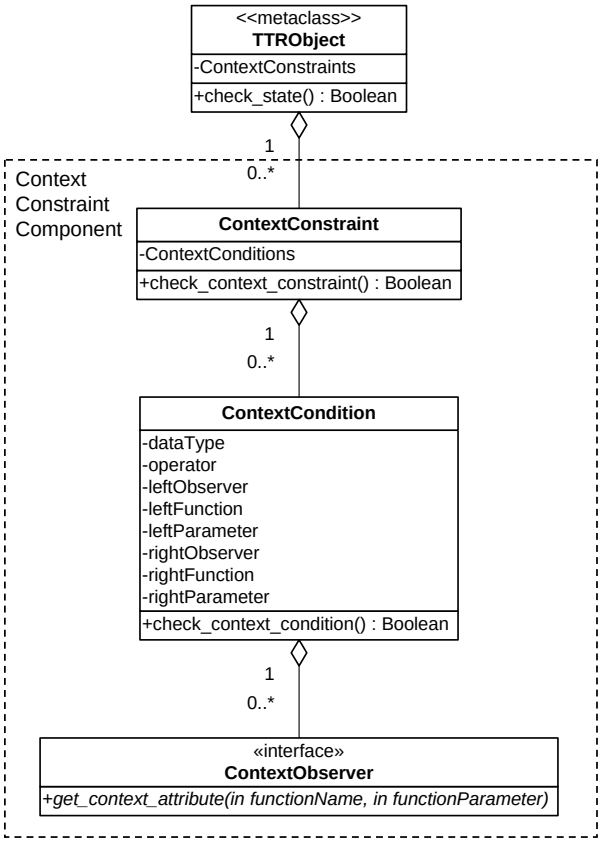


Figure 4.5: Context constraint component.

4.3.3 Context constraint assignments

Context constraints are mainly used to decide if the associated entities should be activated or deactivated at runtime. In this subsection we use an example shown in Figure 4.6 to investigate how the context constraints affect a RBAC system.

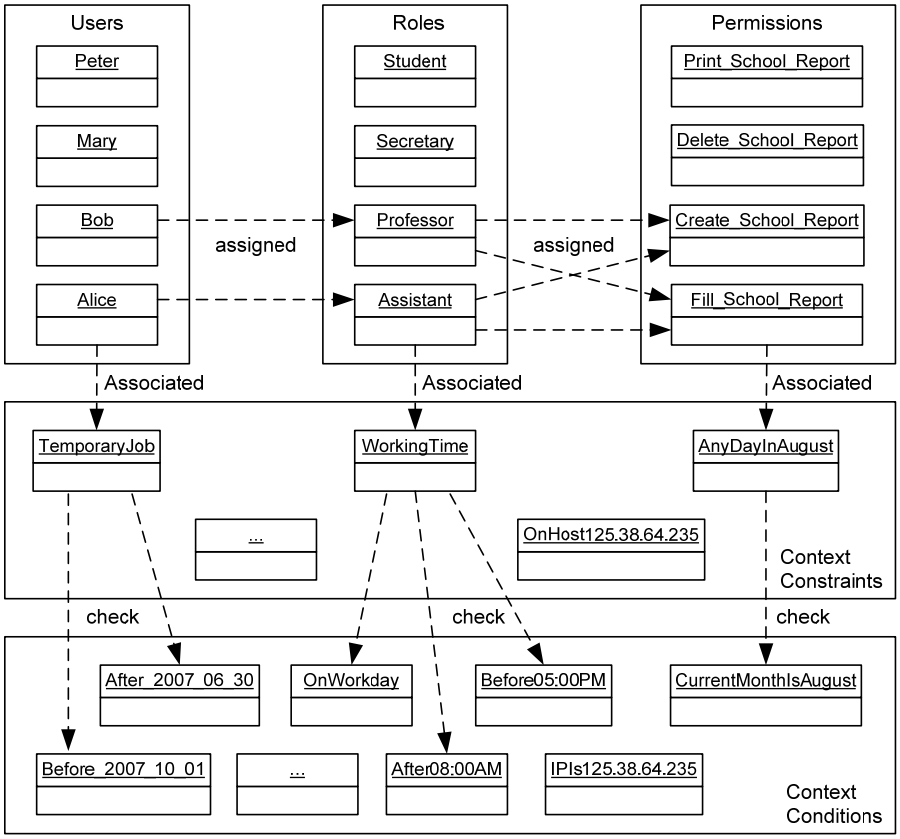


Figure 4.6: Example of RBAC with context constraints.

In this example, *Alice* is assigned to the role *Assistant* that has the permissions *Create_School_Report* and *Fill_School_Report*. The user *Alice* is associated with the context constraint *TemporaryJob* that specifies when she is a valid user. The role *Assistant* is associated with context constraint *WokingTime* that specifies this role can be activated between 08:00AM and 05:00PM. The permission *Fill_School_Report* is associated with context constraint *AnyDayInAugust* that specifies the school report can be filled in August. Finally, after considering the context constraints, the privileges that are available to *Alice* are: (1) she can use the computer system from 01/07/2007 to 30/09/2007; (2) she can create student school report at working time; (3) she can fill the student school report at working time in August.

Similarly, TT-RBAC entities, such as teams and tasks, can also be associated with context constraints. So our approach can give the security officer the maximum ability to define a fine-grained context-aware access control system.

4.3.4 Entity activity effective scope

In TT-RBAC, every entity's state can be *active* or *inactive*. One entity can be used for access

control only when it is in active state. For example, the permissions held by a role are available to a user only when this role is in active state. One entity's state change may affect the permissions that are available to other entities. For example, if an object is deactivated, then all the permissions that relate to this object are deactivated. This effect will be propagated among all the entities that relate to these permissions. One entity's state change may also affect other entities' state. For example, all the employees are assigned to role *Staff*, and the role *Staff* is the prerequisite role to all other roles. If the role *Staff* is deactivated in a session, then all the roles depending on it will also be deactivated. In TT-RBAC different kinds of entities have different effective scopes to other entities. The entity effective scopes are shown in Figure 4.7. The arrows indicate the entities' effective directions. Their meanings are described as follows.

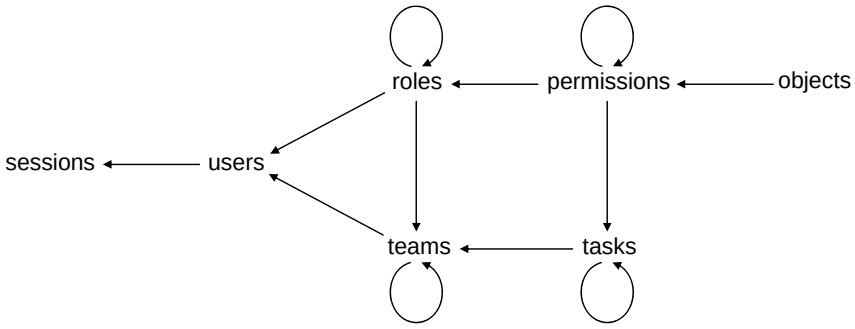


Figure 4.7: Effective scopes of TT-RBAC entities.

- If an object is deactivated, then all the permissions that relate to it will be deactivated.
- If a permission is deactivated, then all the roles and tasks that relate to it will be affected. This permission will be withdrawn from them. This permission's state change may also affect other permissions that depend on it.
- If a role is deactivated, then all the users and teams that relate to it will be affected. This role will be withdrawn from them. This role's state change may also affect other roles that depend on it.
- If a task is deactivated, then all the teams that relate to it will be affected. This task will be withdrawn from them. This task's state change may also affect other tasks that depend on it.
- If a team is deactivated, all the users who relate to it will be affected. This team will be withdrawn from them. This team's state change may also affect other teams that depend on it.
- If a user is deactivated, all the sessions that belong to him/her will be affected. These sessions will be terminated.

4.4 TT-RBAC evaluation process

A user obtains privileges through the active roles in sessions. A role can be activated as a session-role or a session-team-role. If a role is activated as a session-role, then the user will get all the permissions assigned to this role. If a role is activated as a session-team-role, then the permissions that this user can get are also decided by which tasks have been activated in the team. It means that the permissions of a session-team-role are filtered by the permissions of

session-team-tasks. In a session the permissions available to a user are the union of all permissions gotten through session-roles and session-teams. In this section we investigate the TT-RBAC evaluation process.

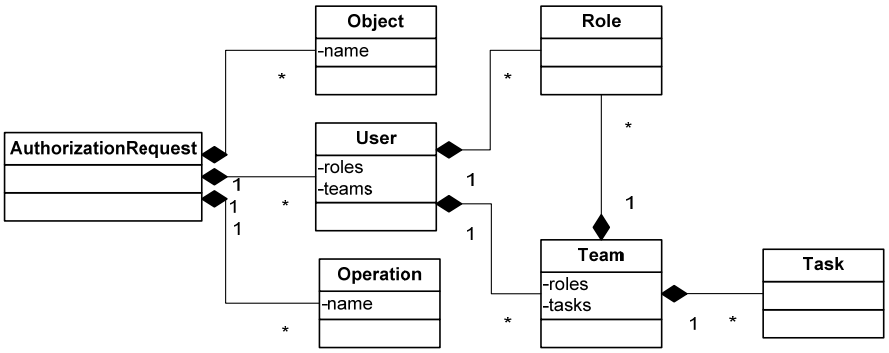


Figure 4.8: Structure of TT-RBAC authorization request.

4.4.1 Authorization request

In TT-RBAC systems authorization requests can be formatted as (user, object, operation). It states that the request *user* wants to perform the request *operation* on the request *object*. The request user information should include all the active session-roles and active session-teams in which also include the active session-team-roles and active session-team-tasks. The structure of TT-RBAC authorization request is shown in Figure 4.8. TT-RBAC authorization requests are evaluated by TT-RBAC decision makers.

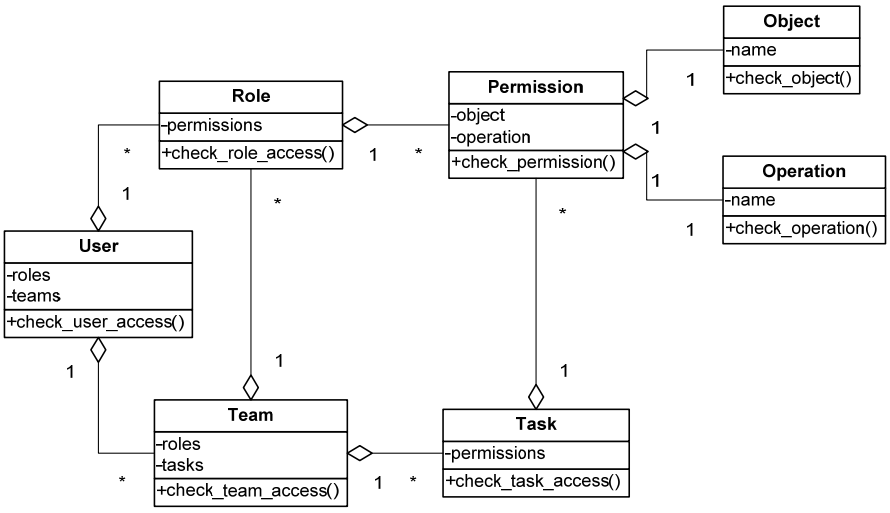


Figure 4.9. Structure of TT-RBAC decision maker.

4.4.2 Decision maker

In order to improve the performance of TT-RBAC evaluation, we developed a novel TT-RBAC evaluation mechanism, in which a decision maker is specially created for each authorization request. This decision maker is then used to evaluate the requested *permission* formed by the request *object* and request *operation*. In fact, this decision maker is a *user* object that is created

according to the request *user*. So the decision maker is also called *user decision maker*. The role, team, team-role and team-task objects contained in a request *user* are not ground. For example, the role objects in a request *user* do not contain permission objects. On the contrary, all the objects inside a decision maker are ground. For example, all the role objects in a decision maker contain all the required permission objects, and these permission objects are also ground. After all the entities relating to a decision maker are ground, this user object is ready for making access control decisions. The structure of TT-RBAC decision maker is shown in Figure 4.9.

In order to accelerate the speed of creating user decision makers, some or all the role, team, task, permission, object and operation objects can be initialized and saved in some object containers when a TT-RBAC evaluation system starts up.

4.4.3 Evaluation sequence

When an authorization request arrives, the evaluation system creates a special user decision maker according to the request user object, and then checks if this user can perform the required permission (ob, op) that is formed by the request object (ob) and request operation (op). The whole authorization check process is shown in figure 4.10 and described as follows.

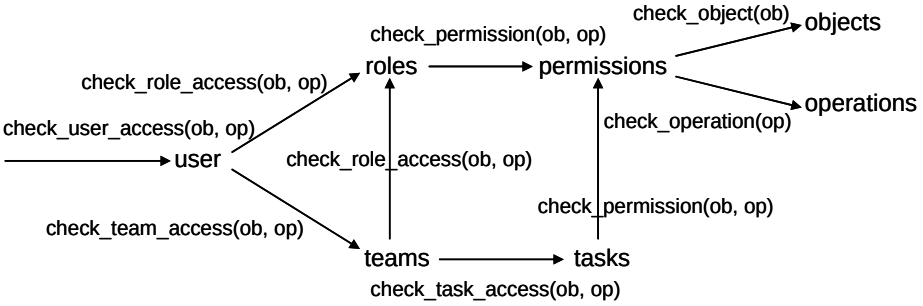


Figure 4.10: TT-RBAC authorization check process.

- *check_user_access* is a method defined in *User* class. It checks if a request is permitted by a user object (user decision maker). This method first checks the state of the user object. Only when it is in active state, the check process continues to do the role permission check and team permission check, otherwise the check process stops and returns the value “false”. The role permission check is through invoking the method *check_role_access* defined in role objects. The team permission check is through invoking the method *check_team_access* defined in team objects. If there is any role or team permits this permission, the check process stops and returns the value “true”.
- *check_role_access* is a method defined in *Role* class. It checks if a request is permitted by a role object. This method first checks the state of the role object. Only when it is in active state, the check process continues to do the role permission check, otherwise the check process stops and returns the value “false”. The role permission check is through invoking the method *check_permission* defined in permission objects. If the request is permitted by the role, it returns the value “true”.
- *check_team_access* is a method defined in *Team* class. It checks if a request is permitted by both team-roles and team-tasks. This method first checks the state of the team object. Only when it is in active state, the check process continues to do the team-role and team-task permission checks, otherwise the check process stops and returns the value “false”. The

team-task permission check is through invoking the method *check_task_access* defined in task objects. If the request is permitted by both the team-roles and team-tasks, it returns the value "true".

- *check_task_access* is a method defined in *Task* class. It checks if a request is permitted by a task. The check process is similar to *check_role_access*.
- *check_permission* is a method defined in *Permission* class. It checks if a request is permitted by a permission object. This method first checks the state of the permission object. Only when it is in active state, the check process continues to do the permission check, otherwise the check process stops and returns the value "false". The permission check is through invoking the method *check_object* defined in (permission) object object, and the method *check_operation* defined in operation object. Only when both of them return the value "true", this method returns the value "true", otherwise returns the value "false".
- *check_object* is a method defined in *Object* class. It checks if the input request object is equal to a (permission) *Object* object. This method first checks the state of the object. Only when it is in active state, the check process continues to do the object comparison, otherwise the check process stops and returns the value "false". The object comparison is through comparing two objects' names. If their names are equal, this method returns the value "true", otherwise returns the value "false".
- *check_operation* is a method defined in *Operation* class. It checks if the input request operation is equal to an *Operation* object. The check process is similar to the check process of *check_object*.

4.5 Implementation

We have developed a TT-RBAC access control system that consists of an access control engine and a GUI-based administration tool. The kernel of the access control engine is the runtime TT-RBAC module that is based on TT-RBAC enforcement mechanism introduced in this chapter. The system's major functionality is listed as follows.

- TT-RBAC access control engine. It consists of TT-RBAC runtime module, context constraints runtime module and authorization constraints runtime module. The TT-RBAC runtime module makes access control decisions. The other two modules are used to activate/deactivate TT-RBAC entities.
- Editors for users, roles, permissions, objects, operations, teams and tasks.
- Editors for permission-role, permission-task, user-role, user-team, role-team and task-team assignments.
- Editors for context constraints and their assignments.
- Editors for function-based authorization constraints and their assignments. (This work is introduced in Chapter 5.)
- TT-RBAC function test platform. Its kernel part is the TT-RBAC access control engine. This platform can be used to test TT-RBAC functions, context constraints and dynamic authorization constraints. It can simulate multiple users' sessions and make access control decisions according to the input requests. A user can open multiple sessions. Its screen snapshot is shown in Figure 4.11.

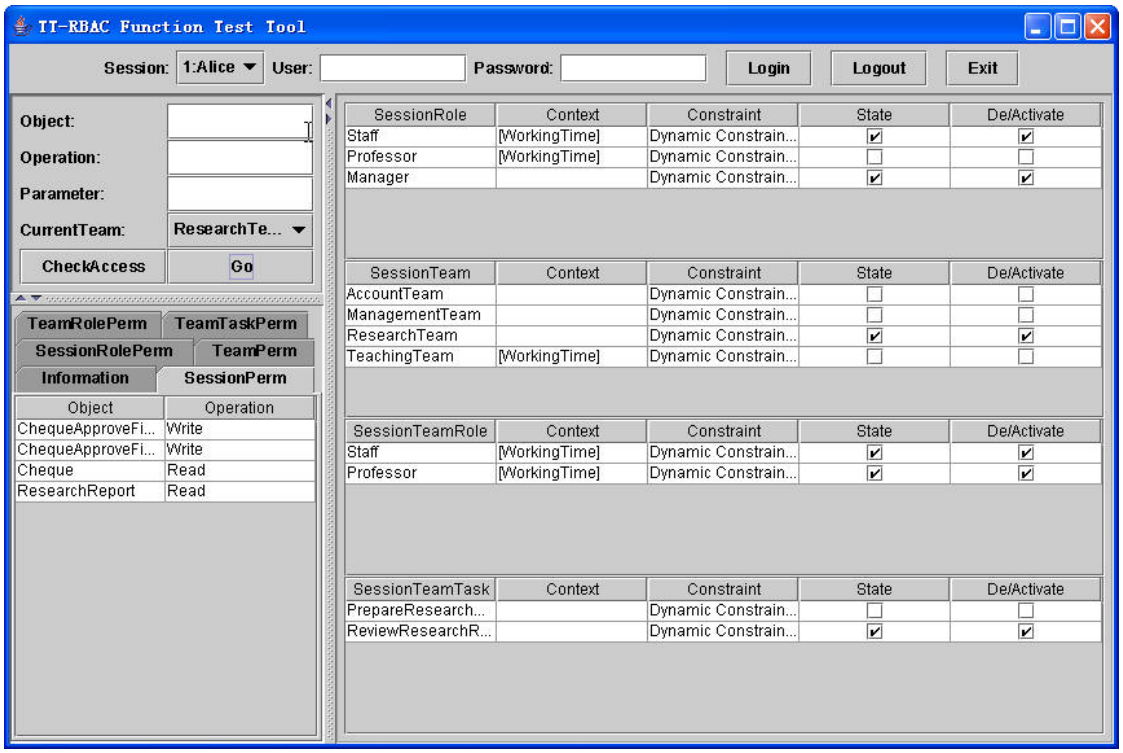


Figure 4.11: TT-RBAC function test tool.

4.6 Summary

The major contributions of the work described in this chapter can be summarized as follows. (1) Provide a software structure for TT-RBAC implementation with object-oriented technology and detail how the TT-RBAC functionalities are arranged into the TT-RBAC core classes. (2) Integrate fine-grained context constraints into TT-RBAC systems. (3) Present a novel access control decision making mechanism that can accelerate the speed of making access control decisions.

Chapter 5

Authorization Constraints

5.1 Introduction

Authorization constraints (also simply called constraints) as an important protection mechanism for handling important business processes or information should be integrated with all access control mechanisms, such as Discretionary Access Control (DAC), Mandatory Access Control (MAC), and especially Role-Based Access Control (RBAC) since most tasks within organizations are performed by roles [CXOL07]. Constraints as an important aspect of RBAC are powerful mechanism for laying out higher-level organizational policy. They are often regarded as one of the principal motivations behind RBAC and have been part of most RBAC models of recent years [SCFY96, LS97, GI97, BFA99, SBM99, FSGK01].

Separation of Duty (SoD) is an important control principle in management whereby sensitive combinations of duties are partitioned between different individuals in order to prevent the violation of business rules [Cra03]. It is widely considered to be a fundamental principle in computer security [SS75, CW87, Bis03]. The purpose of this principle is to discourage fraud by spreading the responsibility and authority for an action or task over multiple people, thereby raising the risk involved in committing a fraudulent act by requiring the involvement of more than one individual. One example of this is that checks may require two different signatures.

There are two important issues relating to constraints: their specification and their enforcement. Although the importance of constraints in RBAC has been recognized for a long time and various approaches have been proposed to model authorization constraints, there are still some issues have not received much attention in the research literature. The early work mainly addresses constraint expression rather than constraint enforcement. Currently there is still no useful approach for both expressing and enforcing constraints. On the other hand, the early research effort mainly concentrates on SoD. Other kinds of constraints, such as Binding of Duty (BoD), have received less attention. An example of BoD constraint is that a user can be assigned to role *A* only when he/she is already a member of role *B*. This kind of constraints is also called prerequisite constraints. Another example of BoD constraint is that one user is required to perform two different tasks in the same workflow instance. To our knowledge, there is still no dedicated work on this topic.

In this chapter, we introduce two novel authorization constraint schemes that can be used to express and enforce authorization constraints. To our knowledge ours is the first constraint specification language that could be used for both expression and enforcement aims. Unlike

existing approaches, these schemes are not confined to RBAC and some predefined entity relations such as user-role assignments. On the contrary, they can be used in various access control models and the entity relations can be arbitrary. These schemes are strongly bound to authorization entity set functions and authorization entity relation functions, so they are also called *function-based authorization constraint schemes*. Once the corresponding functions are defined and developed, then various constraint schemes that are based on these functions can be easily expressed and enforced. A function-based authorization constraint system is scalable through defining new entity set and relation functions. On the other hand, this approach goes beyond the well known separation of duty constraints, and considers other kinds of authorization constraints such as binding of duty constraints.

The rest of this chapter is organized as follows. Section 5.2 provides the background of authorization constraints and introduces the context for the constraint schemes. Section 5.3 gives the formal definition of the constraint schemes. Section 5.4 describes these constraint schemes' evaluation processes. Section 5.5 shows the constraint schemes' expressive power. Section 5.6 introduces the authorization constraint schema. Section 5.7 briefly introduces the implementation of function-based authorization constraints. Finally, Section 5.8 summarizes the results of this chapter.

5.2 Background

In this section we first discuss the related work and enumerate various constraint forms identified in the literature. We then introduce the TT-RBAC access control model that provides the context for investigating constraint specification and enforcement in this chapter. Finally, we discuss the constraint classification used by the constraint schemes.

5.2.1 Related work

Natural language is originally used to describe authorization constraints in the context of RBAC. In RBAC96 [SCFY96], the role-based separation of duty is described as "the same user can be assigned to at most one role in a mutually exclusive set". Simon and Zurko [SZ97] also develop a readable rule format to express the constraint policy at architectural level. Natural language specification has the advantage of ease to be understood by human beings, but may be prone to ambiguities, and the specifications do not lend themselves to the analysis of properties of the set of constraints [AS00]. For example, one may want to check if there are conflicting constraints in a set of authorization constraints. Another major drawback of using natural language is that constraint specification cannot be automatically dealt by computer systems.

In order to overcome the drawback of the informal definition of constraints, a variety of formal rule-based approaches have been proposed. Giuri and Iglio [GI96] defined a formal model for constraints on role-activation. Gligor et al. [GGF98] formalize separation of duty constraints enumerated informally by Simon and Zurko [SZ97]. This important theme is also addressed by Kuhn [Kuh97], Lupu and Sloman [LS99], Sandhu et al. [SBM99], and Ferraiolo et al [FSGK01]. Unfortunately, rule-based systems, while highly expressive, are harder to visualize and thus to use; thus far they have been avoided by practitioners [JT01].

Ahn and Sandhu [AS00] propose a limited logical language called RCL 2000 for expressing separation of duty constraints in the context of RBAC. RCL 2000 reduces the length of the statement of the constraints. However, some constraints require iteration over the members of one set or the other, and the addition of this expression starts to make the constraints complex. The combination of quantification functions and modeling concept functions makes the

constraints expressed in the language difficult to visualize. Thus, this approach is an improvement over a completely general logical language, but it is still too complex [JT01].

Nyanchama and Osborn [NO99] define a graphical model for role-role relationships that includes a combined view of role inheritance and separation of duty constraints based on roles. Osborn and Guo [OG00] extended the model to include constraints involving users. However, neither the basic model nor the extended model distinguishes between accidental relationships and explicitly constructed relationships. Thus, these models do not support policies with a historical component. Jaeger and Tidswell [JT01] proposed a graphical constraint model for constraint specification. In this graphical model the nodes represent sets (e.g., of subjects, objects, etc.) and the edges represent binary relationships on those sets and constraints are expressed using a few, simple set operators on graph nodes. This model has been designed to be applicable in a general access control model, not just in role-based access control models. The major advantage of a graphical model is as an aid to visualize a system's policies rather than enforce them.

Recently, constraint enforcement has received more attention in the research literature. The rule-based and graph-based approaches still provide significant expressive power in constraint expression, but they are not designed for constraint enforcement. To address this problem, several scheme-based approaches have been proposed.

Crampton [Cra03] proposed a simple specification scheme for separation of duty constraints in the context of RBAC. The specification scheme is set-based and has a simpler syntax than the early approaches. This constraint scheme is defined as a triple (s, c, x), where s is the scope set, c is the constraint set and x is the context and takes one of the following values: *static*, *dynamic* and *historical*. But this specification scheme cannot specify those constraints that are based on the aggregation of entities with quantification over sets and members of sets. For example, in the object-based separation of duty constraint, this scheme cannot express that a subject is restricted from performing an operation on an object twice. This shortcoming will limit its usage in many application cases.

The role-based constraint scheme designed by Li et al. [LBT04] is SMER($\{r_1, \ldots, r_n\}$, m) where r_i is a role, and n and m are integers such that $1< m \leq n$. This constraint forbids a user from being a member of m or more roles in $\{r_1, \ldots, r_n\}$. Chadwick et al. [CXOL07] extend this constraint scheme through adding application context for supporting separation of duty constraints among multiple sessions. The extended role-based constraint scheme is MMER($\{r_1, \ldots, r_n\}$, m, BC), where BC identifies the particular business context to which the m mutually exclusive roles apply, in which r_i is a role, and $1< m \leq n$. This constraint forbids a user from activating m or more roles among $\{r_1, \ldots, r_n\}$ in the same business context. Similarly, the permission-based constraint scheme is defined as MMEP($\{p_1, \ldots, p_n\}$, m, BC). The major drawback of these constraint schemes is that they cannot explicitly specify the scope set and assume the scope set is always a user set. So these constraint schemes cannot express certain constraints, such as mutually exclusive permissions cannot be assigned to the same role.

5.2.2 RBAC constraints

Various constraint forms have been identified in the literature. In the standard RBAC language, the taxonomy of constraints is summarized by Jaeger et al. [JT01] is:

- *User-user conflicts* are defined to exist if a pair of users should not be assigned to the same role.
- *Privilege-privilege conflicts* are defined to occur between two privileges (permissions)

when they should not both be assigned to the same role.

- *Static user-role conflicts* exclude users from ever being assigned to the specified roles.
- *Static separation of duty* exists if two particular roles should never be assigned to the same person.
- *Simple dynamic separation of duty* disallows two particular roles from being assigned to the same person due to some dynamic event (e.g., Chinese Wall).
- *Session-dependent separation of duty* disallows a principal from activating two particular roles at the same time (e.g., within the same session).
- *Object-based separation of duty* constrains a user never to act on the same object twice. They can also be specified to constrain the same role from acting on the same object twice.
- *Operational separation of duty* breaks a business task into a series of stages and ensures that no single person can perform all stages. Thus, the roles that are entitled to perform each stage may have users in common so long as no user is a member of all the roles entitled to perform each stage of a business task.
- *Order-dependent history constraints* restrict operations on business tasks based on a predefined order in which actions may be taken.
- *Order-independent history constraints* restrict operations on business tasks requiring two distinct actions (such as two distinct signatures) where there is no ordering requirement between the actions.

5.2.3 TT-RBAC constraints

TT-RBAC model extends the RBAC model through adding sets of two basic data elements called teams and tasks. Central to TT-RBAC is the concept of team relations, around which users, roles and tasks are connected together. Besides the entity relations defined in the RBAC model, Figure 3.6 illustrates user-team assignment (UMA), role-team assignment (RTA), task-team assignment (KMA) and permission-task assignment (PKA) relations defined in the TT-RBAC model. Similar to the constraints defined in RBAC, some constraints should be defined to restrict the ability to form these relations. In Chapter 3 we only discuss three team related separation of duty constraints such as conflict roles or tasks cannot be assigned to the same team.

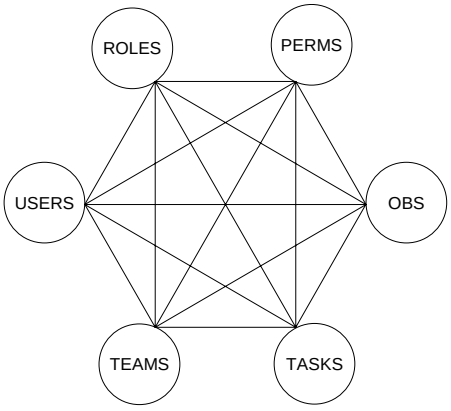


Figure 5.1: Possible entity relations in TT-RBAC.

In fact, the new added two entity sets make the entity relations in TT-RBAC more complicated than in RBAC. The Figure 5.1 shows the possible entity relations in TT-RBAC. Potentially, constraints can exist in any relations among these entities. Besides the separation of duty constraints, some other kinds of constraints need to be considered. For example, at least five users should be assigned to a team, each team must have one team leader and two vice-team leaders, a medical team can be activated only when at least one physician is assigned to the team, and so on.

It is futile to try to enumerate all interesting and practically useful constraints because there are too many possibilities and variations. Instead, we should pursue an intuitively simple yet rigorous approach for specifying and enforcing various constraints. The function-based constraint schemes introduced in this chapter can be used to express and enforce constraints for arbitrary entity relations.

5.2.4 Constraint classification

Simon and Zurko [SZ97] use two broadest categories of separation of duty variations: *static separation of duty* and *dynamic separation of duty*. Static separation of duty prevents mutually exclusive roles from assigning to the same user and conflict privileges from assigning to the same role. Dynamic separation of duty prevents some roles from being activated at the same time. History-based separation of duty is classified into this category. One example of history-based constraints is that one user cannot perform all the steps in a workflow instance. This kind of constraint classification is widely adopted in research literature (see, for example, Gligor et al. [GGF98], Bertino et al. [BFA99] and Li et al. [LBT04]).

NIST RABC [FSGK01] also classifies the constraints into two categories: static separation of duty and dynamic separation of duty. However, in NIST RBAC the static separation of duty only considers that mutually exclusive roles cannot be assigned to the same user, and the dynamic separation of duty only considers that conflict roles cannot be activated at the same time. History-based separation of duty is not supported by this model.

Crampton [Cra03] systematically discusses the history-based constraints that are classified into two categories: static historical constraints and dynamic historical constraints. One example of static historical constraint is that once a user u has been assigned to the role r_1, then u can never be assigned to the role r_2. One example of dynamic historical constraint is that once u has activated r_1, then u can never activate r_2.

Chadwick et al. [CXOL07] propose multi-session separation of duty to model the business processes which include multiple tasks enacted by multiple users over many user access control sessions in dynamic virtual organization environment. Basically, the multi-session separation of duty belongs to the history-based separation of duty.

In this chapter we adopt two axes on which to classify authorization constraints. The first axis is the *objective of constraints*. The second axis is the *enforcement context of constraints*.

According to the objective of constraints, we classify constraints into two categories: *prohibition constraints* and *obligation constraints*. Their definitions are taken from [AS00]. The prohibition constraints are constraints that forbid the entities from being (or doing) something which it is not allowed to be (or do). Separation of duty constraints belong to this category. Obligation constraints are constraints that force the entities to be (or do) something. Prerequisite constraints belong to this category. An example of obligation constraint is that a user can be assigned to one role only when he/she is already a member of another role. We designed two authorization constraint schemes that are used to express prohibition constraints and obligation constraints, respectively.

According to the enforcement context of constraints, we classify constraints into three categories: *static constraints*, *dynamic constraints* and *historical constraints*. Static constraints are enforced in privilege assignment stage; for example, a user is prevented from being assigned to mutually exclusive roles. Dynamic constraints are enforced at runtime within or across users' sessions; for example, a user is allowed to be authorized for two or more roles that do not create a conflict of interest when acted on independently, but produce policy concerns when activated simultaneously. Historical constraints allow the individual access of each user to be constrained based on what they have been (or done); for example, the same user cannot access the same object a certain number of times. In the workflow environment, one user may have the privileges to perform several steps, but the same user can only perform one step in the same workflow instance. Such kind of constraints is also classified into historical constraints. Both prohibition constraint scheme and obligation constraint scheme can express static, dynamic and historical constraints.

5.3 Constraint schemes

5.3.1 Functions for constraint schemes

To be able to use constraints to ensure safety, we must find a suitable formalism to express constraints and then enforce these constraints. Jaeger and Tidswell [JT01] identifies that the constraints in access control environment are set comparisons. There are two steps in expressing a set comparison: (1) expressing the sets to be compared and (2) expressing the comparison to be made. We designed two types of constraint schemes named *prohibition constraint scheme* and *obligation constraint scheme* to express the sets and their comparisons for various authorization constraints. These constraint schemes are strongly bound to some functions that can be classified into two categories: *entity set functions* and *entity relation functions*.

An entity set function is used to get a set of authorization entities according to some data query criterion. The set functions are used to represent and get data when it is not possible or not suitable to enumerate all data items, e.g. all the employees of a university. Set functions are used to represent entity sets in constraint schemes and obtain the entity sets at runtime. For example, the set function *get_account_users* can be used to represent and obtain all the users belonging to the financial department.

An entity relation function is used to get authorization entity relations between different types of entities. For example, the relation function *assigned_user_roles* returns the set of roles assigned to a given user, and the relation function *assigned_role_users* returns the set of users assigned to a given role.

Now we introduce the function naming rules adopted in this chapter. For static assignment functions we use the prefix *assigned_*, e.g. *assigned_user_roles*; for single session functions we use the prefix *session_*, e.g. *session_user_roles*; for multiple sessions functions we use the prefix *sessions_*, e.g. *sessions_user_roles*; for historical assignment functions we use the prefix *ever_assigned_*, e.g. *ever_assigned_user_roles*. In the presences of entity hierarchies we use the prefixes *authorized_* or *ever_authorized_* to replace the prefixes *assigned_* and *ever_assigned_*, respectively. All the entity relation functions are set valued functions. As a general notational device we have the following convention. For any set valued function f defined on set X, We understand

$f(X) = f(x_1) \cup f(x_2) \cup \ldots \cup f(x_n)$, where $X = \{x_1, x_2, \ldots, x_n\}$.

For example, assume we want to get all the roles assigned to a set of users $U = \{u_1, u_2, u_3\}$. We can express this using the function *assigned_user_roles*(U) as equivalent to *assigned_user_roles*(u_1) $\cup$ *assigned_user_roles*(u_2) $\cup$ *assigned_user_roles*(u_3).

5.3.2 Prohibition constraint scheme

The prohibition constraint scheme can be formatted as a triple (*S*, *C*, *T*), where *S* is the *scope element* that specifies what entities are applicable to the constraint scheme, *C* is the *constraint element* that specifies what constraint should be applied to each entity defined in the scope element, and *T* is the *constraint context* that takes one of the following values: *SPC*, *DPC* and *HPC*, which denote *Static Prohibition Constraint*, *Dynamic Prohibition Constraint*, and *Historical Prohibition Constraint*, respectively.

The scope element *S* is further defined as a 4-tuple (*SS*, *SF*, *SO*, *SN*). *SS* is the *scope set* that needs to be constrained, e.g. a user set. The scope set can be represented by an entity set function, called *scope set function*. For example, the function *get_users* returns a set of users. *SF* is the *scope relation function* that maps a value belonging to the constraint set defined in the constraint element to a set of values that have the same type with the scope set. For example, the function *assigned_role_users* returns a set of users assigned to a given role. *SO* is a relational operator that can be “>”, “>=”, “<”, “<=”, “=” or “≠”. *SN* as the right operand of *SO* is a natural number. The (*SO*, *SN*) pair expresses the cardinality constraint to the scope set. If *SF* is not specified, then all the members in the scope set are applicable to the constraint element. An example of scope element is ($\{u_1, u_2, u_3\}$, *assigned_role_users*, <, 3) that states that less than three users defined in the scope set can be assigned to a given role.

The constraint element *C* is further defined as a 4-tuple (*CS*, *CF*, *CO*, *CN*). *CS* is the *constraint set* that expresses the constraint entities applied to the scope set, e.g. a role set. The constraint set can be represented by an entity set function, called *constraint set function*. For example, the function *get_roles* returns a set of roles. *CF* is the *constraint relation function* that maps a value belonging to the scope set defined in the scope element to a set of values that have the same type with the constraint set. For example, the function *assigned_user_roles* returns a set of roles assigned to a given user. *CO* is a relational operator which can be “>”, “>=”, “<”, “<=”, “=” or “≠”. *CN* as the right operand of *CO* is a natural number. The (*CO*, *CN*) pair expresses the cardinality constraint to the constraint set. An example of constraint element is ($\{r_1, r_2, r_3\}$, *assigned_user_roles*, <, 2) that states that less than two roles defined in the constraint set can be assigned to a given user.

In TT-RBAC, the scope set and constraint set are a subset of the following data sets: *USERS*, *ROLES*, *PERMS*, *TEAMS*, *TASKS* and *OBJS*. Any constraints can be defined among these entity sets if the corresponding entity set functions and entity relation functions are defined. An example of prohibition constraint scheme is shown as:

($\{u_1, u_2, u_3\}$, *assigned_role_users*, <, 3), ($\{r_1, r_2, r_3\}$, *assigned_user_roles*, <, 2), *SPC*).

It states that no user defined in the scope set $\{u_1, u_2, u_3\}$ can be assigned to more than one role defined in the constraint set $\{r_1, r_2, r_3\}$, and less than three users defined in the scope set can be assigned to any role defined in the constraint set.

5.3.3 Obligation constraint scheme

The obligation constraint scheme can be formatted as a 4-tuple (*S*, *R*, *C*, *T*), where *S* is the *scope element* that defines the entities needs to be constrained, *R* is the *request element* that defines the

entities that the entities defined in the scope element want to be assigned to or activate, *C* is the constraint element that expresses what kind of constraints should be applied to each entity defined in the scope element, and *T* is the *constraint context* that takes one of the following values: *SOC*, *DOC* and *HOC*, which denote *Static Obligation Constraint*, *Dynamic Obligation Constraint* and *Historical Obligation Constraint*, respectively.

The scope element *S* only contains the *scope set SS*, in which the entities need to be constrained, e.g. a user set. The request element *R* only contains the *request set RS*, which members the entities defined in the scope set want to be assigned to or activate, e.g. a role set. The scope set and request set can be represented by entity set functions, called *scope set function* and *request set function* respectively. The constraint element *C* has the same structure and meaning as in the prohibition constrain scheme.

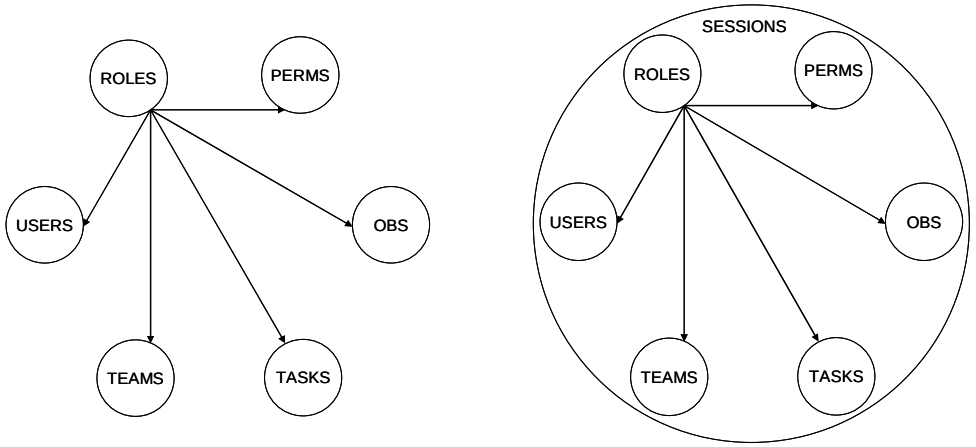
In TT-RBAC, the scope set, request set and constraint set are a subset of one of the following sets: *USERS*, *ROLES*, *PERMS*, *TEAMS*, *TASKS* and *OBJS*. An example of obligation constraint scheme is shown as:

$(\{u_1, u_2, u_3\}, \{r_3, r_4\}, (\{r_1, r_2\}, assigned\_user\_roles, >, 1), SOC)$.

It states that any user defined in the scope set $\{u_1, u_2, u_3\}$ can be assigned to any role defined in the request set $\{r_3, r_4\}$ if and only if he/she is already be assigned to more than one role defined in the constraint set $\{r_1, r_2\}$.

5.3.4 Entity relation functions for TT-RBAC

The entity relation functions are the glue to connect scope set and constraint set in constraint schemes. In this section we investigate what kinds of relation functions should be defined in the context of TT-RBAC. In our constraint schemes there is no any limitation to which entity sets can be in scope set, request set or constraint set. These data sets may be directly enumerated or represented by set functions. The constraint set shows what our major concern is. If the entity type of a constraint set is role, then we call it role-based constraint scheme. In TT-RBAC, constraint schemas can be based on user, role, permission, object, team or task. Any entity relations among of them can be created if the corresponding relation functions are developed.



(a) static relations (b) dynamic relations

Figure 5.2: Relations between roles and other entities in TT-RBAC.

Here we investigate what relation functions should be defined in the role-based constraint schemes in the context of TT-RBAC. The relation functions based on other types of entities could be gotten in the similar way. Figure 5.2 (a) shows the static relations among roles and other entities. For static constraints, we need to consider the relations of static assignments and historical assignments. Figure 5.2 (b) shows the dynamic relations among roles and other entities. These entities are encapsulated in sessions. For dynamic constraints, we need to consider the relations in a single session and multiple sessions.

As shown in Figure 5.2 (a) there are five types of entities related to the role entities. They are users, permissions, objects, teams and tasks. For static constraints we need to consider the relations of static assignments and historical assignments. Then we need to define five role-based static assignment functions and five role-based historical assignment functions. They are listed in Table 5.1 and Table 5.2, respectively.

Name	*Parameter*
assigned_user_roles	(user_set: 2^{NAME}; result: 2^{ROLES})
assigned_permission_roles	(permission_set: 2^{NAME}; result: 2^{ROLES})
assigned_object_roles	(object_set: 2^{NAME}; result: 2^{ROLES})
assigned_task_roles	(task_set: 2^{NAME}; result: 2^{ROLES})
assigned_team_roles	(team_set: 2^{NAME}; result: 2^{ROLES})

Table 5.1: Functions for role-based static assignment constraints.

Name	*Parameter*
ever_assigned_user_roles	(user_set: 2^{NAME}; result: 2^{ROLES})
ever_assigned_permission_roles	(permission_set: 2^{NAME}; result: 2^{ROLES})
ever_assigned_object_roles	(object_set: 2^{NAME}; result: 2^{ROLES})
ever_assigned_task_roles	(task_set: 2^{NAME}; result: 2^{ROLES})
ever_assigned_team_roles	(team_set: 2^{NAME}; result: 2^{ROLES})

Table 5.2: Functions for role-based historical assignment constraints.

In the presences of entity hierarchies we use the prefixes *authorized_* or *ever_authorized_* to replace the prefixes *assigned_* and *ever_assigned_*, respectively. For a prohibition constraint scheme, if there is a constraint to the scope set, then the scope relation function should be specified. For example, if the constraint relation function is *assigned_user_roles*, then the corresponding scope relation function is *assigned_role_user* that is defined in the user-based constraint relation functions. In order to avoid the redundancy, we will not list the corresponding scope relation functions here. Some examples of role-based static constraint schemes are listed as follows:

- ((*U*, , ,), (*R*, *assigned_user_roles*, <, *m*), *SPC*)

  is a static prohibition constraint scheme that states each user defined in the scope set *U* can only be assigned to less than *m* roles defined in the constraint set *R*.

- ((*U*, , ,), (*R*, *ever_assigned_user_roles*, <, *m*), *HPC*)

  is a historical prohibition constraint scheme that states each user defined in the scope set *U* can only ever be assigned to less than *m* roles defined in the constraint set *R*.

- (*U*, *RR*, (*R*, *assigned_user_roles*, >, *n*), *SOC*)

  is a static obligation constraint scheme that states each user defined in the scope set *U* can be assigned to a role defined in the request set *RR* only when this user has already been assigned to more than *n* roles defined in the constraint set *R*.

The Figure 5.2 (b) shows the dynamic relations among role and other entities. There are five types of entities related to the role entities. They are users, permissions, objects, teams and tasks. For dynamic constraints, we need to consider the activation relations under single session and multiple sessions, respectively. Thus, five single session entity activation functions and five multiple sessions entity activation functions should be defined for the dynamic role-based constraint schemes. They are listed in Table 5.3 and Table 5.4, respectively.

Name	*Parameter*
session_user_roles	(session: NAME; user_set: 2^{NAME}; result: 2^{ROLES})
session_permission_roles	(session: NAME; permission_set: 2^{NAME}; result: 2^{ROLES})
session_object_roles	(session: NAME; object_set: 2^{NAME}; result: 2^{ROLES})
session_task_roles	(session: NAME; task_set: 2^{NAME}; result: 2^{ROLES})
session_team_roles	(session: NAME; team_set: 2^{NAME}; result: 2^{ROLES})

Table 5.3: Functions for role-based single session activation constraints.

Name	*Parameter*
sessions_user_roles	(session_set: 2^{NAME}; user_set: 2^{NAME}; result: 2^{ROLES})
sessions_permission_roles	(session_set: 2^{NAME}; permission_set: 2^{NAME}; result: 2^{ROLES})
sessions_object_roles	(session_set: 2^{NAME}; object_set: 2^{NAME}; result: 2^{ROLES})
sessions_task_roles	(session_set: 2^{NAME}; task_set: 2^{NAME}; result: 2^{ROLES})
sessions_team_roles	(session_set: 2^{NAME}; team_set: 2^{NAME}; result: 2^{ROLES})

Table 5.4: Functions for role-based multiple sessions activation constraints.

Some examples of role-based dynamic constraint schemes are listed as follows:

- ((*U*, , ,), (*R*, *session_user_roles*, <, m), *DPC*)

  is a dynamic prohibition constraint scheme that states each user defined in the scope set *U* can only activate less than *m* roles defined in the constraint set *R* in any session.

- ((*U*, , ,), (*R*, *sessions_user_roles*, <, *m*), *DPC*)

  is a dynamic prohibition constraint scheme that states each user defined in the scope set *U* can only activate less than *m* roles defined in the constraint set *R* in anytime.

- (*U*, *RR*, (*R*, *session_user_roles*, >, *n*), *DOC*)

  is a dynamic obligation constraint scheme that states each user defined in the scope set *U* can activate a role defined in the request set *RR* only when this user has already activated more than *n* roles defined in the constraint set *R*.

5.4 Constraint scheme evaluation

The prohibition constraint schemes and obligation constraint schemes are not only used for expressing authorization constraints, but also used for enforcing authorization constraints. In this section, we investigate how these constraint schemes are evaluated at runtime.

5.4.1 Functions for constraint scheme evaluation

An authorization constraint request can be expressed as: (*s*, *o*, *a*), where *s* is the request subject, *o* is the request object, and *a* is the request action. An example of constraint request is (*u*, *r*, *RUA*) that is a role-user assignment request; the *RUA* denotes the action of assigning a role to a user. *CRS*, *CRO* and *CRA* are three functions used to get constraint request subject, object and action, respectively. For example, if $q = (u, r, RUA)$ is a constraint request, then $CRS(q) = u$, $CRO(q) = r$, and $CRA(q) = RUA$. Other functions related to the constraint scheme evaluation are defined as follows.

- *SRF*(*X*) represents the scope relation function that maps a set value *X* to another set value that has the same type with the scope set. At runtime, the *SRF* is replaced by the scope relation function of a constraint scheme, and the value *X* is replaced by the constraint set of the constraint scheme. For example, $assigned\_role\_users(\{r_1, r_2\}) = \{u_1, u_2, u_3\}$.
- *CRF*(*x*) represents the constraint relation function that maps a value *x* to a set value that has the same type with the constraint set. At runtime, the *CRF* is replaced by the constraint relation function of a constraint scheme, and the value *x* is replaced by the constraint request subject. For example, $assigned\_user\_roles(u) = \{r_1, r_2\}$.
- *EN*(*X*) is a function to get the element number of a set value *X*. For example, $EN(\{r_1, r_2, r_3\}) = 3$.

5.4.2 Prohibition constraint scheme evaluation

The prohibition constraint scheme evaluation comprises two steps. One is the *scope element evaluation* that is composed of *scope element applicable check* and *scope element cardinality check*. The other is the *constraint element evaluation* that is composed of *constraint element applicable check* and *constraint element cardinality check*. When ((*SS*, *SF*, *SO*, *SN*), (*CS*, *CF*, *CO*, *CN*), *T*) is the prohibition constraint scheme and *q* is the constraint request, the prohibition scope element evaluation can be formulated as:

$$ESE(q) = \begin{cases} CRS(q) \in SS & \text{if } SF = null \\ CRS(q) \in SS \wedge & \\ EN((SRF(CS) \cup CRS(q)) \cap SS)\,\text{satisfy}(SO, SN) & \text{if } SF \neq null \end{cases}$$

ESE is the function for the scope element evaluation. It first does the scope element applicable check. A scope element is "Applicable" if and only if the request subject is defined in the scope set, otherwise is "NotApplicable". If the scope element is applicable, the *ESE* then does the scope element cardinality check. There are two cases for this evaluation. In the case that *SF* is *null*, the *ESE* returns the value of "Permit". In the case that *SF* is not *null*, the *ESE* then checks if it still satisfies the scope element cardinality constraint after the object is assigned to the subject or activated by the subject. If so, the *ESE* returns the value of "Permit", otherwise returns the value of "Deny".

The prohibition constraint element evaluation can be formulated as:

$$ECE(q) = CRO(q) \in CS \wedge EN((CRF(CRS(q)) \cup CRO(q)) \cap CS) \text{satisfy}(CO, CN)$$

ECE is the function for the constraint element evaluation. It first does the constraint element applicable check. A constraint element is "Applicable" if and only if the request object is defined in the constraint set, otherwise is "NotApplicable". If the constraint element is applicable, the *ECE* then checks if it still satisfies the constraint element cardinality constraint after the object is assigned to the subject or activated by the subject. If so, the *ECE* returns the value of "Permit", otherwise returns the value of "Deny".

A prohibition constraint scheme evaluation result is "Permit" if and only if both scope element check and constraint element check return the value of "Permit". The prohibition constraint scheme evaluation can be formulated as:

$$EPC(q) = ESE(q) \wedge ECE(q).$$

The *EPC* is the function for the prohibition constraint scheme evaluation. The prohibition constraint scheme evaluation true table is shown in Table 5.5.

Value of ESE(q)	Value of ECE(q)	Value of EPC(q)
"Permit"	"Permit"	"Permit"
Do not care	"Deny"	"Deny"
"Deny"	Do not care	"Deny"
"NotApplicable"	"Permit", "NotApplicable" or "Indeterminate"	"NotApplicable"
"Permit", "NotApplicable" or "Indeterminate"	"NotApplicable"	"NotApplicable"
"Indeterminate"	"Permit", "Applicable" or "Indeterminate"	"Indeterminate"
"Permit", "Applicable" or "Indeterminate"	"Indeterminate"	"Indeterminate"

Table 5.5: Prohibition constraint scheme evaluation truth table.

5.4.3 Obligation constraint scheme evaluation

The obligation constraint scheme evaluation comprises three steps. The first is the *scope element evaluation* that is composed of *scope element applicable check*. The second is the *request element evaluation* that is composed of *request element applicable check*. The third is the *constraint element evaluation* that is composed of *constraint element cardinality check*. When (*SS*, *RS*, (*CS*, *CF*, *CR*, *CN*), *T*) is the obligation constraint scheme and *q* is the constraint request, the obligation scope element evaluation can be formulated as:

$$ESE(q) = CRS(q) \in SS.$$

ESE is the function for the scope element evaluation. It only needs to do the scope element applicable check. The scope element is "Applicable" if and only if the constraint request subject is defined in the scope set, otherwise is "NotApplicable".

The obligation request element evaluation can be formulated as:

$$ERE(q) = CRO(q) \in RS .$$

ERE is the function for the request element evaluation. It only needs to do the request element applicable check. The request element is "Applicable" if and only if the constraint request object is defined in the request set, otherwise is "NotApplicable".

The obligation constraint element evaluation can be formulated as:

$$ECE(q) = EN(((CRF(CRS(q)) \cup CRO(q)) \cap CS)\text{satisfy}(CO, CN)).$$

ECE is a function for the constraint element evaluation. It only needs to do the constraint element cardinality check. It checks if the constraint element still satisfies the constraint element cardinality constraint after the object is assigned to the subject or activated by the subject. If so, the *ECE* returns the value of "Permit", otherwise returns the value of "Deny".

An obligation constraint scheme evaluation result is "Permit" if and only if both scope element applicable check and request element applicable check return the value of "Applicable" and constraint element cardinality check returns the value of "Permit". The obligation constraint scheme evaluation can be formulated as:

$$EOC(q) = ESE(q) \wedge ERE(q) \wedge ECE(q).$$

EOC is the function for the obligation constraint scheme evaluation. The obligation constraint scheme evaluation true table is shown in Table 5.6.

Value of ESE(q)	Value of ERE(q)	Value of ECC(q)	Value of EOC(q)
"Applicable"	"Applicable"	"Permit"	"Permit"
"Applicable"	"Applicable"	"Deny"	"Deny"
"Applicable"	"Applicable"	"Indeterminate"	"Indeterminate"
"NotApplicable"	Do not care	Do not care	"NotApplicable"
Do not care	"NotApplicable"	Do not care	"NotApplicable"
"Indeterminate"	"Applicable" or "Indeterminate"	Do not care	"Indeterminate"
"Applicable" or "Indeterminate"	"Indeterminate"	Do not care	"Indeterminate"

Table 5.6: Obligation constraint scheme evaluation truth table.

5.4.4 Examples of constraint scheme evaluation

In this subsection we show the prohibition constraint scheme evaluation process and the obligation constraint scheme evaluation process via two examples, respectively.

Example 1: Prohibition constraint scheme evaluation

The following static prohibition constraint scheme states that each user defined in the scope set $\{u_1, u_2, u_3\}$ can only be assigned to less than two roles defined in the constraint set $\{r_1, r_2, r_3\}$, and less than three users defined in the scope set can be assigned to any role defined in the constraint set.

$((\{u_1, u_2, u_3\}, assigned\_role\_users, <, 3), (\{r_1, r_2, r_3\}, assigned\_user\_roles, <, 2), SPC)$

Next we use this prohibition constraint scheme to check a series of constraint requests. We assume that user u_1 is already assigned to role r_1. Thus, there are:

$assigned\_user\_roles(u_1) = \{r_1\}$
$assigned\_role\_users(r_1) = \{u_1\}$

- Constraint request1: $q = (u_2, r_2, RUA)$.

  (1) Scope element evaluation:

  $u_2 \in \{u_1, u_2, u_3\} \Rightarrow scope\ applicable\ check$ = Applicable $\wedge$
  $EN((SRF(CS) \cup CRS(q)) \cap SS) =$
  $EN((assigned\_role\_users(\{r_1, r_2, r_3\}) \cup \{u_2\}) \cap \{u_1, u_2, u_3\}) =$
  $EN((\{u_1\} \cup \{u_2\}) \cap \{u_1, u_2, u_3\}) = EN(\{u_1, u_2\}) = 2 < 3$
  $\Rightarrow ESE(q)$ = Permit

  (2) Constraint element evaluation:

  $r_2 \in \{r_1, r_2, r_3\} \Rightarrow constraint\ applicable\ check$ = Applicable $\wedge$
  $EN((CRF(CRS(q)) \cup CRO(q)) \cap CS) =$
  $EN((assigned\_user\_roles(u_2) \cup \{r_2\}) \cap \{r_1, r_2, r_3\}) =$
  $EN((\{\} \cup \{r_2\}) \cap \{r_1, r_2, r_3\}) = EN(\{r_2\}) = 1 < 2$
  $\Rightarrow ECE(q)$ = Permit

  (3) Constraint scheme evaluation:

  $EPC(q) = ESE(q) \wedge ECE(q)$ = Permit $\wedge$ Permit $\Rightarrow$ Permit

  (4) So this request is permitted. After r_2 is assigned to u_2, there are:

  $assigned\_user\_roles(u_1) = \{r_1\}$
  $assigned\_user\_roles(u_2) = \{r_2\}$
  $assigned\_role\_users(r_1) = \{u_1\}$
  $assigned\_role\_users(r_2) = \{u_2\}$

- Constraint request2: $q = (u_1, r_2, RUA)$.

  (1) Scope element evaluation:

  $u_1 \in \{u_1, u_2, u_3\} \Rightarrow scope\ applicable\ check$ = Applicable $\wedge$
  $EN((SRF(CS) \cup CRS(q)) \cap SS) =$
  $EN((assigned\_role\_users(\{r_1, r_2, r_3\}) \cup \{u_1\}) \cap \{u_1, u_2, u_3\}) =$
  $EN((\{u_1, u_2\} \cup \{u_1\}) \cap \{u_1, u_2, u_3\}) = EN(\{u_1, u_2\}) = 2 < 3$
  $\Rightarrow ESE(q)$ = Permit

  (2) Constraint element evaluation:

  $r_2 \in \{r_1, r_2, r_3\} \Rightarrow constraint\ applicable\ check$ = Applicable $\wedge$
  $EN((CRF(CRS(q)) \cup CRO(q)) \cap CS) =$
  $EN((assigned\_user\_roles(u_1) \cup \{r_2\}) \cap \{r_1, r_2, r_3\}) =$
  $EN((\{r_1\} \cup \{r_2\}) \cap \{r_1, r_2, r_3\}) = EN(\{r_1, r_2\}) = 2\ !< 2$
  $\Rightarrow ECE(q)$ = Deny

  (3) Constraint scheme evaluation:

$EPC(q) = ESE(q) \wedge ECE(q) = \text{Permit} \wedge \text{Deny} \Rightarrow \text{Deny}$

(4) So this request is denied, and role r_2 cannot be assigned to user u_1.

Example 2: Obligation constraint scheme evaluation

The following dynamic obligation constraint scheme states that each user defined in the scope set $\{u_1, u_2\}$ can activate any role defined in the request set $\{r_3, r_4\}$ if and only if this user has already activated at least one role defined in the constraint set $\{r_1, r_2\}$.

$(\{u_1, u_2\}, \{r_3, r_4\}, (\{r_1, r_2\}, session\_user\_roles, >, 0), DPC)$

Next we use this dynamic obligation constraint scheme to check a series of constraint requests. We assume that user u_1 already activated role r_1. Thus there are:

assigned_user_roles(u_1) = $\{r_1, r_3\}$
assigned_user_roles(u_2) = $\{r_2, r_4\}$
$session\_user\_roles(u_1) = \{r_1\}$

- Constraint request1: $q = (u_1, r_3, RUA)$.

  (1) Scope element evaluation:

  $CRS(q) = u_1, u_1 \in \{u_1, u_2\} \Rightarrow$ *scope applicable check* = Applicable
  $\Rightarrow ESE(q) = \text{Applicable}$

  (2) Request element evaluation:
  $CRO(q) = r_3, r_3 \in \{r_3, r_4\} \Rightarrow$ *request applicable check* = Applicable
  $\Rightarrow ERE(q) = \text{Applicable}$

  (3) Constraint element evaluation:

  $EN((CRF(CRS(q)) \cup CRO(q)) \cap CS) =$
  $EN((session\_user\_roles(u_1) \cup \{r_3\}) \cap \{r_1, r_2\}) =$
  $EN((\{r_1\} \cup \{r_3\}) \cap \{r_1, r_2\}) = EN(\{r_1\}) = 1 > 0$
  $\Rightarrow ECE(q) = \text{Permit}$

  (4) Constraint scheme evaluation:

  $EOC(q) = ESE(q) \wedge ERE(q) \wedge ECE(q) = \text{Applicable} \wedge \text{Applicable} \wedge \text{Permit} \Rightarrow \text{Permit}$

  (5) So this request is permitted. After u_1 activate to r_2, there are:

  $session\_user\_roles(u_1) = \{r_1, r_3\}$

- Constraint request2: $q = (u_2, r_4, RUA)$.

  (1) Scope element evaluation:

  $CRS(q) = u_2, u_2 \in \{u_1, u_2\} \Rightarrow$ *scope applicable check* = Applicable
  $\Rightarrow ESE(q) = \text{Applicable}$

  (2) Request element evaluation:

  $CRO(q) = r_4, r_4 \in \{r_3, r_4\} \Rightarrow$ *request applicable check* = Applicable
  $\Rightarrow ERE(q) = \text{Applicable}$

  (3) Constraint element evaluation:

$EN((CRF(CRS(q)) \cup CRO(q)) \cap CS) =$
$EN((session\_user\_roles(u_2) \cup \{r_4\}) \cap \{r_1, r_2\}) =$
$EN((\{\} \cup \{r_4\}) \cap \{r_1, r_2\}) = EN(\{\}) = 0 \not> 0$
$\Rightarrow ECE(q) =$ Deny

(4) Constraint scheme evaluation:

$EOC(q) = ESE(q) \wedge ERE(q) \wedge ECE(q) =$ Applicable $\wedge$ Applicable $\wedge$ Deny $\Rightarrow$ Deny

(5) So this request is permitted, and user u_2 cannot activate role r_4.

5.5 Expressive power of constraint schemes

In this section, we demonstrate the expressive power of our constraint schemes by showing how they can be used to express a variety of constraints. For comparative purposes, we indicate the correspondence between our examples and those in the paper by Jaeger and Tidswell [JT01], which provides probably the most comprehensive set of examples in the literature.

Example 1. A user-user conflict separation of duty constraint. It is forbidden for two users to both be assigned to any common authorization type (role). It belongs to the user-based separation of duty constraint and the constraint set is a subset of *USERS*. This constraint is expressed as:

$((R, , ,), (\{u_1, u_2\}, assigned\_role\_users, <, 2), SPC)$.

It states that no role defined in the scope set R can be assigned to both u_1 and u_2.

Example 2. A privilege-privilege conflict separation of duty constraint. It is forbidden for two permissions to both be assigned to a common authorization type (role). It belongs to the permission-based separation of duty constraint and the constraint set is a subset of *PERMS*. This constraint is expressed as:

$((R, , ,), (\{p_1, p_2\}, assigned\_role\_permissions, <, 2), SPC)$.

It states that no role defined in the scope set R can be assigned to both p_1 and p_2.

Example 3. A role-role conflict separation of duty constraint. It is forbidden for two roles to both be assigned to the same user. It belongs to the role-based separation of duty constraint and the constraint set is a subset of *ROLES*. This constraint is expressed as:

$((U, , ,), (\{r_1, r_2\}, assigned\_user\_roles, <, 2), SPC)$.

It states that no user defined in the scope set U can be assigned to both r_1 and r_2.

Example 4. A user can access one permission or the other, but not both. This constraint can be used to enforce a Chinese Wall restriction. That is, the history of users granted permission p_1 must not overlap with the history of users granted permission p_2. It belongs to the permission-based separation of duty constraint and the constraint set is a subset of *PERMS*. This constraint can be expressed as:

$((U, , ,), (\{p_1, p_2\}, ever\_assigned\_user\_permissions, <, 2), HPC)$.

It states that no user defined in the scope set U can be assigned to both p_1 and p_2.

Example 5. A object-object conflict separation of duty constraint. It is forbidden for two objects

to both be assigned to a common authorization type (role). It belongs to the object-based separation of duty constraint and the constraint set is a subset of *OBS*. This constraint can be expressed as:

((*R*, , ,), ({o_1, o_2}, *assigned_role_objects*, <, 2), *SPC*).

It states that no role defined in the scope set *R* can be assigned to both o_1 and o_2.

Example 6. A user is restricted from accessing an object more than once. It belongs to the object-based separation of duty constraint and the constraint set is a subset of *OBS*. This constraint can be expressed as:

((*U*, , ,), ({o_1}, *ever_assigned_user_objects*, <, 2), *HPC*).

It states that each user defined in the scope set *U* can only be assigned to o_1 less than two times.

Example 7. An alternative interpretation of the user-user conflict constraint expressed in Example 1 in which two sets of users are restricted form being assigned to any common authorization type (role). This constraint can be expressed with two prohibition constraint schemes specifying that two teams are restricted from being assigned to any common user or authorization type (role). In both constraint schemes, the constraint sets are the subsets of *TEAMS*. The constraint scheme

(({u_1, u_2, …, u_n}, , ,), ({m_1, m_2}, *assigned_user_teams*, <, 2), *SPC*)

states that no user is assigned to both m_1 and m_2. The constraint scheme

(({r_1, r_2, …, r_n}, , ,), ({m_1, m_2}, *assigned_role_teams*, <, 2), *SPC*)

states that no role is assigned to both m_1 and m_2.

Example 8. Another user-user conflict separation of duty constraint. Here two users are restricted from sharing any authorization type (role) in the set of restricted types (roles). In this case, it belongs to the user-based separation of duty and the constraint set is a subset of *USERS*. The constraint can be expressed as:

(({r_1, r_2, …, r_n}, , ,), ({u_1, u_2}, *assigned_role_users*, <, 2), *SPC*).

It states that no role defined in the scope set {r_1, r_2, …, r_n} can be assigned to both u_1 and u_2.

Example 9. A user-role conflict separation of duty constraint. A user or set of users are prohibited from being assigned to any authorization type (role) in a set. It belongs to the role-based separation of duty constraint and the constraint set is a subset of *ROLES*. This constraint can be expressed as:

(({u_1, …, u_n}, , ,), ({r_1, r_2, …, r_m}, *assigned_user_roles*, <, 1), *SPC*).

It states that no user defined in the scope set {u_1, …, u_n} can be assigned to any role defined in the constraint set{ r_1, r_2, …, r_m}.

Example 10. Operational separation of duty. In this constraint, no user is permitted to obtain all the permissions necessary to perform all the tasks in a process. Typically, each task in a process is represented by an authorization type (role), and then we can express this constraint in terms of these types (roles), i.e. each user can only be assigned to a subset of all these authorization types (roles). In this case, it belongs to the role-based separation of duty constraint and the constraint set is a subset of *ROLES*. This constraint can be expressed as:

((*U*, , ,), ({r_1, …, r_n}, *assigned_user_roles*, <, m), *SPC*).

It states that each user defined in the scope set *U* can only be assigned to less than *m* roles defined in the constraint set {r_1, …, r_n}, where $1 < m \leq n$.

Example 11. A session-dependent separation of duty constraint. In this constraint, all the users in an aggregate are prevented from being assigned to all the authorization types (roles) during their sessions. In this case, it belongs to the role-based dynamic separation of duty constraint and the constraint set is a subset of *ROLES*. This constraint can be expressed as:

((*U*, , ,), ({r_1, …, r_n}, *session_user_roles*, <, m), *DPC*).

It states that each user defined in the scope set *U* can only activate less than *m* roles defined in the constraint set {r_1, …, r_n} in a session, where $1 < m \leq n$.

Example 12. A universal quantification. In this constraint, all users are restricted from being assigned to more than one conflicting authorization type (role). In this case, it belongs to the role-based separation of duty constraint and the constraint set is a subset of *ROLES*. The constraint can be expressed as:

((*USERS*, , ,), ({r_1, …, r_n}, *assigned_user_roles*, <, 2), *SPC*).

It states that no user can be assigned to more than one role defined in the constraint set {r_1, …, r_n}.

Example 13. Reconsider the Example 10 in the context of authorization type (role) hierarchy. In this case, it belongs to the static role-based separation of duty constraint and the constraint set is a subset of *ROLES*. This constraint can be expressed as:

((*U*, , ,), (*authorized_role_roles*({r_1, …, r_n}), *authorized_user_roles*, <, *m*), *SPC*).

It states that each user defined in the scope set *U* can only be assigned to less than *m* roles defined in the constraint set {r_1, …, r_n} and the roles inherited by these roles. The function *authorized_role_roles* returns the roles that are inherited by a given role.

Example 14. Kuhn [Kuh97] identified that there may exist a mutual exclusion between authorization types whereby some permissions not involved in the mutual exclusion may be shared. In this constraint a mutual exclusion is set between types *A*' and *B*', but the inheritance of this constraint only excludes the permissions of *B*' from *A* and *A*' from *B*. It is possible for both *A* and *B* to inherit permissions from another authorization type *C* as long as the permissions inherited from *C* are disjoint from those in *A*' and *B*'. It means only part of the roles gotten through role hierarchies are mutually exclusive. We can use several simple constraint schemes to accomplish this complex constraint. These constraint schemes are shown as follows:

((*PERMS*, , ,), ({*A*', *B*'}, *authorized_user_roles*, <, 2), *SPC*),

((*A*, , ,), ({*B*' }, *authorized_role_roles*, <, 1), *SPC*),

((*B*, , ,), ({*A*'}, *authorized_role_roles*, <, 1), *SPC*),

((*C*, , ,), ({*A*', *B*'}, *authorized_role_roles*, <, 1), *SPC*).

Example 15. Both order-dependent and order-independent history constraints specify that a certain history must have taken place before an operation can be executed. For example, two *sign signature* tasks must be executed before the *approve task* can be performed in a workflow instance. It belongs to the order-dependent historical task-based obligation constraint and the constraint set is a subset of *TASKS*. If the two *sign signature* tasks and the *approve* task are

denoted with t_1, t_2 and t_3, respectively, this constraint can be expressed by the following task-based historical obligation constraint scheme.

(*USERS*, {t_3}, ({t_1, t_2}, *ever_performed_tasks*, >, 1), *HOC*).

It states that any user who can perform t_3 only when t_1 and t_2 have been performed. The function *ever_performed_tasks* is used to obtain the performed tasks in a workflow instance. If the task performing sequence is t_1, t_2 and t_3, then we can use the following two obligation constraint schemes to specify this execution order.

(*USERS*, {t_2}, ({t_1}, *ever_performed_tasks*, >, 0), *HOC*)

(*USERS*, {t_3}, ({t_2}, *ever_performed_tasks*, >, 0), *HOC*)

5.6 Constraint schema

Sometime we may need to combine multiple constraint schemes together to express and enforce a complicated authorization constraint, such as Example 14 and Example 15 discussed in Section 5.5. A *Constraint schema* is used to organize a set of constraint schemes together for expressing and enforcing authorization constraints. In a constraint schema, the algorithm used to combine multiple constraint schemes is "deny-overrides" that is described in the following.

In the entire set of schemes in a schema, if any scheme evaluates to "Deny", then the result of the scheme combination is "Deny". If any scheme evaluates to "Permit" and all other schemes evaluate to "NotApplicable", then the result of the scheme combination is "Permit". If all schemes evaluate to "NotApplicable", then the scheme combination is "NotApplicable". If an error occurs while evaluating a scheme, then the scheme combination is "Indeterminate".

Now we illustrate how the constraint schema is used to encapsulate a set of constraint schemes for expressing and enforcing authorization constraints. The application scenario is: only the users (*U*) who have the role *Staff* can be assigned to the role *President* or *Vice-President*; there is only one user can be assigned to the role *President*, and no more than two users can be assigned to the role *Vice-President*; role *President* and role *Vice-President* are mutually-exclusive roles. In order to implement these constraints, the corresponding constraint schema should include the following schemes:

1. (*U*, {*President*, *Vice-President*}, ({*Staff*}, *assigned_user_roles*, >, 0), *SOC*),
2. ((*U*, *assigned_role_user*, <, 2), ({*President*}, *assigned_user_roles*, <, 2), *SPC*),
3. ((*U*, *assigned_role_user*, <, 3), ({*Vice-President*}, *assigned_user_roles*, <, 2), *SPC*),
4. ((*U*, , ,), ({*President*, *Vice-President*}, *assigned_user_roles*, <, 2), *SPC*).

The first scheme specifies that only users who have the role *Staff* can be assigned to roles *President* and *Vice-President*. The second scheme specifies that less than two users can be assigned to the role *President*. The third scheme specifies that less than three users can be assigned to the role *Vice-President*. The fourth scheme specifies that *President* and *Vice-President* are mutually-exclusive roles. In order to express and enforce these constraint schemes, the functions *get_employees*, *assigned_user_roles* and *assigned_role_users* should be developed.

Multiple constraint schemas could also be combined together to express and enforce more complicated authorization constraints. How to deal with multiple constraint schemas is out of the scope of this thesis.

5.7 Implementation

In this section we introduce the implementation of our function-based authorization constraints. We first introduce the structure of constraint request, and then present the classes designed for constraint schema, finally, look at the evaluation processes of prohibition constraint schemes and obligation constraint schemas.

5.7.1 Constraint request

Constraint request is a data structure that can be formulated as (*s*, *o*, *a*, *c*), where *s* is the request subject, *o* is the request object, *a* is the request action, and *c* is the constraint context through which some external context information can be passed into an authorization constraint engine. For example, a database object from which more information about the request subject, object and action can be gotten.

In our implementation the constraint request is a Java class named *ConstraintRequest*, which holds the authorization constraint information of subject, object, action and context. The request subject, object and action are expressed by a class named *ConstraintObject* which holds their identity and type information. The context is a Java Hashtable that is a container used to provide various kinds of information needed by the authorization constraint engine. There is not strict definition about what kinds of content should be put into the constraint context. It will depend on the application requirements. For example, database objects, user session objects or constants can all be put into the constraint context.

5.7.2 Constraint component structure

The function-based authorization constraint component is developed with object-oriented technology. The essential classes that compose this component and their relations are shown in Figure 5.3. *ConstraintSchema* is the class holding and enforcing constraint schemas. *ConstraintSPC*, *ConstraintDPC*, *ConstraintHPC*, *ConstraintSOC*, *ConstraintDOC* and *ConstraintHOC* are classes used to hold and enforce the six types of constraint schemes, respectively. All the prohibition constraint scheme classes inherit from the superclass *ConstraintPC*. All the obligation constraint scheme classes inherit from the superclass *ConstraintOC*. *ConstraintPC* and *constraintOC* further inherit from the superclass *Constraint*. Their functionalities are described as follows:

- *ConstraintSchema* serves as an interface for the constraint component, that is, it hides the internal structures from other components that use this service. Thus, other components use the authorization constraint service through a well-defined API offered by the class *ConstraintSchema*. Each *ConstraintSchema* instance holds only one constraint schema that specifies which constraint schemes should be satisfied before agreeing on a constraint request.
- *ConstraintSPC*, *ConstraintDPC*, *ConstraintHPC* are used to hold and evaluate static, dynamic and historical prohibition schemes, respectively. Each class instance holds one prohibition constraint scheme. They are responsible for extracting data from the constraint requests and invoking the methods defined in the superclass to complete the corresponding prohibition constraint scheme evaluation.
- *ConstraintSOC*, *ConstraintDOC*, *ConstraintHOC* are used to hold and evaluate static, dynamic and historical obligation schemes, respectively. Each class instance holds one

obligation constraint scheme. They are responsible for extracting data from the constraint request and invoking the methods defined in the superclass to complete the corresponding obligation constraint scheme evaluation.

- *ConstraintPC* defines the variables used to hold a prohibition constraint scheme and the methods used to evaluate prohibition constraint schemes.
- *ConstraintOC* defines the variables used to hold an obligation constraint scheme and the methods used to evaluate obligation constraint schemes.
- *Constraint* defines the common variables and methods used by both *ConstraintPC* and *ConstraintOC*. The entity set functions and relation functions are implemented or registered in this class.

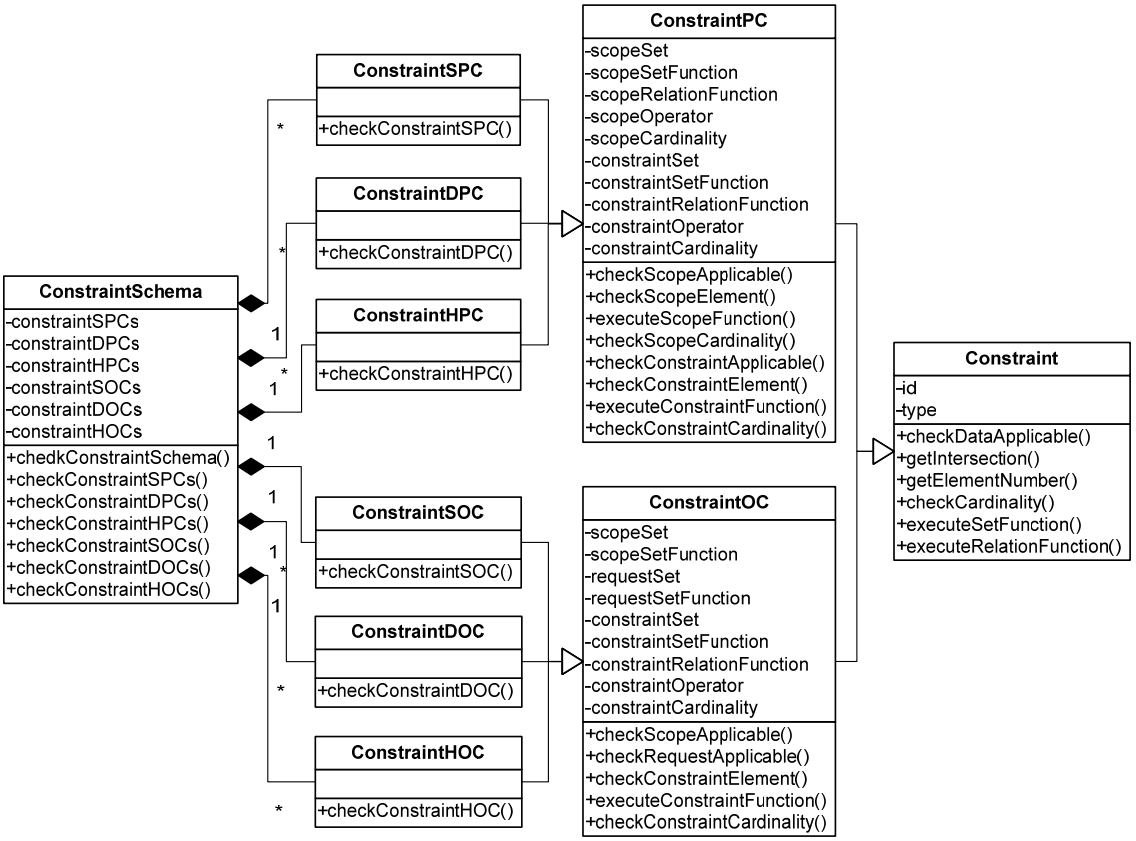


Figure 5.3: Essential class relations of authorization constraint component

Each constraint schema instance holds only one constraint schema. When a constraint request is evaluated by a constraint schema, the request is passed to the class method *checkConstraintSchema* that sequentially calls other six methods for checking the six types of constraint schemes. The combining algorithm for these constraint schemes are "deny-overrides", i.e. if any one of them evaluates to "deny", then the final evaluation result is "deny". The evaluation result could be "Deny", "Permit", "NotApplicable" or "Indeterminate". These constraint schemes' evaluation processes can be classified into two categories. One is the prohibition constraint scheme related, and the other is the obligation constraint scheme related. We investigate their evaluation processes in the next two subsections.

5.7.3 Prohibition constraint evaluation process

The three kinds of prohibition constraint schemes follow the similar evaluation process. The *ConstraintSPC*, *ConstraintDPC* and *ConstraintHPC* classes perform the function of converting the nonstandard outside constraint requests into the standard inside constraint requests, and then passing these standard constraint requests to *ConstraintPC* class that checks if these constraint requests satisfy a given constraint scheme.

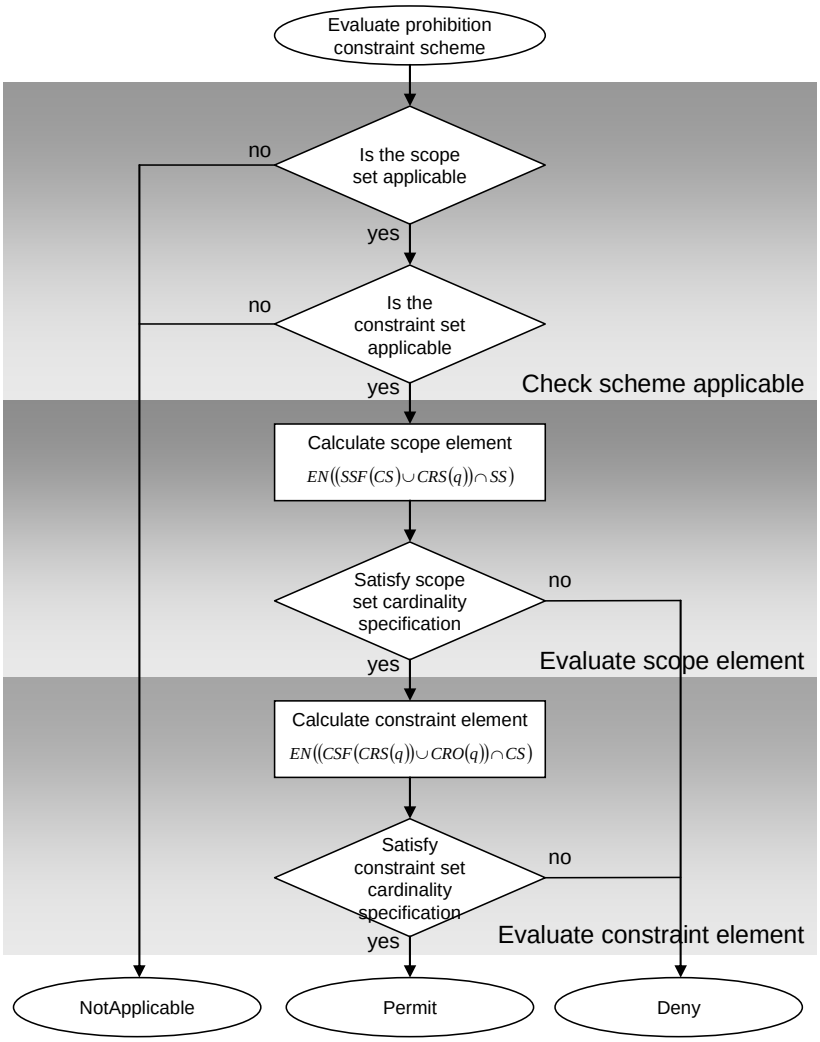


Figure 5.4: Prohibition constraint scheme evaluation steps.

Prohibition constraint schemes are held and evaluated in *ConstraintPC* class. Its major variables and methods are shown in Figure 5.3. The prohibition constraint scheme evaluation is separated into two steps. One is the constraint scheme applicable check. The other is the constraint scheme cardinality check.

A prohibition constraint scheme is applicable if and only if the request subject matches the scope set and the request object matches the constraint set. The constraint scheme scope set match and constraint set match are executed through the methods *checkScopeApplicable* and *checkConstraintApplicable*, respectively.

The prohibition constraint scheme cardinality check comprises scope element cardinality check and constraint element cardinality check. They are executed by the methods *checkScopeElement* and *checkConstraintElement*, respectively. The flow chart of the prohibition constraint scheme evaluation process is shown in Figure 5.4.

5.7.4 Obligation constraint evaluation process

The three kinds of obligation constraint schemes follow the similar evaluation process. The *ConstraintSOC*, *ConstraintDOC*, *ConstraintHOC* classes perform the function of converting the nonstandard outside constraint requests into the standard inside constraint requests, and then passing these standard constraint requests to *ConstraintOC* class that checks if these constraint requests satisfy a given constraint scheme.

Obligation constraint schemes are held and evaluated in *ConstraintOC* class. The major variables and methods are shown in Figure 5.3. The obligation constraint scheme evaluation is separated into two steps. One is the constraint scheme applicable check. The other is the scheme cardinality check.

An obligation constraint scheme is applicable if and only if the request subject matches the scope set and the request object matches the request set. The constraint scheme scope set match and request set match are executed through the methods *checkScopeApplicable* and *checkRequestApplicable*, respectively.

The obligation constraint scheme cardinality check only needs to do the constraint element cardinality check that is executed by the method *checkConstraintElement*. The flow chart of the obligation constraint scheme evaluation process is shown in Figure 5.5.

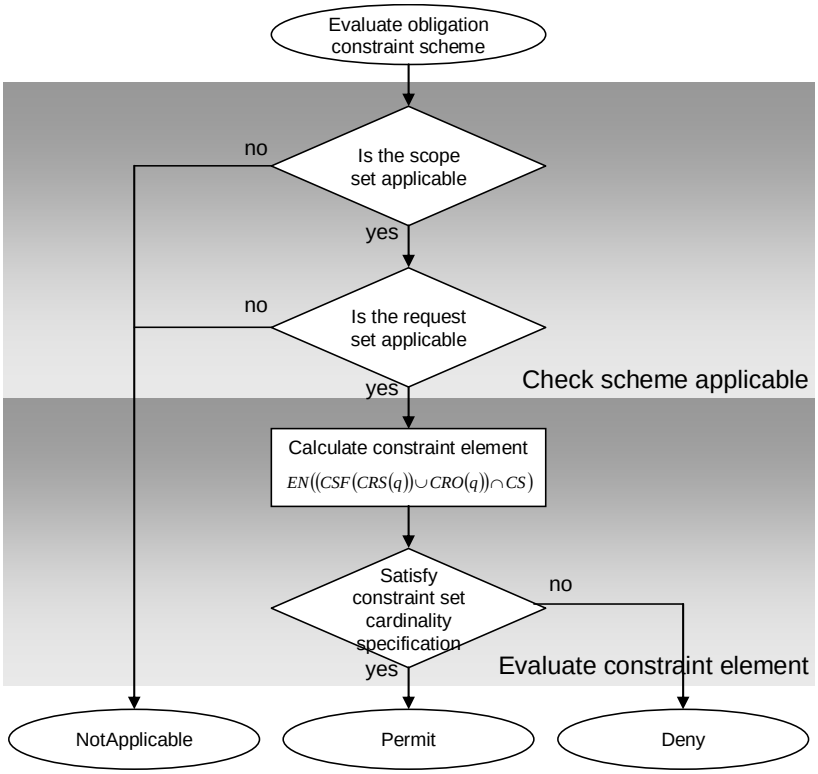


Figure 5.5: Obligation constraint scheme evaluation steps.

5.8 Conclusion

The major contribution of the function-based authorization constraints is summarized as follows. (1) Providing two novel authorization constraint schemes that can be used for both expressing and enforcing authorization constraints. These constraint schemes can be used in various access control models and the entity relations are arbitrary. (2) These constraint schemes are strongly bound to some functions that could be directly mapped to the functions that need to be developed in application systems. Thus, these schemes provide the system developers a clear view about which functions should be defined in an authorization constraint system. Based on these functions, various constraint schemes can be easily defined, and then enforced. The security administrators can use these functions to create constraint schemes for their day-to-day operations. (3) An authorization constraint system could be scalable through defining new entity set functions and entity relation functions. (4) On the hand, this approach goes beyond the well known separation of duty constraints, and considers many aspects of entity relation constraints. We believe that our approach is far simpler to understand, much closer to the real world, and has less cumbersome syntax.

Function-based authorization constraints have been applied in some real application systems. Details bout these application cases can be found in Section 8.2 and Section 8.3.

Chapter 6

Label Policy

6.1 Introduction

To effectively participate in modern collaborations, member organizations must be able to share specific data and functionality with collaboration partners, while ensuring their resources are safe from inappropriate access. In collaborative environments the participants and trust relationships may dynamically change. The dynamic and multi-institutional nature introduces challenging security issues. Various access control models and policies have been developed to protect the shared resources. In this chapter we also call these policies as *normal authorization policies*. As collaborative systems becoming more complex, it is possible that there are some information leaks caused by the improperly defined authorization policies. This chapter specifically addresses the following problem: how to enhance the normal access control policies so that the information leaks caused by these policies could be avoided or reduced.

In collaborative environments, access control policies may need to explicitly specify which users from which organizations can perform what operations on what resources. When there are many organizations involved, especially those organizations need to be dynamically added or removed, considerable burden could be brought to the security administration. So it is better to move the dynamic control information to some separated control policies, and then merge them to the normal access control policies at runtime. Dealing with some security administration, such as adding or removing participants from collaborative systems, the administrators only need to modify these control policies rather than the normal policies. This process is obviously easier and less error-prone than modifying the normal access control policies. At the same time, detaching the dynamic control information also simplifies the normal access control policies.

As access control policies becoming more complex, some information leaks could be caused by the improperly defined access control policies. For example, the access control policies are too coarse or complicated to withdraw some privileges in time. If there is a mechanism through which some extra constraints could be applied to the normal access control policies so that their usages could be restricted or some functions could be easily turned on/off, then some potential information leaks could be avoided or reduced.

Role-Based Access Control (RBAC) [SCFY96, FSGK01] is an alternative to traditional discretionary and mandatory access controls. In RBAC, permissions are associated with roles, and users are made members of appropriate roles thereby acquiring the roles' permissions. Roles can be created for various job functions, and users are assigned to roles based on their

responsibilities and qualifications. Roles can be granted new permissions as new applications and systems are incorporated, and permissions can be revoked from roles when needed. Users can be easily reassigned from one role to another. RBAC is thus more scalable than user-based security specifications and greatly reduces the cost and administrative overhead. But subsequent attempts to apply RBAC in various application environments reveal some limitations of RBAC.

- RBAC assumes that all permissions needed to perform a job function can be neatly encapsulated. In fact, role engineering has turned out to be a difficult task [HFK06]. The challenge of RBAC is the contention between strong security and easier administration. For stronger security, it is better for each role to be more granular, thus having multiple roles per user. For easier administration, it is better to have fewer roles to manage.
- RBAC lacks the ability to specify a fine-grained control on individual users in certain roles and on individual object instances. For collaborative environments, it is insufficient to have role permissions based on object types. Rather, it is often the case that a user in an instance of a role might need a specific permission on an instance of an object type [TAPH05]. On the other hand, RBAC does not provide an abstraction to capture a set of collaborative users operating in different roles.
- RBAC products have sometimes proved challenging to implement and will, for some organizations, need to be combined with rule-based and other more time-tested access control methods to achieve the most practical value [Des03].

In this paper we introduce a novel access control policy, called Label-Based Access Control Policy (LBACP) that could be used to address the issues mentioned above [ZM08a]. The concept of LBACP was originally presented in [ZRMA06]. The LBACP is used to enforce some constraints to the normal access control policies from the view of information flow. Its basic principle is defining some labels that specify the information flow constraints, and then assigning these labels to the normal access control policy components. The usages of the labeled policy components must obey the information flow constraints defined by the labels in order to avoid being misused. The LBACP can also be used to improve access control policy management. Through assigning multiple labels to policy components, these policy components can be managed and enforced in multiple dimensions (various application aspects). The LBACP is a high level security policy layered on the top of normal access control policies.

The rest of this chapter is organized as follows. Section 6.2 introduces the label model and its basic operations. Section 6.3 describes label policy composition. Section 6.4 investigates label policy assignments and information flow. Section 6.5 investigates label policies in hierarchical contexts. Section 6.6 looks at some label policy application examples. Section 6.7 gives a brief introduction to our label policy implementation. Section 6.8 discusses some related work. Finally, Section 6.9 summarizes the results of this chapter.

6.2 Label model

In LBACP model there are three essential elements: *contexts*, *label polices* and *labels*. Their definitions and operations are described in the following subsections.

6.2.1 Context and information flow

C*ontexts* are the entities whose privacy is protected by the model. Information is owned by, updated by and released to contexts, which are the basic elements in LBACP model.

Information could be any access control related entities, such as users, roles, files and so on. Contexts could be created based on different criterions. Contexts could be created based on data ownership such as finance, engineering and human resources, or created based on data sensitivity level such as top-secret, secret, confidential and unclassified, or created based on data categories such as marketing, financial, sales and personnel.

Information can flow from a *source context* to a *target context*, only when the source context permits its information can be exposed to the target context and the target context also permits to receive the information from the source context. Figure 6.1 shows an example of information flow between two contexts.

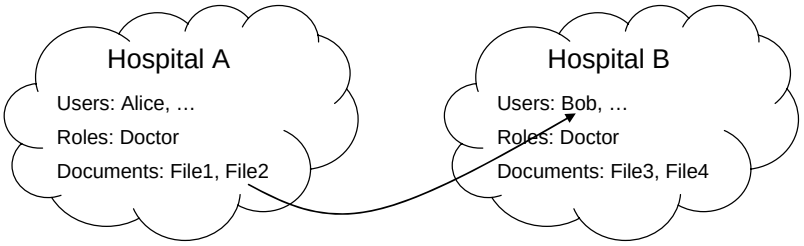


Figure 6.1: Example of information flow between contexts.

In this example *Hospital A* and *Hospital B* are two contexts. The role *Doctor* is defined for sharing information between two hospitals. This role is permitted to read *File*1 and *File*2 in *Hospital A* and *File*3 and *File*4 in *Hospital B*. From the view of RBAC, any user who has role *Doctor* can read *File*1, *File*2, *File*3 and *File*4. When information flow constraints are considered, besides the normal access control policy evaluation, we also need to check if information can flow from the object (source) context to the subject (target) context. In the case shown in Figure 6.1, we need to check if *File*2 is permitted to flow to context *Hospital B*, and if *Bob* is permitted to receive information from context *Hospital A*. Our label-based access control policies are dedicated to specify and enforce information flow constraints among access control entities, such as users, roles and objects in the environment of RBAC.

6.2.2 Label policy structure

Label policies are the ways that contexts express their privacy concerns. A label policy has three parts: an *owner context*, a set of *import contexts* and a set of *export contexts*. The owner context is a context whose data is observed. The import contexts are a set of contexts, from which the owner context accepts information flow. The export contexts are a set of contexts, to which the owner context allows information flow. It is also implicitly understood that the owner of the policy permits its information flow to itself, even if it is not explicitly in import context set or export context set. An example of an expression that denotes a label policy P is the following: $P = \{o: i_1, i_2: e_1, e_2\}$ where o, i_1, i_2, e_1, e_2 denote contexts. Colons separate the owner context, import context set and export context set within the label policy. A label policy with no import/export contexts means that no information is permitted to flow to/from this context. An example of a label policy containing no import and export contexts is $\{o::\}$, which is equivalent to the label policy $\{o:o:o\}$. It means there is no information can flow to or from this owner context. We use wildcard "*" to denote all the possible contexts. So, $\{o:*:\}$, $\{o::*\}$ and $\{o:*:*\}$ denote that the owner context permits information can flow from all the possible contexts, information can flow to all the possible contexts, and information can flow from and to all the possible contexts, respectively.

6.2.3 Label structure

A *label* is composed of a set of label policies. Labels are assigned to the access control policy components that need information flow constraints. The intuitive meaning of a label is that every label policy in the label must be obeyed as data flows through the system, so labeled information is permitted to flow in or out only by the consensus of all the label policies. Because the intersection of all of the policies is enforced, adding more policies to a label only restricts the propagation of the labeled data. An example of an expression that denotes a label L with two label policies is the following: $L = \{o_1: i_1, i_2: e_1, e_2; o_2: i_2, i_3: e_2, e_3\}$, where o_1, o_2, i_1, i_2, i_3, e_1, e_2, e_3 denote contexts. Semicolon separates two policies within the label. The owners of these policies are o_1 and o_2. The import sets for the policies are $\{i_1, i_2\}$ and $\{i_2, i_3\}$ respectively, thus the import context set consented by them is $\{i_2\}$. The export sets for the policies are $\{e_1, e_2\}$ and $\{e_2, e_3\}$ respectively, thus the export context set consented by them is $\{e_2\}$. The least restrictive label is an empty label containing no label policies, because no context has expressed an interest in restraining the data with this label. An empty label is written as $\{\}$. A data labeled with an empty label equals to no label is associated with it.

6.2.4 Information flow between label policies

Label policies are used to define information flow related to their owner contexts. Information can flow from a context into a label policy owner context only when this context is defined in the label policy import context set. Information can flow from a label policy owner context into a context only when this context is defined in the label policy export context set. If P is a label policy, then the notation $o(P)$ denotes the label policy owner context, the notation $i(P)$ denotes the set of import contexts including the owner context, the notation $e(P)$ denotes the set of export contexts including the owner context. If I and J are two label policies and C represents the set of all contexts, the information flow between them is defined as:

$$I \mapsto J \Leftrightarrow (I = J) \vee ((o(J) \in e(I) \vee e(I) = C) \wedge (o(I) \in i(J) \vee i(J) = C))$$

6.2.5 Information flow between labels

One label may contain multiple label policies. In order to define the information flow between labels, we first define a label's owner, import and export contexts. A label's owner contexts are the union of all its label policies' owner context. A label's import contexts are the intersection of all its label policies' import contexts plus its owner contexts. A label's export contexts are the intersection of all its label policies' export contexts plus its owner contexts. Information can flow from a label's import contexts to the label's owner contexts or from a label's owner contexts to the label's export contexts. If L is a label, then the notations $lo(L)$, $li(L)$ and $le(L)$ denote the label's owner, import and export contexts, respectively. If P_1, P_2, …, P_n are the label policies of a label L, then the $lo(L)$, $li(L)$ and $le(L)$ are formally defined as:

$$lo(L) = \bigcup_{i=1}^{n} o(P_i)$$

$$li(L) = \left(\bigcap_{i=1}^{n} i(P_i)\right) \cup \left(\bigcup_{i=1}^{n} o(P_i)\right)$$

$$le(L) = \left(\bigcap_{i=1}^{n} e(P_i)\right) \cup \left(\bigcup_{i=1}^{n} o(P_i)\right)$$

If L_1 and L_2 are two labels, then information flow between them is permitted only if L_1 equals L_2, or the export contexts of L_1 contain the owner contexts of L_2 and the import contexts of L_2 contain the owner contexts of L_1. Formally this is defined as:

$$L_1 \mapsto L_2 \Leftrightarrow (L_1 = L_2) \vee ((le(L_1) \subseteq lo(L_2)) \wedge (lo(L_1) \subseteq li(L_2)))$$

6.2.6 Information flow channels

In real applications we more concern the *information channel* between two labels. A channel is used to describe that information can flow from which owner contexts in label *A* to which owner contexts in label *B*. A channel is composed of *input channel* and *output channel*. Input channel describes which contexts are the source contexts of a channel. Output channel describes which contexts are the target contexts of a channel. Information can flow from any input channel contexts to any output channel contexts. We can understand that $L_1 \mapsto L_2$ is a full information channel between two labels, i.e. information can flow from all owner contexts of L_1 to all owner contexts of L_2. If L_1 and L_2 are two labels and the information flow direction is from L_1 to L_2, then the input channel, output channel and information channel are defined as:

input_channel(L_1, L_2) = $lo(L_1) \cap li(L_2)$
output_channel(L_1, L_2) = $le(L_1) \cap lo(L_2)$
information_channel(L_1, L_2) = (*input_channel*, *output_channel*)

For example, C_1, C_2, C_3, C_4, and C_5 are five contexts, L_1 and L_2 are two labels defined as follows:

L_1 = {C_2: :C_1}
L_2 = {C_1: C_2, C_5: ; C_3: C_2, C_4:}

The information channel (from L_1 to L_2) calculation process is described as follows:

$lo(L_1) = \{C_2\}$
$le(L_1) = \{C_1, C_2\}$
$lo(L_2) = \{C_1, C_3\}$
$li(L_2) = \{C_1, C_2, C_3\}$
input_channel(L_1, L_2) = $\{C_2\} \cap \{C_1, C_2, C_3\} = \{C_2\}$
output_channel(L_1, L_2) = $\{C_1, C_2\} \cap \{C_1, C_3\} = \{C_1\}$
information_channel(L_1, L_2) = ($\{C_2\}$ $\{C_1\}$)

6.3 Label composition

In LBACP model one policy component can be associated with multiple labels. This section describes label composition algebra with a syntax consisting of labels and composition operators of *conjunction*, *disjunction* and *separation*.

6.3.1 Conjunction

Conjunction permits accesses that are allowed by both its components, i.e. conjunction merges two labels by returning their intersection. If L_1 and L_2 are two labels, the conjunction is formally

defined as:

$$L_1 \cap L_2 = \left\{ \begin{pmatrix} O, \\ \bigcap \{ i(K) \mid K \in L_1 \vee K \in L_2, o(K) = O \}, \\ \left(\bigcap \{ e(K) \mid K \in L_1 \vee K \in L_2, o(K) = O \} \right) \end{pmatrix} \middle| \; O \in o(L_1) \vee O \in o(L_2) \right\}$$

Intuitively, conjunction enforces minimum privilege. For example, consider a virtual organization in which participant organizations share certain documents. An access to a document may be allowed only if all the authorities agree on it. Some information, such as valid organization domains to which this document is permitted to be released, can be organized into labels. An example of label conjunction is shown as follows:

$L_1 = \{o_1: i_1, i_2: e_1, e_2\}$
$L_2 = \{o_2: i_2, i_3: e_2, e_3\}$
$L_2 \cap L_2 = \{o_1: i_2: e_2; o_2: i_2: e_2\}$

6.3.2 Disjunction

Disjunction permits accesses that are allowed under either of its components, i.e. disjunction merges two labels by returning their union. If L_1 and L_2 are two labels, the disjunction is formally defined as:

$$L_1 \cup L_2 = \left\{ \begin{pmatrix} O, \\ \bigcup \{ i(K) \mid K \in L_1 \vee K \in L_2, o(K) = O \}, \\ \left(\bigcup \{ e(K) \mid K \in L_1 \vee K \in L_2, o(K) = O \} \right) \end{pmatrix} \middle| \; O \in o(L_1) \vee O \in o(L_2) \right\}$$

Intuitively, disjunction enforces maximum privilege. For example, there is a portal used by a virtual organization that manages a collection of resources and users belonging to different organizations. Access to this portal could be authorized by any of the participant organizations. Some information, such as valid user domains, can be organized into labels. The totality of the accesses to the portal should be the union of the statements of each organization. An example of label disjunction is show as follows:

$L_1 = \{o_1: i_1: e_1 \}$
$L_2 = \{o_2: i_2: e_2 \}$
$L_2 \cup L_2 = \{o_1: i_1, i_2: e_1, e_2; o_2: i_1, i_2: e_1, e_2\}$

6.3.3 Separation

Separation allows multiple labels are assigned to one component, but there is not any relation among of them. At runtime, only one of them is used for a particular access control. If L_1 and L_2 are two labels, the separation is formally defined as:

$L_1 ; L_2$

Separation allows policy components being segmented along multiple dimensions (application aspects). For example, there are a set of rules for the access control to patient medical information. These rules are assigned with labels L_{surgery}, L_{medicine} or $L_{\text{surgery}};L_{\text{medicine}}$. At runtime access control engine may only use the rules labeled with L_{surgery} or L_{medicine} for a particular access control.

6.4 Label assignments

Labeled policy components have to conform to the information flow constraints specified by the labels. To enable fine-grained information flow constraints, labels could be assigned to policy components at different levels, such as policies, rules, rule elements and so on. In order to investigate the information flows among authorization policy components, we first need a suitable way to express authorizations. To make our approach generally applicable, we do not make any assumption on the subjects, objects or actions with respect to which authorization specifications should be stated. For this purpose, Authorization Specification Language (ASL) [JSSS01] is adopted for specifying authorization policies. ASL is a logical language that expresses authorizations in the form of rules. Here we give a brief introduction to some ASL notations that we used in this chapter.

Data symbols used in ASL are *Obj*, *T*, *U*, *G*, *R*, *A* and *SA* that represent the sets of *objects*, *types*, *users*, *groups*, *roles*, *unsigned actions* and *signed actions*, respectively. The *authorization subject hierarchy* captures the ordering relationships among authorization subjects. It is represented with $ASH = (U, G \cup R, \leq_{AS})$, where $\leq_{AS}$ is defined as follows:

$$x \leq_{AS} y \text{ iff } \{x, y\} \subseteq U \cup G \;\&\; x \leq_{UG} y \text{ or } \{x, y\} \subseteq R \;\&\; x \leq_{R} y$$

The *authorization object hierarchy* captures the ordering relationships among authorization objects. It is represented with $AOH = (Obj, T \cup R, \leq_{AO})$, where $\leq_{AO}$ is defined as follows:

$$x \leq_{AO} y \text{ iff } \{x, y\} \subseteq Obj \cup T \;\&\; x \leq_{OT} y \text{ or } \{x, y\} \subseteq R \;\&\; y \leq_{R} x$$

ASL defines many predicate symbols. Here we only introduce two predicate symbols used in this thesis. They are *in* and *cando*.

in is a ternary predicate symbol that takes as arguments two elements of authorization subjects or authorization objects and whose third argument is a ground term that equals to either *ASH* or *AOH*. This predicate captures the ordering relationships in *ASH* and *AOH* hierarchies. The *in* predicate symbol is defined as:

Let $H=\{X, Y, \leq_H\}$ be a hierarchy whose nodes $(X \cup Y)$ are ordered by relation $\leq_H$. *H* satisfies in(x, y, *H*) for $x, y \in X \cup Y$ iff $x \leq_H y$.

cando is a ternary predicate symbol. The first argument is an authorization object, the second is an authorization subject and the third is a signed action. This predicate represents the access that the system security officer wishes to allow or deny depending on the sign associated with the action. The *cando* predicate symbol is defined as:

cando(o, s, <*sign*>a)$\leftarrow L_1 \& \ldots \& L_n$, where o, s, and a are elements of authorization objects, authorization subjects and actions respectively, $n \geq 0$, <*sign*> is either + or -, and $L_1, \ldots, L_n$ are predicate symbols.

If *p* is one of the above predicate symbols with arity *n*, and t_1, …, t_n are *terms* appropriate for *p* as defined above, then $p(t_1, \ldots, t_n)$ is an *atom*. An atom is denoted with word *literal*. For example, if *OT*, *ST* and *SAT* are authorization object, authorization subject and signed action respectively, then cando(*OT*, *ST*, *SAT*) and ¬cando(*OT*, *ST*, *SAT*) are examples of literals. *OT*, *ST* and *SAT* are literal terms. Some authorization rules written in ASL are shown as follows:

in(Alice, Clerk, ASH)←
in(Bob, Manager, ASH)←
in(File1, Report, AOH)←
in(File2, Report, AOH)←
cando(file, user, +read) ← in(user, Clerk, ASH) & in(file, Report, AOH)
cando(file, user, +write) ← in(user, Manager, ASH) & in(file, Report, AOH)

The first four *in* rules consist of information about subject and object hierarchies, and the next two rules describe how accesses propagate along these hierarchies. *ASH* and *AOH* denote authorization subject and object hierarchies, respectively. *ASH* consists of *Clerk* and *Manager* two roles. Alice is assigned to role *Clerk*, and Bob is assigned to role *Manager*. *AOH* has one data type *Report* with members of *File*1 and *File*2. The two *cando* rules specify that all clerks are allowed to read reports and only managers are allowed to write reports.

In LBACP, labels could be assigned to *literal terms*, *literals*, *rules*, *policies* and *data elements*. Each label assignment has its applicable scope. We investigate these label assignments in the following subsections.

6.4.1 Information flow in policy rules

Authorization policies are used to specify which users can perform what actions on which objects. For the purpose of easy administration, user *groups*, *roles* and object *types* are widely used in various authorization systems. Administrators can specify the permissions that are needed by the group of users by granting the group permission in an access control list (ACL) for the object. In RBAC, access rights are associated with roles, and users are assigned to roles thereby acquiring the corresponding permissions.

The main purpose of label policies are adding extra constraints on the subjects and objects involved in access control from the view of information flow. So the entities affected by the label policies are the authorization subjects and authorization objects. We specify that the information flow direction between a subject (e.g. a user) and an object (e.g. a file) is from the object to the subject. Groups, roles and types are the data elements that serve as intermediaries between subjects and objects. The information flows among of them are shown in Figure 6.2, and described as follows.

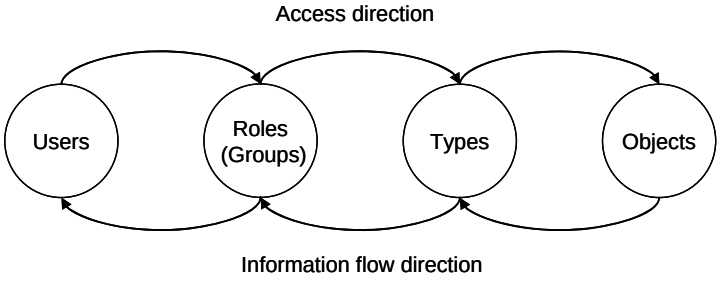


Figure 6.2: Information flow among different entities.

- If an object is labeled, then the effects of import contexts are ignored, and this object can only be exposed to the export contexts.
- If a type is labeled, then only the objects belonging to import contexts are valid data members, and these data members can only be exposed to export contexts.
- If a role/group is labeled, then only the objects belonging to import contexts are valid data members, and these data members can only be exposed to export contexts. In other words, only the users from export contexts can activate this role, and access the objects belonging to import contexts.
- If a user is labeled, then the effects of export contexts are ignored, and only the objects belonging to import contexts are valid data for this user.

Authorization rules are the most elementary unit for specifying entity relations among subjects, objects and actions in authorization policies. One subject can access an object also implies that information can flow from the object context to the subject context. Here we use a role-based authorization rule to analyze information flows among rule elements. This authorization rule expressed in ASL and the information flow existing in it are shown in Figure 6.3. This rule states that the users who have *Researcher* role are authorized to read the files that belong to *Report* type.

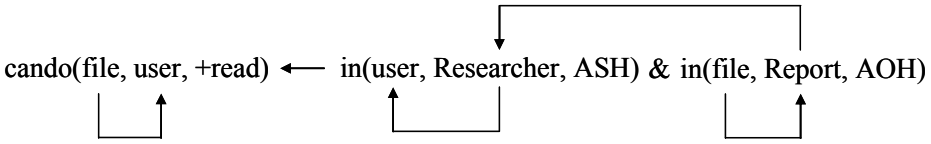


Figure 6.3: Information flow in a RBAC authorization rule.

6.4.2 Label specification for literal terms

Some rule literal terms can be associated with labels. Literal terms could be constants or variables. For simplicity, we specify that only constants can be associated with labels.

Label specification for in literal terms

For the literal *in*(*x*, *y*, *H*), both terms *x* and *y* belong to the same hierarchy that is either an authorization subject hierarchy or an authorization object hierarchy, and *y* is the data type of which *x* wants to be a member. Both of them can be associated with labels. The information flow direction between *x* and *y* is decided by the relation type that this *in* literal is used to describe.

When *in* literals are used to describe subject relations, the information flow check between *x* and *y* is if information can flow from *y* to *x*. If data member *x* is associated with a label, then *x* can only access the data belonging to the contexts specified by the label import contexts. If data type *y* is associated with a label, then only the data members coming from the contexts specified by the label export contexts are valid and these data members can only access the data belonging to the contexts specified by the label import contexts. If both *x* and *y* are associated with labels, then we need to check if the information can flow from the label assigned to *y* to the label assigned to *x*. The following labeled authorization rule states that only the users who have *Researcher* role and come from *Computer Science* and *Mathematics* are authorized to read the *Reports* of *Computer Science*. Without considering the effects of the information flow

constraints, this rule states that all the users who have *Researcher* role are authorized to read all the *Reports*.

L1 = (Computer Science: : Mathematics)
cando(file, user, +read) ← in(user, Researcher^{L1}, ASH) & in(file, Report, AOH)

When *in* literals are used to describe object relations, the information flow check between *x* and *y* is if information can flow from *x* to *y*. If data member *x* is associated with a label, then *x* can only be exposed to the contexts specified by the label export contexts. If data type *y* is associated with a label, then only the data members belonging to the contexts specified by the label import contexts are valid and these data members can only be exposed to the contexts specified by the label export contexts. If both *x* and *y* are associated with labels, then we need to check if the information can flow from the label assigned to *x* to the label assigned to *y*. The following labeled authorization rule states that all the users who have *Researcher* role and come from *Computer Science* are authorized to read the *Reports* of *Computer Science* and *Mathematics*.

L2 = (Computer Science: Mathematics:)
cando(file, user, +read) ← in(user, Researcher, ASH) & in(file, Report^{L2}, AOH)

Label specification for cando literal terms

For a *cando*(*o*, *s*, <sign>*a*) literal, the terms *o* and *s* are elements of authorization objects and authorization subjects, respectively. Both of them can be associated with labels. The information flow check between *o* and *s* is if information can flow from *o* to *s*. If the object term *o* is associated with a label, then the object can only be exposed to the contexts specified by the label export contexts. If the subject term *s* is assigned with a label, then only the data belonging to the contexts specified by the label import contexts can be accessed by the subject *s*. If both *o* and *s* are assigned with labels, then we need to check if the information can flow from the label assigned to *o* to the label assigned to *s*.

The following example shows an authorization rule in which the object term of the *cando* literal is associated with a label. Without considering the effects of information flow constraints, this authorization rule states that all the users who have *Staff* role are authorized to read *File*1. After considering the effects of information flow constraints, this authorization rule states that the *File*1 can only be read by the users who have *Staff* role and come from *Enrollment Office*.

L3 = (Enrollment Office: :)
cando(File1^{L3}, user, +read) ← in(user, Staff, ASH)

The following example shows another authorization rule in which the subject term of the *cando* literal is associated with a label. Without considering the effects of information flow constraints, this authorization rule states that *Alice* is authorized to read all *Letters*. After considering the effects of information flow constraints, this authorization rule states that *Alice* can only read the *Letters* from *USA*, *UK* and *AU*.

L4 = (Enrollment Office: USA, UK, AU:)
cando(o, Alice^{L4}, +read) ← in(o, Letter, AOH)

6.4.3 Label specification for literals

Rule literals can be associated with labels that specify the overall information flow constraints to these literals. All the literal terms that can be associated with labels should conform to the information flow constraints specified by the literal labels. The effects of information flow constraints to a given literal term is the intersection of the labels assigned to the literal and the labels assigned to the literal term. For example, the literal in^{L1}(user, $Staff^{L2}$, ASH) is equal to the literal in(user, $Staff^{L1 \cap L2}$, ASH).

6.4.4 Label specification for rules

Rules can be associated with labels that specify the overall information flow constraints to these rules. All the rule literals should conform to the information flow constraints specified by the rule labels. The effects of information flow constraints to a given literal is the intersection of the labels assigned to the rule and the labels assigned to the literal. For example, the following two labeled authorization rules are equal.

$(\text{cando}(\text{File1}, \text{user}, +\text{read}) \leftarrow \text{in}^{L1}(\text{user}, \text{Staff}, \text{ASH}))^{L2}$
$\text{cando}^{L2}(\text{File1}, \text{user}, +\text{read}) \leftarrow \text{in}^{L1 \cap L2}(\text{user}, \text{Staff}, \text{ASH})$

6.4.5 Label specification for policies

The whole policy can be associated with labels that specify the overall information flow constraints to this policy. All the policy rules should conform to the information flow constraints specified by the policy labels. The effects of information flow constraints to a given rule is the intersection of the labels assigned to the policy and the labels assigned to the rule. For example, the following two labeled policies are equal.

$\text{policy} = \{\text{rule}_1^{L1}, \text{rule}_2, \ldots, \text{rule}_n\}^{L2}$
$\text{policy} = \{\text{rule}_1^{L1 \cap L2}, \text{rule}_2^{L2}, \ldots, \text{rule}_n^{L2}\}$

6.4.6 Label specification for data elements

Normally, authorization policies have data element definition part that specifies which subjects, objects and actions are used in these policies. If labels are assigned to the entities specified in this part, the effect scope of these labels will automatically scatter to the whole application scope of these entities. For example, if a role is labeled in its definition part, then its subsequent usages should conform to the information flow constraints specified by these labels.

6.5 Hierarchical contexts

In big organizations, such as universities and governments, there are many departments, and some of them may also have their own groups. These organizations, departments and groups could be modeled as contexts in different levels. Thus, context hierarchies could be constructed. In this section we investigate information flow constraints in context hierarchies. Context hierarchies introduce layer structure in context specification. An example of context hierarchy is shown in Figure 6.4. In this example, all the groups (contexts) of school of engineering are grouped together under the context *Engineering*. School of engineering and other departments

(contexts) are grouped together under the context *University*.

In context hierarchies, participant contexts grouped under a superior context can be referred to collectively via this superior context. In the above example, at the university level the *Engineering* context can be used to specify information flow to the school of engineering, while within the school of engineering the information flow can be further refined.

With the context hierarchies introduced in this section it would be possible to specify information flow within a group. For example, we can specify that information may freely flow from the context *Engineering* into any other context in this group, which in this case is *Manufacture*, *Mechanics* and *Electronic*. This context group can be extended later without the need to modify the information flow restriction for the parent context *Engineering*.

One problem that could be addressed with hierarchical contexts is the distributed administration of contexts. In the above example the school of engineering takes care of its *Engineering* context, but its groups are responsible for their own contexts.

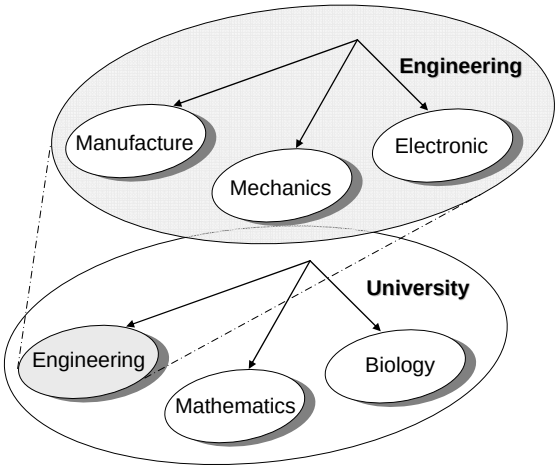


Figure 6.4: Example of context hierarchy.

6.5.1 Context hierarchy definition

Context hierarchies define an inheritance relation among contexts. Inheritance is described in terms of parent and child context relations. Context hierarchy is a limited hierarchy that imposes restrictions resulting in a simple tree structure, i.e. one context may have one or more immediate descendants, but is restricted to a single immediate ascendant.

Although limited context hierarchies do not support multiple inheritances, they nonetheless provide clear administrative advantages to the information flow control. We represent c_1 as an immediate ascendant of c_2 by $c_2 << c_1$, if $c_2 \leq c_1$, but no context in the context hierarchy lies between c_1 and c_2. That is, there exists no context c_3 in the context hierarchy such that $c_2 \leq c_3 \leq c_1$, where $c_2 \neq c_3$ and $c_1 \neq c_3$. We now define limited context hierarchies as a restriction on the immediate ascendants of the general context hierarchy. We shall use *CS* to refer to all the hierarchical contexts. The context hierarchies are formally defined as follows.

$CH \subseteq CS \times CS$ is a partial order on *CS* called the inheritance relation written as $\leq$, where $c_2 \leq c_1$ only if c_1 is the parent context of c_2.This relation pairs together parent context and child context. Contexts in a context hierarchy are restricted to a single immediate parent context. Therefore the context hierarchy must satisfy the following restriction:

$$\forall c, c_1, c_2 \in CS, c << c_1 \wedge c << c_2 \Rightarrow c_1 = c_2$$

We also introduce the predicate *Croot* to denote the root context, i.e. one that has no parent:

$Croot(c : CS) \rightarrow BOOL$, where *BOOL* is either true or false.

$$\forall a \in CS : Croot(a) = (\neg \exists b \in CS : a \leq b)$$

6.5.2 Hierarchical context specification

In order to specify the contexts in a context hierarchy, we specify a function *Cname* that returns the name associated with such a hierarchical context. In the case of non-hierarchical contexts, we required the names of contexts are unique. In the case of hierarchical contexts, we require the root context names are unique. Similarly, sibling contexts must use different names. This is expressed by the following constraints:

$$\forall a, b \in CS, (Croot(a) \wedge Croot(b)) \vee (\exists c \in CS : a << c \wedge b << c) \Rightarrow$$
$$Cname(a) \neq Cname(b)$$

In order to describe the location of contexts in a context hierarchy, we specify a function *Cpath* that returns the names separated by '.' on a path from the context hierarchy root to the given context. The *Cpath* is defined as follows:

$$Cpath(x) = \begin{cases} Cpath(y) + '.' + Cname(x) & \text{if } \exists y \in CS : x << y \\ Cname(x) & \text{if } Croot(x) \end{cases}$$

Note that as a path from a context hierarchy root to a context is unambiguous, and because of the restrictions, there is a bijection between the above path strings and the context. Consequently, the *Cpath* function is invertible, with inverse $Cpath^{-1}$. This bijection enables policy administrators to refer more easily to hierarchical contexts through a path expression.

6.5.3 Information flow in context hierarchy

The information flow specification in context hierarchies is not the same as in non-hierarchical ones. In a context hierarchy, one context can only have relations with the contexts that it directly links to. They are the *parent context*, *sibling contexts* and *child contexts*. When wildcard "*" is used, the meaning of the phrase "information is allowed from or to everywhere" can be interpreted now to any context that one context directly links in a context hierarchy. In other words, when one context uses wildcard to specify its import/export contexts in a context hierarchy, the information can flow from or to its parent context, all sibling contexts, all child contexts and itself. Therefore we shall introduce wildcard information flow definition in context hierarchies. The wildcard symbol will have the form of *expand*(*X*) denoting the context *X* and all its directly linked contexts. In order to specify information flows in context hierarchies, we define three functions that are used to express the relations of one context to its parent context, all sibling contexts and all child contexts. These functions are summarized as follows:

- *Cparent*(*x*: *CS*) → *CS*, the mapping of context *x* onto its parent context. Formally: $Cparent(x) = \{y \in CS \mid x << y\}$.

- *Csibling*(*x*: *CS*) → *CS*, the mapping of context *x* onto a set of sibling contexts. Formally: $Csibling(x) = \{s, y \in CS \mid x << y \wedge s << y\}$.
- *Cchild*(*x*: *CS*) → *CS*, the mapping of context *x* onto a set of child contexts. Formally: $Cchild(x) = \{y \in CS \mid y << x\}$.

Now we redefine the meaning of wildcard in context hierarchies. For a given context that uses the wildcard '*' to represent its import contexts or export contexts in a context hierarchy, we use the function *Cexpand* to interpret the wildcard '*'. This function is defined as follows:

$Cexpand(x) = \{x\} \cup Cparent(x) \cup Csibling(x) \cup Cchild(x)$

We also introduce the set function *Cconvert* to convert sets of contexts, parent context expressions, sibling context expressions, child context expressions and wildcard expressions to a set of contexts. For the set valued function *Cconvert* defined on set *S*, we understand

$Cconvert(S) = Cconvert(s_1) \cup Cconvert(s_2) \cup \ldots \cup Cconvert(s_n)$, where $S = \{s_1, s_2, \ldots, s_n\}$.

The following example shows how the *Cconvert* function expands the child contexts of a given context in the context hierarchy shown in Figure 6.4.

$$Cconvert\left(\begin{Bmatrix} Uinversity.Mathematics, \\ Uinversity.Engineering, \\ Cchild(Engineering) \end{Bmatrix}\right) = \begin{Bmatrix} Uinversity.Mathematics, \\ Uinversity.Engineering, \\ Uinversity.Engineering.Manufacture, \\ Uinversity.Engineering.Mechanics, \\ Uinversity.Engineering.Electronic \end{Bmatrix}$$

Instead of considering the information flow relation as it is, we shall consider its transitive closure. Consequently, we get the following behavior for parent and child contexts. Information may flow from a child to the parent if this is either explicitly stated, or the child allows information to flow via a wildcard to its parent and other contexts. Similarly, information may flow from a parent to a child if the information flow is either explicitly specified, or information flow is permitted via a wildcard from the parent to the child context and other contexts.

6.5.4 Context hierarchy application

According to the redefinition of wildcard, we know that the information flow is more restricted in hierarchical contexts than in non-hierarchical contexts. In a context hierarchy the information flow related to a given context cannot beyond its parent, sibling and child contexts. This characteristic makes it possible to define a group of contexts that are totally isolated from the outside. One example of applying context hierarchy is shown in Figure 6.5. In this example, *Secure* is the parent context of contexts *Normal* and *High* that represent different sensitivities of information. The dash lines with arrows show the parent-child relations. The solid line shows permitted information flow that is from *Normal* to *High*. Here the context *Secure* groups its two child contexts together.

In this example, the parent context does not specify any permitted information flow from or to its child contexts. It means that no information beyond the parent context can flow into its

child contexts and no information belonging to the child contexts can flow out of the parent context, because any information exchange must go through their parent context and the parent context prohibits such kinds of information exchange. Even the parent context cannot obtain the information belong to its child contexts. So the security information is totally isolated inside the parent context. The following example shows label policies applying in a context hierarchy.

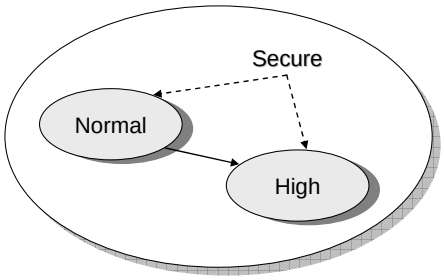


Figure 6.5: Example of applying context hierarchy.

NoSecure, Secure, Normal, High $\in$ CS
NoSecure $\in$ Csibling(Secure)
Normal $\leq$ Secure, High $\leq$ Secure
L1 = {Normal: :High}
L2 = {High: :NoSecure}
SecurityInfo1^{L1}, SecurityInfo2^{L2}

In this example there define two labels. One specifies that the labeled objects can only be exposed to the contexts *Normal* and *High*. It is assigned to the object *SecurityInfo*1. Another specifies that the labeled objects can only be exposed to the contexts *High* and *NoSecure*. It is assigned to the object *SecurityInfo*2. Without considering the effects of context hierarchy, both labels are valid. After considering the effects of context hierarchy, the label *L*2 is invalid, because *NoSecure* is a sibling of *Secure* and is invisible to any child context of *Secure*. Thus, the parent context isolates all its child contexts from the outside world.

There are two major benefits of using context hierarchies. One is that they can group the related contexts into multiple levels, thus we do not need to explicitly enumerate these contexts in label policies. Another is that they can provide perfect ways for isolating unrelated information. It is important for protecting sensitive information.

6.6 Applications

LBACP could be used for both access control and policy management. In this section we show some possible applications of LBACP through examples.

6.6.1 Enhance role-based access control

With the help of LBACP, some issues of RBAC mentioned in Section 6.1 could be addressed. In this subsection, we give four label policy application examples in which label policies are assigned to *users*, *roles* and *resources*, respectively.

Labels assigned to users

At a university, the enrollment office is responsible for dealing with application letters from all over the world. The employees who have *Staff* role are authorized to read these application letters. But for a given employee, he/she is only responsible for dealing with the letters from some specified countries. In this example, countries are modeled as contexts. Labels are designed to specify information flow constraints related to these contexts, and employees are associated with these labels according to their responsibility. The labels and their assignments are shown in Figure 6.6.

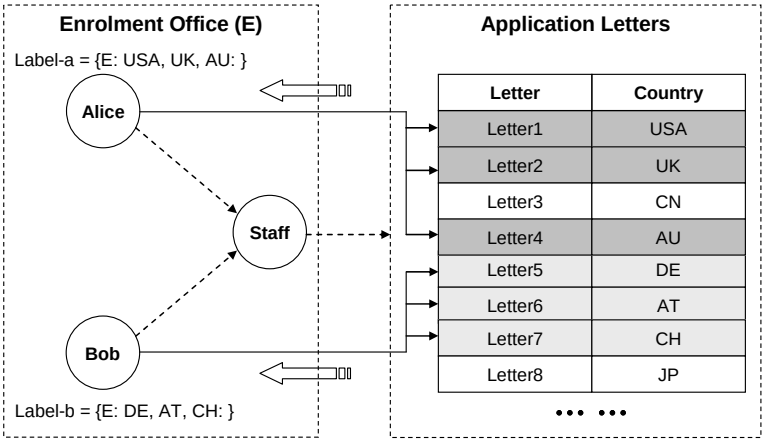


Figure 6.6: Example of labels assigned to users.

In this example, without considering information flow constraints, *Alice* and *Bob* are authorized to read all *Letters*. After considering information flow constraints, *Alice* can only read the letters belonging to the contexts USA, UK and AU, and *Bob* can only read the letters belonging to the contexts DE, AT and CH. This example shows that LBACP can be used to provide fine-grained access control in RBAC systems.

Labels assigned to roles

In a bank, for the reason of separation of duty, an employee who is authorized to be a *teller* may not be allowed to be an *auditor* in the same bank branch. But it is possible that an employee is a teller in one branch and an auditor in another branch. If RBAC is used, then many auditor roles have to be defined for mapping the users from some branches to the audit objects belonging to other branches. This certainly increases the administrative burden. For simplicity, we assume that there are only three bank branches involved, and they are *A*, *B* and *C*. We also assume that auditors of branch *A* can access audit objects of branch *B*, auditors of branch *B* can access audit objects of branch *C*, and auditors of branch *C* can access audit objects of branch *A*. If pure RBAC is used, then three auditor roles have to be defined.

The LBACP approach is shown in Figure 6.7. The *Auditor* role has the privileges of accessing all audit objects. There are four contexts are involved in this example, they are contexts *O*, *A*, *B* and *C* represent bank, bank branch *A*, bank branch *B* and bank branch *C*, respectively. As described previously, users who have *Auditor* role in branch *A* are permitted to access the audit objects of branch *B*. From the view of information flow, information is permitted to flow from *B* to *A*. This information flow is expressed by the label policy {*O*: *B*: *A*}. Similarly, we can define other two label policies {*O*: *C*: *B*} and {*O*: *A*: *C*}. These label policies are organized into labels *L*1, *L*2 and *L*3, respectively. These labels are combined with label separation operators and assigned to the role *Auditor*. At runtime, according to a user's context,

the system decides which contexts' audit objects are available to him/her. From the view of administration, the LBACP approach is much easier then the pure RBAC approach, in which three auditor roles should be defined. Defining too many roles will bring administrative problems. This example shows how the LBACP could be used to reduce the number of roles needed in a RBAC system.

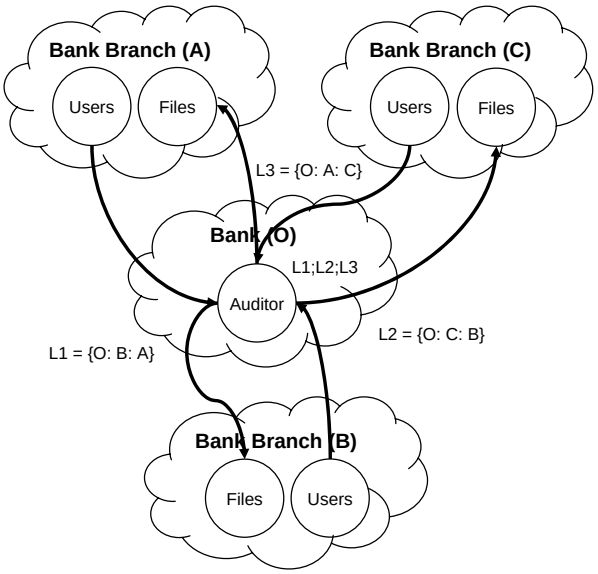


Figure 6.7: Example of labels assigned to roles.

Labels assigned to resources

In a hospital clinical staffs who are in various roles, such as *Doctor* and *Nurse*, are organized into different care teams. Team members can access the medical records of patients who are assigned to the team. From the view of RBAC, all the users who have role *Doctor* are permitted to access all patients' medical records. But from the view of information security, a clinical staff can only access the medical records of patients whom he/she is caring for. Thus, pure RBAC cannot deal with this situation. Next we show how the LBACP can be used to resolve this problem.

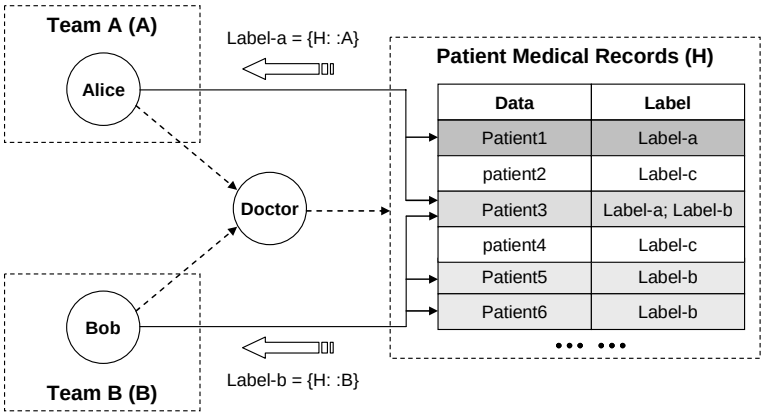


Figure 6.8: Example 1 of labels assigned to resources.

As shown in Figure 6.8, the care teams are modeled as contexts, and patients' medical records are associated with labels that specify to which contexts these records are permitted to expose. The medical records of *Patient*1 and *Patient*3 are labeled with *label-a* that specifies information can flow from *H* to *A*, so they are available to the users of *team A*. The medical records of *Patient*3, *Patient*4 and *Patient*5 are labeled with *label-b* that specifies information can flow from *H* to *B*, so they are available to the users of *team B*. After considering the effects of RBAC policies and LBACP policies, *Alice* can access the medical records of *Patient*1 and *Patient*3; Bob can access the medical records of *Patient*3, *Patient*5 and *Patient*6. This example shows that RBAC and LBACP can work together to provide an abstraction to capture a set of collaborative users operating in different roles on individual object instances.

Now we consider another example that labels are assigned to the resources. In an IT company there are two departments. One is *Development department* in which the employees can access the projects that are in developing phase. The other is *Test department* in which the employees can access the projects that are in testing phase. If pure RBAC is used, then two roles need to be defined for accessing developing and testing projects respectively. If one project needs to be tested, then the administrator has to do two operations. One is withdrawing this project from the *developing* role; the other is assigning this project to the *testing* role. Next we investigate how the LBACP can be used to simplify the privilege management in this scenario.

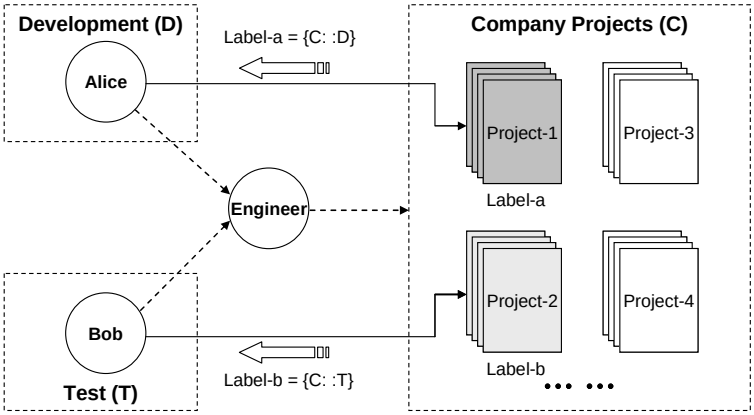


Figure 6.9: Example 2 of labels assigned to resources.

The LBACP solution is shown in Figure 6.9. Here we only need to define one role *Engineer* that has the privilege of accessing all projects, including the developing projects and testing projects. In both departments the employees are assigned to the role *Engineer*. The company and its departments are modeled as different contexts. In order to implement the access control requirements described above, two labels are defined. The *label-a* permits that information can flow from the resource context *C* to the development context *D*, and the *label-b* permits that information can flow from the resource context *C* to the test context *T*. The developing projects are associated with *label-a*, and the testing projects are associated with *label-b*. After considering the effects of RBAC policies and LBACP policies, *Alice* who is in *Development department* can access *Project*-1 and Bob who is in *Test department* can access *Project*-2. When *Project*1 is finished and needs to be tested, it is then relabeled with *label-b* so that the employees of test department can access it. If it is necessary, one testing project can also be relabeled back to the development department. The re-labeling process is obviously easier than changing the access control policies. This is another example that the LBACP can be used to simplify privilege management.

6.6.2 Data separation

Information could be classified into different categories according to their sensitive levels. Through enforcing information flow constraints to these categories, some privacy control could be achieved. Considering a web application in which all the pages are organized into *NormalWeb* and *SecureWeb* two categories (contexts). Three roles are defined. They are *Clerk*, *Manager* and *Auditor*. Pages are assigned to these roles according to their job functions. The permission-role assignments and label-role assignments are shown in Figure 6.10.

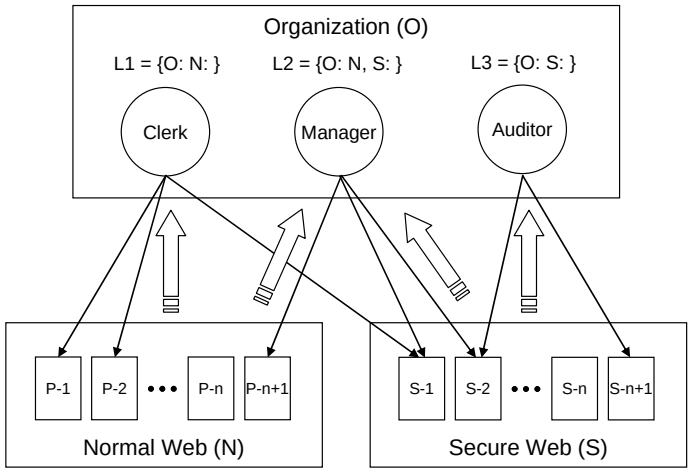


Figure 6.10: Example of label policies used for data separation.

In this example, from the view of information flow constraints, *Clerk* can access the pages belonging to *NormalWeb*, *Manager* can access the pages belonging to *NormalWeb* and *SecureWeb*, and *Auditor* can access the pages belonging to *SecureWeb*. *Clerk* is assigned to pages *P*-1, *P*-2 and *S*-1. Without considering the effects of information flow constraints, *Clerk* can access all three pages. After considering the effects of information flow constraints, *S*-1 is blocked by the system. This is an example that LBACP is used for avoiding information leaks caused by improper policy definition.

6.6.3 Policy component grouping

A label policy without import and export contexts means that no information can flow to/from the owner context. This kind of label policies can be used to group policy components for both access control and policy management. Policy components could be associated with multiple labels combined with label separation operators. It allows administrators to group policy components along various dimensions (application aspects).

Consider a hospital scenario, there are four departments in a hospital, they are *Administration*, *Medicine*, *Surgery* and *Laboratory*. These departments have different views to the patient medical information. With the help of label polices, the administrator can get clear views about how the patient medical information are shared among these departments. Here four labels are defined for these departments respectively; they are L_{admin} = {*Admin*::}, L_{med} = {*Med*::}, L_{surg} = {*Surg*::} and L_{lab} = {*Lab*::}. These label policies only permit information flowing to themselves. According to the usage, patient medical information is divided into *Admission*, *Billing*, *Examination*, *Diagnosis* and *Treatment* five parts. The label assignments to these data segments are shown as follows:

MedicalRecord = {Admission, Billing^{Ladmin}, Examination$^{Lmed;Lsurg;Llab}$, Diagnosis$^{Lmed;Lsurg}$, Treatment$^{Lmed;Lsurg;Llab}$}

The *Admission* data holds patients' general information, such as name, age, sex and so on. The *Admission* data is not associated with any label means that it is available to all the departments. The *Billing* data are only available to the *Administration* department. For the purposes of test and therapy, the *Laboratory* department can access *Examination* data and *Treatment* data. Except the *Billing data*, *Medicine* department and *Surgery* department can access all other kinds of data. Through the label policies assigned to the different parts of patient medical record, the administrators can get intuitive views about what kinds of patient information have been shared with which departments.

Label policies could be used for segregating unrelated policy components. At runtime only the rules associated with a given label are used for making some particular access control decisions. In the above example, we define L_{med}, L_{surg} and L_{lab} three labels for *Medicine*, *Surgery* and *Laboratory* three departments, respectively. *P* is an authorization policy that contains five authorization rules. The label assignments are shown as follows:

P = {rule1$^{Lsurg;Llab}$, rule2$^{Lmed;Lsurg}$, rule3$^{Lmed;Llab}$, rule4$^{Lmed;Lsurg;Llab}$, rule5}

There is no label assigned to *rule*5 means that rule5 is applicable to all departments. In this example, it is similar to *rule*4 that is also applicable to all three departments. After considering the effects of the labels, we can get three rule sets that specially used for the three departments. We use P_{med}, P_{surg} and P_{lab} to represent them, respectively. These rule sets are shown as follows:

P_{med} = {rule2, rule3, rule4, rule5}
P_{surg} = {rule1, rule2, rule4, rule5}
P_{rad} = {rule1, rule3, rule4, rule5}

6.7 Implementation

The label policies are not used independently. They must be combined with other authorization systems. So the implementation of LBACP is tightly connected with the host authorization systems to which the LBACP will add information flow constraints. Many large application systems use databases as their data repositories, and their authorization systems are not policy-based, the authorization information are stored in different tables or files and the authorization logics are implemented in the programs. Applying LBACP in such kinds of systems strongly depends on how these application systems are designed. For example, in which table one column should be added for storing the labels. Applying LBACP in such kinds of application systems needs to redesign the authorization systems. It nevertheless gives the possibility of freely adopting various label assignments described in Section 6.4.

However, many authorization systems are policy-based. It is not easy to freely assign labels to their policy components without dramatically changing their original architecture, but it is easy to directly add information flow constraints to these policies. We have developed a label-based access control system for such kind of application environment. The major components of the system are described as follows.

- *Label policies* are written in XML. They specify the contexts, context hierarchies, labels, users, roles, objects and the label assignments to the users, roles and objects.

- *Label policy evaluator* is responsible for checking if information can flow from the object context to the subject context against to a given label policy.
- *Label policy editor* is a GUI-based tool used to create and maintain the label policies. Its screen snapshot is shown in Figure 6.11.

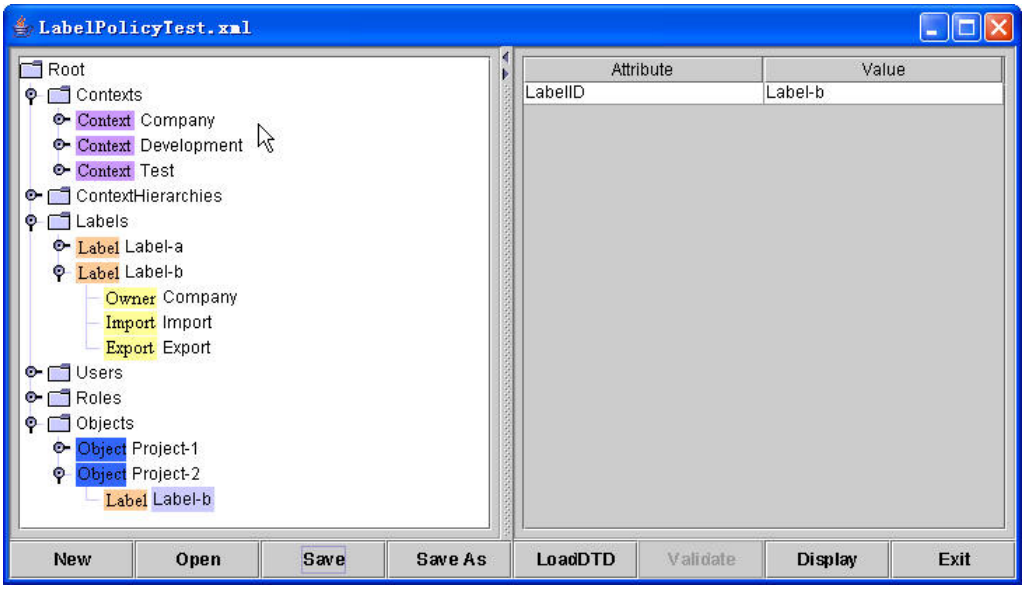


Figure 6.11: Label policy editor.

The label policies are not used independently. They should be combined with normal policies in some ways. We have developed a policy combination system named *root policy* system that is used to combine multiple heterogeneous authorization policies. The root policy system will be introduced in Chapter 7. The root policy system is not specially developed for combining the label policies and normal policies. But it can be used for this purpose very well. The evaluation process is: firstly, a request is evaluated by a normal authorization policy; if this request is permitted, then it is further checked by the label policy that is bound to the normal policy; only when both checks return the value "Permit", the root policy system returns the value "Permit".

The following is a segment of a label policy. It specifies a label and an object that is associated with the label.

```
<Labels>
  <Label LabelID="Label-b">
    <OwnerContext>
      <ContextReference ContextID="Company" />
    </OwnerContext>
    <ImportContexts />
    <ExportContexts>
      <ContextReference ContextID="Test" />
    </ExportContexts>
  </Label>
</Labels>
<Objects>
  <Object ObjectID="Project-2">
    <Attributes>
      <Attribute AttributeID="ObjectID" DataType="String" Issuer="hpi.uni-potsdam.de">
```

```
      <AttributeValue>Project-2</AttributeValue>
    </Attribute>
  </Attributes>
  <AssociatedLabels>
   <LabelReference LabelID="Label-b" />
  </AssociatedLabels>
 </Object>
</Objects>
```

6.8 Related work

Multi-Level Security (MLS) [BL76] is a type of mandatory access control system. MLS is developed in the military as a means to manage classified information. In MLS every target is given a *security label*, which includes a classification, and every subject is given a *clearance*, which includes a classification list. The classification list specifies which type of classified target the subject is allowed to access. A typical classification scheme used by the military is: *unclassified*, *restricted*, *confidential*, *secret*, and *top secret*. As MLS was developed with a military environment in mind, confidentiality was a major motivating issue, which was achieved by information flow restrictions among users and resources. A typical security policy, designed to stop information leakage, is "read down and write up". This specifies that a subject can read targets with a lower classification than his clearance, and can write to targets with a higher classification. However, MLS only expresses the privacy concerns in a single context. On the contrary, LBACP express privacy concerns among multiple contexts.

Role Templates (*RT*) [GI97] is proposed by Giuri and Iglio in their content-based access control model. They define *parameterized roles* through the use of role templates. This work is based on extending privileges to include parameterized constraints that are evaluated against the content of the requesting subject and the requested object at access time. A role is regarded as a set of privileges, so the parameters of a role template correspond to those of its privileges. For example, instead of defining an own doctor role for each division within a hospital, a generic parameterized role may be defined. Users are assigned to a specific division-role, and the name of the division is used as a parameter for the operation assignments. Hence, according to the concrete parameter value, a user may only access patient records that are allocated to his/her division. In RT a regular role can be considered a special case of role template without parameters.

There are some differences between RT and LBACP. The RT provides fine-grained access control through defining multiple similar roles that are bound with different constraints. On the contrary, the LBACP tries to avoid this situation. For example, in above example the parameterized role may be defined as Doctor<division>, where the *division* is the parameter representing the division. The role instances, such as Doctor<Cardiovascular>, are assigned to the users. In LBACP there is only one role defined, and the system decides who can do what according to the information flow constraints that could be associated with users, roles, permissions or resources in the context of RBAC. Both approaches can be used to realize the authorization requirements shown in Figure 6.8. In RT approach we need to define the parameterized roles as many as the number of teams, and assign these roles to the team members. In LBACP approach there is only one role defined. To this kind of scenarios, we think LBACP can provide more clear administration view than RT approach. On the other hand, the constraints bound to the parameterized roles can also contain context-related information, such as time. These kinds of constraints are out of the scope that LBACP tries to address.

Decentralized Label Model (*DLM*) [ML98, ML00] is a model for control of information flow in systems with mutual distrust and decentralized authority. This model defines two kinds of label policies. One is confidentiality policy that allows the owner of the policy to specify which principals are permitted to read a given piece of information. Another is the integrity policy that allows the owner to specify which principals are permitted to affect the value of a given piece of information. The model improves on existing MLS by allowing users to declassify information in a decentralized way, and by improving support for fine-grained data sharing. These label policies could be associated to the values used in programs. The DLM permits programs using it to be checked statically to avoid information leaks, in a manner similar to type checking.

Even our work is closely related to the DLM, there are some crucial differences. The first difference is the information flow definition. In DLM the information flows are defined according to the read or write operations. The confidentiality (reader) policies specify where the information can flow to, and the integrity (writer) policies specify where the information can flow in. In LBACP, the information flows are not defined according to the read or write operations, the information direction is always from objects to subjects. For example, a user can read or write an object only when information can flow from the object to the user. The second difference is that the DLM is a self contained security system, whereas the LBACP must be combined with other authorization systems. The third difference is their labeled targets. The DLM labels are directly attached to the data manipulated by the computing system. The LBACP labels are assigned to the access control policy components in order to restrict their usage.

OASIS RBAC Meta-Policy (*ORMP*) [BEM03, BME04] is an approach for subdividing the administration of large-scale security environments and for enforcing information flow restrictions over policies. This approach introduces the concept of *contexts* to group and classify policy components according to various aspects. These contexts are applied to control information flows between system entities. With the help of information flow relation administrators can restrict the use of policy components alongside components belonging to certain other groups, and organize access control policies into a hierarchical, multidimensional structure.

There are two major differences between ORMP and LBACP. In ORMP there is no real label structure. Since it simply assigns the context names to the policy components, these direct assignments make it difficult to provide flexible policy management in multiple dimensions as they hope. Whereas in LBACP there is a clear label structure that specifies the owner, import and export contexts, and one component can be associated with multiple labels that can be combined in different ways. Thus, LBACP provides a really flexible approach for policy management. The second difference is that the ORMP is not directly used for access control but used for describing and restricting policies at policy specification time. On the contrary, the LBACP tightly binds to normal access control policies. They will be checked after normal access control policy evaluation.

Oracle Label Security (OLS) [Oracle] is an add-on security option for the Oracle Enterprise Edition. This product enables administrators to add label based access control to the access mediation process. It mediates access to rows in database tables based on a label contained in the row, a label associated with each database session, and OLS privileges assigned to the session. Labels are assigned to both data row in a database table and users. A *data row label* indicates the level and nature of the row's sensitivity and species the additional criteria that a user must meet to gain access to that row. A *user label* specifies that user's sensitivity level plus any compartments and groups that constrain the user's access to labeled data. Each user is

assigned a range of levels, compartments, and groups, and each session can operate within that authorized range to access labeled data within that range.

OLS are used to provide fine-grained access control within Oracle databases. Our LBACP are used to provide information flow constraints among access control entities that are not limited to users and database table rows. Even their label structures and design purposes are different; some functionality of OLS could be implemented by LBACP. The functionality supported by OLS and the alternative LBACP solutions are described as follows.

1. OLS can be configured to keep data from different organizations separate within a single database instance, so that organizations can share database tables but only see data that pertains to them. In LBACP the label policies that only permit information flowing to itself can be used for this purpose. For example, if one data is labeled with {*o*: : }, then only the users from the organization *o* can access it.
2. OLS is particularly useful for hosting environments in which access to information can be formalized by means of sensitive levels, access categories, or user groups. In LBACP the information belonging to different sensitivity levels, access categories or user groups can be organized into different contexts in order to define some labels that specify information flow constraints among these contexts.
3. OLS is also ideal for enforcing privacy concerns. With OLS, data can be labeled to express where the data should be released. Here the labels are used to restrict information flow directions of the labeled data. This functionality is well supported by LBACP through defining the labels' export contexts. From the point of information flow control, LBACP can provide more flexibility than OLS.

6.9 Conclusion

With LBACP we can get four major benefits. (1) Information flow related control information can be detached from the normal access control policies. Thus, some dynamic configuration could be done through redefining the information flow constraints specified by label policies rather than modifying the normal access control policies. This is less error-prone, and caters the requirements of modern dynamic collaboration activities, such as dynamically changing participant relationships. It also simplifies the design of normal access control policies. (2) By applying some information flow constraints to the normal access control policy components in order to restrict their usages, some information leaks caused by these components could be avoided or reduced. (3) LBACP can also be used to improve access control policy management. Through assigning multiple labels to policy components, these components can be managed and enforced in multiple dimensions. (4) LBACP is a high level security policy layered on the top of normal access control policies. The layer structure makes it easy to integrate the LBACP with existing access control systems.

Chapter 7

Root Policy

7.1 Introduction

Authorization is the process of ascertaining that an entity with a particular identity or set of attributes has the permission to perform a particular action on a particular resource. This step is preceded by authentication where the identity of the entity is established. Authorization policies play an important role in this era where computing resources of organizations with diverse privacy protection and information sharing requirements are increasingly connected together to carry out joint or common tasks. Authorization policies are enforced through mechanisms consisting of authorization functions and authorization data that together map a user's authorization request to a decision whether to grant or deny access [FGHK05].

For information protection, various authorization systems and mechanisms have been developed. For example, in response to the need to protect classified information, there are mechanisms to enforce Multi-Level Security (MLS) [BL76] policies, and in recognition of the needs of industry, there are mechanisms to enforce Role-based Access Control (RBAC) [SCFY96] policies. Recently, collaborative systems are becoming popular. For the purpose of authorization in collaborative systems, some new authorization systems and mechanisms have been proposed or developed, such as Akenti Authorization Service [TEM03], Cardea [Lep03], Community Authorization Service (CAS) [PKWF03], PRIMA [LAKK03], Permis Authorization Infrastructure [CO03] and Virtual Organization Membership Services (VOMS) [ACCA05]. Thus, authorization systems come in a wide variety of forms, each with their individual (and often proprietary) attributes, functions, and methods for configuring policy, and a tight coupling to a class of policies.

Unfortunately, in many application scenarios one size does not fit all, especially in the collaborative environments such as grid computing systems. A grid system is a virtual organization that is composed of several autonomous domains. Authorization in such a system needs to be flexible and scalable to support multiple authorization mechanisms or security policies. A further example is provided by laws concerning privacy issues. In a modern information system, the security policy of the organization should combine internally specified constraints with externally imposed privacy regulations [BD99].

Even there is only one authorization mechanism being used, it also possible to require an authorization system to combine multiple policies to complete some complex decision tasks. Consider now a large organization composed of different divisions, each of which can

independently specify security policies; the global policy of the organization results from the combination of all these components. Finally, as security policies become more sophisticated, it may be desirable to formulate the policy incrementally by assembling small, manageable, and independently conceived modules [BVS02].

Some authorization policy languages already provide comprehensive functionality to combine multiple policies, such as eXtensible Access Control Markup Language (XACML) [XACML]. Some other authorization policy languages do not support multiple policies, such as the PERMIS X.500 PMI RBAC Policy [CO02]. It is better to have a mechanism that can flexibly specify and combine these authorization policies. The existing approaches are either too simple to support complex policy combination or tightly integrated into an operation system.

To our knowledge there is still no dedicated work that comprehensively supports heterogeneous authorization policies' trust management, combination and enforcement in distributed authorization environments. Motivated by this requirement, we developed a policy language to specify multiple heterogeneous policies' combination and a mechanism to enforce these policies [ZM07a]. This policy specification language is called *root policy specification language*. A policy written in root policy specification language is called *root policy*. In this chapter we introduce the root policy model and enforcement mechanism.

The rest of this chapter is organized as follows. Section 7.2 introduces the major related work. Section 7.3 presents the root policy authorization model. Section 7.4 describes the root policy language model. Section 7.5 introduces the root policy evaluation rules. Section 7.6 introduces the root policy component combining algorithms. Section 7.7 looks at the root policy enforcement mechanism. Finally, Section 7.8 gives the conclusions.

7.2 Related work

Recent years there is considerable work on access control models and languages. Many approaches have been proposed to increase expressiveness and flexibility of authorization languages by supporting multiple policies within a single framework [Hos92, WL93, BJS99, LFG99, JSSS01]. These proposals, while based on powerful languages able to express different policies, assume a single monolithic specification of the entire policy. Such an assumption does not fit many real-world situations, where access control might need to combine independently stated restrictions that should be enforced as one.

Since different organizations operate under different requirements for protection and control of their data, inevitably their security mechanisms do not share common principles, are not implemented in similar languages, and do not run on compatible operating platforms. This situation is recognized by both [BVS02, WJ03]. They propose algebras for combining security policies with formal semantics. Complex policies are formulated as expressions of the algebras. These frameworks provide descriptions of policies that are language and implementation mechanism independent. Such descriptions can be examined for completeness, consistency, and unambiguity. Environment related policy composition also be considered by [SCH03].

NIST has initiated a project in pursuit of a standardized access control mechanism, referred to as the Policy Machine (PM) that requires changes only in its configuration in the enforcement of arbitrary and organization specific attribute-based access control policies [FGHK05]. Included among the PM's enforceable policies are combinations of policy instances (e.g., RBAC and MLS). The core features of the PM are capable of configuring, combining and enforcing arbitrary attribute-based policies. PM categorizes users and objects and their attributes into policy classes, and transparently enforces these policies through a series of fixed PM

functions, that are invoked in response to subject (process) access requests. However, this approach cannot be used to combine the various existing authorization policies without dramatically changing their enforcement mechanisms.

XACML is a very rich and flexible language. Security administrators can directly represent a large variety of authorization policies in XACML. XACML is becoming popular these years. More and more systems adopt XACML as their authorization language. However, XACML has not been built to manage security in large distributed systems in which virtual enterprises are dynamically built with the collaboration of multiple independent subjects sharing their resources to provide new services to customers [MBCS06].

The LINUX Rule Set Based Access Control (RSBAC) system is an open source security extension to current Linux kernels [RSBAC]. RSBAC is a framework that comes with several fully functional access control modules, such as MAC, ACL and RBAC. Users just have to choose which security modules suit their needs best. They can also add new access control model in RSBAC. However, the RSBAC is tightly integrated into the Linux operating system. It means that many application systems cannot use RSBAC for access control. For example, RSBAC is not suitable for providing access control in database environments. On the other hand, the RSBAC does not support distributed authorization policy enforcement and complex policy combination. Once a security module is turned on, then all the authorization requests will be sent to this module.

Multipolicy Authorization Framework for Grid Security by Lang et al [LFSA06] is the most relevant work to our research. Basing on the special security needs of the Grid computing, they constructed an authorization framework in the Globus Toolkit 4 [Globus] that can support multiple authorization policies. For each authorization policy, the framework constructs a Policy Decision Point (PDP). There is a Master PDP that is responsible for coordinating the PDPs to render a final decision. The PDPs managed by the Master PDP are specified in the security configuration file. However their approach does not touch some important issues, such as policy storage and management, interactions among PDPs and the collaboration of multiple Master PDPs. This approach does not support complex policy combinations. It simply adds policy evaluators' class names into the security configuration file. So all PDPs are valid for all the decision requests and the policy combining algorithm is either deny-override or permit-override.

To address these problems, we developed the root policy specification language that is used to specify multiple heterogeneous authorization policies' combination and a mechanism that is used to enforce these root policies. In a root policy, each involved authorization policy's storage, trust management and enforcement can be defined independently. Various policy combinations and flexible context constraints enable the root policy to deal with complex application scenarios. On the other hand multiple root policies can cooperate together to complete more complex authorization tasks in distributed environments.

7.3 Root policy authorization model

A root policy authorization system consists of a root policy and a root policy evaluator. The root policy evaluator reads a root policy in, evaluates input authorization requests against the root policy, and then renders the finial authorization decisions. In this section we give an overview of the root policy authorization model.

7.3.1 Root policy data flow model

The root policy data flow model as shown in Figure 7.1 is the same as the XACML data flow

model. This data flow model mainly contains Policy Enforcement Point (PEP), Policy Decision Point (PDP), Context Handler (CH), Policy Information Point (PIP) and Policy Administration Point (PAP). The PEP performs access control by making decision requests and enforcing authorization decisions. The PDP makes access decisions according to the security policy written by the PAP. The PEP and PDP are separated by the CH through which an application system can communicate with a XACML evaluator or other policy evaluators, such as our root policy evaluator. This is the major reason that we adopt this data flow model.

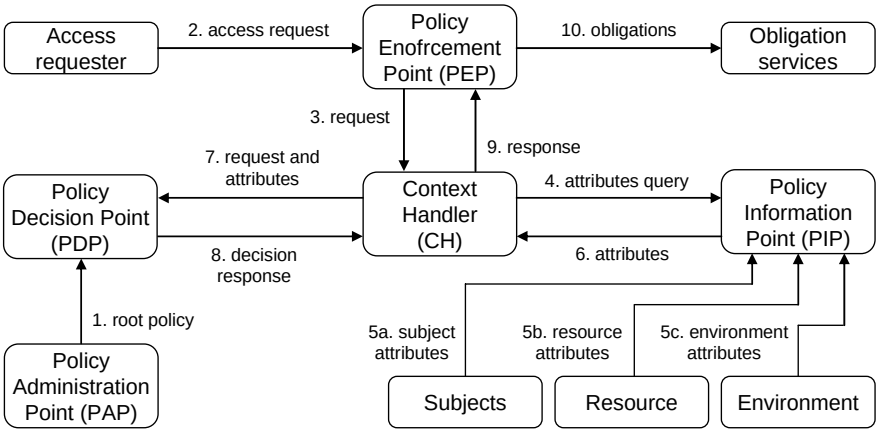


Figure 7.1: Root policy data flow model.

The PIP collects information about the request subject, resource and environment. It is useful to separate this collection process into its own module so that different authorization algorithms and policies can be configured with the same collection process. Examples of PIPs are the VOMS [ACCA05] PIP that parses a caller's VOMS credential for attributes, the Shibboleth [Shibboleth] PIP that retrieves SAML [SAML] attributes; and the Permis [CO03] PIP that parses a caller's X.509 attribute certificates [AC02] for user roles. But the root policy PDP does not concern PIPs. It is the responsibility of CH to decide which kinds of information should be extracted from the PIPs.

The CH constructs root policy canonical requests based on the requests sent by the PEP and the additional attributes obtained from the PIPs, and then presents them to the PDP. The PDP evaluates the root policy and renders the authorization decisions. The CH then converts the authorization decisions to the native response format supported by the PEP.

7.3.2 Root policy authorization framework

The root policy security framework is shown in Figure 7.2. The authorization policies managed by a root policy are organized in one or multiple policy schemas. Policy schema groups a set of authorization policies according to their application scope. Various policy combinations can be defined among these policies. Authorization requests are evaluated by policy schemas. If there are multiple policy schemas involved in making access control decisions for a given request, the decisions returned by these schemas should to be combined in order to render a final decision. The root policy evaluation process is described as follows.

1. A client accesses the service protected by a PEP. The PEP intercepts the request and sends a decision request to the context handler.
2. The context handler encapsulates the decision request and additional information about the

subject, object and environment collected from the PIPs into a root policy request, and then presents it to a root policy PDP.

3. The root policy request is handled by the PDP coordinator. According to the content of the request and the application scope of each policy schema in a root policy, the PDP coordinator invokes all the applicable policy PDPs with this request.
4. Each policy PDP evaluates to an independent decision.
5. The PDP coordinator receives the decisions returned by all the involved policy PDPs, and then combines these decisions according to the policy combination logic specified by various root policy component combining algorithms, such as policy combining algorithms and policy schema combining algorithms, for the finial decision.
6. The PDP coordinator returns the finial decision to the invoker, i.e. the context handler, through a root police response.
7. The context handler returns the decision to the PEP in the format used by the PEP.
8. The PEP then executes the decision, either denying or permitting the access request.

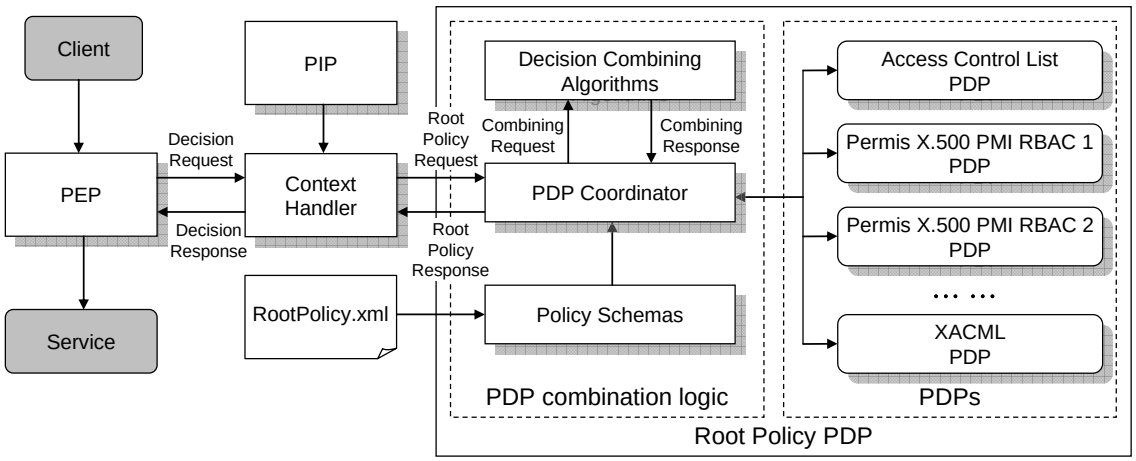


Figure 7.2: Root policy authorization framework.

7.3.3 Root policy trust management

In our root policy security framework, authorization policies can be stored and enforced in distributed way. In order to ensure that the genuine policies are used, Privilege Management Infrastructure (PMI) is adopted. PMI was specified by the ITU-T and ISO/IEC [ITUT01]. The main function of PMI is providing a strong authorization after the authentication has taken place. It has a number of similarities with PKI [HFPS99]. The basic data structure in a PMI is a X.509 Attribute Certificate (AC) [FH02]. Like Public Key Certificate (PKC) strongly binds a public key to its subject, AC strongly binds a set of attributes to its holder. In fact, attribute certificates have been designed to be used in conjunction with identity certificates, i.e. PMI and PKI infrastructures are linked by the information contained in the attribute certificates and public key certificates. For example the holder field in an AC contains the serial number and issuer of a PKC. The identity certificate, attribute certificate and their relation are shown in Figure 7.3.

In a PKI, the entity that digitally signs a PKC is called a Certification Authority (CA). The trusted root of a PKI is called root CA or trust anchor. A CA may have subordinate CAs which

it trust, and to which it delegate the powers of authentication and certification. Subordinate CAs may also delegate their powers of authentication to further subordinate CAs. Then a CA hierarchy can be established. When a user needs to have his signing key revoked, a CA will issue a Certificate Revocation List (CRL) containing the list of PKCs not to be trusted. In a PMI, the entity that digitally signs an AC is called an Attribute Authority (AA). The trusted root of a PMI is called Source of Authority (SOA). A SOA may delegate its powers of authorization to subordinate AAs. Subordinate AAs may also delegate their powers of authorization to further subordinate AAs. Then an AA hierarchy can be established. When a user's authorization permissions need to be revoked, an AA will issue an Attribute Certificate Revocation List (ACRL) containing the list of ACs not to be trusted.

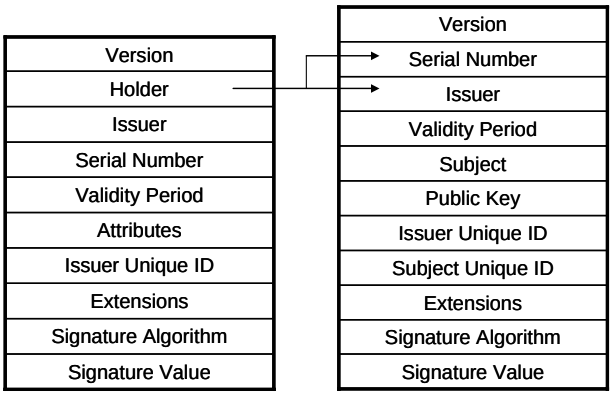


Figure 7.3: Relation between identity and attribute certificates.

There are two primary models for distribution of ACs: the "push" and "pull" models. In the "push" model, the client supplies his AC to a server. The "push" model is suitable when the client's rights should be assigned within the client's home domain. In the "pull" model, the server retrieves the client's AC from an AC repository. The "pull" model is suitable when the client's rights should be assigned within the server's domain. The choice to use the "push" or "pull" model is dependent on system requirements and the available infrastructure. More information about PKI and PMI may be obtained from reference [HFPS99, FH02].

ACs may be used with various security services, including access control, data origin authentication and non-repudiation. Since ACs are digitally signed by the SOA who issued them, they are tamper-resistant, and therefore there is no modification risk from allowing them to be stored in a publicly accessible repository. This also means that authorities who issue digital ACs can store them locally, but give global access to them.

In a root policy system ACs are used to store the authorization policies managed by the root policy. These ACs are pulled into the system. At runtime the root policy evaluator tries to get each authorization policy's AC and PKC, and verify if a policy AC is issued by a valid AA. With the help of PMI, a root policy can guarantee all its authorization policies' non-repudiation and integrity even they are stored around the world.

7.4 Root policy language model

The root policy language model is shown in Figure 7.4. The main components of the model are *subject domain*, *resource domain*, *context constraint*, *policy*, *policy hierarchy*, *policy subschema* and *policy schema*. They are introduced in the following subsections.

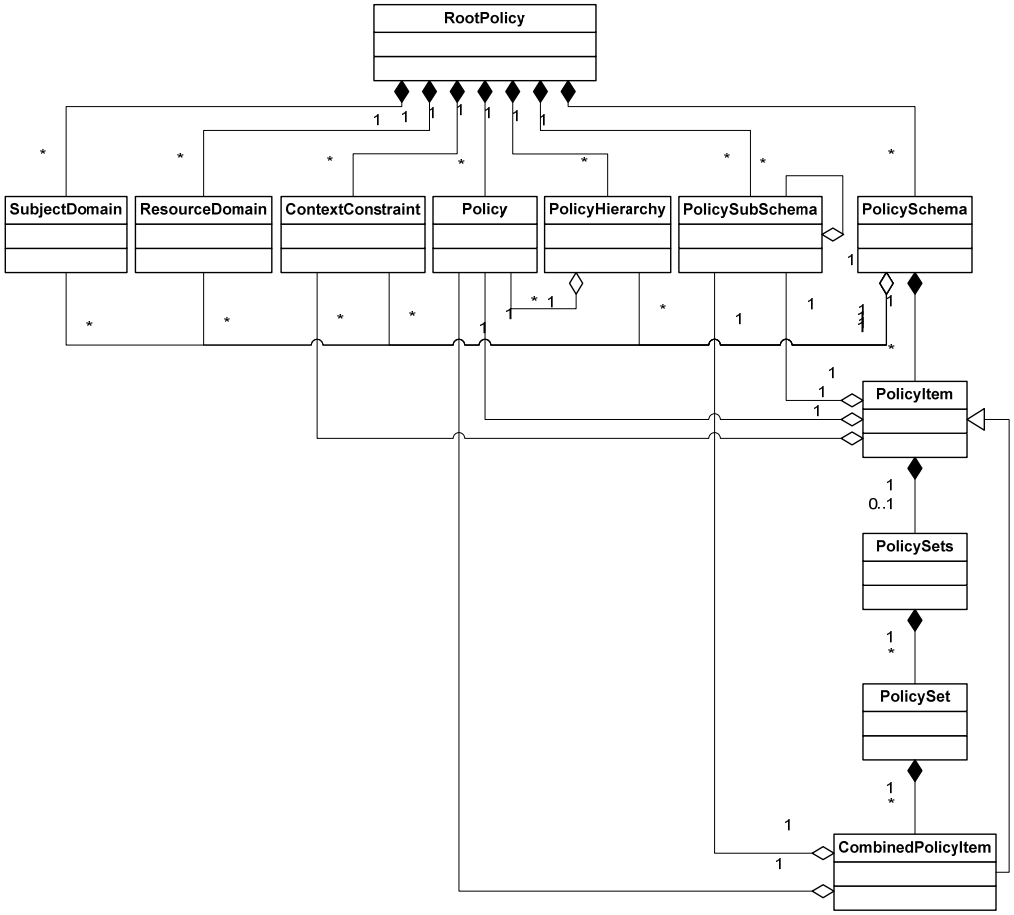


Figure 7.4: Root policy language model.

7.4.1 Subject domain

SubjectDomain specifies the domain of users who may be granted within the overall root policy. Each domain is specified as a collection of LDAP sub-trees, using Include and Exclude statements. The Include statement specifies the LDAP DN of the root node of a subject domain, and the Exclude statement specifies which subordinate sub-trees to be excluded from the domain. Using a null in an Include statement specifies the domain of all users in the world.

An example of directory information tree is shown in Figure 7.5. The example of subject domain specification below specifies the employees of company *ABC*, excluding the Germany branch.

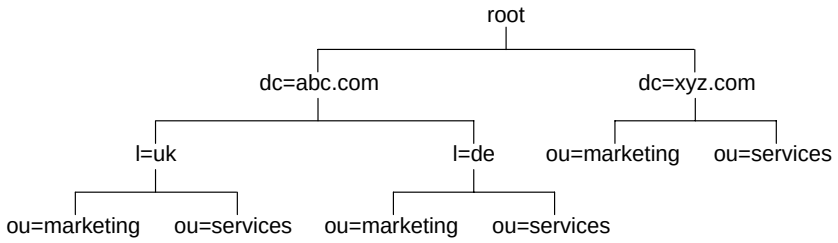


Figure 7.5: Example of directory information tree.

```
<SubjectDomains>
  <SubjectDomain SubjectDomainID=" CompanyABCEmployees ">
    <Include>
      <LDAPDN>dc=abc.com</LDAPDN>
    </Include>
    <Exclude>
      <LDAPDN>l=de, dc=abc.com</LDAPDN>
    </Exclude>
  </SubjectDomain>
</SubjectDomains>
```

7.4.2 Resource domain

ResourceDomain specifies the resource domain covered by a root policy. Resource domains are specified as LDAP DN sub-trees, using Include, Exclude and ObjectClasses statements. The Include statement specifies the LDAP DN of the root node of a domain, and the Exclude statement specifies subordinate sub-trees to be excluded from the domain. Using a null in an Include statement specifies the domain of all resources in the world. A domain may optionally be refined by specifying a set of object classes. An object class is a general description of an object as opposed to the description of a particular object. Only resources with the full set of object classes are included in the domain. A null set of object classes implies all resources in the domain are included.

The following example comprises a resource domain shown in Figure 7.5, which specifies all the customer information in the company *ABC*.

```
<ResourceDomains>
  <ResourceDomain ResourceDomainID="CompanyABCCustomers">
    <Include>
      <LDAPDN>ou=services, l=uk, dc=abc.com</LDAPDN>
    </Include>
    <ObjectClasses>
      <ObjectClass>CustomerInformation</ObjectClass>
    </ObjectClasses>
    </ResourceDomain>
</ResourceDomains>
```

7.4.3 Context constraint

ContextConstraint specifies that certain context attributes must meet certain conditions to permit a specific operation. Detail about the context constraint is described in Section 4.3. A conditional root policy component is granted for making access control decisions if and only if all the associated context constraints evaluate to “true”.

The following example specifies a context constraint that specifies the working days are from Monday to Friday.

```
<ContextConstraints>
  <ContextConstraint ContextConstraintID="WorkingTime">
    <ContextCondition ContextConditionID="DaysOfWeek">
      <Operator DataType="WeekDay">in</Operator>
      <LeftOperand>
        <Observer>LocalHostObserver</Observer>
        <Function>GetWeekDay</Function>
      </LeftOperand>
      <RightOperand>
        <Parameter>Monday;Tuesday;Wednesday;Thursday;Friday</Parameter>
      </RightOperand>
```

```
    </ContextCondition>
</ContextConstraints>
```

7.4.4 Policy

Policy specifies the authorization policy used in a root policy. The major items defined for specifying an authorization policy are described as follows.

- *OID* specifies the unique object identifier of an authorization policy.
- *ValidityPeriod* specifies the valid time of a policy. The actual validity time is the intersection of the policy validity time the attribute certificate validity time.
- *Critical* specifies how to deal with a policy after it is expired. If its value is "true", then all the requests related to this policy will be denied after this policy is expired, otherwise this policy should be ignored after it is expired.
- *Evaluator* specifies the program used to evaluate this policy.
- *ACURI* specifies the URI used to retrieve the policy AC.
- *ACRLURI* specifies the location of AC revocation list.
- *PKCURI* specifies the URI used to retrieve the PKC used to verify the policy AC.
- *CRLURI* specifies the location of PKC revocation list.
- *RealizedBy* specifies where an authorization request should be further sent when it needs to be evaluated by another root policy.

If a policy specification only contains the *PolicyID*, we call it *empty policy*. An empty policy always evaluates to "Permit". Empty policies are used to realize policy combination logic. For example, three policies are bound to an empty policy, and the relation among of them is conjunction. A policy can be evaluated at local site or a remote site that is indicated by the *RealizedBy*. The following example specifies two policies. One is evaluated at local site. Another is evaluated at a remote site.

```
<Policies>
  <Policy PolicyID="Permis_Policy">
    <OID>1.2.826.0.1.3344810.1.1.13.1</OID>
    <ValidityPeriod Start="2006-06-01T00:00:00" End="2007-08-31T23:59:59" />
    <Critical>TRUE</Critical>
    <Evaluator>PDP_PERMIS</Evaluator>
    <ACURI>ldap://localhost:389/cn=PermisPolicy1,ou=Security,dc=uni-trier.de</ACURI>
    <PKCURI>ldap://localhost:389/cn=aaRSA1024Trier,ou=Security,dc=uni-trier.de</PKCURI>
  </Policy>
  <Policy PolicyID="XACML_Policy">
    <OID>1.2.826.0.1.3344810.1.1.15.1</OID>
    <Critical>TRUE</Critical>
    <RealizedBy>http://remotehost:8080/root-policy/coordinator</RealizedBy>
  </Policy>
</Policies>
```

7.4.5 Policy hierarchy

PolicyHierarchy specifies the inheritance relations among policies. At the top of a hierarchy, policy is the most general of all policies. Policies near the bottom of the hierarchy provide more specialized specification. Except the root node policy, any policy has one and only one direct

superior policy. A policy hierarchy example is shown in Figure 7.6. When an authorization request is checked by a policy that is in a policy hierarchy, the policies from the given policy to the root node policy may be checked. Each policy hierarchy is associated with a policy combining algorithm used to combine these policies.

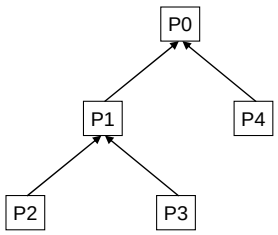


Figure 7.6: Example of policy hierarchy.

Policy hierarchy caters the requirement of large organizations composed of different divisions, each of which can independently specify security policies. The global policy of the organization results from the combination of the organization scope policies and the division scope policies. Similar requirement also exists in virtual organizations (VOs) where need to combine the VO scope policies and the local organization scope policies.

The following example specifies a policy hierarchy shown in Figure 7.6, and the associated policy combining algorithm is “deny-overrides”.

```
<PolicyHierarchies>
  <PolicyHierarchy PolicyHierarchyID="Policy_Hierarchy" PolicyCombiningAlgID="policy-combining-
algorithm:deny-overrides">
    <PolicyReference PolicyID="P0">
      <SubPolicyReference PolicyID="P1" />
      <SubPolicyReference PolicyID="P4" />
    </PolicyReference>
    <PolicyReference PolicyID="P1">
      <SubPolicyReference PolicyID="P2" />
      <SubPolicyReference PolicyID="P3" />
    </PolicyReference>
    <PolicyReference PolicyID="P2" />
    <PolicyReference PolicyID="P3" />
    <PolicyReference PolicyID="P4" />
  </PolicyHierarchy>
</PolicyHierarchies>
```

7.4.6 Policy schema

PolicySchema specifies what requests should be checked by which policies, and how these policies are combined together to make decisions. As shown in Figure 7.4, each schema contains five data elements; they are *SchemaSubjectDomains*, *SchemaResourceDomains*, *SchemaPolicyHierarcys*, *PolicyItems* and *Context*.

The *SchemaSubjectDomains* specifies the domains from which the request subjects are valid. The *SchemaResourceDomains* specifies the domains in which the request objects are valid. A policy schema is applicable if and only if the request subject matches the schema subject domains, the request object matches the schema resource domains and the associated context constraints evaluate to “true”. The *SchemaPolicyHierarchies* specifies which policy hierarchies should be considered in this schema.

PolicyItem is the basic data element used to describe policy relations. Each *PolicyItem* holds

a policy reference or a policy subschema reference. The relevant policies and policy subschemas are organized in *CombinedPolicyItem* elements that are organized into *PolicySet* elements which be further organized in a *PolicySets* element. Algorithms are defined to combine multiple *CombinedPolicyItem* elements and multiple *PolicySet* elements. The relation between *PolicyItem* and *PolicySets* is conjunction. A policy reference held by a *CombinedPolicyItem* can also be held by another *PolicyItem*. Through this way, any relations among policies can be defined. A *PolicyItem* can be associated with context constraints that are used to limit its usage according to the context information.

In each policy schema there is a policy item that is defined as the start point, from which the authorization requests begin to be checked by the policy schema. If there are multiple policy schemas satisfying one request, then these schemas' evaluation results should be combined with a policy schema combining algorithm.

The following example shows a policy schema. In this example there defines two subject domains, one resource domain and one policy hierarchy. A context constraint is associated to this schema. There two policies relate to the policy item, they are organized in one policy set. The start point of the policy schema is the policy item "Permis_Policy_1".

```
<PolicySchemas SchemaCombiningAlgID="schema-combining-algorithm:permit-overrides">
 <PolicySchema PolicySchemaID="B2BMCPolicySchema1">
  <StartPolicyItem PolicyItemID="Permis_Policy_1" />
  <SchemaSubjectDomains>
   <SubjectDomainReference SubjectDomainID="TrierEmployees" />
   <SubjectDomainReference SubjectDomainID="PotsdamEmployees" />
  </SchemaSubjectDomains>
  <SchemaResourceDomains>
   <ResourceDomainReference ResourceDomainID="B2BMCFiles" />
  </SchemaResourceDomains>
  <SchemaPolicyHierarchies HierarchyCombiningAlgID="hierarchy-combining-algorithm:deny-
overrides">
   <PolicyHierarchyReference PolicyHierarchyID="Policy_Hierarchy_1" />
  </SchemaPolicyHierarchies>
  <SchemaPolicyRelations>
   <PolicyItem PolicyItemType="Policy" PolicyItemID="Permis_Policy_1">
    <PolicySets PolicySetCombiningAlgID="policy-set-combining-algorithm:permit-overrides">
     <PolicySet PolicyItemCombiningAlgID="policy-item-combining-algorithm:permit-
overrides">
      <CombinedPolicyItem PolicyItemType="Policy" PolicyItemID="XACML_Policy_1" />
      <CombinedPolicyItem PolicyItemType="Policy" PolicyItemID="XACML_Policy_2" />
     </PolicySet>
    </PolicySets>
   </PolicyItem>
  </SchemaPolicyRelations>
  <Context>
   <ContextConstraintReference ContextConstraintID="WorkingTime" />
  </Context>
 </PolicySchema>
</PolicySchemas>
```

7.4.7 Policy subschema

PolicySubSchema is a simplified policy schema. It can be invoked by policy schemas or other policy subschemas. The major difference between policy subschema and policy schema is that subschema does not specify schema subject domains and schema resource domains. This characteristic makes policy subschemas can be invoked by multiple policy schemas that have different application scopes.

The root policy schema is included in Appendix A.

7.5 Root policy evaluation

The major functionality of the root policy is to decide what authorization requests should be evaluated by which authorization policies and how to combine their evaluation results in order to render a finial decision. Root policy evaluation results could be “Permit”, “Deny”, “NotApplicable” or “Indeterminate”. In this section we investigate the root policy evaluation rules. Policy combining algorithms are introduced in next section.

7.5.1 Schema domain match evaluation

Schema domain match evaluation checks if a schema is applicable to an authorization request. The schema domain shall be “Match” if the schema subject domains and resource domains both match a request’s subject and resource, respectively. If one schema does not specify schema subject/resource domain, it means there is no any limitation to the subject/resource domains, and then any domain check returns the value “Match”. If any one of them is “Indeterminate”, the policy schema domain match shall be “Indeterminate”. The policy schema match table is shown in Table 7.1.

Schema Subject Domains	Schema Resource Domains	Schema Domain Match
“Match”	“Match”	“Match”
“No match”	Do not care	“No match”
Do not care	“No match”	“No match”
“Indeterminate”	“Match” or “Indeterminate”	“Indeterminate”
“Match” or “Indeterminate”	“Indeterminate”	“Indeterminate”

Table 7.1: Policy schema match table.

The schema subject/resource domains shall match the values in a request if at least one of its subject/resource domains matches the subject/resource value in the request. If an error occurs while evaluating a schema subject/resource domain, such as no such domain reference, then this domain match shall evaluate to “Indeterminate”. The schema subject/resource domains match table is shown in Table 7.2.

Schema Subject/Resource Domain	Schema Subject/Resource Domains
At least one “Match”	“Match”
All “No match”	“No match”
"None matches and at least one “Indeterminate”	“Indeterminate”

Table 7.2: Policy schema subject/resource domains match table.

A subject domain shall match the subject value in a request if one of its include LDAPDN matches the value in the request, and in the same time no one of its exclude LDAPDN match the value in the request. If include LDAPDN is null, it means there is no any limitation to the subject domains and any comparisons return the value “Match”. If exclude LDAPDN is null, it means there is no any exception to the subject domains and any comparisons return the value “No match”. The subject domain match table is shown in Table 7.3.

Subject Domain Include LDAPDN	Subject Domain Exclude LDAPDN	Subject Domain
“Match”	“No match”	“Match”
“No match”	Do not care	“No match”
Do not care	“Match”	“No match”

Table 7.3: Subject domain match table.

A resource domain shall match the resource value in a request if one of its include LDAPDN matches the value in the request, and in the same time no one of its exclude LDAPDN match the value in the request. If include LDAPDN is null, it means there is no any limitation to the resource domains and any comparisons return the value “Match”. If exclude LDAPDN is null, it means there is no any exception to the resource domains and any comparisons return the value “No match”. A domain may optionally be refined by a set of object classes. Only resources with the full set of object classes are included in the domain. A null object class set implies all the resources in the domain are included. The resource domain match table is shown in Table 7.4.

Resource Domain Include LDAPDN	Resource Domain Exclude LDAPDN	Object Class	Resource Domain
“Match”	“No match”	“Match”	“Match”
“No match”	Do not care	Do not care	“No match”
Do not care	“Match”	Do not care	“No match”
Do not care	Do not care	“No match”	“No match”

Table 7.4: Resource domain match table.

7.5.2 Policy item evaluation

The evaluation of a policy item comprises three steps. The first step is evaluating context constraints associated to the policy item. If there is no context constraint or all the context constraints are evaluated to “true”, the system continues to do the second step. The second step is evaluating this policy item. If the policy item is evaluated to “Permit”, the system continues to do the third step. The third step is evaluating policy sets bound to this policy item. If there is no policy set, then we think the evaluation result is “Permit”. The policy item evaluation truth table is shown in Table 7.5.

Context Constraints	Policy Item	Policy Sets	Policy Item Evaluation
“True”	“Permit”	“Permit”	“Permit”
“True”	“Permit”	“Deny”	“Deny”
“True”	“Permit”	“Indeterminate”	“Indeterminate”
“True”	“Deny”	Do not care	“Deny”
“True”	“Indeterminate”	Do not care	“Indeterminate”
“False”	Do not care	Do not care	“NotApplicable”

Table 7.5: Policy item evaluation true table.

7.5.3 Policy sets evaluation

The value of a policy sets shall be determined by its contents, considered in relation to the contents of the request. A policy sets' value shall be determined by evaluation of the policy sets according to the specified policy-set-combining-algorithm. The policy sets truth table is shown in Table 7.6.

Combined Policy Set Values	Policy Sets Value
At least one policy set is its decision	Specified by the policy set combining algorithm
All policy set values are "NotApplicable"	"NotApplicable"
At least one policy set is "Indeterminate"	Specified by the policy set combining algorithm

Table 7.6: Policy sets evaluation truth table.

7.5.4 Policy set evaluation

The value of a policy set shall be determined by its contents, considered in relation to the contents of the request. A policy set's value shall be determined by evaluation of the combined policy items according to the specified policy-item-combining-algorithm. The policy set truth table is shown in Table 7.7.

Combined Policy Item Values	Policy Set Value
At least one policy item is its decision	Specified by the policy item combining algorithm
All policy item values are "NotApplicable"	"NotApplicable"
At least one policy item is "Indeterminate"	Specified by the policy item combining algorithm

Table 7.7: Policy set evaluation truth table.

7.5.5 Policy schema evaluation

The value of a policy schema shall be determined by its contents, considered in relation to the contents of the request. Policy schema evaluation is composed of associated context constraints evaluation, schema domain match and start policy item evaluation. The policy schema evaluation truth table is shown in Table 7.8.

Context Constraints	Schema Domain Match	Start Policy Item	Schema Evaluation
"True"	"Match"	"Permit"	"Permit"
"True"	"Match"	"Deny"	"Deny"
"True"	"Match"	"NotApplicable"	"NotApplicable"
"True"	"Match"	"Indeterminate"	"Indeterminate"
"True"	"No-match"	Do not care	"NotApplicable"
"True"	"Indeterminate"	Do not care	"Indeterminate"
"False"	Do not care	Do not care	"NotApplicable"

Table 7.8: Policy schema evaluation truth table.

7.6 Policy combining algorithms

In root policy there are five kinds of policy components that need to be combined, they are *policy*, *policy hierarchy*, *policy item*, *policy set* and *policy schema*. Policies are the basic elements that constitute other policy components. In this section, we first give the general description of combining algorithms, and then define the various policy component combing algorithms.

7.6.1 Combining algorithm description

The combining algorithms defined to combine root policy components are "deny-overrides" and "permit-overrides". They are described as follows.

- *Deny-overrides*. In the entire set of authorization components, if any component evaluates to "Deny", then the result of the combination shall be "Deny". In other words, "Deny" takes precedence, regardless of the result of evaluating any of the other components. If any component evaluates to "Permit", and all other components evaluate to "Permit" or "NotApplicable", then the result of the combination shall be "Permit". If all components are found to be "NotApplicable", then the combination shall evaluate to "NotApplicable". If an error occurs while evaluating a component, the evaluation shall continue looking for a result of "Deny". If no other component evaluates to "Deny", then the combination shall evaluate to "Indeterminate".
- *Permit-overrides*. In the entire set of authorization components, if any component evaluates to "Permit", then the result of the combination shall be "Permit". In other words, "Permit" takes precedence, regardless of the result of evaluating any of the other components. If any component evaluates to "Deny", and all other components evaluate to "Deny" or "NotApplicable", then the result of the combination shall be "Deny". If all components are found to be "NotApplicable", then the combination shall evaluate to "NotApplicable". If an error occurs while evaluating a component, the evaluation shall continue looking for a result of "Permit". If no other component evaluates to "Permit", then the combination shall evaluate to "Indeterminate".

7.6.2 Component combining algorithms

- *Policy combining algorithm* specifies the procedure for combining the decisions from multiple *policies*. The policy set is gotten from a *PolicyHierarchy* element. The algorithm is identified by the *PolicyCombiningAlgId* attribute which can take one of the following values:
  - policy-combining-algorithm:deny-overrides,
  - policy-combining-algorithm:permit-overrides.
- *Policy hierarchy combining algorithm* specifies the procedure for combining the decisions from multiple *policy hierarchies*. The policy hierarchy set is gotten from a *SchemaPolicyHierarchies* element. The algorithm is identified by the *HierarchyCombiningAlgId* attribute which can take one of the following values:
  - policy-hierarchy-combining-algorithm:deny-overrides,
  - policy-hierarchy-combining-algorithm:permit-overrides.

- *Policy item combining algorithm* specifies the procedure for combining the decisions from multiple *policy items*. The policy item set is gotten from a *PolicySet* element. This algorithm is identified by the *PolicyItemCombiningAlgId* attribute which can take one of the following values:
  - policy-item-combining-algorithm:deny-overrides,
  - policy-item-combining-algorithm:permit-overrides.
- *Policy set combining algorithm* specifies the procedure for combining the decisions from multiple *policy sets*. The collection of policy sets is gotten from a *PolicySets* element. This algorithm is identified by the *PolicySetCombiningAlgId* attribute which can take one of the following values:
  - policy-set-combining-algorithm:deny-overrides,
  - policy-set-combining-algorithm:permit-overrides.
- *Policy schema combining algorithm* specifies the procedure for combining the decisions from multiple *policy schemas*. The policy schema set is gotten from the *PolicySchemas* element. This algorithm is identified by the *PolicySchemaCombiningAlgId* attribute which can take one of the following values:
  - schema-combining-algorithm:deny-overrides,
  - schema-combining-algorithm:permit-overrides.

7.7 Root policy enforcement

In this section we introduce the root policy enforcement mechanism. The implementation uses object-oriented technology. The classes used to implement a root policy system can be classified into two categories. One is the classes used to statically describe the root policies. The other is the classes used to dynamically enforce the root policies. They are described in Section 7.7.1 and Section 7.7.2, respectively. Section 7.7.3 introduces the root policy evaluator. Section 7.7.4 looks at the root policy collaboration. Finally, Section 7.7.5 gives an overview to the root policy implementation.

7.7.1 Root policy static description classes

The classes used to statically describe a root policy are *RootPolicy*, *SubjectDomain*, *ResourceDomain*, *ContextConstraint*, *Policy*, *PolicyHierarchy*, *PolicySchema*, *PolicyItem*, *PolicySets*, *PolicySet* and *CombinedPolicyItems*. The *PolicySchema* is also used to hold policy subschemas. Their relations are shown in Figure 7.4. These classes are used to provide various root policy information to the root policy evaluator at runtime. In order to initialize a *RootPolicy* instance, a root policy filename should be passed to the class constructor. In the root policy initialization process, all the root policy components are saved in the corresponding classes.

7.7.2 Root policy dynamic runtime classes

The major classes used to enforce a root policy and their relations are shown in Figure 7.7. Their functions are described as follows.

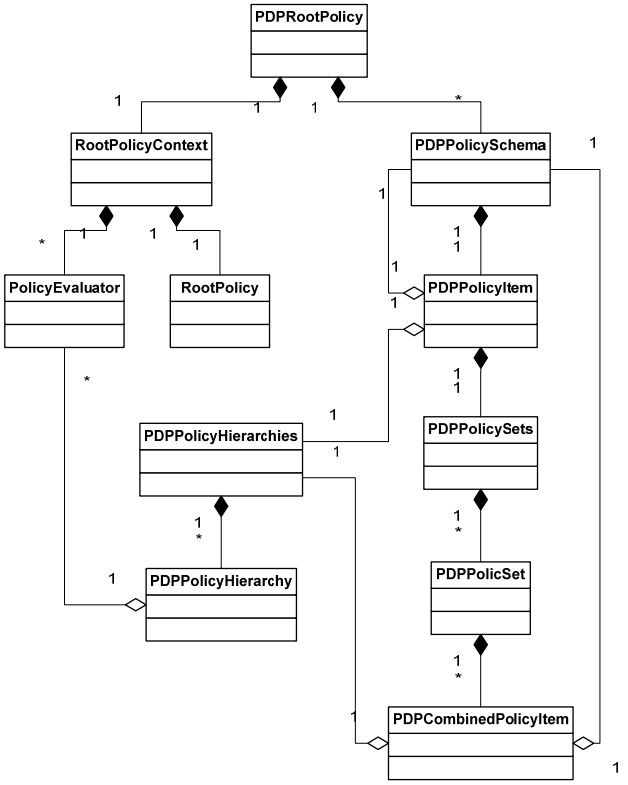


Figure 7.7: Root policy dynamic runtime class relations

- *PDPRootPolicy* creates the root policy PDP for a root policy. In order to initialize a *PDPRootPolicy* instance, a root policy filename is passed to the class constructor. If the *RootPolicyContext* instance is created successfully, the *PDPRootPolicy* instance is ready to make authorization decisions. It is done through checking *PDPPolicySchema* instances.
- *RootPolicyContext* does two tasks. One is initializing a *RootPolicy* instance through passing a root policy filename to the class constructor. Another is initialing all the policy evaluators according to the policy specification in the root policy.
- *PDPPolicySchema* evaluates requests against a policy schema. It first checks if the schema is applicable. If the schema is applicable, it passes the check process to the schema's start policy item, otherwise returns the value "NotApplicable".
- *PDPPolicyItem* evaluates requests against a policy item. It first checks if the policy item is applicable. If the policy item is applicable, it continues to evaluate the policy item, otherwise returns the value "NotApplicable". If the policy item holds a policy reference, it evaluates the policy in the policy hierarchies specified in the schema. If the policy item holds a policy subschema reference, it passes the check process to the subschema. If the evaluation result is "Permit", the *PDPPolicyItem* continues to evaluate the associated policy sets, and returns the evaluation result.
- *PDPPolicyHierarchies* evaluates requests against a set of policy hierarchies specified in a schema. It passes a policy reference and request to each of them, and combines the evaluation results with a policy-hierarchy-combining algorithm.
- *PDPPolicyHierarchy* evaluates requests against a policy hierarchy. It first gets a set of

policies from the policy hierarchy according to a policy id, and then uses them to check the given request. The evaluation results are combined with a policy-combining algorithm.

- *PDPPolicySets* evaluates requests against a collection of policy sets that associated to a policy item. The evaluation results are combined with a policy-set-combining algorithm.
- *PDPPolicySet* evaluates requests against a set of combined policy items. The evaluation results are combined with a policy-item-combining algorithm.
- *PDPCombinedPolicyItem* evaluates requests against a combined-policy item. A combined-policy item holds either a policy reference or a policy subschema reference. If it holds a policy reference that is also held by another policy item, then the check process is passed to that policy item, otherwise the evaluation process is the same as evaluating a policy item.

7.7.3 Authorization policy evaluators

The root policy evaluator is designed for policy agnostic. The basic idea is that every kind of policies has its own evaluator, and different policy evaluators have the same interface so that the root policy evaluator can treat them in a unified way. Thus, new policy evaluator can be dynamically added or removed from the system at runtime. The classes related to policy evaluator dynamic loading is shown in Figure 7.8.

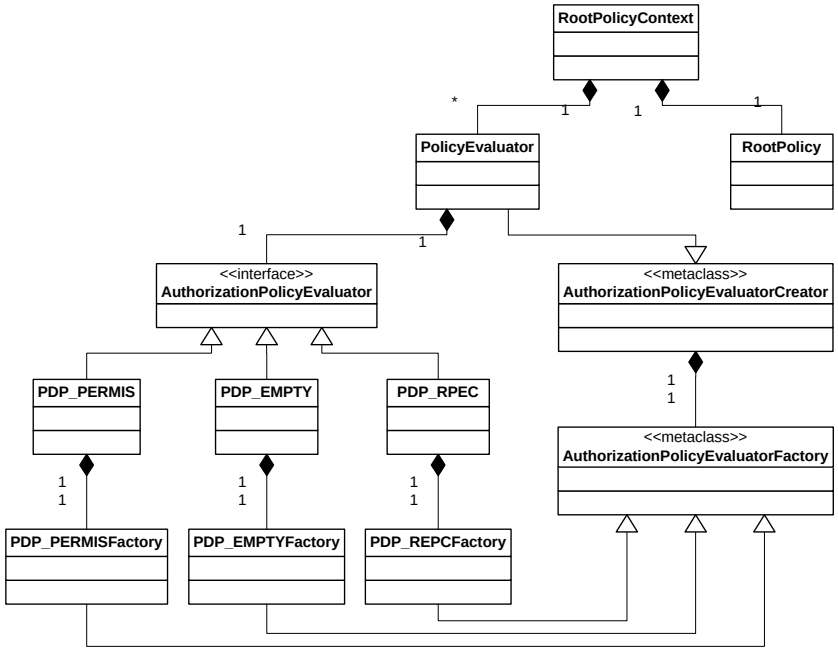


Figure 7.8: Essential class relations of root policy context.

A *RootPolicyContext* instance is the container of a *RootPolicy* instance and a collection of *PolicyEvaluator* instances. Each *PolicyEvaluator* instance handles one authorization policy and acts as a PDP managed by a root policy system. In order to initialize a policy evaluator, the policy evaluator class name and policy filename are passed into the *PolicyEvaluator* class constructor. The *PolicyEvaluator* handles a concrete authorization policy evaluator through the interface *AuthorizationPolicyEvaluator*. Each concrete policy evaluator is implemented separately and independently. They can be plugged into a root policy evaluator at runtime. The

only restriction is that all of them must share a similar interface, i.e. they have the same methods in common. The Java interface is defined as:

```
public interface AuthorizationPolicyEvaluator {
  public int evaluate(RequestContext requestContext);
}
```

The interface method *evaluate* must be implemented in a concrete authorization policy evaluator. Through this interface authorization requests are passed into the policy evaluator. The class *PolicyEvaluator* inherits from the abstract class *AuthorizationPolicyEvaluatorCreator* that is responsible for creating an *AuthorizationPolicyEvaluator* instance that is saved in the *PolicyEvaluator* instance. The *AuthorizationPolicyEvaluatorCreator* class is defined as follow.

```
public abstract class AuthorizationPolicyEvaluatorCreator {
 public static Hashtable authorizationPolicyEvaluatorFactories = new Hashtable();
  protected static AuthorizationPolicyEvaluator
createAuthorizationPolicyEvaluator(String policyEvaluatorName, String policyFilename)
throws AuthorizationPolicyEvaluatorNotCreatedException {
    AuthorizationPolicyEvaluatorFactory f = (AuthorizationPolicyEvaluatorFactory)
authorizationPolicyEvaluatorFactories.get(policyEvaluatorName);
    if(f == null) {
      try {
       Class.forName("RootPolicy.AuthorizationPolicyEvaluator.Evaluators."+policyEvaluat
       orName);
        f = (AuthorizationPolicyEvaluatorFactory)
authorizationPolicyEvaluatorFactories.get(policyEvaluatorName);
        if(f == null) {
          throw (new AuthorizationPolicyEvaluatorNotCreatedException());
        }
      } catch(ClassNotFoundException e) {
        throw(new AuthorizationPolicyEvaluatorNotCreatedException());
      }
    }
    return(f.create(policyFilename));
  }
}
```

The class static variable *authorizationPolicyEvaluatorFactories* is a Java Hashtable from which the *AuthorizationPolicyEvaluatorFactory* for a given authorization policy evaluator can be retrieved. The class static method *createAuthorizationPolicyEvaluator* creates instances of *AuthorizationPolicyEvaluator*.

The *AuthorizationPolicyEvaluatorFactory* is an abstract class that creates the instance of an *AuthorizationPolicyEvaluator* for the *AuthorizationPolicyEvaluatorCreator*. The idea of using a *Factory* is that in order to create the instance of an object, we use another object which has better access to the object. Through this way, the *AuthorizationPolicyEvaluator* can require some information that cannot be done with its own method. Each authorization policy evaluator is a subclass of *AuthorizationPolicyEvaluatorFactory* and provides its own *Factory*. The abstract class *AuthorizationPolicyEvaluatorFactory* is defined as:

```
public abstract class AuthorizationPolicyEvaluatorFactory {
  public abstract AuthorizationPolicyEvaluator create(String filename);
}
```

Now we show how to implement an authorization policy evaluator through a example. In Figure 7.8 the PDP_PERMIS is the evaluator of X.509_PMI_RBAC_Policy policies [CO02]. In PDP_PERMIS, the codes used to create *PDP_PERMISFactory*, initialize instance and make authorization decisions are shown as follows.

```
public class PDP_PERMIS implements AuthorizationPolicyEvaluator {
  public PDP_PERMIS() {}
  public PDP_PERMIS(String filename) {
    initPolicy(filename);
  }
  static class PDP_PERMISFactory extends AuthorizationPolicyEvaluatorFactory {
    public AuthorizationPolicyEvaluator create(String filename){
      return(new PDP_PERMIS(filename));
    }
  }
  static {
AuthorizationPolicyEvaluatorCreator.authorizationPolicyEvaluatorFactories.put("PDP_PERMI
S", new PDP_PERMISFactory());
  }
  public boolean initPolicy(String filename){
    //Initialize Permis RBAC Policy specified by the filename.
  }
  public int evaluate(RequestContext requestContext) {
    PermisRequest permisRequest = buildRequest(requestContext);
    return makeDecision(permisRequest);
  }
  public PermisRequest buildRequest(RequestContext requestContext) {
    PermisRequest permisRequest;
    //Build Permis RBAC Policy Request according to the root policy request context.
    return permisRequest;
  }
  public int makeDecision(PermisRequest permisRequest){
    int decision;
    //Make an access control decision according to the Permis RBAC Policy.
    return decision;
  }
}
```

The *evaluate* method is the implementation of the interface method defined in *AuthorizationPolicyEvaluator*. The *buildRequest* method converts a root policy request into the request format used by the Permis policy evaluator, and then calls *makeDecision* method to make an access control decision. The *makeDecision* is also responsible for converting the decision result from its own native format into a root policy response. For example, the decision result "Permit" should be converted to an *int* value "0" that represents "Permit" in root policy.

In Figure 7.8 the PDP_EMPTY is a predefined evaluator used to evaluate empty policies. The evaluation results of PDP_EMPTY are always "Permit". The codes used to implement PDP_EMPTY are shown as follows.

```
public class PDP_EMPTY implements AuthorizationPolicyEvaluator {
  public PDP_EMPTY() {}
  static class PDP_EMPTYFactory extends AuthorizationPolicyEvaluatorFactory {
    public AuthorizationPolicyEvaluator create(String policyID){
      return(new PDP_EMPTY());
    }
  }
  static {
       AuthorizationPolicyEvaluatorCreator.authorizationPolicyEvaluatorFactories.put("PD
P_EMPTY", new PDP_EMPTYFactory());
  }
  public int evaluate(RequestContext requestContext) {
    return 0;
  }
}
```

7.7.4 Root policy collaboration

Multiple root policies can collaborate to complete more complicated authorization tasks. The root policy collaboration is implemented through remote policy evaluation calling, i.e. an

authorization policy specified in one root policy is realized by another root policy. In this case the policy specification field *RealizedBy* provides a URI to which all the requests related to this policy will be sent. Every root policy or authorization policy has a unique object identifier (OID) in the scope where it is applied. If one policy is realized by another root policy, then their OID is the same. An example of root policy collaboration is shown in Figure 7.9.

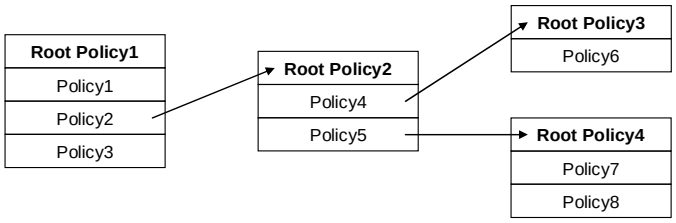


Figure 7.9: Example of root policy collaboration.

In this example there are four root policies. In *Root Policy*1 there define three authorization policies, and the *Policy*2 is realized by the *Root Policy*2. In the *Root Policy*2 there define two authorization policies, both of them are realized by some other root policies, i.e. the *Policy*4 is realized by the *Root Policy*3 and the *Policy*5 is realized by the *Root Policy*4. Through this way, one authorization request check can be propagated among many root policies.

In Figure 7.8 the PDP_RPEC is a predefined evaluator that is responsible for dealing with remote policy evaluation calling. The PDP_RPEC has similar structure with other authorization policy evaluator except the parameter structure. Initializing a normal authorization policy evaluator, the passed parameter is a policy filename. Initializing a PDP_RPEC evaluator, the passed the parameter is a root policy OID by which the decision requests should be evaluated and a URI to which the remote policy calling messages should be sent. The codes used to implement PDP_RPEC are shown as follows.

```
public class PDP_RPEC implements AuthorizationPolicyEvaluator {
  private String policyOID;
  private String realizedByURI;
  public PDP_RPEC() {}
  public PDP_RPEC(String realizedBy) {
    initPolicy(realizedBy);
  }
  static class PDP_RPECFactory extends AuthorizationPolicyEvaluatorFactory {
    public AuthorizationPolicyEvaluator create(String realizedBy){
      return(new PDP_RPEC(realizedBy));
    }
  }
  static {
       AuthorizationPolicyEvaluatorCreator.authorizationPolicyEvaluatorFactories.put("PD
P_RPEC", new PDP_RPECFactory());
  }
  public void initPolicy(String realizedBy){
    String [] paras = realizedBy.split(";");
    policyOID = paras[0];
    realizedByURI = paras[1];
  }
  public int evaluate(RequestContext requestContext) {
    Hashtable permisParam = buildRequest(requestContext);
    return makeDecision(permisParam);
  }
  public Hashtable buildRequest(RequestContext requestContext) {
    Hashtable rpecParam = new Hashtable();
    rpecParam.put("PolicyOID", policyOID);
    rpecParam.put("RequestContext", requestContext);
    return rpecParam;
```

```
  }
  public int makeDecision(Hashtable paramTable){
    CoordinatorRequest cr = new CoordinatorRequest(realizedByURI);
    int decisionResult = cr.sendRequest(paramTable);
    return decisionResult;
  }
}
```

The *initPolicy* method is responsible for initializing a PDP_RPEC instance. It extracts the values *oid* and *uri*. When a decision request is passed into this instance, the *evaluate* method first invokes the *buildRequest* method to build a new request object that consists of policy *oid* and the request, and then invokes the *makeDecision* method to get the evaluation result. The *makeDecision* method will try to get an evaluation result via the class *CoordinatorRequest* that belongs to the component *root policy coordinator*. The *root policy coordinator* is responsible for dealing with the communications among root policies. The structure of root policy coordinator and its relation with other components are shown in Figure 7.10.

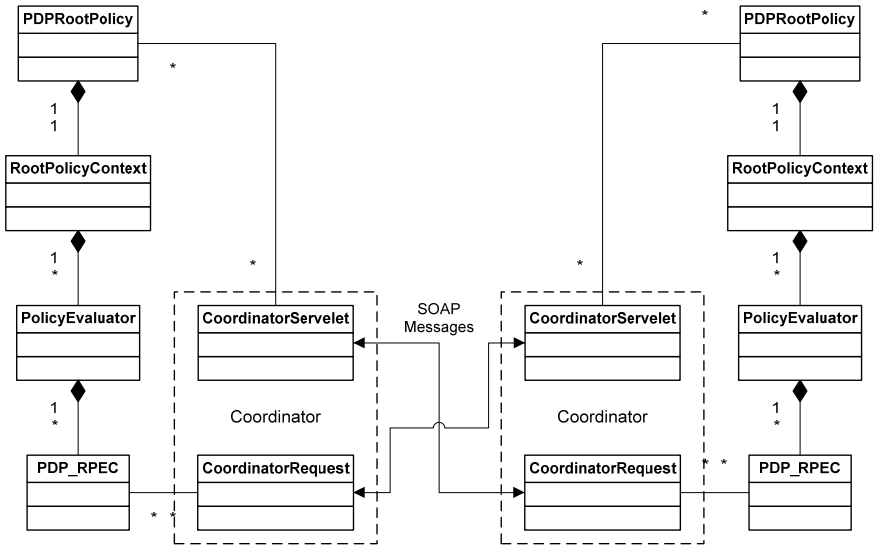


Figure 7.10: Structure of root policy coordinator.

CoordinatorRequest is responsible for converting root policy requests into SOAP messages and send them to the destination sites. The parameters passed by an invoker are *policyOID*, *realizedByURI* and *requestContext*. *policyOID* and *requestContext* are encapsulated into a SOAP message that will be sent to an endpoint specified by *realizedByURI*. If a valid response message is received, the *CoordinatorRequest* converts this message into a root policy response and then returns it to the invoker, otherwise a response with the value "Indeterminate" is returned. An example of remote root policy calling message is shown as follows.

```
<?xml version="1.0" encoding="UTF-8" ?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header />
  <SOAP-ENV:Body>
    <PolicyOID>1.2.826.0.1.3344810.1.1.14.1</PolicyOID>
    <RequestContext>
      <RequestElement ElementID="Subject">
        <RequestAttribute AttributeID="Domain"
DataType="LDAPDN">cn=alice,ou=marketing,l=uk,dc=abc.com</RequestAttribute>
      <RequestAttribute AttributeID="SubjectID"
```

```
DataType="RFC822Name">alice@abc.com</RequestAttribute>
      <RequestAttribute AttributeID="Role" DataType="TIRole">manager</RequestAttribute>
      </RequestElement>
      <RequestElement ElementID="Resource">
        <RequestAttribute AttributeID="Domain"
DataType="LDAPDN">ou=services,l=uk,dc=abc.com</RequestAttribute>
        <RequestAttribute AttributeID="ResourceName"
DataType="String">cn=products,l=uk,dc=abc.com</RequestAttribute>
      </RequestElement>
      <RequestElement ElementID="Action">
      <RequestAttribute AttributeID="ActionName" DataType="String">read</RequestAttribute>
    </RequestElement>
   </RequestContext>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

CoordinatorServlet is responsible for receiving remote root policy calling messages, converting them into root policy requests and then invoking corresponding root policy evaluators according to the policy OID carried by the messages. The evaluation result is returned back to the requester via a root policy decision response SOAP message. An example of a response message is shown as follows.

```
<?xml version="1.0" encoding="UTF-8" ?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header />
  <SOAP-ENV:Body>
    <ResponseContext>
      <ResponseElement ElementID="DecisionResult">
        <ResponseAttribute AttributeID="Decision">DECISION_PERMIT</ResponseAttribute>
      </ResponseElement>
    </ResponseContext>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

One root policy coordinator may manage multiple root policy evaluators. These evaluators must register to the policy coordinator so that they can be invoked. The registration information is saved in a XML file. An example of root policy registration file is shown as follows.

```
<?xml version="1.0" encoding="UTF-8" ?>
<RegisteredRootPolicy>
  <RootPolicy OID="1.2.826.0.1.3344810.1.1.14.1">
    <URI>C:/RootPolicy/RootPolicy1.XML</URI>
  </RootPolicy>
  <RootPolicy OID="1.2.826.0.1.3344810.1.1.14.2">
    <URI>C:/RootPolicy/RootPolicy2.XML</URI>
  </RootPolicy>
</RegisteredRootPolicy>
```

7.7.5 Root policy implementation

Based on the proposed root policy enforcement mechanism, we have developed a root policy system that is written in Java, and uses the following software:

- Jakarta Tomcat servlet container [Tomcat] is used for running the root policy coordinators and the root policy evaluators
- SOAP with Attachments API for Java (SAAJ) [SAAJ] is used for implementing the communications among root policy coordinators.

- IBM XML Security Suite [IBMSS] provides the security features, such as digital signature and encryption, for the SOAP messages.
- IAIK-JCE [IAIK] provides PKI and PMI related functions, e.g. creates X.509 attribute certificates used to hold authorization policies.
- OpenLDAP [OLDAP] is used as the repositories for X.509 public key certificates and X.509 attribute certificates.

To simplify the root policy creation and modification we develop a GUI-based root policy editor. Its screen snapshot is shown in Figure 7.11. We also developed an administration tool that can create key pair, X.509 PKCs and X.509 policy ACs.

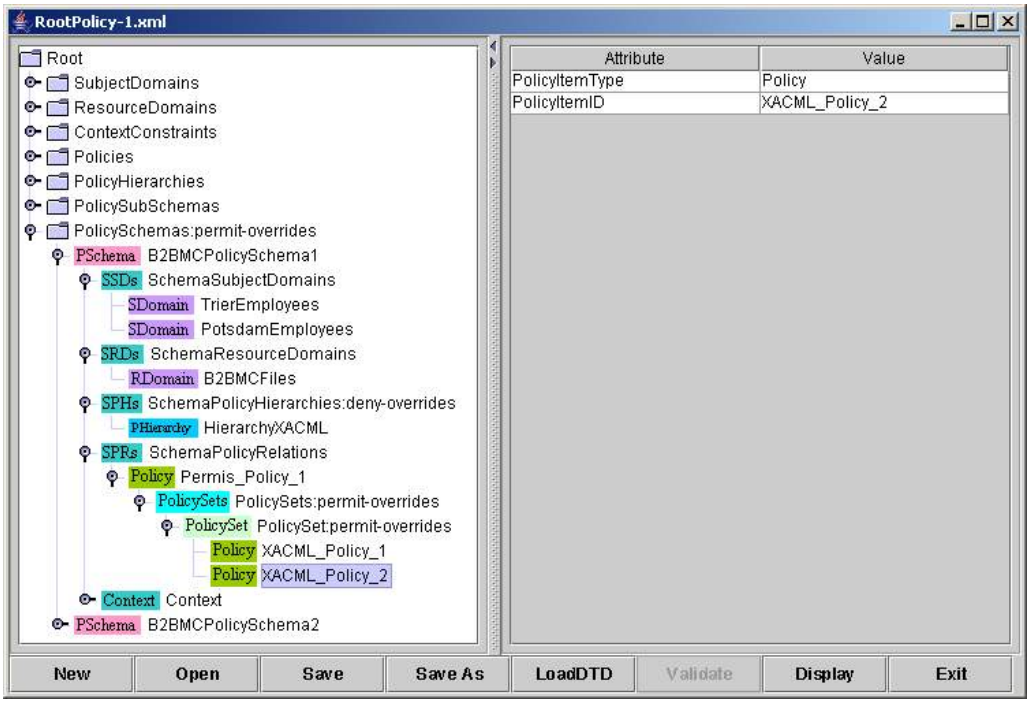


Figure 7.11: Root policy editor.

7.8 Conclusion

Root policy is used to specify and enforce multiple heterogeneous authorization policies. With the root policy we can get four major benefits. (1) It allows security administrators to freely define each involved authorization policy's storage, trust management and enforcement independently. New kinds of policy evaluators and policies can be dynamically added into a root policy system at runtime. (2) With the help of policy schemas, policy subschemas and policy hierarchies, complex authorization policy relations can be easily defined. (3) The context constraint component makes the root policy is a context-aware authorization system. It is important for collaborative systems to specify and change policies at runtime depending on the environment or collaboration dynamics. Flexible context constraints enable the root policy to deal with complex authorization scenarios (4) On the other hand multiple root policies can cooperate together to complete more complicated authorization tasks.

Chapter 8

Applications

The focus of this chapter is to describe some case studies, through which to show how some of our research results introduced in this thesis could be used in the real world. Section 8.1 uses TT-RBAC to analyze a real hospital authorization system. Section 8.2 provides a real case study of function-based authorization constraints used in a management information system. Section 8.3 provides a real case study of function-based authorization constraints integrated into a business process management system.

8.1 Case study: TT-RBAC in a hospital information system

8.1.1 A hospital authorization system introduction

Chuangye Software Corporation (www.bsoft.com.cn) is the largest software company in the field of health care information system in China. One of its major products is Chuangye Hospital Information System (CHIS) that is adopted by over 1000 large hospitals in China. The CHIS adopts client/server architecture and uses group-based authorization mechanism. In this subsection we briefly introduce CHIS's authorization mechanism that is shown in Figure 8.1 and described as follows.

1. *Database level*. CHIS adopts Oracle database that supports role-based access control. Several users are defined for accessing database. Each of them is assigned to a role that is associated with the permissions of accessing database. Different subsystems may use different database users to access database. For example, the user used by finance related subsystems and the user used by health care related subsystems are different.
2. *Subsystem level*. CHIS consists of lots of subsystems, such as clinic management information system, medicine management information system, clinic doctor workstation system, residency doctor workstation, and so on. Different subsystems access different kinds of data.
3. *Group level*. In a subsystem the access rights are assigned to different user groups. User groups are created according to the application requirements. There is no limitation on how many groups can be defined. For example, in medicine management information system there are manager group, clerk group and auditor group.

4. *User level*. In a user group the capability lists are used to manage group members' privileges. For example, the users handling concrete transactions do not need to access statistical reports. After a group user login a subsystem, the menu items are activated or deactivated according to the permissions assigned to him/her. The maximum permissions assigned to a group user are the permissions assigned to the group.

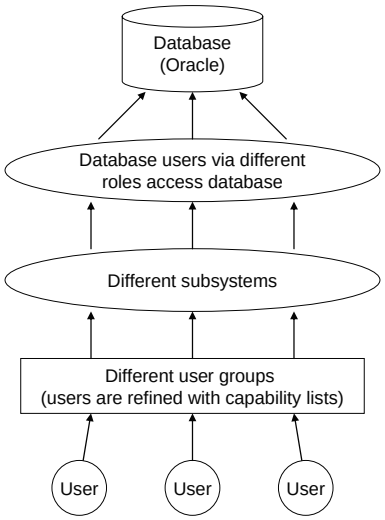


Figure 8.1: A hospital information system's authorization mechanism.

The authorization mechanism used by CHIS is not policy-based. Its major drawbacks are:

- In each subsystem, one user can only takes part in one group. If there is no group can cater to some application requirements, then a new group should be defined.
- The application scope of a user group is confined to one subsystem. So the collaboration among different subsystem is not possible, except that a new subsystem is developed.
- To accurately determine what permissions have been granted to a user or withdraw all permissions assigned to a user, administrators have to examine all the subsystems and possible all the groups in each subsystem. Since there are too many objects to query, the state of access control with respect to a particular user is sometimes rarely verified.
- As group collections grow, the number of places where administrators need to manage permissions grows. It requires consistent practice and coordination among administrators and precise definitions of user groups. For example, one administrator *A* assigned user *Peter* to the group *G*, but he/she did not tell this assignment to another administer *B*, then administrator *B* does not know *Peter* has some new privileges. Because one administrator normally does not check all the user groups to find what privileges have been assigned to a given user. These processes slow down the administrative process.

8.1.2 Using user group

Here we zoom into the *residency doctor workstation subsystem* of CHIS. In a large hospital, there may define several parallel groups that perform similar tasks in one department. For example, there may define several parallel care teams in the *medicine department* and *surgery department*. Any user who uses the *residency doctor workstation subsystem* can only belong to

one user group. Group members can only access the information of patients who they are caring. The privileges assigned to the group members are different. This is implemented through defining some subgroups which have different privileges. Users belonging to different subgroups will have different privileges. For example, members of *medicine department group* are typically organized into *consultant*, *associate consultant*, *principal doctor* and *residency doctor* four subgroups. A hospital user group definition example is shown in Figure 8.2. The permissions assigned to the four subgroups are described as follows.

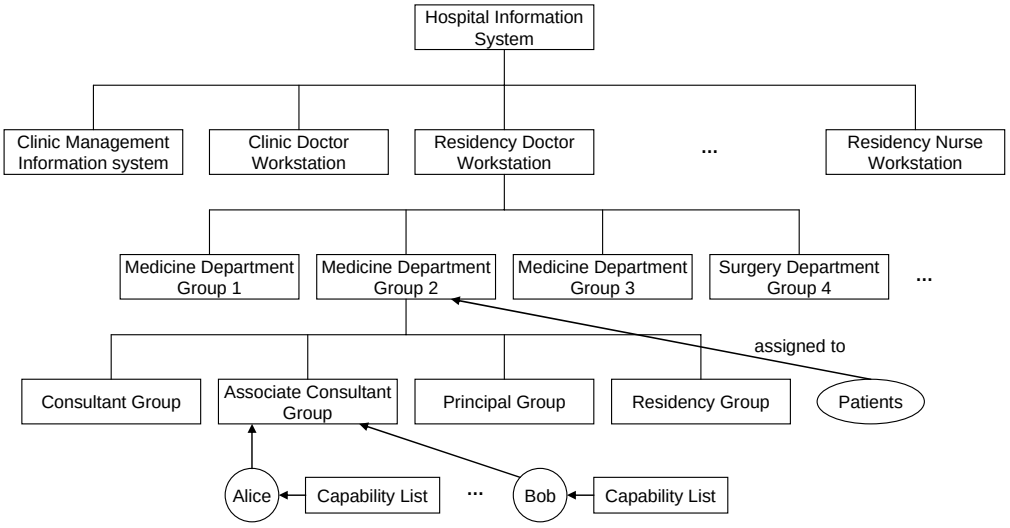


Figure 8.2: Example of using user group in a hospital information system.

- *Consultant group* groups top experts who do not take part in day-to-day ward rounds. They are available through making appointments.
- *Associate consultant group* groups experts who are junior to the users in the consultant group. They take part in day-to-day ward rounds. They give treatment advices to patients.
- *Principal group* groups doctors who are junior to the users in the associate consultant group. They prescribe for the patients according to the advices given by the members of consultant and associate group.
- *Residency group* groups the residencies. They write the patient records and perform the treatments.

In CHIS, the permissions are represented by the menu items. According to the application requirements, all or parts of the menu items are assigned to a group. Subgroups are created based on the job classifications in one group. The permissions assigned to a subgroup are a subset of the group permissions. The permissions assigned to a subgroup member are a subset of the subgroup permissions; the assigned permissions are stored in a Capability-list. Patients' ids are assigned to the groups. At runtime a clinic staff logs in to a CHIS subsystem with his/her username and password; the system actives some menu items according to his/her Capability-list; the group member can then access the medical records of patients who are assigned to his/her group.

An example of permission assignments in CHIS is shown in Figure 8.3. In this example, *Peter* belongs to the *consultant* group, and has all the permissions assigned to the group. *Bob* belongs to the *residency* group, and has the least permissions.

Residency doctor workstation menu items

{M1, M2, M3, M4, M5, M6, M7, M8 , M9, M10 , M11, M12}

Residency doctor workstation group definition

Medicine department group 1: {M1, M2, M3, M4, M5}
Medicine department group 2: {M1, M2, M3, M4, M5}
Medicine department group 3: {M1, M2, M3, M4, M5}

Medicine department group 2 permission definition

Consultant group: {M1, M2, M3, M4, M5}
Associate consultant group: {M1, M2, M3, M4}
Principal group: {M1, M2, M3}
Residency group: {M1, M2}

Medicine department group 2 user definition

Consultant group: {Peter, …}
Associate consultant group: {Mary, …}
Principal group: {Alice, …}
Residency group: {Bob, …}

Medicine department group 2 user permissions

Peter: {M1, M2, M3, M4, M5}
Mary: {M1, M2, M3, M4}
Alice: {M1, M2, M3}
Bob: {M1, M2}

Figure 8.3: Example of permission assignments in CHIS.

8.1.3 Using TT-RBAC

Now we investigate how the TT-RBAC can be used to replace the traditional group-based authorization system introduced in Section 8.1.2. Here the menu items are modeled as permissions, and the groups are replaced with teams. Because subgroups are created according to users' job functions, they can be modeled as roles, i.e. *Consultant*, *Associate consultant*, *Principal* and *Residency*. These roles are assigned to users and teams. The group permissions are replaced with team tasks. The patients' medical records are the targets that the team members can access. The corresponding TT-RBAC solution is shown in Figure 8.4.

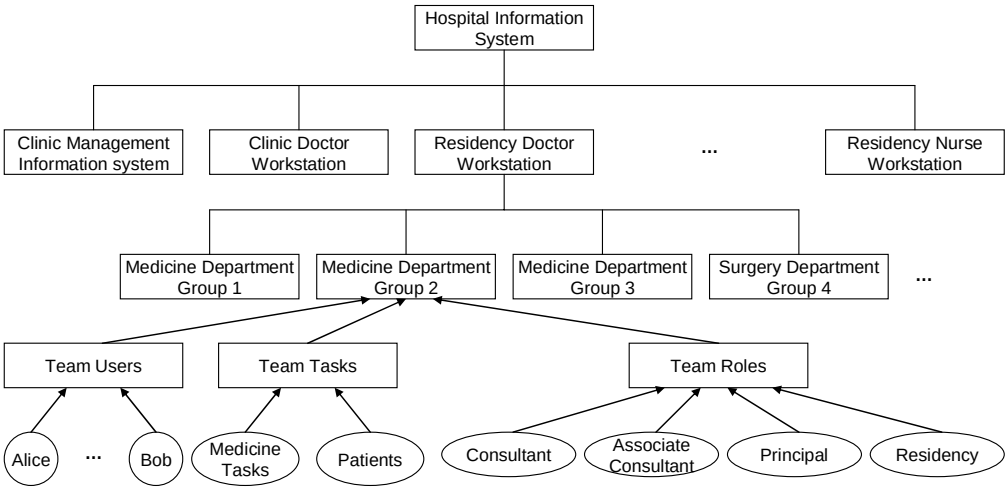


Figure 8.4: Example of using TT-RBAC in a hospital information system.

An example of TT-RBAC privilege assignments shown in Figure 8.5 are the same as what is in Figure 8.3. In this example, the team *medicine department group*2 is assigned to roles *Consultant*, *Associated consultant*, *Principal* and *Residency*. This team is also assigned to task *T*1 that contains *M*1, *M*2, *M*3, *M*4 and *M*5. *Peter* has the role *Consultant* that is permitted by the team. So *Peter* can active role *Consultant*, and has all the permissions assigned to the team. *Bob* has the role *Residency* that is also permitted by the team. The role *Residency* has the permissions *M*1, *M*2, *M*12. But *M*12 is not permitted by the team. So *Bob* can only have permissions *M*1 and *M*2.

Residency doctor workstation menu items

{M1, M2, M3, M4, M5, M6, M7, M8 , M9, M10 , M11, M12}

Role-permission assignment

Consultant: {M1, M2, M3, M4, M5, …}
Associate consultant: {M1, M2, M3, M4, …}
Principal: {M1, M2, M3 , M12, …}
Residency: {M1, M2, M12, …}

User-role assignment

Peter: {Consultant, …}
Mary : {Associate consultant, …}
Alice: {Principal, …}
Bob: {Residency, …}

Residency doctor workstation task definition

T1: {M1, M2, M3, M4, M5}

Medicine department group 2 definition

Medicine department group 2 users: {Peter, Mary, Alice, Bob, …}
Medicine department group 2 roles: {Consultant, Associate consultant, Principal, Residency}
Medicine department group 2 tasks: {T1}

Medicine department group 2 user permissions

Peter: {M1, M2, M3, M4, M5}
Mary: {M1, M2, M3, M4}
Alice: {M1, M2, M3}
Bob: {M1, M2}

Figure 8.5: Example of permission assignments in TT-RBAC.

From the previous description, we show that the TT-RBAC model can basically replace the original group-based approach. The replacement is summarized as follows:

1. Groups are replaced by teams.
2. Group members now are team members.
3. Subgroups are replaced by roles that are assigned to both users and teams.
4. Permissions assigned to groups are organized into tasks that are assigned to teams.
5. Capability lists used in subgroups can be implemented through defining some new roles.

Using roles may not be as flexible as using capability lists. But role-based approach can dramatically improve administration when there are many users involved. Moreover, through carefully analyzing business processes we can define a limited role set and assign multiple roles to users to obtain the flexibility provided by capability lists. The major benefits of using TT-RBAC compare to using group-based approach are described as follows.

1. Compare to the subgroups, roles can be used in the range of whole system or beyond it.
2. Users can join in different teams in one subsystem.
3. Users can be assigned to multiple roles, and activate them in different teams. Thus, they have different privileges in different teams.
4. Assigning tasks to a team is much easier and less error-prone than selecting permissions from the set of all permissions.
5. Modifying the permissions assigned to team tasks will affect the permissions that are available to the team members. So we do not need to change users' privileges one by one.
6. Team roles are dual approach to team tasks to affect the permissions that are available to the team members. Modifying team roles may only affect part of the team members. Changing team tasks will affect all the team members. In the above example, all the four roles have the permission $M1$. If we remove the $M1$ from the team tasks, then $M1$ will be unavailable to all the team members. If we remove the *Residency* role from the team roles, then this change will only affect *Bob*, whereas other team members can still perform $M1$.

8.1.4 Supporting dynamic collaboration with TT-RBAC

Collaboration is the general requirement in hospital information systems. In CHIS the group is used to encapsulate a collection of hospital staffs with the objective of caring a set of patients. The subgroups are used to classify the staffs who have different responsibilities in a given group. We call these kinds of collaboration as *static collaboration*. In hospital environment, there are some other kinds of collaboration. For example, a patient's primary doctor may consult other doctors for the diagnosis and treatment. In this case, the primary doctor needs to decide what kinds of patient information should be shared and with whom. In fact, the primary doctor creates a team in which the team members have the privilege of accessing the patient's medical record and provide their advices. In this case the team should be created dynamically without the help of system administrators. Here the team members and team tasks are created at runtime. We call this kind of collaboration as *dynamic collaboration*.

For the dynamic collaboration, the CHIS adopts a mechanism that is similar to an email system to deal with the collaboration among different groups. These messages can be sent to the selected doctors or groups. If the message is sent to a group, then any doctors belonging to this group can view and answer this message. If the message is sent to a user, then only this user can view and answer this message. When a patient's primary doctor wants to get some advices from other doctors, he creates an advice request, attaches the patient's medical record to the advice request, and then selects the doctors or groups from the doctor or group lists, to which the message will be sent. The answers are also sent as messages. Based on the answer messages, the patient's primary doctor makes the finial decision. The merit of this approach is simple and easy to implement. But there are some shortcomings. (1) Because the team is not defined explicitly, there is no interaction among the team members. They cannot discuss each other. There is also no way to ask for some extra information, except making a phone call or talking face to face. (2) The primary doctor does not have the right to decide what kind of information should be shared with his/her colleagues. It has been hard coded in the program. It may cause the problems of exposing too much/less information to other doctors. (3) The primary doctor does not have any control about who can see and answer a message when it is sent to a group.

Now we investigate how this application scenario is implemented with TT-RBAC model. Its process could be described as follows.

1. A patient's primary doctor creates a temporary team for discussing the patient's case. He assigns the patient to the team and writes his requirements. In fact, this doctor is acting a team administrator.
2. He selects the doctors or other teams as the team members.
3. If he wants only the doctors with role *Consultant* or *Associate consultant* can view the patient's case and provide advice, then he adds these roles as the team roles. Otherwise there is no such limitation, and all the team members are valid.
4. The primary doctor can also decides how much patient's information can be released through defining team tasks. But he cannot release the information beyond he can view.

Compare to the message-based approach, the major benefits of using TT-RBAC are described as follows.

1. In message-based approach, the patient's information is encapsulated into messages. Once these messages are sent out, they are out of control. In TT-RBAC approach, the system always keeps the control power. For example, based on privacy policy, some information may be blocked to some people whose position is lower than a predefined level. The team administrator (primary doctor) may not be familiar to with these policies.
2. In message-based approach, the amount of patient's information that can be sent is limited, and the information is not organized in a good format. But in TT-RBAC approach, the entire patient's history record could be available to the team members, and the information can be presented through a well designed interface.
3. In message-based approach, the sent messages cannot be withdrawn. Adding new information can only be done through resending messages. These will give the senders and receivers management burdens. In TT-RBAC approach, the team administrator can easily add/remove team members, roles and tasks. These changes can take into effects at once.
4. In message-based approach, the sender cannot decide when the answer messages come. Some messages may come too late. In TT-RBAC approach, once a team is closed, then all the relevant information will not be available to the team members.

8.1.5 Conclusion

In summary, the TT-RBAC model can replace the group-based authorization system very well and, in the same time, also provide many extra advantages. On the other hand, the TT-RBAC can well support the dynamic collaborative activities happened in hospital scenarios.

8.2 Case study: Authorization constraints in a MIS system

We have applied the function-based authorization constraints in two real application systems. One is the "CASC Material Management Information System Integration Tool". The other is the "CASC Material Management Information System". Both of them are from a R&D project that cooperates with the Beijing Shenzhou Aerospace Software Technology Co., Ltd (www.bjsasc.com) that belongs to the China Aerospace Science and Technology Corporation (CASC). In this section we show how the function-based authorization constraints are applied in a management information system. In next section we will show how the function-based authorization constraints are integrated into a business process management system.

8.2.1 Background

The "CASC Material Management Information System Integration Tool (CASC-MMISIT)" is developed by the Beijing Shenzhou Aerospace Software Technology Co., Ltd and used by the group's subsidiary companies and institutes. The CASC-MMISIT manages many subsystems, such as *public basic data management*, *supplier management*, *material management* and so on. Currently, each subsystem has its own authentication and authorization systems. They adopt role-based access control mechanism. Each subsystem manages its own users and roles. But these subsystems are not totally independent. There are data exchanges among of them. For example, one user is filling a purchase contract through the subsystem *material management*, and finds that one purchase item cannot be classified into any existent material category. In this case, he/she has to log in to the subsystem *public basic data management* in order to apply a temporary material code for this item. It is possible for a user who has different usernames and passwords in different subsystems. So this situation brings the remembrance burden to the users and the administration burden to the administrators. It also brings the problem in log management. The log records the information about who has done what job at what time. On the other hand, the data exchanges among these subsystems are not convenient. The integration tool is used to resolve these problems. Its major functions are described as follows.

1. The integration tool will provide *single sign on* function for all the subsystems. The authorization mechanism is RBAC. Subsystems will not do user authentication anymore, but still keep their own authorization system.
2. Data exchange among subsystems must go through the integration tool.
3. The integration tool also provides comprehensive functions for log management.

8.2.2 Authorization constraint requirements

Because their businesses are special and sensitive, information security is one of the important issues that must be considered. In the system RBAC is applied at two levels. One is at level of integration tool; the other is at the level of subsystems. At the level of integration tool, RBAC decides who can access which subsystems. At the level of subsystem, RBAC decides who can do what functions in a subsystem. The relations among integration tool, subsystems and subsystems' functions are shown in Figure 8.6.

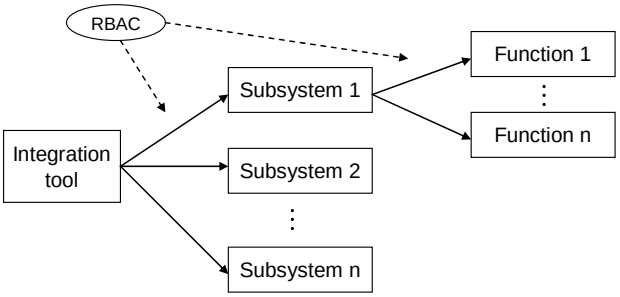


Figure 8.6: Relation between the integration tool and subsystems.

For the integration tool, there are two major security concerns in privilege management. One is that the administrative power should be distributed to several persons so that the damage caused by either a single member's mistake, an accident or deception can be avoided or

minimized and the accountability can be enforced. For this purpose, some predefined administrative roles are defined and should be assigned to different users. There are also some cardinality constraints to these roles. For example, a role must be assigned to two users.

Because this information system is used to manage material's purchase, storage and distribute in a very special business. Many access control constraints need to be performed. For example, a purchase contract should not be initialized and approved by the same person. Due to RBAC being used and the requirement of fine-grained access control, many roles are defined. For example, some users can only approve the specified purchase items. When the role number becomes large, how to avoid the conflict permissions being assigned to the same user becomes a big issue. The constraints should be considered are that the mutually exclusive subsystems are not assigned to the same role, the mutually exclusive roles are not assigned to the same user, and the mutually exclusive subsystems are not assigned to the same user. The integration tool does not provide any mechanism to detect these authorization constraints. It totally depends on the administrators to keep an eye on these constraint rules. It gives a heavy burden to the administrators and also gives the possibility for them to misuse their power. So it would be better to have a mechanism that can automatically detect any unsuitable privilege assignment.

In a large corporation, there are many subsidiary companies and institutes, and their employee number ranges from several decades to several thousands. In some organizations a user may take less job functions, and in other organizations a user may take more job functions. It means that the role number and constraint rules in these organizations are different. Thus, it is impossible for these constraint rules to be hard coded into the system.

This issue is addressed by our function-based authorization constraints. In this project we developed an *authorization constraint module* that is invoked by the integration tool. All the user-role assignments and permission-role assignments will be checked by the module according to some authorization constraint rules that are written in XML. With the help of this authorization constraint module, any intentional or unintentional privilege assignment that causes a violation of an authorization constraint will be avoided. We will introduce how the constraint module is integrated into the integration tool in the following subsections.

8.2.3 Data structures and functions

The integrating tool uses ORACLE database to manage its data. The tables related to the user-role assignments and role-permission assignments and their relations are shown in Figure 8.7. These tables' functions are described as follows.

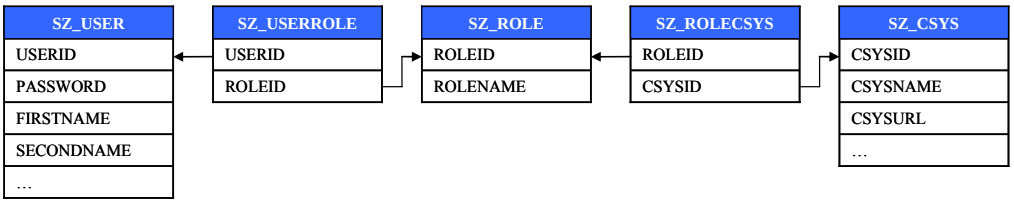


Figure 8.7: Tables related to authorization constraints in integration tool.

- *SZ_USER* stores user information, such as username, password, session-time and so on.
- *SZ_ROLE* stores role information, such as role id, name and so on.
- *SZ_CSYS* stores subsystem description such as system id, system url and so on.
- *SZ_USERROLE* stores user-role assignments. One user can be assigned to multiple roles.

- *SZ_ROLECSYS* stores role-permission (subsystem) assignments. One role can be assigned to multiple subsystems.

According to the description of Section 8.2.2, we summarize the authorization constraints needed by the integration tool as follows.

- Mutually exclusive roles cannot be assigned to the same user.
- Mutually exclusive permissions cannot be assigned to the same role.
- Mutually exclusive permissions cannot be assigned to the same user.
- There are cardinality constraints to some roles.
- There are dependent relations in user-role assignments and role-permission assignments.

So we need both prohibition constraint schemes and obligation constraint schemes for expressing these constraints. Next we analyze which entity set functions and entity relation functions should be defined.

In this application scenario, three types of entities are involved. They are users, roles and permissions. The scope sets and constraint sets defined in constraint schemes are different from one to another, so there may be many set functions. For example, if a constraint scheme is used to specify user-role assignments in the whole organization, then there is a set function that is used to get all the users of the organization. If a constraint scheme is only used to specify user-role assignments in the financial department, then there is a set function that is used to get all the users who belonging to the financial department. In a given constraint scheme, an entity set can be expressed with a set function or through directly enumerating all the involved entities. The entity set and relation functions defined for the integration tool are listed in table 8.1.

Function name	Implementation	Functionality
getSZUsers	Select id from sz_user	Get all users from the table sz_user
getSZRoles	Select id from sz_role	Get all roles from the table sz_role
getSZSubsystems	Select id from sz_csys	Get all subsystems from the table sz_csys
getSZUserRoles	Directly get the roles assigned to a given user from the request context.	Get all roles that will be assigned to a given user from request context
getSZRoleUsers	Select roleid from sz_userrole where roleid=role-id	Get all users assigned to a given role from the table sz_userrole
getSZRoleSubsystems	Directly get the subsystems assigned to a given role from the request context.	Get all subsystems that will be assigned to a given role from request context
getSZSubsystemRoles	Select roleid from sz_rolecsys where csysid=csys-id	Get all roles assigned to a given subsystem from the table sz_rolecsys
getSZUserSubsystems	Select sz_rolepurv.csysid from sz_user, sz_role, sz_userpurv, sz_rolepurv where sz_userpurv.userid=sz_user.id and sz_role.id=sz_rolepurv.roleid and sz_userpurv.roleid=sz_rolepurv.roleid and sz_userpurv.userid=user-id	Get all subsystems assigned to a given user from the table sz_rolecsys

Table 8.1: Entity set and relation functions in case study 2.

In table 8.1, functions *getSZUserRoles* and *getSZRoleSubsystems* get data directly from the request context rather than from the database. *getSZUserRoles* returns all the roles that will be assigned to a given user, and *getSZRoleSubsystems* returns all the subsystems that will be

assigned to a given role. In this case, if the data was gotten from database, it would bring the sequence related problem. For example, if *role*1 is the prerequisite role of *role*2, then the *role*1 must be written into to the database before *role*2, even if both of them are in the same role sets that will be assigned to a given user.

Now we give a concrete example of authorization constraints used for the predefined administrative roles in the integration tool. There are four predefined administrative roles for the integration tool. They are described as follows.

- *SuperAdministrator* has all the permissions of the integration tool.
- *SystemAdministrator* is responsible for system configuration, such as defining organization structure, departments and subsystems.
- *LogAdministrator* has the permission of reading the log except the log related to him/her. The log contains all the information of users' login, logout and their operations.
- *SecurityAdministrator* is responsible for managing users, roles, permissions and their assignments. He/she can also see the log related to the role *LogAdministrator*.

The role *SuperAdministrator* and its privileges are hard coded into the system. The user-role assignments related to other administrative roles are policy-based. These administrative roles are mutual exclusive and each of them can only be assigned to less than three users. These authorization constraints expressed in our constraint schemes are given as follows.

1. ((*getSZUsers*, , ,), ({*SystemAdministrator*, *LogAdministrator*, *SecurityAdministrator*}, *getSZUserRoles*, <, 2), *SPC*)
2. ((*getSZUsers*, *getSZRoleUser*, <, 3), ({*LogAdministrator* }, *getSZUserRoles*, <, 2), *SPC*)
3. ((*getSZUsers*, *getSZRoleUser*, <, 3), ({*SystemAdministrator*}, *getSZUserRoles*, <, 2), *SPC*)
4. ((*getSZUsers*, *getSZRoleUser*, <, 3), ({*SecurityAdministrator*}, *getSZUserRoles*, <, 2), *SPC*)

8.2.4 Integrating with the administration tool

We developed an *authorization constraint module* that is invoked by the integration tool through its API. All the user-role assignments and role-permission assignments will be checked by the module according to some authorization constraint rules. This module is developed in Java and invoked by the Java programs that are responsible for saving user-role assignments and role-permission assignments. These programs create constraint requests, and then pass them to the authorization constraint module for authorization constraint checks. If a constraint check is failed, the saving process will be stopped and an error message is returned. The return value from the module is either the "0" representing "Permit" or a constraint scheme id showing which constraint scheme is violated. The code segment for invoking authorization constraint module is shown as follows.

```
String[] boxid = request.getParameterValues("checkbox"); //subsystems
String roleID = request.getParameter("id"); //role
//create constraint request
SZConstraintRequest constraintRequest = new SZConstraintRequest();
//create constraint request subject
CRObject requestSubject = new CRObject(roleID, "ROLE");
constraintRequest.setSubject(requestSubject);
//create constraint request objects
ArrayList objectSet = new ArrayList();
if(boxid.length>0){
```

```
  for(int k = 0;k<boxid.length;k++){
    CRObject requestObject = new CRObject(boxid[k], "PERMISSION");
    objectSet.add(requestObject);
  }
}
constraintRequest.setObjects(objectSet);
//create constraint request action
CRObject requestAction = new CRObject("ROLE-PERMISSION-ASSIGNMENT", "ACTION");
constraintRequest.setAction(requestAction);
//create request context
Hashtable requestContext = new Hashtable();
ArrayList subsystems = new ArrayList();
for(int k = 0;k<boxid.length;k++){
  subsystems.add(boxid[k]);
}
requestContext.put(roleID, subsystems);
constraintRequest.setContext(requestContext);
//initialize a constraint schema instance
SZConstraintSchema constraintSchema = new SZConstraintSchema(xmlpath);
//check the constraint request
String checkResult = constraintSchema.checkConstraintSchema(constraintRequest);
```

The following codes show the implementation of functions *getSZRoles* that gets all the roles from the database and *getSZRoleSubsystems* that gets all the subsystems that will be assigned to a given role from a request context.

```
public Collection getSZRoles(SZConstraintRequest request){
  Hashtable context = request.getContext();
  SZDBAO dbao = (SZDBAO)context.get("Database");
  Collection roles = dbao.getSZRoles();
  return roles;
}

public Collection getSZRoleSubsystems(SZConstraintRequest request, CRObject role){
  Hashtable context = request.getContext();
  String roleID = role.getID();
  Collection subsystems = (Collection)context.get(roleID);
  return subsystems;
}
```

In this project the constraint schema is written in XML. The following example shows a constraint schema that contains the first constraint scheme given in the Section 8.2.3.

```
<?xml version="1.0" encoding="UTF-8" ?>
<ConstraintSchema ID="SZ_Constraint_Schema_1">
 <ConstraintPCs>
  <ConstraintPC ID="SZ_User_Role_Assignment_Constraint_Rule_1" Type="SPC">
   <ScopeElement>
    <ScopeSetType>USER</ScopeSetType>
    <ScopeSet>
     <SetFunction>getSZUsers</SetFunction>
    </ScopeSet>
   </ScopeElement>
   <ConstraintElement>
    <ConstraintSetType>ROLE</ConstraintSetType>
    <ConstraintSet>
     <SetElement>SystemAdministrator</SetElement>
     <SetElement>LogAdministrator</SetElement>
     <SetElement>SecurityAdministrator</SetElement>
    </ConstraintSet>
    <ConstraintRelationFunction>getSZUserRole</ConstraintRelationFunction>
    <ConstraintOperator>Less</ConstraintOperator>
    <ConstraintCardinality>2</ConstraintCardinality>
   </ConstraintElement>
```

```
    </ConstraintPC>
  </ConstraintPCs>
</ConstraintSchema>
```

A screenshot of the role-permission assignment interface with a constraint check failed message window is shown in Figure 8.8.

Figure 8.8: Screenshot of a role-permission assignment in integration tool.

8.2.5 Discussion and conclusion

When a technology is used in the real world, the efficiency must be considered. In our approach calculating scope relation functions may bring efficiency problems. For example, if the scope relation function is *getUserRoles* and the constraint set function is *getUsers*, then we need to calculate *getUserRoles*(*getUsers*). If the user number is quit large, then the calculation may need to read database many times. It is unacceptable. A possible solution for handling this situation is that we can treat *getUserRoles*(*getUsers*) as one operation. First we get its executing result in some way and save this result. Next time, when the system needs to execute this operation, it directly uses this result instead of recalculating it. When there is a change to the user-role assignment, we change this result directly. Because the execution is hidden inside of the function *getUserRoles*, the system does not know the difference.

In our current implementation, most of the constraint schemes do not have scope set relation functions, e.g. the first constraint scheme given in the Section 8.2.3. For each constraint scheme, system only needs to read database two times, i.e. executing the scope set function and constraint set function. Only several schemes that have cardinality constraints to the constraint set have scope relation functions. But, in these constraint schemes, there is only one element in the constraint set, e.g. the second constraint scheme given in the Section 8.2.3. So the execution of scope relation function only needs to read database one time. It does not have obvious effect to the system efficiency.

After adding the authorization constraint module, the integration tool gets three major benefits. (1) Once the necessary entity set and relation functions are developed, expressing and enforcing various authorization constraints can be done only through modifying some constraint schemes specified in XML files. This caters the requirement that different organizations need different authorization constraint rules. (2) An authorization constraint system is extensible through adding new entity set and relation functions. This extension does not have any influence to the existing system and enforcement mechanism. (3) Once constraint schemas are configured into the system, any privilege assignment through the integration tool is monitored, and then the intentional and unintentional privilege assignments that cause a violation of an authorization constraint will be avoided.

8.3 Case study: Authorization constraints in a BPM system

8.3.1 Background

The "CASC Material Management Information System (CASC-MMIS)" is the major subsystem managed by the "CASC Material Management Information System Integration Tool". The system manages the materials' whole lifecycle, such as planning, purchase, storage, distributing, and so on. As the introduction in Section 8.2, this system will be used by the group's subsidiary companies and institutes. Because these organizations' business and structure are different, their business processes are different. For example, for the purchase contract management workflow, some organizations may only need one person to do the "examine and approve" task. Some organizations may need several persons to do the "examine and approve" task in one workflow step, and some organizations may need more several persons who have different roles to do the "examine and approve" tasks in different workflow steps. Even in the same organization, different types of purchase contracts may need different handling processes. On the other hand, these business processes may be removed or modified, and new processes may also be added. So the traditional workflow management system cannot handle this situation very well.

In order to cater these requirements, the CASC-MMIS adopts a BPM(Business Process Management)-based product named AWS BPM platform (http://www.actionsoft.com.cn) as its application developing platform. This product provides many application models and comprehensive functionalities for workflow management. But there are still some complex applications that cannot be directly implemented with AWS, e.g. authorization constraints in workflow. In the next several subsections we first introduce the differences between BPM and workflow, and then give a brief view of AWS BPM platform. At last we investigate how the function-based authorization constraint mechanism is integrated with this BPM product.

8.3.2 Comparison between workflow and BPM

Here we briefly introduce what is the difference between workflow and BPM [Crosman].

Workflow is concerned with the application-specific sequence of activities via predefined instruction sets, involving either or both automated procedures (software-based) and manual activities (people work). Integration between workflow systems and externalities are comparatively limited, often only allowing the retrieval of documents or data variables and only serving as a pass-through with no awareness of content.

BPM is concerned with the definition, execution and management of business processes defined independently of any single application. BPM is a superset of workflow, further differentiated by the ability to coordinate activities across multiple applications with fine-

grained control. BPM systems allow both the capture and introspection of external documents and data, presenting a closed-loop process for validating the integrity of transactions, data and content, as well as the initiation of compensating activities when necessary.

BPM processes separate execution instructions from process flows. Thus, routing can be tied to process outcomes and milestones. As workflow processes are tied to single applications, process flow is hardwired and accommodates alternative means for reaching the same task or goal. Distilled into single-word definitions, workflow is about repetition and BPM is about coordination (also automation and orchestration, respectively).

8.3.3 AWS BPM platform introduction

AWS BPM platform developed by ActionSoft (http://www.actionsoft.com.cn) is a BPM-based application platform. The ActionSoft is one of the leading companies in providing commercial BPM products in China. The AWS platform architecture is shown in Figure 8.9. AWS based on JAVA/J2EE provides integrated BPM developing, operation and maintenance platforms. It is model-driven-based multi-layer internet application service infrastructure.

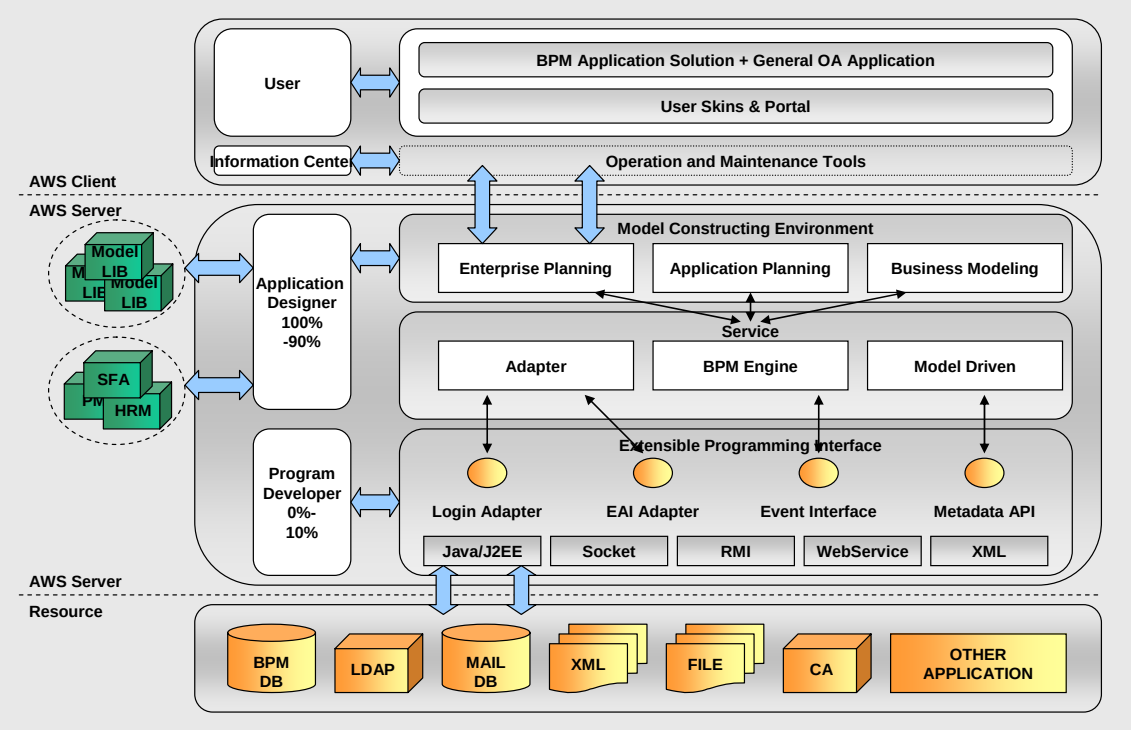


Figure 8.9: AWS BPM platform architecture.

The AWS platform provides integrated tools for model application installation, design, deployment, Optimization and unloading. It supports GUI-based design, execution, maintenance and optimization for organization, repository, form, workflow, report and navigation models. It also provides convenient model import/export tool.

Currently, the application models supported by AWS are Enterprise Asset Management (EAM), Project Management (PM), Sales Force Automation (SFA), Distribution Resource Planning (DRP), Human Resource Management and Office Automation (OA). According to the complexity of the requirement, the platform model constructing tool can reach over 90% zero

coding. Through the extensible programming interface, the AWS platform can meet the special requirements in system integration, complex business process and security.

Though the AWS platform provides comprehensive and flexible model description approach for various complex workflows, there are still some business logics cannot be directly implemented. In this case, the AWS developing interface can be used to implement these business logics. The AWS platform provides various event triggers for invoking interface codes so that the system can automatically invoke these codes when some events happen. The trigger that we use to add the authorization constraints is *AWS workflow node trigger*. As shown in Figure 8.10, the trigger permits AWS developers register the external Java codes to the actions of a given node. Once an action is triggered, the registered Java code will be automatically executed. These Java codes are called *BizEvent*. The major events that can be triggered by the workflow nodes and their functions are listed in Table 8.2.

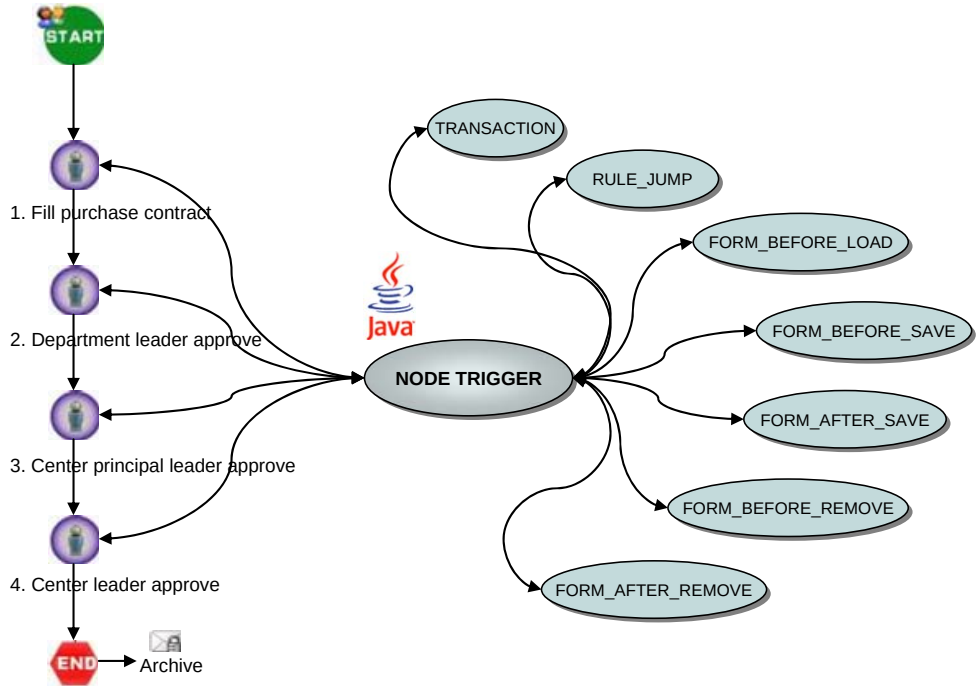


Figure 8.10: AWS workflow node BizEvent.

In order to automatically search for the operators for different nodes in a workflow, the workflow designers need to set a route policy for each node. The AWS platform predefines many route schemes. For example, route *scheme*1 permits current node's operator selects an operator for the next node, route *scheme*2 permits workflow designers appoint a node operator at design time, route *scheme*3 permits the system automatically looks for a valid operator according to users' roles, and so forth. All the route schemes do not consider any authorization constraint. Considering the workflow example shown in Figure 8.12, if all the nodes are set with route *scheme*3, then one user who has all these roles can perform all four steps in a workflow instance. In many cases, this should be avoided. This is the major motivation of integrating our function-based authorization constraints into the CASC-MMIS.

Event	Function
TRANSACTION	Executed before a task is passed to next step. Return false, then prevent this task be executed further.
TRANSACTION_AFTER	Executed after a task is passed to next step.
FORM_BEFORE_LOAD	Executed before a form is loaded.
FORM_AFTER_LOAD	Executed after a form is loaded.
FORM_BEFORE_SAVE	Executed before saving the information of a form.
FORM_AFTER_SAVE	Executed after saving the information of a form.
FORM_BEFORE_REMOVE	Executed before deleting the information of a form.
FORM_AFTER_REMOVE	Executed after deleting the information of a form.
RULE_JUMP	Executed adding external routing rules for next node and node operator.

Table 8.2: Events triggered by workflow nodes.

8.3.4 AWS BPM transaction trigger

There are two ways to integrate the authorization constraint mechanism into the AWS BPM application systems. One is through the RULE_JUMP event trigger, the other is through the TRANSACTION event trigger. In our current implementation, we use the transaction trigger that is mainly used to implement the business requirements brought by the operation of a workflow node. The major tasks that can be done through the transaction trigger are:

- Execute some Java codes before entering next workflow step,
- Implement complex data valid check and decide if the operation is permitted,
- Exchange data with external systems, or start a new workflow.

All the triggers must inherit from some predefined AWS classes. The transaction triggers inherit from the class *com.actionsoft.loader.core.WorkFlowStepRTClassA*. These trigger classes override the *execute* method defined in the superclass. The *execute* method returns a Boolean value. The transaction trigger executive sequence is shown in Figure 8.11.

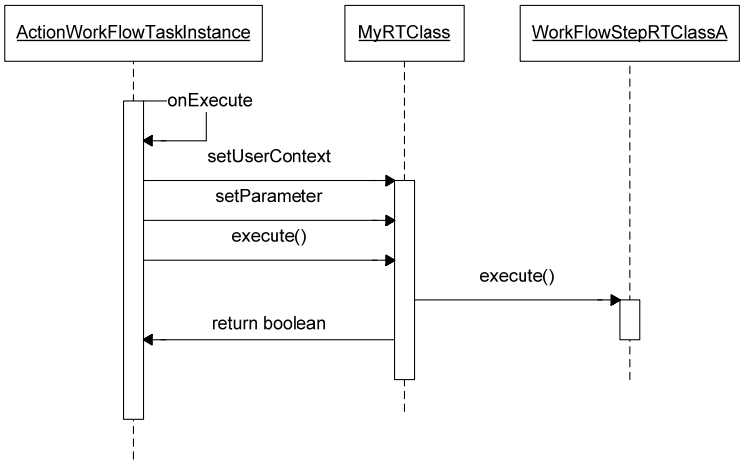


Figure 8.11: Transaction trigger executive sequence.

If the *execute* method returns false, the AWS will prevent a task instance from being passed into next node. The actions of the transaction trigger are shown in Figure 8.12.

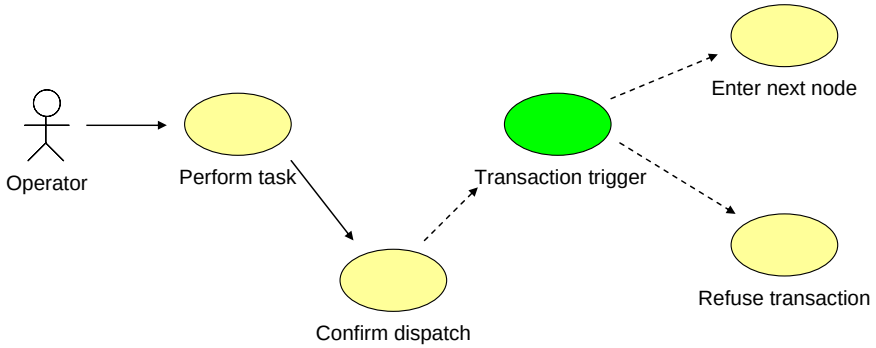


Figure 8.12: Actions of transaction trigger.

8.3.5 Integrating with the AWS BPM platform

In this subsection we use the workflow *purchase contract management* shown in Figure 8.10 to demonstrate how the authorization constraint mechanism is integrated into the AWS BPM application systems. This workflow is composed of *fill purchase contract*, *department leader approve*, *center principal leader approve* and *center leader approve* four steps. We assume that all the nodes are set with route *scheme*3, i.e. the node operators are automatically selected by the system according to users' roles. We also assume that the business rules prohibit that one user perform more than one step in any workflow instance.

In this project our major target is to enforce Separation of Duty (SoD) policy among the tasks (steps) in workflow environment. These SoD policies are expressed by the historical prohibition constraint schemes. For example, the following scheme states that no user defined in the scope set specified by the entity set function *getWorkflowUsers* can perform more than one task defined in the constraint set $\{t_1, t_2, t_3, t_4\}$ in any workflow instance. The entity relation function *getUserWorkflowInstanceTasks* is used to get all the tasks assigned to a given user in a workflow instance.

((getWorkflowUsers, , ,), ($\{t_1, t_2, t_3, t_4\}$, getUserWorkflowInstanceTasks, <, 2), HPC).

The AWS can adopt ORACLE, SQLServer or MYSQL as its database. The tables related to authorization constraints and their relations are shown in Figure 8.13. These tables' functions are described as follows.

- *ORGUSER* stores user information managed by the AWS platform, such as user id, password, role and so on.
- *ORGROLE* stores roles used by the AWS platform and the application systems.
- *SYSFLOW* stores workflow general information, such as workflow id, name and so on.
- *SYSFLOWSTEP* stores what steps a workflow is composed of and their definitions. The step definition contains the workflow id, step id, step number, name, user, role, route scheme and so on.
- *WF_TASK* stores all the dispatched tasks that need to be handled. Each task description contains workflow instance id (bind_id), person dispatching this task (owner), person

performing this task (target), workflow id, step id, and some other information, such as expire time.

- *WF_TASK_LOG* stores all the completed tasks. Its major fields are the same as the table *WF_TASK*. A task is moved from *WF_TASK* into this table only when this task is completed and the workflow instance is moved into next step.

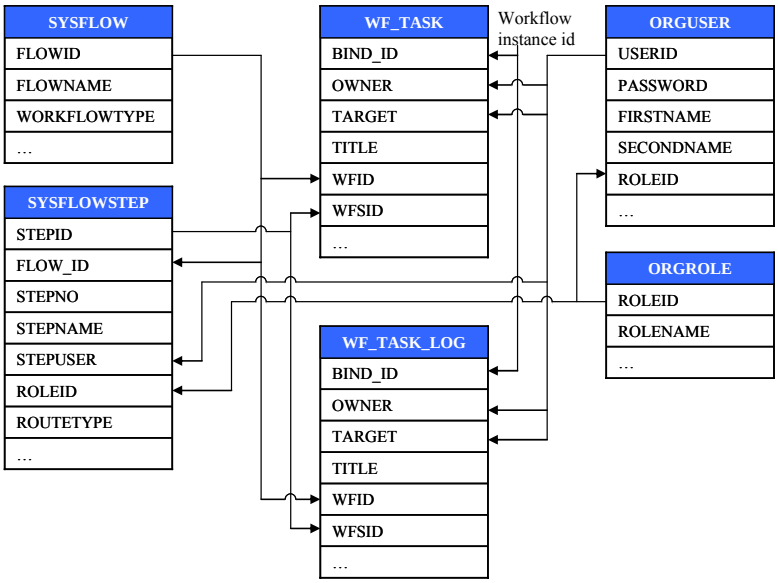


Figure 8.13 Tables related to authorization constraints in AWS.

Now we investigate how the entity set and relation functions used in the above constraint scheme example are designed. We do not use the table *SYSFLOWSTEP* to obtain valid user set, because the *stepuser* filed is null except that the node operators are directly designated. Here we assume that all the users defined in the table *ORGUSER* are valid for this constraint scheme. In this authorization constraint check, the constraint scheme needs to make a constraint decision according to a workflow instance's executive history, i.e. what tasks have been executed and by whom. So, for the entity relation function, the related information is stored in the table *WF_TASK_LOG*. The entity set and relation functions defined for the AWS platform are listed in table 8.3.

Function name	SQL statement	Functionality
getWorkflowUsers	Select userid from orguser	Get all users from the table orguser
getWorkflowInstanceUserTasks	select wfsid from wf_task_log where bind_id = + workflowInstanceID + and target = + userID	Get all the tasks performed by a given user in a workflow instance from the table wf_task_log

Table 8.3: Entity set and relation functions in case study 3.

In this project we developed an *authorization constraint monitor* that is responsible for enforcing authorization constraints in workflow environments. Each constrained request that could potentially cause a violation to an authorization constraint rule is passed to the constraint monitor. The constraint monitor checks whether granting the request would violate an authorization constraint rule. The authorization constraint monitor is integrated into the

workflow systems through AWS developing interface. The authorization constraint rules are written in XML. Once the necessary entity set and relation functions are developed, various authorization constraints can be created and enforced. With the help of authorization constraint monitor, various historical authorization constraints, such as Separation of Duty (SoD) and Binding of Duty (BoD), among users, roles and tasks in workflows can be realized.

The following codes shows a BizEvent designed for the *purchase contract management* workflow. Its major functions are creating a constraint request, invoking the authorization constraint monitor, and then taking an action according to the check result. The return value of the monitor is either the "0" representing "Permit" or a constraint scheme id showing which constraint scheme is violated. If this BizEvent return the value of "true", the transaction will continue, otherwise be cancelled together with an error message.

```
public class SZConstraintCheck extends WorkFlowStepRTClassA {
  public SZConstraintCheck(UserContext uc) {
    super(uc);
    setVersion("1.0.1");
    setDescription("Check if the user-task assignment conforms to the authorization
constraints.");
  }

  public boolean execute() {
    //create constraint request
    SZConstraintRequest constraintRequest = new SZConstraintRequest();
    //create request context
    Hashtable requestContext = new Hashtable();
    String
workflowInstanceID=String.valueOf(this.getParameter(this.PARAMETER_INSTANCE_ID).toInt());
    requestContext.put("WorkflowInstanceID", workflowInstanceID);
    constraintRequest.setContext(requestContext);
    //create constraint request subject
    String userID=this.getUserContext().getUID();
    CRObject requestSubject = new CRObject(userID, "USER");
    constraintRequest.setSubject(requestSubject);
    //create constraint request objects
    ArrayList objectSet = new ArrayList();
    String workflowStepID =
String.valueOf(this.getParameter(this.PARAMETER_WORKFLOW_STEP_ID).toInt());
    CRObject requestObject = new CRObject(workflowStepID, "TASK");
    objectSet.add(requestObject);
    constraintRequest.setObjects(objectSet);
    //create constraint request action
    CRObject requestAction = new CRObject("USER-TASK-ASSIGNMENT", "ACTION");
    constraintRequest.setAction(requestAction);
    //initialize a constraint schema instance
    SZConstraintSchema constraintSchema = new SZConstraintSchema(xmlpath);
    //check the constraint request
    String constraintCheckResult =
constraintSchema.checkConstraintSchema(constraintRequest);
    if (!constraintCheckResult.equalsIgnoreCase("0"))
{       MessageQueue.getInstance().putMessage(super.getUserContext().getUID(), "User-
Task assignment violates constraint rule: ["+ constraintCheckResult + "]");
      return false;
    } else {
      return true;
    }
  }
}
```

The following codes show the implementation of functions *getWorkflowUsers* that gets all the users who are valid for workflow operation and *getWorkflowInstanceUserTasks* that gets all the tasks that has been done by a given user in a workflow instance.

```
public Collection getWorkflowUsers(SZConstraintRequest request){
  Hashtable context = request.getContext();
  SZDBAO dbao = (SZDBAO)context.get("Database");
  Collection users = dbao.getWorkflowUsers();
  return users;
}

public Collection getWorkflowInstanceUserTasks(SZConstraintRequest request) {
  Hashtable context = request.getContext();
  SZDBAO dbao = (SZDBAO)context.get("Database");
  String workflowInstanceID = (String)context.get("WorkflowInstanceID");
  String userID = request.getSubject().getID();
  Collection tasks = dbao.getSZUserWorkflowTasks(userID, workflowInstanceID);
  return tasks;
}
```

The following example shows an authorization constraint schema that contains one constraint scheme discussed in this subsection. The “11239”, “11250”, “35777” and “35778” are the workflow step ids representing the nodes of *fill purchase contract*, *department leader approve*, *center principal leader approve* and *center leader approve*, respectively.

```
<?xml version="1.0" encoding="UTF-8" ?>
<ConstraintSchema ID="SZ_Constraint_Schema_2">
 <ConstraintPCs>
  <ConstraintPC ID="SZ_User_Task_Constraint_Rule_1" Type="HPC">
   <ScopeElement>
    <ScopeSetType>USER</ScopeSetType>
    <ScopeSet>
     <SetFunction>getWorkflowUsers</SetFunction>
    </ScopeSet>
   </ScopeElement>
   <ConstraintElement>
    <ConstraintSetType>TASK</ConstraintSetType>
    <ConstraintSet>
     <SetElement>11249</SetElement>
     <SetElement>11250</SetElement>
     <SetElement>35777</SetElement>
     <SetElement>35778</SetElement>
    </ConstraintSet>
<ConstraintRelationFunction>getWorkflowUserTasks</ConstraintRelationFunction>
    <ConstraintOperator>Less</ConstraintOperator>
    <ConstraintCardinality>2</ConstraintCardinality>
   </ConstraintElement>
  </ConstraintPC>
 </ConstraintPCs>
</ConstraintSchema>
```

The program SZConstraintCheck is organized into package com.bjsasc.constraint and registered to the node event trigger TRANSACTION at nodes *department leader approve*, *center principal leader approve* and *center leader approve* (step2, step3, step4). A screenshot of registering the authorization constraint monitor to the node center leader approve is shown in Figure 8.14.

At runtime, once an operator submits his/her operation to the system, a constraint request is created and passed to the constraint monitor registered at the node that this operator is operating. The constraint monitor checks whether granting the request would violate an authorization constraint rules and returns the check result. Only when the monitor returns the value of “0”, this transaction can be completed, otherwise be cancelled together with an error message window.

Figure 8.14 Screenshot of event trigger registration in AWS.

8.3.6 Conclusion

This case study shows how the function-based authorization constraint mechanism is integrated into a commercial BPM product. After integrating the authorization constraint monitor into the AWS BPM Platform, the CASC-MMIS becomes a system that supports policy-based authorization constraints. Even there is only one relation function being defined, this authorization system can support various authorization constraints at workflow instance level, such as separation of duty and binding of duty, among users and workflow steps. Adding authorization constraints to a given node is quite simple. What we need to do is registering an authorization constraint monitor to a workflow node that needs authorization constraints. The constraint schemas are writing in XML.

Chapter 9

Conclusions and future work

As more businesses engage in globalization, inter-organizational collaborative computing is becoming important. In collaborative systems, a set of organizations share their computing resources, such as computer cycles, storage space, or online services, to establish virtual organizations aiming at achieving a particular task. In order to effectively participate in modern collaborations, member organizations must be able to share specific data and functionality with collaboration partners, while ensuring that their resources are safe from inappropriate access. This requires access control models, policies, and enforcement mechanisms for collaboration resources. This thesis reviewed several research issues in this area. They are listed as follows.

- Access control model that caters the requirements for access control in collaborative environments.
- Authorization constraints that could be used for both expressing and enforcing authorization constraints.
- How to enhance and simplify the access control policies used in collaborative environments.
- Policy and mechanism used for managing and enforcing multiple heterogeneous authorization policies in distributed authorization environments.

To address the above research issues, we developed an access control model and three kinds of authorization policies. These access control model, policies and their enforcement mechanisms have been presented in this thesis.

This chapter concludes this thesis. Section 9.1 highlights the main contributions of this work. Section 9.2 suggests some possible future research directions. Section 9.3 gives the conclusion of this thesis.

9.1 Summary of contributions

The main contributions of our work are providing an access control model and three kinds of authorization policies for access control in collaborative environments. In detail these are the following:

Team and Task-based RBAC: We proposed the TT-RBAC model that extends the RBAC

model through adding sets of two basic data elements called teams and tasks. Central to TT-RBAC is the concept of team relations, through which users, roles and tasks are connected together. Through the relations of user-team assignments, role-team assignments and task-team assignments, users, roles and permissions are introduced into a team, respectively. Thus, the team defines a small and specific RBAC application zone, through which we can preserve the advantages of scalable security administration that RBAC-style models offer and yet offers the flexibility to specify fine-grained control on individual users in certain roles and on individual object instances. This characteristic enables a team to organize a collection of users with different privileges collaborating together to complete some specific tasks assigned to the team.

Context-aware TT-RBAC enforcement: We designed and implemented a TT-RBAC system with object-oriented technology. It is used to demonstrate how the TT-RBAC functionalities are arranged into the TT-RBAC core classes and how these classes work together to meet the functional requirements for administrative operations, session management and administrative review. We developed a mechanism through which any TT-RBAC entities can be associated with context constraints. We also introduced a novel access control decision making mechanism that can accelerate the speed of making access control decisions. The integrated context constraints make the TT-RBAC be an active security model.

Function-based authorization constraints: We designed two novel authorization constraint schemes named prohibition constraint scheme and obligation constraint scheme respectively. Prohibition constraint schemes are used to express the constraints that forbid the entities from doing or being something. Obligation constraint schemes are used to express the constraints that force the entities to do or be something. These schemes are strongly bound to the authorization entity set and relation functions that could be directly mapped to the functions that need to be developed in an authorization constraint system. Thus, these schemes are not only used for expressing constraints but also used for enforcing constraints. A constraint system could be scalable through defining new entity set and relation functions. Based on these functions, various constraint schemes can be easily defined and then enforced.

Label-based access control policies: We designed label policies that specify information flow constraints. Label policies are organized into labels that are assigned to the authorization policy components. The usage of the labeled policy components must conform to the information flow constraints defined by the labels in order to avoid them being misused. Thus, some information leaks caused by these policy components could be avoided. We have shown that the LBACP could be used for enhancing the traditional RBAC, data separation and classification, policy components grouping and supporting distributed access control. LBACP can simplify other access control policies through detaching some dynamic control information from them. The LBACP could be used to improve access control policy management. Through assigning multiple labels to policy components, these components can be managed and enforced according to multiple application aspects.

Root policies: We developed the root policy specification language and its enforcement mechanism for managing and enforcing multiple heterogeneous authorization policies in distributed authorization environments. In a root policy, each involved authorization policy's storage, trust management and enforcement can be defined independently. At runtime, these policies are combined together and enforced as one. Policy schemas, policy

subschemas, policy hierarchies and various root policy component combining algorithms can be used to flexibly describe various authorization policy relations. Flexible context constraints make the root policy be a context-aware authorization system. On the other hand, multiple root policies can cooperate to complete more complex authorization tasks in distributed fashion.

Implementation and applications: We either presented a proof of concept implementation that we have used to test the ideas described in this thesis, or provided well developed modules that could be directly used in real application systems. We gave in-depth descriptions of several real case studies. Two of them show how our authorization constraint mechanism has been integrated into the real application systems.

9.2 Future work

This section outlines several avenues for future research highlighted by this thesis. Based on the work presented in this thesis, the future research is to explore some specific application fields in order to further enhance or extend our work. We shall discuss some of them as follows.

- The major reason that the TT-RBAC supports fine-grained access control is because of the flexibility provided by the task-assignments. In TT-TBAC, the tasks assigned to the teams could be type-based or instance-based. In many application cases, we may statically assign a type-based permission to a team, and then enforce some constraints to the task using some parameters at runtime. For example, a health care team is assigned to a task that has the permission of reading all the patients' medical records. At runtime, the team members can only read the medical records of patients who are assigned to the team. It means that the team-task is parameterized. We are planning to systematically investigate the parameterized team-tasks in the near future.
- Our function-based authorization constraints system has been well defined and developed. However, there are still some research issues in this area. For example, in a bank, it is possible for a staff to be assigned to the roles teller and auditor if these roles can only be activated in different branches. We call these context related constraints as context-based authorization constraints. Context-based authorization constraints are not considered by the function-based authorization constraints. This issue has been discussed by Chadwick et al. [CXOL07]. However, their approach only considers the constraints related to separation of duty. In the area of authorization constraints, our future research is mainly related to the context-based authorization constraints.
- We are considering two directions about the future research to the label-based access control policies. One is developing a XML-based TT-RBAC policy in which the labels can be assigned to the policy components in different levels in order to get all the benefits described in Section 6.4. The other is investigating the enforcement mechanism when the labels are assigned to the data rows in database tables.

9.3 Conclusion

This thesis presented an access control model and three kinds of authorization policies, which address the authorization issues in collaborative environments. We first introduced the TT-RBAC access control model that permits a collection of users with different roles collaborating

together to complete some specific tasks assigned to them. We next described the function-based authorization constraint schemes that can be used to express and enforce prohibition constraints and obligation constraints. We then described the label-based access control policies that enforce information flow constraints to other access control policies or their components. Finally, we introduced the root policies that are used to integrate multiple heterogeneous authorization policies in distributed authorization environments. Moreover, we outlined some future work.

Appendix A

TT-RBAC Functional Specification

In the following table we list the administrative functions, supporting system functions and review functions required for Core TT-RBAC and Hierarchical TT-RBAC. For each kind of functions, we first list the functions required for RBAC that are defined in [FSGK01], then list the additional functions required for TT-RBAC.

Function Name	***Function Description***
Administrative functions required for Core TT-RBAC	
Administrative functions for Core RBAC	
AddUser	Creates a new user.
DeleteUser	Deletes an existing user from the database.
AddRole	Creates a new role.
DeleteRole	Deletes an existing role from the database.
AssignUser	Assigns a user to a role.
DeassignUser	Deletes the assignment of the user to the role.
GrantPermission	Grants a role the permission to perform an operation on an object to a role.
RevokePermission	Revokes the permission to perform an operation on an object from the set of permissions assigned to a role.
Administrative functions for Core TT-RBAC	
AddTeam	Creates a new team.
DeleteTeam	Deletes an existing team from the database.
AddTask	Create a new task.
DeleteTask	Deletes an existing task from the database.
AssignTeamUser	Assigns a user to a team.
DeassignTeamUser	Deletes the assignment of the user to the team.
AssignTeamRole	Assigns a role to a team.
Dea ssignTeamRole	Deletes the assignment of the role to the team.
AssignTeamTask	Assigns a task to a team.
DeassignTeamTask	Deletes the assignment of the task to the team.
GrantTaskPermission	Grants a task the permission to perform an operation on an object to a task.
RevokeTaskPermission	Revokes the permission to perform an operation on an object from the set of permissions assigned to a task.

Supporting system functions required for Core TT-RBAC	
Supporting system functions for Core RBAC	
CreateSession	Creates a user session and provides the user with a default set of active roles.
DeleteSession	Deletes a given session with a given owner user.
AddActiveRole	Adds a role as an active role for the current session.
DropActiveRole	Deletes a role from the active role set for the current session.
CheckAccess	Determines if the session subject has permission to perform the requested operation on an object.
Supporting system functions for Core TT-RBAC	
AddActiveTeam	Adds a team as an active team for the current session.
DropActiveTeam	Deletes a team from the active team set for the current session.
AddActiveTeamRole	Adds a role as an active team-role for the current session in the given team.
DropActiveTeamRole	Deletes a role from the active team-role set for the current session in a given team.
AddActiveTeamTask	Adds a task as an active team-task for the current session in the given team.
DropActiveTeamTask	Deletes a task from the active team-task set for the current session in the given team.
CheckTeamAccess	Determines if the session subject has permission to perform the requested operation on an object in a team.
Review functions required for Core TT-RBAC	
Review functions for Core RBAC	
AssignedUsers	Returns the set of users assigned to a given role.
AssignedRoles	Returns the set of roles assigned to a given user.
RolePermissions	Returns the set of permissions granted to a given role.
UserPermissions	Returns the set of permissions a given user gets through his/her assigned roles.
SessionRoles	Returns the set of active roles associated with a session.
SessionPermissions	Returns the set of permissions available in the session.
RoleOperationsOnObject	Returns the set of operations a given role may perform on a given object.
UserOperationsOnObject	Returns the set of operations a given user may perform on a given object.
Review functions for Core TT-RBAC	
AssignedTeamUsers	Returns the set of users assigned to a given team.
AssignedUserTeams	Returns the set of teams assigned to a given user.
AssignedTeamRoles	Returns the set of roles assigned to a given team.
AssignedRoleTeams	Returns the set of teams assigned to a given role.
AssignedTeamTasks	Returns the set of tasks assigned to a given team.
AssignedTaskTeams	Returns the set of teams assigned to a given task.
TaskPermissions	Returns the set of permissions granted to a given task.
TeamRolesPermissions	Returns the set of permissions granted to a given team gets through the assigned roles.
TeamTasksPermissions	Returns the set of permissions granted to a given team gets through the assigned tasks.
TeamPermissions	Returns the set of permissions granted to a given team.
UserTeamRolesPermissions	Returns the set of permissions granted to a given team-user gets through the assigned team roles.
UserTeamPermissions	Returns the set of permissions granted to a given team-user gets through the assigned team roles and team tasks.
SessionTeams	returns the set of active teams associated with a session.
SessionTeamRoles	Returns the set of active team-roles of a given team associated with a session.

SessionTeamTasks	Returns the set of active team-tasks of a given team associated with a session.
SessionTeamRolesPermissions	Returns the set of permissions available through the team-roles of a given team associated with a session.
SessionTeamTasksPermissions	Returns the set of permissions available through the team-tasks of a given team associated with a session.
SessionTeamPermission	Returns the set of permissions available through a given team associated with a session.
TaskOperationsOnObject	Returns the set of operations a given task may perform on a given object.
TeamRolesOperationsOnObject	Returns the set of operations a given team roles may perform on a given object.
TeamTasksOperationsOnObject	Returns the set of operations a given team tasks may perform on a given object.
TeamOperationsOnObject	Returns the set of operations a given team may perform on a given object.
UserTeamOperationsOnObject	Returns the set of operations a given user gets through a team may perform on a given object.
Administrative functions required for Hierarchical TT-RBAC	
Administrative functions for Hierarchical RBAC	
AddInheritance	Establish a new immediate inheritance relationship between two existing roles.
DeleteInheritance	Delete an existing immediate inheritance relationship between two roles.
AddAscendant	Create a new role and add it as an immediate ascendant of an existing role.
AddDescendant	Create a new role and add it as an immediate descendant of an existing role.
Administrative functions for Hierarchical TT-RBAC	
AddTeamInheritance	Establish a new immediate inheritance relationship between two existing teams.
DeleteTeamInheritance	Delete an existing immediate inheritance relationship between two teams.
AddTeamAscendant	Create a new team and add it as an immediate ascendant of an existing team.
AddTeamDescendant	Create a new team and add it as an immediate descendant of an existing team.
AddTaskInheritance	Establish a new immediate inheritance relationship between two existing tasks.
DeleteTaskInheritance	Delete an existing immediate inheritance relationship between two tasks.
AddTaskAscendant	Create a new task and add it as an immediate ascendant of an existing task.
AddTaskDescendant	Create a new task and add it as an immediate descendant of an existing task.
Supporting system functions required for Hierarchical TT-RBAC	
Supporting system functions for Hierarchical RBAC	
CreateSession	The default active role set created as result of the new session shall include not only roles directly assigned to a user but also some or all of the roles inherited by those “directly assigned roles” as well.
AddActiveRole	A user can activate a directly assigned role or one or more of the roles inherited by the “directly assigned role”.
Supporting system functions required for hierarchical TT-RBAC	
AddActiveTeam	A user can activate a directly assigned team or one or more of the teams inherited by the “directly assigned team”.
AddActiveTeamRole	A user can activate a directly assigned team-role or one or more of the team-roles inherited by the “directly assigned team-role” in a given team.
AddActiveTeamTask	A user can activate a directly assigned team-task or one or more of the team-tasks inherited by the “directly assigned team-task” in a given team.
Review functions required for Hierarchical TT-RBAC	
Review functions for Hierarchical RBAC	
AuthorizedUsers	returns the set of users directly assigned to a given role as well as those who were members of those “roles that inherited the given role”
AuthorizedRoles	returns the set of roles directly assigned to a given user as well as those “roles

	that were inherited by the directly assigned roles"
Review functions required for hierarchical TT-RBAC	
AuthorizedTeamUsers	returns the set of users directly assigned to a given team as well as those who were members of those "teams that inherited the given team"
AuthorizedUserTeams	returns the set of teams directly assigned to a given user as well as those "teams that were inherited by the directly assigned teams"
AuthorizedRoleTeams	returns the set of teams directly assigned to a given role as well as those who were members of those "roles that inherited the given role"
AuthorizedTeamRoles	returns the set of roles directly assigned to a given team as well as those who were members of those "roles that were inherited by the directly assigned roles"
AuthorizedTaskTeams	returns the set of teams directly assigned to a given task as well as those who were members of those "tasks that inherited the given task"
AuthorizedTeamTasks	returns the set of tasks directly assigned to a given team as well as those "tasks that were inherited by the directly assigned tasks"
Redefined review functions required for Hierarchical TT-RBAC	
Review functions for Hierarchical RBAC	
RolePermissions	Returns the set of all permissions either directly granted to or inherited by a given role.
UserPermissions	Returns the set of permissions of a given user through his or her authorized roles (union of directly assigned roles and roles inherited by those roles).
RoleOperationsOnObject	Returns the set of operations a given role may perform on a given object (obtained either directly or by inheritance).
UserOperationsOnObject	Returns the set of operations a given user may perform on a given object (obtained directly or through his/her assigned roles or through roles inherited by those roles).
Review functions for Hierarchical TT-RBAC	
TaskPermissions	Returns the set of all permissions either directly granted to or inherited by a given task.
TeamRolesPermissions	Returns the set of permissions of a given team through its authorized roles (union of directly assigned roles and roles inherited by those roles).
TeamTasksPermissions	Returns the set of permissions of a given team through its authorized tasks (union of directly assigned tasks and tasks inherited by those tasks).
TeamPermissions	Returns the set of all permissions of a given team through its authorized tasks and roles.
UserTeamRolesPermissions	Returns the set of permissions of a given user through his or her authorized roles (union of directly assigned roles and roles inherited by those roles).
UserTeamPermissions	Returns the set of permissions of a given user through his or her authorized teams (union of directly assigned teams and teams inherited by those teams).
TaskOperationsOnObject	Returns the set of operations a given task may perform on a given object (obtained either directly or by inheritance).
TeamOperationsOnObject	Returns the set of operations a given team may perform on a given object (obtained either directly or by role inheritance and task inheritance).
UserTeamOperationsOnObject	Returns the set of operations a given user may perform on a given object through a team (union of directly assigned teams and teams inherited by those teams).

Appendix B

Root Policy Schema

The following XML document is the root policy schema that is used for root policy storage and verification.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="RootPolicy">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="Description" type="xs:string" minOccurs="0"/>

    <xs:element name="SubjectDomains">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="SubjectDomain" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
         <xs:sequence>
          <xs:element name="Include" minOccurs="0">
           <xs:complexType>
            <xs:sequence>
             <xs:element name="LDAPDN" type="xs:string" minOccurs="0"
maxOccurs="unbounded"/>
            </xs:sequence>
           </xs:complexType>
          </xs:element>
          <xs:element name="Exclude" minOccurs="0">
           <xs:complexType>
            <xs:sequence>
             <xs:element name="LDAPDN" type="xs:string" minOccurs="0"
maxOccurs="unbounded"/>
            </xs:sequence>
           </xs:complexType>
          </xs:element>
         </xs:sequence>
         <xs:attribute name="SubjectDomainID" type="xs:string" use="required"/>
        </xs:complexType>
       </xs:element>
      </xs:sequence>
     </xs:complexType>
    </xs:element>

    <xs:element name="ResourceDomains">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="ResourceDomain" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
```

```
         <xs:sequence>
          <xs:element name="Include" minOccurs="0">
           <xs:complexType>
            <xs:sequence>
             <xs:element name="LDAPDN" type="xs:string" minOccurs="0"
maxOccurs="unbounded"/>
            </xs:sequence>
           </xs:complexType>
          </xs:element>
          <xs:element name="Exclude" minOccurs="0">
           <xs:complexType>
            <xs:sequence>
             <xs:element name="LDAPDN" type="xs:string" minOccurs="0"
maxOccurs="unbounded"/>
            </xs:sequence>
           </xs:complexType>
          </xs:element>
          <xs:element name="ObjectClasses" minOccurs="0">
           <xs:complexType>
            <xs:sequence>
             <xs:element name="ObjectClass" type="xs:string" minOccurs="0"
maxOccurs="unbounded"/>
            </xs:sequence>
           </xs:complexType>
          </xs:element>
         </xs:sequence>
         <xs:attribute name="ResourceDomainID" type="xs:string" use="required"/>
        </xs:complexType>
       </xs:element>
      </xs:sequence>
     </xs:complexType>
    </xs:element>

    <xs:element name="ContextConstraints">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="ContextConstraint" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
         <xs:sequence>
          <xs:element name="ContextCondition" minOccurs="0" maxOccurs="unbounded">
           <xs:complexType>
            <xs:sequence>
             <xs:element name="Operator" type="xs:string">
              <xs:complexType>
               <xs:attribute name="DataType" type="xs:string" use="required"/>
              </xs:complexType>
             </xs:element>
             <xs:element name="LeftOperand">
              <xs:complexType>
               <xs:sequence>
                <xs:element name="Observer" type="xs:string" minOccurs="0"/>
                <xs:element name="Function" type="xs:string" minOccurs="0"/>
                <xs:element name="Parameter" type="xs:string" minOccurs="0"/>
               </xs:sequence>
              </xs:complexType>
             </xs:element>
             <xs:element name="RightOperand">
              <xs:complexType>
               <xs:sequence>
                <xs:element name="Observer" type="xs:string" minOccurs="0"/>
                <xs:element name="Function" type="xs:string" minOccurs="0"/>
                <xs:element name="Parameter" type="xs:string" minOccurs="0"/>
               </xs:sequence>
              </xs:complexType>
             </xs:element>
            </xs:sequence>
            <xs:attribute name="ContextConditionID" type="xs:string" use="required"/>
           </xs:complexType>
          </xs:element>
```

```
      </xs:sequence>
      <xs:attribute name="ContextConstraintID" type="xs:string" use="required"/>
     </xs:complexType>
    </xs:element>
   </xs:sequence>
  </xs:complexType>
 </xs:element>

 <xs:element name="Policies">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="Policy" minOccurs="0" maxOccurs="unbounded">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="OID" type="xs:string"/>
       <xs:element name="ValidityPeriod" minOccurs="0">
        <xs:complexType>
         <xs:attribute name="Start" type="xs:dateTime" use="optional"/>
         <xs:attribute name="End" type="xs:dateTime" use="optional"/>
        </xs:complexType>
       </xs:element>
       <xs:element name="Critical" type="xs:boolean"/>
       <xs:element name="Evaluator" type="xs:string" minOccurs="0"/>
       <xs:element name="ACURI" type="xs:string" minOccurs="0"/>
       <xs:element name="ACRLURI" type="xs:string" minOccurs="0"/>
       <xs:element name="PKCURI" type="xs:string" minOccurs="0"/>
       <xs:element name="CRLURI" type="xs:string" minOccurs="0"/>
       <xs:element name="CRLURI" type="xs:string" minOccurs="0"/>
       <xs:element name="RealizedBy" minOccurs="0">
        <xs:complexType>
         <xs:attribute name="URI" type="xs:string" use="required"/>
         </xs:complexType>
       </xs:element>
      </xs:sequence>
      <xs:attribute name="PolicyID" type="xs:string" use="required"/>
     </xs:complexType>
    </xs:element>
   </xs:sequence>
  </xs:complexType>
 </xs:element>

 <xs:element name="PolicyHierarchies">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="PolicyHierarchy" minOccurs="0" maxOccurs="unbounded">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="PolicyReference" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
         <xs:sequence>
          <xs:element name="SubPolicyReference" minOccurs="0" maxOccurs="unbounded">
           <xs:complexType>
            <xs:attribute name="PolicyID" type="xs:string" use="required"/>
           </xs:complexType>
          </xs:element>
         </xs:sequence>
         <xs:attribute name="PolicyID" type="xs:string" use="required"/>
        </xs:complexType>
       </xs:element>
      </xs:sequence>
      <xs:attribute name="PolicyHierarchyID" type="xs:string" use="required"/>
      <xs:attribute name="PolicyCombiningAlgID" type="xs:string" use="required"/>
     </xs:complexType>
    </xs:element>
   </xs:sequence>
  </xs:complexType>
 </xs:element>

 <xs:element name="PolicySubSchemas">
```

```
     <xs:complexType>
      <xs:sequence>
       <xs:element name="PolicySubSchema" minOccurs="0" maxOccurs="unbounded">
        <xs:complexContent>
          <xs:extension base="PolicySubSchemaBase">
           <xs:attribute name="PolicySubSchemaID" type="xs:string" use="required"/>
          </xs:extension>
        </xs:complexContent>
       </xs:element>
      </xs:sequence>
     </xs:complexType>
    </xs:element>

    <xs:element name="PolicySchemas">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="PolicySchema" minOccurs="0" maxOccurs="unbounded">
        <xs:complexContent>
          <xs:extension base="PolicySubSchemaBase">
           <xs:sequence>
            <xs:element name="SchemaSubjectDomains">
             <xs:complexType>
              <xs:sequence>
               <xs:element name="SubjectDomainReference" minOccurs="0"
maxOccurs="unbounded">
                <xs:complexType>
                 <xs:attribute name="SubjectDomainID" type="xs:string" use="required"/>
                </xs:complexType>
               </xs:element>
              </xs:sequence>
             </xs:complexType>
            </xs:element>
            <xs:element name="SchemaResourceDomains">
             <xs:complexType>
              <xs:sequence>
               <xs:element name="ResourceDomainReference" minOccurs="0"
maxOccurs="unbounded">
                <xs:complexType>
                 <xs:attribute name="ResourceDomainID" type="xs:string" use="required"/>
                </xs:complexType>
               </xs:element>
              </xs:sequence>
             </xs:complexType>
            </xs:element>
           </xs:sequence>
           <xs:attribute name="PolicySchemaID" type="xs:string" use="required"/>
          </xs:extension>
        </xs:complexContent>
       </xs:element>
      </xs:sequence>
     </xs:complexType>
    </xs:element>

   </xs:sequence>
   <xs:attribute name="RootPolicyID" type="xs:string" use="required"/>
   <xs:attribute name="OID" type="xs:string" use="required"/>
  </xs:complexType>
 </xs:element>

 <xs:complexType name="PolicySubSchemaBase">
  <xs:sequence>
   <xs:element name="StartPolicyItem">
    <xs:complexType>
     <xs:attribute name="PolicyItemID" type="xs:string" use="required"/>
    </xs:complexType>
   </xs:element>
   <xs:element name="SchemaPolicyHierarchies">
    <xs:complexType>
     <xs:sequence>
```

```
      <xs:element name="PolicyHierarchyReference" minOccurs="0" maxOccurs="unbounded">
       <xs:complexType>
        <xs:attribute name="PolicyHierarchyID" type="xs:string" use="required"/>
       </xs:complexType>
      </xs:element>
     </xs:sequence>
    </xs:complexType>
   </xs:element>
   <xs:element name="SchemaPolicyRelations">
    <xs:complexType>
     <xs:sequence>
      <xs:element name="PolicyItem" minOccurs="0" maxOccurs="unbounded">
       <xs:complexType>
        <xs:sequence>
          <xs:element name="PolicySets" minOccurs="0">
          <xs:complexType>
           <xs:sequence>
            <xs:element name="PolicySet" minOccurs="0" maxOccurs="unbounded">
             <xs:complexType>
              <xs:sequence>
               <xs:element name="CombinedPolicyItem" minOccurs="0"
maxOccurs="unbounded">
                <xs:complexType>
                 <xs:attribute name="PolicyItemType" type="xs:string" use="required"/>
                 <xs:attribute name="PolicyItemID" type="xs:string" use="required"/>
                </xs:complexType>
               </xs:element>
              </xs:sequence>
              <xs:attribute name="PolicyItemCombiningAlgID" type="xs:string"
use="required"/>
             </xs:complexType>
            </xs:element>
           </xs:sequence>
           <xs:attribute name="PolicySetCombiningAlgID" type="xs:string"
use="required"/>
          </xs:complexType>
         </xs:element>
         <xs:element name="Context">
          <xs:complexType>
           <xs:sequence>
            <xs:element name="ContextConstraintReference" minOccurs="0"
maxOccurs="unbounded">
             <xs:complexType>
              <xs:attribute name="ContextConstraintID" type="xs:string" use="required"/>
             </xs:complexType>
            </xs:element>
           </xs:sequence>
          </xs:complexType>
         </xs:element>
        </xs:sequence>
        <xs:attribute name="PolicyItemType" type="xs:string" use="required"/>
        <xs:attribute name="PolicyItemID" type="xs:string" use="required"/>
       </xs:complexType>
      </xs:element>
     </xs:sequence>
    </xs:complexType>
   </xs:element>
   <xs:element name="Context">
    <xs:complexType>
     <xs:sequence>
      <xs:element name="ContextConstraintReference" minOccurs="0" maxOccurs="unbounded">
       <xs:complexType>
        <xs:attribute name="ContextConstraintID" type="xs:string" use="required"/>
       </xs:complexType>
      </xs:element>
     </xs:sequence>
    </xs:complexType>
   </xs:element>
  </xs:sequence>
```

```
 </xs:complexType>

</xs:schema>
```

Bibliography

[ACCA05] R. Alfieri, R. Cecchini, V. Ciaschini, L. dell'Agnello, Á. Frohner, K. Lőrentey and F. Spataro. From gridmap-file to VOMS: managing authorization in a Grid environment. Future Generation Computer Systems, 21(4), pp. 549-558, 2005.

[ACCA05] R. Alfieri, R. Cecchini, V. Ciaschini, L. dell'Agnello, Á. Frohner, K. Lőrentey and F. Spataro. From gridmap-file to VOMS: managing authorization in a Grid environment. Future Generation Computer Systems, 21(4), pp. 549-558, 2005.

[Amo94] E. Amoroso. Fundamentals of Computer Security Technology. Prentice Hall, Apr. 1994. ISBN 0-13108-929-3.

[AS00] G. Ahn, R. Sandhu. Role-based authorization constraint specification. ACM Trans. Inf. Syst. Sec. 3, 4 (Nov.), 2000.

[BB99] A. Bullock, S. Benford. An access control framework for multi-user collaborative environments. In ACM GROUP, Phoenix, AZ, pp. 140-149, 1999.

[BD99] D. Banisar, S. Davies. Privacy & Human Rights—An International Survey of Privacy Laws and Developments. EPIC, 1999.

[BEM03] A. Belokosztolszki, D. M. Eyers, K. Moody. Policy contexts: Controlling information flow in parameterised RBAC. In Proceedings of Policy 2003: IEEE Fourth International Workshop on Policies for Distributed Systems and Networks, 2003, pp. 99-110.

[BFA99] E. Bertino, E. Ferrari, V. Atluri. The specification and enforcement of authorization constraints in workflow management systems. ACM Trans. Inf. Syst. Sec. (TISSEC) 1, 2, Feb. 1999.

[Bib75] Ken J. Biba. Integrity consideration for secure computer systems. Technical Report MTR-3153, MITRE Corporation, Bedford, MA, April 1975.

[Bis03] M. Bishop. Computer Security — Art and Science. Addison-Wesley, 2003.

[BJS99] E. Bertino, S. Jajodia, P. Samarati. A flexible authorization mechanism for relational data management systems. ACM Trans. Inf. Syst. 17, 2 (April), pp. 101-140, 1999.

[BL75] David E. Bell, Leonard J. LaPadula. Secure computer systems: Unified exposition and Multics interpretation. Technical Report MTR-2997, The MITRE Corporation, July 1975.

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

[BME04] A. Belokosztolszki, K. Moody, D. M. Eyers. A formal model for hierarchical policy contexts. In Proceedings of Policy 2004: IEEE Fifth International Workshop on Policies for Distributed Systems and Networks, pp. 127-136, 2004.

[BN89] D. Brewer, M. Nash. The Chinese Wall Security Policy. In Proceedings of IEEE Symp Security & Privacy, IEEE Comp Soc Press, pp. 206-214, 1989.

[Bul98] A. Bullock. SPACE: Spatial access control for collaborative virtual environments. PhD. thesis, University of Notingham, 1998.

[BVS02] P. Bonatti, S. Vimercati, P. Samarati. An Algebra for Composing Access Control Policies. ACM Transaction on Information and System security, 5(1):1-35, February 2002.

[CLSD01] M. Covington, W. Long, S. Srinivasan, A. Dey, M. Ahamad, G. D. Abowd. Securing context-aware applications using environment roles. In Proceedings of ACM Symposium on Access Control Model and Technology. Chantilly, VA, 2001.

[CO02] D.W. Chadwick, A. Otenko. RBAC Policies in XML for X.509 Based Privilege Management. SEC 2002, Egypt, May 2002.

[CO03] D. W. Chadwick, A. Otenko. The PERMIS X.509 role based privilege management infrastructure. Future Generation Computer Systems, Volume 19, Issue 2, pp. 277-289, February 2003.

[Cra03] J. Crampton. Specifying and enforcing constraints in role-based access control. In Proceedings of the Eighth ACM Symposium on Access Control Models and Technologies (SACMAT 2003), pp. 43-50, Como, Italy, June 2003.

[Crosman] P. Crosman. What's the Difference Between Workflow and BPM? http://www.transformmag.com/showArticle.jhtml?articleID=16400140.

[CTWS02] E. Cohen, R. K. Thomas, W. Winsborough, D. Shands. Models for Coalition-based Access Control (CBAC). In Proceedings of the ACM Symposium on Access Control Models and Technologies, June 2002.

[CW87] D. D. Clark, D. R. Wilson. A comparision of commercial and military computer security policies. In Proceedings of the 1987 IEEE Symposium on Security and Privacy, IEEE Computer Society Press, pp. 184-194, Los Angeles, CA, May 1987.

[CXOL07] D. W. Chadwick, W. Xu, S. Otenko, R. Laborde, B. Nasser. Multi-Session Separation of Duties (MSoD) for RBAC. In Proceedings of the First International Workshop on Security Technologies for Next Generation Collaborative Business Applications (SECOBAP'07), Istanbul, Turkey, April 2007.

[DataGrid] M. Draoli, G. Mascari, R. Piccinelli. Project Presentation. DataGrid-11-NOT-0103-1_1.

[DataTag] http://www.datatag.org/.

[Des03] J. Desmond. Roles or rules: the access control debate. http://www.esecurityplanet.com/views/article.php/2241671; July 2003.

[Dey2001] A. K. Dey. Understanding and using context, Personal and Ubiquitous Computing. Volume 5 , Issue 1 (February 2001) pp. 4-7.

[Edw96] W. K. Edwards. Policies and roles in collaborative applications. In ACM Conference on Computer-Supported Cooperative Work, Cambridge, MA., 1996.

[FB97] D. Ferraiolo, J. Barkley. Specifying and managing role-based access control within a corporate intranet. In Proceedings of 2nd ACM Workshop on Role-Based Access Control, Fairfax, VA. pp. 77-82, 1997.

[FBK99] D. F. Ferraiolo, J. F. Barkley, D. R. Kuhn. A role-based access control model and reference implementation within a corporate intranet. ACM Transactions on Information and System Security, vol. 2, pp. 34-64, Feb. 1999.

[FCK95] D. F. Ferraiolo, J. A. Cuigini, D. R. Kuhn. Role-based access control (RBAC): Features and motivations. In Proceedings of the 11th Annual Computer Security Applications Conference (ACSAC'95), Dec. 1995.

[FGHK05] D. F. Ferraiolo, S. Gavrila, V. Hu, D. R. Kuhn. Access control policy management: Composing and combining policies under the policy machine. In Proceedings of the SACMAT'05, Stockholm, Sweden, pp. 11-20, 2005.

[FH02] S. Farrell, R. Housley. An Internet Attribute Certificate Profile for Authorization, Internet-draft April 2002, http://www.ietf.org/rfc/rfc3281.txt.

[FK92] D. Ferraiolo, R. Kuhn. Role-based access controls. In Proceedings of 15th NIST-NCSC National Computer Security Conference, pp. 554-563, 1992.

[FKC03] D. Ferraiolo, D. Kuhn, R. Chandramouli. Role-Based Access Control. Artech House, Computer Security Series, 2003.

[FSGK01] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, R. Chandramouli. Proposed NIST standard for role-based access control. ACM Transactions on Information and System Security, vol. 4, pp. 224-274, Aug. 2001.

[GGF98] V. D. Gligor, S. Gavrila, D. Ferraiolo. On the formal definition of separation of duty policies and their composition. In Proceedings of the 1998 IEEE Computer Society Symposium on Research in Security and Privacy, pp. 172-183, Oakland, CA, May 1998.

[GI96] L. Giuri, P. Iglio. A formal model for role-based access control with constraints. In Proceedings of 9th IEEE Workshop on Computer Security Foundations, pp. 136-145, Kenmare, Ireland, June 1996.

[GI97] L. Giuri, P. Iglio. Role templates for content-based access control. In Proceedings of the second ACM workshop on Role-based access control table of contents, pp.153-159, Fairfax, Virginia, United States, 1997.

[Globus] The Globus Project: http://www.globus.org/.

[GMPT01] C. K. Georgiadis, I. Mavridis, G. Pangalos, R. K. Thomas. Flexible Team-Based Access Control Using Contexts. In Proceedings of the ACM Symposium on Access Control Models and Technologies, Virginia, USA, pp. 21-27, May 2001.

[HFK06] V. C. Hu, D. F. Ferraiolo D. R. Kuhn. Assessment of Access Control Systems. National Institute of Standards and Technology (NIST) Interagency Report 7316, September 2006.

[HFPS99] R. Housley, W. Ford, W. Polk, D. Solo. Internet X.509 Public Key Infrastructure Certificate and CRL Profile. January 1999, http://www.ietf.org/rfc/rfc2459.txt.

[Hos92] H. Hosmer. The multipolicy paradigm. In Proceedings of the Fifteenth National Computer Security Conference, pp. 409-422, Baltimore, MD, October 1992.

[HZZM05] W. Huang, W. Zhou, X. Zhang, C. Meinel. A Dynamic, Secure and Multi-Solutions Supported Middleware System. In Proceedings of the 7th IEEE International Conference on Advanced Communication Technology (ICACT2005), pp. 724-729, Republic of Korea, February, 2005.

[IAIK] IAIK Java Cryptography Extension (IAIK-JCE), http://jce.iaik.tugraz.at/index.php.

[IBMSS] IBM alphaworks, XML Security Suite, http://www.alphaworks.ibm.com/tech/xmlsecuritysuite.

[ITUT01] ITU-T Rec. X.509 ISO/IEC 9594-8. The Directory: Public-key and Attribute Certificate Frameworks, May, 2001.

[JP96] T. Jaeger, A. Prakash. Requirements of role-based access control for collaborative systems. In Proceedings of ACM Role-based Access Control Workshop, Gaithersburg, MD., pp. 53-64, 1996.

[JSSS01] S. Jajodia, P. Samarati, M. L. Sapino, V. S. Subrahmanian. Flexible support for multiple access control policies. ACM Transactions on Database Systems (TODS), Volume 26, Issue 2, pp. 214-260, June 2001.

[JT01] T. Jaeger, J. Tidswell. Practical safety in flexible access control models. ACM Transactions on Information and System Security 4, 2, pp. 158-190, 2001.

[KPF01] M. H. Kang, J. S. Park, J. N. Froscher. Access control mechanisms for inter-organizational workflow. In Proceedings of the sixth ACM symposium on Access control models and technologies, Chantilly, Virginia, United States, pp. 66-74, May 2001.

[Kuh97] R. Kuhn. Mutual exclusion as a means of implementing separation of duty requirements in role based access control systems. In Proceedings of the Second ACM Workshop on Role Based Access Control, pp. 23-30, 1997.

[LAKK03] M. Lorch, D. Adams, D. Kafura, M. Koneni, A. Rathi, S. Shah. The PRIMA System for Privilege Management, Authorization and Enforcement in Grid Environments. The 4th Int. Workshop on Grid Computing - Grid 2003, Phoenix, AR, USA, November 2003.

[Lam71] B. Lampson. Protection. In 5th Princeton Symposium on Information Science and Systems, pp. 437-443, 1971. Reprinted in ACM Operat. Syst. Rev. 8,1, 18–24, 1974.

[LBT04] N. Li, Z. Bizri, M. V. Tripunitara. On mutually exclusive roles and separation of duty. In Proceedings of the CCS'04, pp. 42-51, Washington, DC, USA, October 2004.

[Lep03] R. Lepro. Cardea: Dynamic Access Control in Distributed Systems. NASA Technical Report NAS-03-020, November 2003.

[LFG99] N. Li, J. Feigenbaum, B. Grosof. A logic-based knowledge representation for authorization with delegation. In Proceedings of the Twelfth IEEE Computer Security Foundations Workshop (Mordano, Italy, June), pp. 162-174, 1999.

[LFSA06] B. Lang, I. Foster, F. Siebenlist, R. Ananthakrishnan, T. Freeman. A Multipolicy Authorization Framework for Grid Security. In Proceedings of the Fifth IEEE Symposium on Network Computing and Application, Cambridge, USA, pp. 269-272, July 2006.

[LS97] E. C. Lupu, M. Sloman. A policy based role object model. In Proceedings of the 1st IEEE Enterprise Distributed Object Computing Workshop, Calif, Oct. 1997.

[LS99] E. Lupu, M. Sloman. Conflicts in policy-based distributed systems management. IEEE Trans. Softw. Eng. 25, 6, Nov./Dec., 1999.

[MBCS06] P. Mazzoleni, E. Bertino, B. Crispo, S. Sivasubramanian. XACML policy integration algorithms: not to be confused with XACML policy combination algorithms!. In Proceedings of the SACMAT'06, Lake Tahoe, California, USA, pp. 219-227, June 2006.

[ML00] A. C. Myers, B. Liskov. Protecting privacy using the decentralized label model. ACM Transactions on Software Engineering and Methodology, vol 9, issue 4, pp. 410-442, Oct. 2000.

[ML98] A. C. Myers, B. Liskov. Complete, safe information flow with decentralized labels. in Proceedings of IEEE Symposium on Security and Privacy, pp. 186-197, Oakland, CA, USA, May 1998.

[MYSQL] MySQL, the open source SQL database server, http://www.mysql.com/.

[NCSC87] National Computer Security Center (NCSC). A Guide to Understanding Discretionary Access Control in Trusted System. Report NSCD-TG-003 Version1, 30 September 1987.

[NO93] M. Nyanchama, S. Osborn. Role-based security: Pros, cons & some research directions. ACM SIGSAC Review, vol. 2, pp. 11-17, June 1993.

[NO99] M. Nyanchama, S. Osborn. The role graph model and conflict of interest. ACM Transactions on Information and System Security, vol. 2, pp. 3-33, Feb. 1999.

[OG00] S. Osborn, Y. Guo. Modelling users in role-based access control. In Proceedings of the 5th ACM Role-Based Access Control Workshop, July 2000.

[OLDAP] OpenLDAP, the open source Lightweight Directory Access Protocol (LDAP), http://www.openldap.org/.

[OP03] S. Oh, S. Park. Task–role-based access control model. Information Systems 28 (2003) pp.533-562, 2003.

[Oracle] Oracle Label Security Administrator's Guide, 10g Release 1 (10.1), Part Number B10774-01, http://download.oracle.com/docs/cd/B14117_01/network.101/b10774.pdf

[PKWF03] L. Pearlman, C. Kesselman, V. Welch, I. Foster, S. Tuecke. The Community Authorization Service: Status and Future. In Proceedings of Computing in High Energy Physics 03 (CHEP03), La Jolla, California, USA, March, 2003.

[RSBAC] RSBAC, the LINUX Rule Set Based Access Control (RSBAC) system, http://www.rsbac.org/.

[SAAJ] SOAP with Attachments API for Java (SAAJ), http://java.sun.com/xml/saaj/.

[SAML] Security Assertion Markup Language (SAML) v1.0 - OASIS Standard, 5 November 2002: http://www.oasis-open.org/committees/download.php/3406/oasis-sstc-saml-core-1.1.pdf.

[San88] R. S. Sandhu. Transaction control expressions for separation of duties. In 4th Aerospace Computer Security Conference, pp. 282-286, Dec. 1988.

[San90] R. S. Sandhu. Separation of duties in computerized information systems. In IFIP Workshop on Database Security, pp. 179-190, 1990.

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

[SBM99] R. S. Sandhu, V. Bhamidipati, Q. Munawer. The ARBAC97 model for role-based administration of roles. ACM Trans. Inf. Syst. Sec. 1, 2, Feb. 1999.

[SCFY96] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, C. E. Youman. Role-based access control models. IEEE Computer, vol. 29, pp. 38-47, Feb. 1996.

[SCH03] Francois Siewe, Antonio Cau, Hussein Zedan. A Compositional Framework for Access Control Policies Enforcement. In proceedings of the ACM FMSE'03, pp. 32-42, Washington, DC, USA, October 2003.

[Shibboleth] The Shibboleth project, http://shibboleth.internet2.edu/.

[SN2004] M. Strembeck, G. Neumann. An integrated approach to engineer and enforce context constraints in RBAC environments. ACM Transactions on Information and System Security (TISSEC), Volume 7 Issue 3, August 2004.

[SS75] J. H. Saltzer, M. D. Schroeder. The Protection of Information in Computer Systems. In Proceedings of the IEEE, 63(9):1278-1308, September 1975.

[SZ97] R. Simon, M. E. Zurko. Separation of duty in role based access control environments. In Proceedings of the 10th IEEE Workshop on Computer Security Foundations, pp. 183-194, Rockport, MA, June 1997.

[TAPH05] W. Tolone, G. J. Ahn, T. Pai, S. P. Hong. Access Control in Collaborative Systems. ACM Computing Surveys, Vol. 37, No. 1, pp. 29-41, March 2005.

[TEFW02] S. Tuecke, D. Engert, I. Foster, V. Welch, M. Thompson, L. Pearlman, C. Kesselman. Internet X.509 Public Key Infrastructure Proxy Certificate Profile, draft-ggf-gsi-proxy-04, 2002.

[TEM03] M. Thompson, A. Essiari, S. Mudumbai. Certificate-based Authorization Policy in a PKI Environment. ACM Transactions on Information and System Security (TISSEC), Volume 6, Issue 4, pp. 566-588, Nov. 2003.

[Tho97] R. K. Thomas. Team-based access control (TMAC): A primitive for applying role-based access controls in collaborative environments. in Proceedings of the Second ACM Workshop on Role-based Access Control, Virginia, USA, pp. 13-19, November 1997.

[Tomcat] Jakarta Tomcat servlet container, http://jakarta.apache.org/tomcat/.

[TS97] R. K. Thomas, R. S. Sandhu. Task-based authorization controls (TBAC): A family of models for active and enterprise-oriented authorization management. In Proceedings of the IFIP WG 11.3 Workshop on Database Security, Lake Tahoe, California, pp. 166-181, August 1997.

[Wan99] W. Wang. Team-and-role-based organizational context and access control for cooperative hypermedia environments. in Proceedings of the tenth ACM Conference on Hypertext and hypermedia, Darmstadt, Germany, pp. 37-46, February 1999.

[WJ03] D. Wijesekera, S. Jajodia. A Propositional Policy Algebra for Access Control. ACM Transactions on Information and System Security, 6(2):286–325, May 2003.

[WL93] T. Woo, S. Lam. Authorizations in distributed systems: A new approach. J. Comput. Sec. 2, 2,3, pp. 107-136, 1993.

[XACML] XACML and OASIS Security Services Technical Committee. eXtendible Access Control Markup Language (xacml) committee specification 2.0. Feb 2005.

[ZM04] W. Zhou, C. Meinel. Implement role based access control with attribute certificates. In Proceedings of the 6th IEEE International Conference on Advanced Communication Technology (ICACT2004), pp. 536-541, Republic of Korea, February 2004.

[ZM07a] W. Zhou, C. Meinel. Function-Based Authorization Constraints Specification and Enforcement. In Proceedings of the Third International Symposium on Information Assurance and Security (IAS 2007), pp. 119-114, Manchester, United Kingdom, August 2007.

[ZM07b] W. Zhou, C. Meinel. Team and Task Based RBAC Access Control Model. In Proceedings of the 5th Latin American Network Operations and Management Symposium (LANOMS 2007), pp. 84-94, Petrópolis, Brazil, September 2007.

[ZM07c] W. Zhou, C. Meinel. A Policy Language for Integrating Heterogeneous Authorization Policies. In Proceedings of the 4th International Conference on Grid Service Engineering and Management (GSEM 2007), pp. 9-23, Leipzig, Germany, September 2007.

[ZM08a] W. Zhou, C. Meinel. Enforcing Information Flow Constraints in RBAC Environments. In Proceedings of the International Symposium on Electronic Commerce and Security (ISECS 2008), Guangzhou, China, August 2008. (to appear)

[ZMR05] W. Zhou, C. Meinel, V. H. Raja. A Framework for Supporting Distributed Access Control Policies. In Proceedings of the 10th IEEE Symposium on Computers and Communications (ISCC 2005), pp. 442-447, La Manga del Mar Menor, Cartagena, Spain, June 2005.

[ZMXS08b] W. Zhou, C. Meinel, Y. Xiang, Y. Shao. Authorization Constraints Specification and Enforcement. Journal of Information Assurance and Security (JIAS). (to appear)

[ZRM05] W. Zhou, V. H. Raja, C. Meinel. An Authentication and Authorization System for Virtual Organizations. In Proceedings of the 9th world multiconference on systemics, cybernetics and informatics (WMSCI 2005), Vol. VII, pp. 150-155, Orlando, Florida, U.S.A., July 2005.

[ZRMA05] W. Zhou, V. H. Raja, C. Meinel, M. Ahmad. A Framework for Cross-Institutional Authentication and Authorisation. In Proceedings of the eChallenges e-2005 Conference(e-2005), pp. 1259-1266, Ljubljana, Slovenia, October 2005.

[ZRMA06] W. Zhou, V. H. Raja, C. Meinel, M. Ahmad. Label-Based Access Control Policy Enforcement and Management. In Proceedings of the Seventh International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD 2006), Vol. 00, pp. 395-400, Las Vegas, Nevada, June 2006.