Analysis of ANSI RBAC Support in Commercial Middleware

by

Wesam M. Darwish

B.A.Sc., The University of British Columbia, 2001

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

in

The Faculty of Graduate Studies

(Electrical and Computer Engineering)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

April, 2009

Abstract

This thesis analyzes the access control architectures of three middleware technologies: Common Object Request Broker Architecture (CORBA), Enterprise Java Beans (EJB), and Component Object Model (COM+). For all technologies under study, we formalize the protection state of their corresponding authorization architectures in a more precise and less ambiguous language than their respective specifications. We also suggest algorithms that define the semantics of authorization decisions in CORBA, EJB, and COM+. Using the formalized protection state configurations, we analyze the level of support for the American National Standard Institute's (ANSI) specification of Role-Based Access Control (RBAC) components and functional specification in the studied middleware technologies. This thesis establishes a framework for assessing implementations of ANSI RBAC in the analyzed middleware technologies.

Our findings indicate that all of three middleware technologies under study fall short of supporting even Core ANSI RBAC. Custom extensions are necessary in order for implementations compliant with each middleware to support ANSI RBAC required or optional components. Some of the limitations preventing support of ANSI RBAC are due to the middleware's architectural design decisions; however, fundamental limitations exist due to the impracticality of some aspects of the ANSI RBAC standard itself.

Contents

List of Tables

List of Figures

Acknowledgements

I would like to thank my colleagues from the Laboratory for Education and Research in Secure Systems Engineering (LERSSE) for their constructive feedback. Many thanks go to every person who taught me something, and very special thanks to my supervisor, Dr. Konstantin Beznosov, from whom I learned a lot.

Words fall short of expressing my gratitude towards my family, especially my wife, for their unconditional love and endless support throughout this journey.

To my parents, and to my wife, Suzan

Statement of Co-Authorship

This research is based on three technical reports produced by the author of this thesis and Dr. Konstantin Beznosov. The research was designed by Dr. Beznosov. The author of this thesis performed the background research, co-authored Chapter 2, performed the research and data analysis for Chapters 3 and 4. This manuscript is prepared by the author of this thesis.

Chapter 1

Introduction

This thesis studies the access control mechanisms of the Common Object Request Broker Architecture (CORBA), Enterprise Java Beans (EJB), and Component Object Model Plus (COM+) middleware technologies, and analyzes their support for the American National Standard Institute's (ANSI) specification of Role-Based Access Control (RBAC) components and functional specification.

1.1 Motivation

The American National Standard Institute's (ANSI) specification of Role-Based Access Control (RBAC) [ANS04] is a standard for access control in which permissions are associated with roles and users are assigned to appropriate roles. A role can represent competency, authority, responsibility or specific duty assignments. A major purpose of RBAC is to facilitate access control administration and review. RBAC is commonly believed to address the needs of commercial enterprises better than lattice-based Mandatory Access Control (MAC) [BL75] and owner-based Discretionary Access Control (DAC) [Lam71] models. Moreover, Osborn et al. [OSM00] show that an RBAC system can indeed be configured to enforce either a DAC or a MAC policy. Evidence of RBAC becoming a dominant access control paradigm is the approval of ANSI RBAC standard in 2004. The ANSI RBAC standard consists of two main parts: (1) the RBAC Reference Model and (2) the RBAC System and Administrative Functional Specification, each comprising core, hierarchical, and constraint components.

Early research [FK92, SFK00] was designed to formalize RBAC, and subsequent research over the years led to the consensus ANSI standard for RBAC [FKS07]. The ANSI RBAC standard provides a consistent and uniform specification of RBAC features. Without this uniform definition of RBAC, there would be uncertainty and confusion about RBAC's utility and meaning [ANS04].

In order to conform to this standard, an RBAC system must comply with, and implement, all of the core set of RBAC functional specifications.

At the same time as RBAC was introduced and evolving into a mature model ready for standardization, commercial middleware technologies—such as Common Object Request Broker Architecture (CORBA) [OMG99], COM+ [Obe00], and Enterprise Java Beans (EJB) [DYK01]—also matured, and distributed enterprise applications became routinely developed using middleware. The ability of particular middleware technology to support specific types of access control policy is an open and practical research question, for the following three reasons.

First, different middleware technologies and their subsystems are defined in different forms and formats. For example, CORBA is specified in the form of open application programming interfaces (APIs), whereas EJB is defined through APIs as well as the syntax and semantics of the accompanying eXtensible Markup Language (XML) files used for configuring an EJB container. COM+ is defined through an implementation of APIs as well as graphical user interfaces (GUI) for configuring the behavior of a COM+ server on Windows NT, 2000, 2003, XP, and Vista operating systems. The variations in the form, terminology, and format of the middleware definitions and implementations lead to the difficulty of identifying the correspondence among the security (and other) capabilities of any two middleware technologies as well as the degree to which they can support a particular access control model.

Second, the capabilities of the middleware security controls are not defined in the language of any particular access control model. Instead, each middleware provides general access control mechanisms, which are supposed to be adequate for the majority of cases and scenarios, and could be configured to support various access control models. Designed to support a variety of policy types as well as large-scale, diverse distributed applications, the controls seem to be the result of engineering compromises involving, among other factors, perceived customer requirements, the capabilities of the target run-time environment, and their expected usage. For example, CORBA access controls are defined in terms of *principal's attributes*, *required rights*, and *granted rights*, whereas EJB controls are defined using *role mappings* and *role-method permissions*. Assessing the capability of middleware controls to enforce particular types of authorization policies is harder due to the mismatch in terminology between the published access control models and the languages of the controls.

Third, the security subsystem semantics in commercial middleware is defined imprecisely, sometimes ambiguously, leaving room for different interpretations. For example, the EJB specification does not address nor dictate how the EJB security roles should be mapped to the operational environment's security principals, leaving the semantics of this mapping up for interpretation by various vendors. Another example is the CORBA OMG specification, where not always is the functionality of various interfaces precisely defined [BR02], through UML sequence diagrams, for example.

In this thesis, we clarify the semantics of the security subsystem and analyze its ability to support ANSI RBAC for three industrial middleware technologies: CORBA, EJB, and COM+. This thesis establishes a process for assessing implementations of ANSI RBAC using CORBA, EJB, or COM+. The results provide directions for middleware developers supporting ANSI RBAC in their systems and criteria for users and application developers for selecting those middleware implementations that support required and optional components of ANSI RBAC.

1.2 Overview of ANSI RBAC

Role based access control (RBAC) was introduced more than a decade ago [FK92, SCFY96]. Over the years, RBAC has enjoyed significant attention as many research papers were written on topics related to RBAC; and in recent years, vendors of commercial products have started implementing various RBAC features in their solutions.

The National Institute of Standards and Technology (NIST) initiated a process to develop a standard for RBAC to achieve a consistent and uniform definition of RBAC features. An initial draft of a standard for RBAC was proposed in the year 2000 [SFK00]. A second version was later publicly released in 2001 [FSG+01]. This second version was then submitted to the International Committee for Information Technology Standards (INCITS), where further changes were made to the proposed standard. Lastly, INCITS approved the standard for submittal to the American National Standards Institute (ANSI). The standard was later approved in 2004 [ANS04]. The ANSI RBAC standard consists of two main parts as described in the following sections.

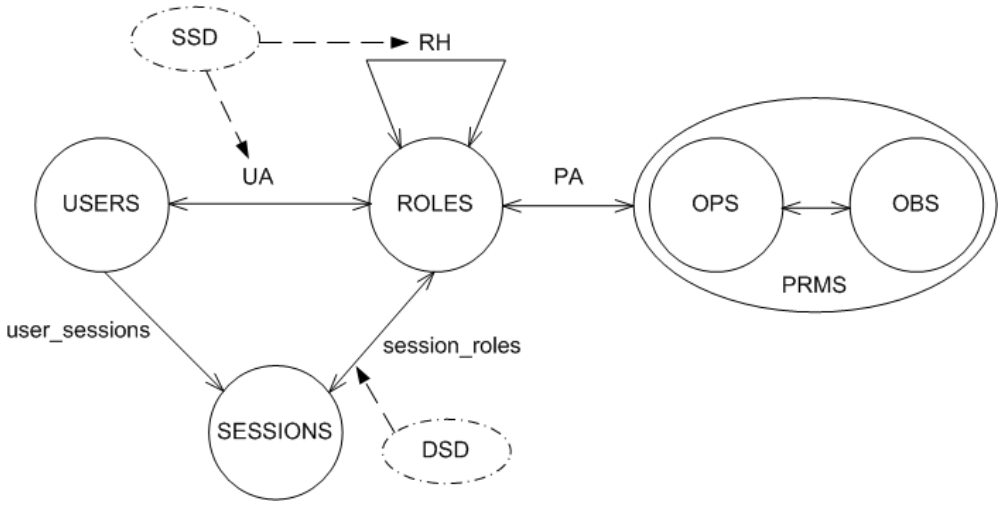


Figure 1.1: RBAC components

1.2.1 Reference Model

The RBAC Reference Model defines sets of basic RBAC elements, relations, and functions that the standard includes. This model is defined in terms of four major RBAC components as described in the following sections. Figure 1.1 depicts the four major components of RBAC.

Core RBAC

Core RBAC defines the minimum set of elements required to achieve RBAC functionality. Core RBAC must be implemented as a minimum in RBAC systems. The other components described below, which are independent of each other, can be implemented separately.

Core RBAC elements are defined as follows [ANS04, pp.4-5]:

Definition 1 [**Core RBAC**]

- $USERS, ROLES, OPS$, *and* OBS *(users, roles, operations, and objects respectively)*
- $UA \subseteq USERS \times ROLES$*, 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:* $assigned\_users(r) = \{u \in USRES | (u, r) \in UA\}$
- $PRMS = 2^{(OPS \times OBS)}$*, the set of permissions*

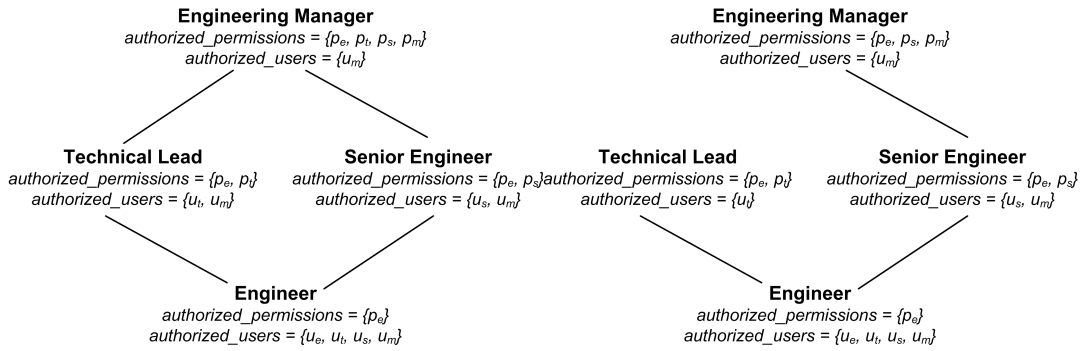
- $PA \subseteq PERMS \times ROLES$, *a many-to-many mapping permission-to-role assignment relation.*
- $assigned\_permissions(r : ROLES) \rightarrow 2^{PRMS}$, *the mapping of role* r *onto a set of permissions. Formally:* $assigned\_permissions(r) = \{p \in PRMS | (p, r) \in PA\}$
- $Op(p : PRMS) \rightarrow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

Hierarchical RBAC

This component adds relations to support role hierarchies. Role hierarchy is a partial order relation that defines seniority between roles, whereby a senior role has at least the permissions of all of its junior roles, and a junior role is assigned at least all the users of its senior roles. A senior role is also said to "inherit" the permissions of its junior roles.

The standard defines two types of role hierarchies. These types are shown in Figure 1.2, and are defined as follows:

- General Role Hierarchies: provide support for arbitrary partial order relations to serve as the role hierarchy. This type allows for multiple inheritance of assigned permissions and users; that is, a role can have any number of ascendants, and any number of descendants
- Limited Role Hierarchies: provide more restricted partial order relations that allow a role to have any number of ascendants, but only limited to one descendant



(a) General role hierarchy (b) Limited role hierarchy

Figure 1.2: Examples of Hierarchical RBAC

In the presence of role hierarchy, the following is defined:

- $authorized\_users(r) = \{u \in USERS | r' \succeq r, (u, r') \in UA\}$ is the mapping of role r onto a set of users
- $authorized\_permissions(r) = \{p \in PRMS | r \succeq r', (p, r') \in PA\}$ is the mapping of role r onto a set of permissions

where $r_{senior} \succeq r_{junior}$ indicates that r_{senior} inherits all permissions of r_{junior}, and all users of r_{senior} are also users of r_{junior}.

Constrained RBAC

Static Separation of Duty (SSD) relations component defines exclusivity relations among roles with respect to user assignments. **Dynamic Separation of Duty (DSD) Relations** component defines exclusivity relations with respect to roles that are activated as part of a user's session.

1.2.2 Functional Specification

For the four components defined in the RBAC reference model, the RBAC System and Administrative Functional Specification defines three categories of various operations that are required in an RBAC system. These categories are defined as follows.

The category of *administrative operations* defines operations required for the creation and maintenance of RBAC element sets and relations. Examples of these operations are listed here. A complete list of these operations, as well as their formal definition is included in the standard.

- Core RBAC administrative operations include AddUser, DeleteUser, AddRole, DeleteRole, AssignUser, GrantPermission, and so on
- Hierarchical RBAC administrative operations include AddInheritance, DeleteInheritance, AddAscendant, and AddDescendant
- SSD Relations administrative operations include CreateSsdSet, AddSsdRoleMember, SetSsdSetCardinality, and so forth
- DSD Relations administrative operations include CreateDsdSet, AddDsdRoleMember, SetDsdSetCardinality, and so on

The *administrative reviews* category defines operations required to perform administrative queries on the system. Examples of Core RBAC administrative review functions include RolePermissions, UserPermissions, SessionRoles, and RoleOperationsOnObjects. Other operations for other RBAC components can be found in the standard.

The *system level functionality* category defines operations for creating and managing user sessions and making access control decisions. Examples of such operations are CreateSession, DeleteSession, AddActiveRole, and CheckAccess.

1.3 Literature Review

Over the past decade, there has been no shortage of papers proposing ways to support RBAC. The overwhelming majority of this work, however, is about support for RBAC96 [SCFY96], which defines the reference models for plain, hierarchical, and constrained RBAC but does not specify the operations to be supported by an RBAC implementation. The paucity of analysis or proposals for supporting ANSI RBAC is not surprising, given the fact that the standard was published in 2004. Because of the lack of research on support for ANSI RBAC, and because of the significant similarities between RBAC96 and ANSI RBAC, we review related work on supporting or implementing

RBAC96 in operating systems, databases, web applications, and distributed systems, including middleware.

Since the mainstream operating systems, with the exception of Solaris [Sun00], do not provide direct support for RBAC, researchers and developers have been employing either groups (e.g., [SA98, AS01]) or user accounts (e.g., [Fad99, Cha03]) to simulate roles. This choice determines whether more than one role can be activated in a session. Role hierarchies are either not supported [Fad99, Sun00] or are simulated by maintaining additional system files with the role hierarchy and various book-keeping data [SA98, AS01]. No implementations we reviewed support static SoD. Just one case of dynamic SoD comes as a side effect with those implementations that simulate roles with user accounts (i.e., [Fad99, Cha03]): the role set in this DSoD is equal to the set of all roles in the system, and the cardinality of the role set is exactly one. In other words, any session can have only one role activated at any given time; the current role is deactivated while another role is activated.

We analyzed DB2 [TM06] and MySQL [MyS07] and updated the analysis of RBAC support in commercial database management systems (DBMS)—conducted by Ramaswamy and Sandhu [RS98]—with the latest versions of the corresponding systems. Commercial DBMS continue to have the most advanced support for RBAC96. Informix Dynamic Server v7.2 [IBM05], IBM DB2 [TM06], Sybase Adaptive Server v11.5 [Syb05], and Oracle Enterprise Server v8.0 [BLL03] directly support roles and role hierarchies. Only Oracle and Sybase allow users to have more than one role activated at any time, though. On the other hand, Informix also provides limited support for dynamic SoD, and Sybase features support for both types of SoD.

In RBAC implementations for client-server systems, including Web applications, roles are either "pushed" from the client to the server in the form of attribute certificates or HTTP cookies, as in [Gut01, PSA01], or "pulled" by the server from a local or remote database, as in [Bar97, FBK99, PSA01, CO02, ZM04]. The former enables selective activation of roles by users, and the latter simplifies the implementation of client authentication but activates all of the assigned roles for the user. However, Web implementation of NIST RBAC [FBK99] has hybrid design, which allows the user to select the roles to be "pulled" by the server. A number of implementations use a database, possibly accessible through the Light-weight Directory Access Protocol (LDAP) [WHK97] front-end, as in [Bar97, Gut01, PSA01, ZM04], to store role and other information. Role hierarchies

are only supported by some implementations, using either manual assignment of permissions of junior roles to senior ones [PSA01], additional files [Giu99], a database [FBK99] or an LDAP server [CO02, ZM04]. JRBAC-WEB [Giu99] and RBAC/Web [FBK99] also support both types of SoD.

The work most relevant to ours addresses support for RBAC in middleware. Ahn [Ahn00] outlines a proposal for enforcing RBAC policies for distributed applications that utilize Microsoft's Distributed Component Object Model (DCOM) [BK98, Mic98]. His proposal employs the following elements of Windows NT's architecture: (1) registry for storing and maintaining the role hierarchy, and permission-to-role assignment (PA), (2) user groups for simulating roles and maintaining user-to-role assignment (UA), and (3) a custom-built security provider that follows the RBAC model to make access control decisions, which are requested and enforced by the DCOM run-time. Since the support for role hierarchy is indicated but not explained in [Ahn00], we assume that the Windows NT registry can be used to encode the hierarchy so that the RBAC security provider can refer to it while making authorization decisions. Similar to the proposals for RBAC support in operating systems, the use of OS user groups for simulating roles enables activation of more than one role. Yet, like with the pull model in client-server systems, all assigned roles are activated, leaving no choice to the user. Ahn does not indicate in [Ahn00] support for any kind of SoD, nor does he explain how RBAC policies can be enforced consistently and automatically in a multi-computer deployment of DCOM-accessible objects.

RBAC-JaCoWeb [WF99, OF02] utilizes the PoliCap [WdSFW$^+$02] policy server to implement CORBASec specification in a way that supports RBAC. PoliCap holds all data concerning security policies within a CORBASec policy domain, including users, roles, user-to-role and role-to-permission assignments, role hierarchy relations, and SoD constraints. Most of the authorization policy enforcement is performed by an RBAC-JaCoWeb CORBA security interceptor. At the time of the client binding to a CORBA object, the interceptor obtains necessary data from the PoliCap server and instantiates CORBASec-compliant DomainAccessPolicy and RequiredRights objects that contain the privilege and control attributes appropriate for the application object. When the client makes invocation requests later, the access decisions are then performed based on the local instances of these objects. Initially, the client security credentials object—created as part of the binding—has no privilege attributes, only AccessId, which is obtained from the client's X.509

certificate used in the underlying SSL connection. If the invocation cannot be authorized with the current set of client privilege attributes, the interceptor "pulls" additional user's role attributes from the PoliCap server. Only those roles that are (1) assigned to the user, (2) necessary for the invocation in question to be authorized, and (3) not in conflict with any DSoD constraints are activated. These role attributes are added to the client's credentials and are later re-used on the server for other requests from the same principal. The extent to which RBAC-JaCoWeb conforms to the CORBASec specification is unclear from [WF99, OF02]. Nevertheless, RBAC-JaCoWeb serves as an example of implementation-specific extensions to CORBAsec that enable better support for RBAC advanced features, such as role hierarchies and SoD, which—as will be seen from the results of our analysis—cannot be supported without extending a CORBASec implementation with additional operations.

1.4 Contributions

Each chapter analyzes access control mechanisms of a specific middleware technology, and defines a configuration of that middleware's protection system in a more precise and less ambiguous language than the corresponding middleware specification. Using these configurations, we suggest algorithms that formally specify the semantics of authorization decisions in each middleware technology. We analyze the level of support for the ANSI RBAC components and functional specification in each middleware. Each chapter sets up a process for assessing implementations of ANSI RBAC for the corresponding middleware system.

The following is a summary of our findings related to each middleware technology in each chapter.

- Chapter 2 addresses the above contributions for CORBA. The results indicate that CORBA Security falls short of supporting even Core RBAC. Custom extensions are necessary in order for implementations compliant with CORBA Security to support ANSI RBAC required or optional components.

- Chapter 3 addresses the above contributions for the Enterprise Java Beans (EJB) architecture. Our results indicate that the EJB specification falls short of supporting even Core ANSI

RBAC. EJB extensions dependent on the operational environment are required in order to ANSI RBAC required components. Other vendor-specific extensions are necessary in order to support ANSI RBAC optional components. Fundamental limitations exist, however, due to impracticality of some aspects in the ANSI RBAC standard itself.

- Chapter 4 addresses the above contributions for the COM+ architecture. Our findings indicate that COM+ falls short of supporting even Core RBAC. The main limitations exist due to the tight integration of the COM+ architecture with the underlying operating system. Other limitations exist due to impracticality of some aspects in the ANSI RBAC standard itself.

1.5 Structure

Each of the following chapters of the thesis is structured as follows: (1) Overview of the middleware technology and its security subsystem; (2) Description of the access control architecture of the middleware under analysis, and formalization of a configuration of the middleware's protection state; (3) Analysis of support for ANSI RBAC in the middleware being analyzed, which also includes an example, where appropriate, that illustrates the level of this support; (4) Discussion, elaborating on interpretations of the analysis; and (5) Conclusions, summarizing the findings of each chapter.

Chapter 5 summarizes the contributions of this thesis, the limitations of our research, and directions for future research.

Chapter 2

Analysis of ANSI RBAC Support in CORBA

This chapter analyzes access control mechanisms of CORBA Security, and defines a configuration of this middleware's protection system in a more precise and less ambiguous language than the corresponding CORBA Security specification. We also suggest algorithms that formally specify the semantics of authorization decisions in CORBA Security. We then analyze the level of support for the ANSI RBAC components and functional specification in CORBA, and conclude with our findings.

2.1 Overview of CORBA Security

The following sub-section provides a brief and informal overview of CORBA. More information can be found in the corresponding CORBA specifications. Readers familiar with CORBA are advised to proceed to Section 2.1.2, which provides background on CORBA Security.

2.1.1 CORBA

CORBA specifications, including the CORBA Security Service [OMG02], define a general-purpose interface definition language and OS-independent infrastructure for developing and deploying distributed applications. The distributed computing model that CORBA adheres to is outlined in the book *Object Management Architecture Guide* [SS96]. The model and all other CORBA specifications are developed by the Object Management Group (OMG), a consortium of software vendor and user organizations. Application systems and the CORBA infrastructure, including the Security Service, are defined using standard CORBA declarative facilities.

All entities in the CORBA computing model are specified by means of data structures and

interfaces defined in the OMG Interface Definition Language (IDL) [OMG04]. The IDL resembles declarative elements of C++ in its syntax and constructs. A CORBA interface is a collection of three elements: operations, attributes, and exceptions. Interface definitions can inherit other interfaces to allow for interface evolution and composition. The CORBA standards also define how IDL constructs are translated into various programming languages. The OMG standardized multiple language bindings, which means that CORBA objects—the implementations of the interfaces—can be coded in different programming languages and yet interoperate with clients and each other.

When CORBA objects are deployed, they reside in OS processes and utilize CORBA middleware in the form of Object Request Broker (ORB) and object adapters to make their functionality available to the clients as well as to receive and process invocations and to return the results. Objects can act as clients as well, that is, make invocations on other objects, creating chains of invocations. Clients and targets may reside in the same or different processes or on different hosts. A CORBA ORB is responsible for core middleware functions, such as registering, keeping track of, and finding interface implementations, aiding clients in connecting to the objects, and providing communication transport from a client to a target.

Even though in theory, any CORBA client can invoke any CORBA object as long as the client has a valid IOR for that object, the overwhelming majority of practical scenarios involve clients and objects from same CORBA *deployments*, where all entities share the underlying security technology and often belong to same administrative domain. Such deployments are commonly limited by intranet boundaries or are subject to pre-established business relationships among organizations. An example of the latter kind is Parlay [3GP07], a standardized CORBA-based service for accessing functionality of a telecom network.

2.1.2 Security Subsystem

CORBA Security service [OMG02] (CORBASec for short) defines the content of security-specific General Inter-ORB Protocol (GIOP) service contexts, interoperable object reference (IOR) components, and, most importantly, interfaces to a collection of objects for enforcing a range of security policies. It provides abstraction from an underlying security technology so that CORBA-based applications can be independent from the particular security infrastructure provided by the un-

derlying computing environment.

CORBASec has an extensible model for *subject security attributes* to enable security run-time and administration scalability with possibly large numbers of subjects. Another example of grouping in CORBA security is *policy domains*, which allow scaling on the number of objects. Domains are used for most security policies in CORBA. A third grouping mechanism—also specific to access control—employs *required* and *effective* rights to allow scaling on the number of operations.

Another design goal of CORBASec architecture was to provide totally unobtrusive protection to applications. Most CORBA objects should be *security-unaware*, that is, run securely without any special programming of the application. At the same time, it should be possible for an object to exercise security policies that are application-specific and/or of finer granularity than those enforced by CORBASec run-time. Such objects are referred in CORBASec terminology as *security-aware*. For the purposes of this thesis, we will focus on the means for protecting security-unaware objects.

The three main parts of CORBASec are client security service (CSS), target security service (TSS), and secure channel. Listed in [BD07, Section 2.2.2], their responsibilities are fairly standard. The secure channel between CSS and TSS is established and managed via service context in the GIOP messages. Any GIOP Request/Reply message contains a list of service context elements, which is used by different services for inserting service-specific information into the stream of communications between client and server. CORBASec defines a SecurityAttributeService (SAS) data type, which may be used in GIOP message service context to associate security-specific identity, authorization, and client authentication contexts with GIOP Request and Reply messages.

Similar to other middleware security technologies, security policies in CORBA are enforced completely outside of an application system. Everything, including obtaining the information necessary for making policy decisions, is done before the method invocation is dispatched to the target object. As Figure 2.1 shows, the security enforcement code is executed inside a CORBA security service when a message from a client application to a target object is passed through the ORB. The CORBASec subsystem intercepts an invocation, determines what policy domain(s) a target or a client belongs to, and enforces the policies associated with the domain(s). In the rest of this section, we describe two key CORBASec functions—authentication and security administration. We describe access control in Section 2.3.

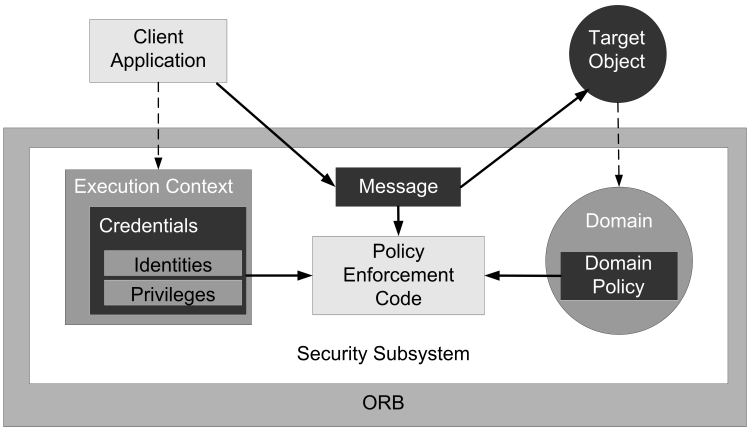


Figure 2.1: Enforcement of policies in CORBA security

The concept of a user is absent from CORBASec. Instead, CORBASec uses the more generic and abstract notion of *principal*. "A principal is a human user or system entity that is registered in and authentic to the system" [OMG02, p.2-3]. It is important for the analysis of RBAC support in CORBA to note that the notion of a session is indistinguishable from the notion of a principal's credentials. Thus the same principal might be represented by multiple, and possibly different, credentials. Just like a session in the ANSI RBAC model, once a Credentials object has been created, it begins to exist completely independently from other such objects, even those created to represent the same principal.

To create credentials, a CORBA application uses a UserSponsor to authenticate the principal to the CORBA Security environment. A UserSponsor is an implementation artifact that authenticates on behalf of a principal with and obtains authenticated credentials from a PrincipalAuthenticator. Instances of UserSponsor implement user interfaces specific to the authentication methods supported by the concrete implementations of CORBASec. CORBASec does not mandate any particular authentication method; what it does specify, however, is the interface of a PrincipalAuthenticator. This object conducts the actual authentication and creates a Credentials object for a new principal. Based on the authentication data it received from the UserSponsor and on the underlying security technology (e.g., Kerberos [NT94], SESAME [PP95]), PrincipalAuthenticator instantiates the Credentials with various information. The client's ORB associates Credentials object with requests on CORBA objects.

The authenticated security attributes of the principal are part of the information stored in the

Credentials object. Hereafter, we understand attribute to mean security attribute. The TSS uses these attributes to decide which operations this principal can invoke on the target object ("target" for short). A variety of privilege attributes may be available, depending on the access policies. At any given time, the principal may be using only a subset of these permitted attributes, chosen either by the principal or by using a default set specified for the principal. There may be limits on the duration these privilege attributes are valid for and controls on where and when they can be used. These attributes, once established through principal authentication, are carried from CSS to TSS in the security-specific service context elements of GIOP messages.

CORBASec administration architecture rests on three constituents—*administrative interfaces*, defined on *policy* objects, each associated with a *policy domain*. CORBASec specifies administrative interfaces for managing most security runtime mechanisms described above, except authentication.[1] As with anything else in CORBA, these interfaces are defined in IDL. Since the mechanisms for user-account management are beyond CORBASec's scope, the interfaces for administering user-attribute assignment policies are as well. There are several types of policies; one of them is *access policy*.

The policy enforcement code uses three sources of information: (1) the information from the client's credentials, (2) the message itself, which specifies the target object and the name of the method to be invoked, and (3) the policy of the domain to which the target belongs.

Any policy is associated with a policy domain—an abstraction that allows security administrators to group objects in groups and assign policies to the groups. Domains allow the application of access control and other policies to security-unaware objects without requiring changes to their implementations or interfaces. Policy domains are also the means by which CORBASec runtime and administration mechanisms achieve scalability on the number of objects in a system. Policies of more than one type (for example, access, audit, message protection) can be associated with the same policy domain.

The policy domain abstraction is represented in CORBASec by DomainManager objects. Whereas the management of domain membership is implementation dependent, an application can invoke the get_domain_managers operation on an object reference to obtain a list of the immediately en-

[1] For authentication, an administrator can still specify whether a target can be authenticated and/or requires its clients to authenticate.

closing domain managers for that object. The structure of the domain organization is determined by the relationships among DomainManagers. Even though an object can belong to more than one policy domain, CORBASec v1.8 specification states that it "does not require support for overlapping or hierarchical security policy domains" [OMG02, p. F-6]. As a result, there is no standard semantics for making access control decisions for object belonging to several domains or for domain hierarchies. Before describing access control architecture of CORBASec in detail, we review related work.

This work builds on an earlier analysis of support for RBAC96 in CORBASec [BD99], where Beznosov and Deng suggest how RBAC_{0-3} models could be implemented in CORBA. Similarly to this work, their results indicate that—aside from conforming to the CORBASec specification—additional functionality needs to be implemented in order to support RBAC96 data models in CORBA. The main differences with [BD99] are in (1) the target model and, as a consequence, the outcomes of the analysis, and (2) extension and correction of the previous analysis. While the data models (e.g., roles, users, permissions, assignments of users and permissions to roles) defined for ANSI RBAC and RBAC96 are largely the same, ANSI RBAC additionally provides a detailed functional specification, which constitutes about 3/4 of the standard. Results of our analysis show that it is the functional specification of ANSI RBAC that CORBASec mostly fails to support. As summarized in Section 4.4, only 3 out of 21 defined functions can be completely supported by a CORBASec implementation.

We extend and correct analysis by Beznosov and Deng as follows. First, we extend their definition of the CORBA protection state with the operational definition of the function access_allowed, demonstrating that the definition of the protection state is sufficient for computing access control decisions in CORBA (Section 2.4). Second, we introduce a formal translation from RBAC to CORBASec (Section 2.7). Third, we identify three major shortcomings of CORBASec (Sections 2.8 and 4.4), specifically (1) the lack of a standard mechanism for enumerating all objects that implement the DomainAccessPolicy interface in a CORBA deployment, (2) the lack of the notion of user accounts and support for their management, as well as the lack of explicit user representation, and (3) the inability to enumerate all CORBA principals related to a specific user. Based on these findings, we conclude, unlike [BD99], that CORBASec is largely inadequate for implementing ANSI RBAC functions without resorting to vendor-specific extensions of a CORBAsec

implementation.

2.2 CORBA Protection State

One of the two major contributions of this thesis is a formalization of the CORBA protection state, which is defined in Section 2.4. We explain first the architecture of the access control mechanisms in CORBA.

2.3 CORBA Access Control Architecture

Due to its general nature, CORBASec is not tailored to any particular access control model. Instead, it defines a general mechanism that is supposed to be adequate for the majority of cases and can be configured to support various access control models. For example, implementing lattice-based mandatory access control (MAC) using CORBASec is shown in [Kar00].

Access control policies in CORBASec are expressed through *security attributes* of principals, attributes of objects, and operations implemented by those objects. Because CORBASec defines an extensible attribute model, it enables access control policies based on roles, groups, clearance, and any other security-related attributes of the principal. From the access control model point of view, a Credentials object is nothing but a set of authenticated attributes. An attribute is a four-tuple ($a = \{\tau, \alpha, \upsilon, \delta\}$) with certain type τ, defining authority α, value υ, and delegation state δ, where $\delta \in DS = \{i, d\}$. State i indicates an attribute possessed by the immediate invoker, and d – by the intermediate one (i.e., delegated). Attribute types are partitioned into two families: privilege attributes and identity attributes. The family of privilege attributes enumerates attribute types that identify principal privileges: access id, primary and secondary groups the principal is a member of, clearance, capabilities, etc. Identity attributes, if present, provide additional information about the principal: audit id, accounting id, and non-repudiation id, reflecting the fact that a principal might have various identities used for different purposes. Principal credentials may contain zero or more attributes of the same family or type. The role attribute is one of the standard CORBA attribute types. Due to the extensibility of the schema for defining security attributes, an implementation of CORBASec can support attribute types that are not defined by the CORBASec standard. Although the normative part of CORBASec does not mandate the way

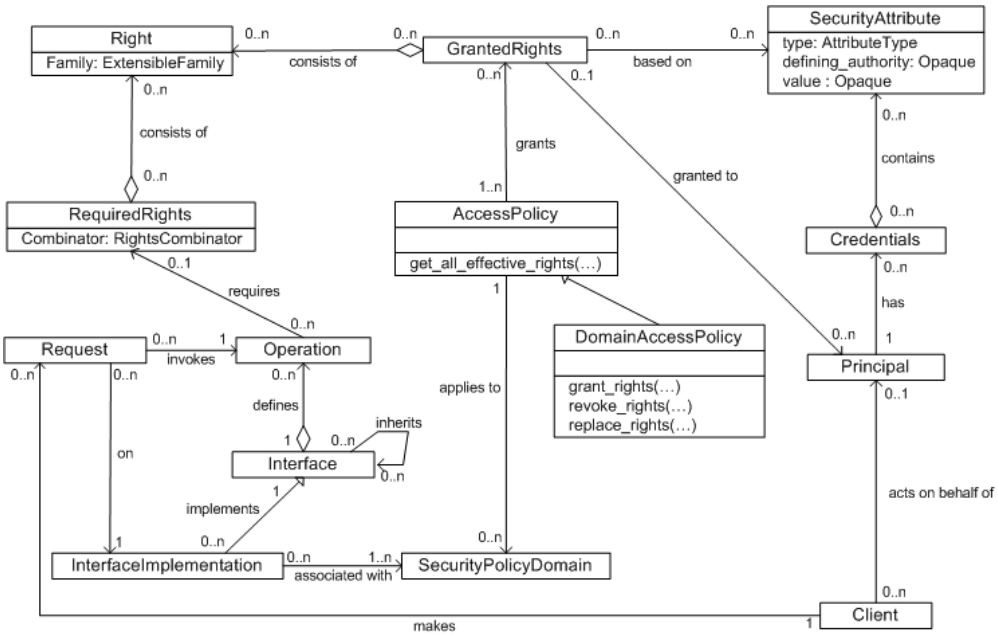


Figure 2.2: A model of CORBASec access control architecture in UML notation.

attributes are managed, assignment of such attributes to users is meant to be performed by user administrators.

In the CORBA computational model, all a principal does is to invoke operations on corresponding objects. In order to make a request, one needs to know two things: object reference, which uniquely identifies an object, and operation name. An operation name is unique for an interface.[2] Thus, any operation is uniquely identified by its name and by the name of the interface it is defined in. In this thesis, we use the notation $i.m$, to refer to operation (a.k.a. method) m on interface i. There is a global[3] set of *required rights* for each operation defined by its interface's required rights mapping. The required rights set, together with a combinator (*all* or *any* rights), defines what rights a principal has to have in order to invoke the operation. Table 2.1b provides an example of required rights for operations on three interfaces, i_1, i_2, i_3. It is assumed that required rights are defined and that their semantics are precisely documented by application developers who best know what each operation does. CORBASec Level 2 API defines the operation set_required_rights(operation, interface, rights, rights combinator) for managing required rights.

[2]Interface inheritance in CORBA does not allow inheritance from interfaces with operations of the same name. This rule resolves the problem of operation name overloading.

[3]"Global" in the context of required rights means that they are independent of the policy domain in which the object is located.

Figure 2.2 is useful for illustrating our discussion. Depending on the access policy (DomainAccessPolicy) enforced in a particular access control policy domain, a principal is granted different rights (GrantedRights) according to what SecurityAttributes it has.[4] Each DomainAccessPolicy defines what rights are granted for each security attribute. An example of a mapping between principal privilege attributes and granted rights is provided in Table 2.1c. Security administrators are responsible for defining what rights are granted to what security attributes in what delegation state on a domain by domain basis. CORBASec Administrative API defines operations grant_rights(attribute, rights) (as well as revoke_rights, and replace_rights) for managing rights granted for an attribute in the scope of a particular policy domain.

Whenever a principal attempts to invoke an operation, its effective rights are computed via operation AccessPolicy::get_all_effective_rights(...). CORBASec purposely does not define how the operation combines rights granted through the different privilege attribute entries in Table 2.1c. The specifiers let CORBASec implementors define the operation's semantics ([OMG02, p. 2-123]). The simplest implementation of get_all_effective_rights would be when the set of rights granted to a principal is a union of rights granted to every security attribute possessed by the principal. For the rest of this chapter, we will assume these semantics for the operation. If we use our example of security attributes assigned to principals p_1, p_2, p_3, and p_4 (Table 2.1a), and granted rights (Table 2.1c), then Table 2.1d shows what "effective" rights the principals have in each domain.

The use of effective rights and policy domains makes the correspondence between the ANSI RBAC OBS set and CORBA objects nontrivial. Note that the effective rights of the invoking principal are computed for the object's policy domain. At the same time, all instances of the same interface implementation that belong to the same domain are indistinguishable for the purpose of making access control (and other policy) decisions. That is, a principal has exactly the same permissions on all objects that implement the same interface(s) and belong to the same domain. To accommodate this important detail, we defined the ANSI RBAC OBS set as a cross-product between CORBA interfaces and access policy domains: $I \times D$.

Once the principal's effective rights are determined, they are compared to the rights required for the operation. If the match is successful, the request is authorized. Given the required rights in

[4] For the sake of brevity, we omit the delegation state qualifier for granted rights. This omission does not change the correctness of the discussion, as we show below.

Principal	Attributes
p_1	a_1
p_2	a_2, a_6
p_3	a_2, a_3
p_4	a_4, a_5

(a) An example of security attributes possessed by authenticated principals.

Operations	**Required Rights**	**Combinator**	**Meaning**
$i_1.m_1$	r_1	all	Only a principal who is granted right r_1 can invoke the operation.
$i_1.m_2$	r_1, r_2	any	Any principal who is granted either r_1 or r_2 right can invoke the operation.
$i_2.m_1$	r_2, r_3	all	Only a principal who is granted both r_2 and r_3 rights can invoke the operation.
$i_2.m_2$	r_2, r_3, r_4	all	Only a principal who is granted all r_2, r_3, r_4 rights can invoke the operation.
$i_3.m_1$	r_1, r_2, r_3, r_4	all	Only a principal who is granted r_1, r_2, r_3, and r_4 rights can invoke the operation.

(b) Required rights matrix

Attributes	**Granted Rights**	
	Domains	
	$\mathbf{d_1}$	$\mathbf{d_2}$
a_1	r_1	r_2
a_2	–	r_1
a_3	r_2, r_3	–
a_4	r_3	r_1, r_4
a_5	r_1, r_2, r_3	r_2, r_3, r_4
a_6	r_6	r_1

(c) Granted rights per attribute

Principal	**Granted Rights**	
	Domains	
	$\mathbf{d_1}$	$\mathbf{d_2}$
p_1	r_1	r_2
p_2	r_6	r_1
p_3	r_2, r_3	r_1
p_4	r_1, r_2, r_3	r_1, r_2, r_3, r_4

(d) Effective rights of principals in each of the two domains

Principals	**Permitted Operations**	
	Domains	
	$\mathbf{d_1}$	$\mathbf{d_2}$
p_1	$i_1.m_1, i_1.m_2$	$i_1.m_2$
p_2	–	$i_1.m_1, i_1.m_2$
p_3	$i_1.m_2, i_2.m_1$	$i_1.m_1, i_1.m_2$
p_4	$i_1.m_1, i_1.m_2, i_2.m_1$	$i_1.m_1, i_1.m_2, i_2.m_1, i_2.m_2, i_3.m_1$

(e) Operations that principals from the example can invoke

Table 2.1: Sample CORBASec configuration (adapted from [BD99])

Table 2.1b and the rights granted to the principals in Table 2.1d, Table 2.1e shows what operations can be invoked by the principals from our example.

2.4 Formalization of the Protection State

In this section, we formalize the semantics of the CORBA access control architecture.

Definition 2 ***[CORBA privilege attributes]*** *CORBA privilege attributes are* $A \subseteq T \times AUTH \times V \times DS$*, where* $T, AUTH, V, DS$ *are interpreted as follows:*

- $T = \{\_Public, AccessId, PrimaryGroupId, GroupId, Role, AttributeSet, Clearance, Capability\}$ *is the set of types.*
- $AUTH$ *is the set of authorities.*
- V *is the set of values.*
- $DS = \{i, d\}$ *is the set of delegation states.*

Definition 3 ***[CORBA Protection State]*** *A configuration of a CORBA system protection state is a tuple (*I*,* OPS*,* $IOPS$*,* $RIGHTS$*,* RR*,* D*,* $DOBS$*,* A*,* GR*,* get_all_effective_rights*) interpreted as follows:*

- I *is the set of interfaces.*
- OPS *is the set of operations on CORBA objects.*
- $IOPS \subseteq I \times OPS$ *specifies which operations are defined on which interfaces.*
- $RIGHTS$ *is the set of rights.*
- $RR \subseteq IOPS \times 2^{RIGHTS}$ *defines rights required for invoking operations on interfaces.*
- D *is the set of security policy domains.*
- $Inst$ *is the set of CORBA objects.*
- $DOBS \subseteq Inst \times D$ *associates each object with zero or more policy domains.*

- *A is the set of privilege attributes as specified in Defintion 2.*
- *GR* $\subseteq A \times (D \times RIGHTS)$ *associates an attribute with a domain and a right;* $(a, d, r) \in GR$ *means that attribute* a *is granted right* r *in domain* d*.*
- get_all_effective_rights*:* $D \times 2^A \rightarrow 2^{RIGHTS}$*, a function computing rights that are in effect for a given set of privilege attributes in a given domain. Although this function uses GR to obtain rights granted for each attribute, the semantics of combining the granted rights are implementation-specific.*

An implementation of security service compliant with CORBASec is supposed to yield the same access control decision as that described by Algorithms 1 and 2. Employed by Algorithms 1 and 2, functions get_domain_policy and get_all_effective_rights are defined by CORBASec.

access_allowed$(u : 2^A, m : OPS, o : Inst, i : I) \rightarrow \{true, false\}$
Require: $(i, m) \in IOPS$
1: **for all** $(o, d) \in DOBS$ **do**
2: {Find an access policy domain.}
3: $p \leftarrow get\_domain\_policy(d,$"AccessPolicy")
4: **if** $p \neq NULL$ **then**
5: **return** $is\_authorized(u, i, m, d)$
6: **end if**
7: **end for**
8: **return** $is\_authorized(u, i, m, NULL)$

Algorithm 1: Operational definition of function access_allowed. This function makes the access control decision with regard to principal u accessing operation m on instance o of interface i. If no policy domain with an access policy found, then this case is signaled with NULL for the domain parameter.

CORBASec standard is unclear about cases when an object does not belong to any domain that has *AccessPolicy* or it belongs to several such domains. To resolve the ambiguity, we chose Algorithm 1 to use first domain of the object that has *AccessPolicy*. Because a policy domain might not have *AccessPolicy*, the algorithm iterates until it finds a domain that does (lines 1-7). If no such domain is found, then the algorithm passes NULL for the domain argument to is_authorized (line 8), which will result in the empty set of effective rights. Whether this would lead to a denial of access, depends on the required rights. If an object has no required rights specified, the request would be denied to all users, as Karjoth points out [Kar00].

is_authorized$(u : 2^A, i : I, m : OPS, d : D) \rightarrow \{true, false\}$

```
1: er ← get_all_effective_rights(d, u)
2: if ∃ (i, m, rr) ∈ RR : rr ⊆ er then
3:    return true
4: else
5:    return false
6: end if
```

Algorithm 2: Operational definition of function is_authorized.

We separated authorization logic into two functions. This separation is purely syntactical and its only purpose is to demonstrate in Section 2.8 to the reader the capability of the CORBASec to provide an implementation of ANSI RBAC's CheckAccess in the form of is_authorized. This function is the same as access_allowed, except that it makes an authorization decision for a given domain d and particular operation m on CORBA interface i to be accessed by principal u. In Algorithm 2, the operation get_all_effective_rights retrieves granted rights and combines them according to its implementation semantics. Effective rights of the principal in the object's domain are checked then against RR. If the match succeeds, then access is granted. Otherwise, access is denied. An example of an algorithm for get_all_effective_rights that returns a union of the rights granted per each attribute are shown in Algorithm 3.

get_all_effective_rights$(d : D, u : 2^A) \rightarrow 2^{RIGHTS}$

```
1: if d ≡ NULL then
2:    return ∅
3: end if
4: er ← ∅
5: for all a ∈ u do
6:    for all (a, d, r) ∈ GR do
7:       er ← er ∪ r
8:    end for
9: end for
10: return er
```

Algorithm 3: Operational definition of a sample function get_all_effective_rights that returns a union of all rights granted to principal u in domain d.

We simplified the semantics of the support for the required rights combinator in the definition of RR and Algorithm 2. Combinator value "any" is supported via separate elements of RR. For example, if either r_1 or r_2 are required for operation $i.m$, then $(i.m, \{r_1\}) \in RR$ and $(i.m, \{r_2\}) \in RR$. Whereas, combinator value "all" is supported by listing all the required rights

Subjects	**Interfaces**		
	i_1	i_2	i_3
p_1	$i_1.m_1$		
p_2	$i_1.m_1, i_1 m_2$		
p_3	$i_1.m_1, i_1.m_2$		
p_4	$i_1.m_1, i_1.m_2$	$i_2.m_1, i_2.m_2$	$i_3.m_1$

Table 2.2: Access matrix for domain d_2

in one element of RR, e.g., $(i.m, \{r_1, r_2\})$.

For each domain, a Lampson's access matrix [Lam71], such as that one in Table 2.2, can be constructed. Three general observations are worth noting regarding an access matrix constructed for any CORBASec system. First, subjects cannot be objects, i.e., the CORBA access control model does not support the concept of operations on principals. It only has the concept of operations on interfaces, which are objects according to the terminology of the access matrix [Lam71]. Second, since $i_k.m_p \equiv i_l.m_q \Longleftrightarrow k \equiv l \wedge p \equiv q$ (i.e., just $p \equiv q$ is not enough for $i_k.m_p \equiv i_l.m_q$), the semantics of the operations with same names but defined on different interfaces in a general case might be different. Thus, for each subject s and object o, the content of cell $[s, o]$ is specific to the object. That is, no operations permitted on one object can be permitted on another, because operations are semantically different for every interface unless the interfaces are related through inheritance. Third, since those implementations of the same interface that are located in the same access policy domain are indistinguishable from the access control point of view, all such interface implementations are represented by the same object in the access matrix. This is one of the reasons policy domains are important in the CORBA access control model.

Before we proceed to our analysis of the support for ANSI RBAC in CORBA, we would like to note that not all sets from Definition 3 can be enumerated. Particularly, we could not find operations in CORBA specifications that allow enumerating $RIGHTS, Inst, A, D, OPS, I, IOPS$ sets. As a consequence, a number of ANSI RBAC functions cannot be supported without resorting to implementation specifics. However, membership in the last three sets can be tested through the operation get_required_rights specified on the interface RequiredRights.

The lack of standard mechanisms for enumerating all objects (*Inst*) in a given CORBA deployment accounted for the inability of CORBASec to support *RolePermissions* and, consequently,

SessionPermissions functions of the ANSI RBAC functional specification. The implementation of these two functions requires enumeration of effective rights for a given role or principal. In order to do so, it is necessary to obtain a reference to every object that implements interface DomainAccessPolicy and to invoke the operation get_all_effective_rights on it.

2.5 Analysis of ANSI RBAC support in CORBA

Recall that among the four sets of ANSI RBAC features (also referred as *model components*), Core RBAC is the required minimum for any implementation compliant with the standard. A system supporting Core RBAC must implement functions for administering user accounts, roles, sessions, objects, operations, and permissions. Hierarchical RBAC has hierarchies of roles in addition to everything Core RBAC has. The last two standard's components, Static Separation of Duty (SSD) Relations and Dynamic Separation of Duty (DDS) Relations, define relations among roles with respect to user assignments as well as role activation in user sessions.

We first examine in Section 3.3.1 the extent to which a CORBA protection state—as formalized in Definition 3—can support each of the four ANSI RBAC model components. Second, we describe in Section 2.7 steps for translating an ANSI RBAC policy into a CORBA protection state. Two examples that illustrate the results of our analysis can be found in [BD07, Section 5.3]. Finally, we analyze in Section 2.8 the degree to which the programming interfaces defined in CORBASec and other related parts of the CORBA specification support the functional specification of ANSI RBAC. We discuss results of our analysis in Section 4.4.

2.6 Reference Model

2.6.1 Core RBAC

The five sets of Core RBAC identities are represented in CORBA Security as follows: *Users* in RBAC map to user accounts in CORBASec; *Roles* are represented by a set of privilege attributes of type *role*; each RBAC object is a collection of CORBA objects that implement the same interface(s) and belong to the same access policy domain(s), and are thus indistinguishable from the point of view of a CORBA protection system; *Permissions* are operation-object pairs; RBAC

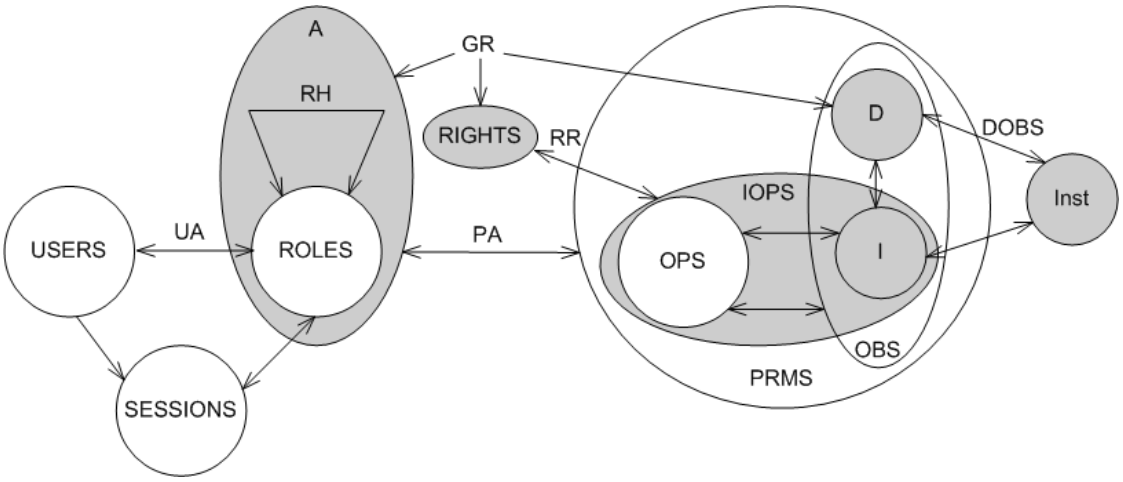


Figure 2.3: RBAC (with white background) and CORBA (with light grey background) sets and relations.

Sessions are equivalent to CORBA principals, which can be reduced for the purpose of this thesis to just sets of security attributes. We do not mention CORBASec access control domains because, as shown in [BD07], ANSI RBAC models can be supported in CORBA using either a single domain or multiple domains. To aid with the understanding of the correspondence between elements in the RBAC model and CORBASec, we present in Figure 2.3 RBAC (with white background) and CORBA (with light grey background) sets and relations. The reader is encouraged to compare it to the diagram in Figure 1.1.

The Core RBAC in the language of CORBA Security is formally defined as follows:

Definition 4 ***[Core RBAC in CORBASec]*** *Core RBAC in the language of CORBA Security is defined by the CORBA system protection state outlined in Definition 3, as well as the following additional elements:*

- *USERS is the set of user accounts.*
- *ROLES* $\subseteq$ *A roles, which are CORBA privilege attributes of type* role.

  Formally: $ROLES = \{a | a \in A \wedge T(a) \equiv Role\}$.
- *OBS* $\subseteq I \times D$ *set of objects distinguishable from the point of view of access control in CORBA. That is, for any two elements of OBS, there could be a CORBA protection system state in which the same principal p have different access rights on these elements, even if they both*

implement the same interface(s). An ANSI RBAC object is mapped into a tuple (i, d)*, i.e., a CORBA interface and the access policy domain it is a member of.*

- $UA \subseteq USERS \times ROLES$*, a many-to-many user-account-to-role assignment relation.*
- *assigned_users*$(r : ROLES) \rightarrow 2^{USERS}$*, the mapping of role* r *onto a set of user accounts, as in ANSI RBAC.*
- $PRMS \subseteq OPS \times OBS$ *the set of permissions. A permission can be considered as a three-tuple* (op, i, d)*, i.e., operation, interface, and domain.*
- *assigned_permissions*$(r : ROLES) \rightarrow 2^{PRMS}$*, the mapping of role* r *onto a set of permissions. Function* assigned_permissions *is specified operationally by Algorithm 4.*
- $Op(p : PRMS) \rightarrow OPS$*, the permission to operation mapping, which gives the operation associated with permission.*
- $Ob(p : PRMS) \rightarrow OBS$*, the permission to RBAC object mapping, which gives the object associated with permission.*
- $domain(p : PRMS) \rightarrow D$*, the permission to CORBA access policy domain mapping, which gives the domain associated with the permission. The mapping is used by Algorithm 4.*
- $interface(p : PRMS) \rightarrow I$*, the permission to CORBA interface mapping, which gives the interface associated with the permission.*
- $PA \subseteq PRMS \times ROLES$*, a many-to-many permission-to-role assignment relation, defined through the function assigned_permissions.*
- $SESSIONS \subseteq 2^A$*. RBAC sessions are represented by CORBA principals, which in their turn can be treated for the purpose of access control as sets of security attributes from* A*.*
- *session_users*$(s : SESSIONS) \rightarrow USERS$*, the mapping of a session onto the corresponding user account.*
- *session_roles*$(s : SESSIONS) \rightarrow 2^{ROLES}$*, the mapping of a session onto a set of roles.*

  Formally: $session\_roles(s_i) \subseteq \{r \in ROLES | (session\_users(s_i), r) \in UA\}$*.*

- *avail_session_perms*$(s : SESSIONS) \rightarrow 2^{PRMS}$, *the permissions available to a session* $= \bigcup_{r \in session\_roles(s)} r \in assigned\_permissions(r)$.

assigned_permissions$(r : ROLES) \rightarrow 2^{PRMS}$

```
1:  AP ← ∅ {Initialize the set of assigned permissions to return}
2:  for all p ∈ PRMS do
3:     i ← interface(p)
4:     m ← Op(p)
5:     d ← domain(p)
6:     if is_authorized({r}, i, m, d) then
7:        AP ← AP ∪ p
8:     end if
9:  end for
10: return AP
```

Algorithm 4: Operational definition of function *assigned_permissions*, which determines permissions assigned to a given role in a CORBA system.

Definition 4 specifies all elements of Core RBAC. The elements *PRMS*, *Op*, and *Ob* require further elaboration. The definition of *PRMS* in ANSI RBAC allows each permission to comprise multiple operation-object pairs. A CORBA permission, on the other hand, consists of only one such pair, which can be considered as a more restricted case of ANSI RBAC *PRMS*, i.e., $OPS \times OBS \subset 2^{OPS \times OBS}$. The ranges of functions *Op* and *Ob* are elements, not subsets, of *OPS* and *OBS*, respectively. Given that an element of a set also comprises a subset of the set, ANSI RBAC versions of *Op* and *Ob* functions can be substituted by their counterparts from Definition 4. Thus, *PRMS*, *Op*, and *Ob* from Definition 4 can be used instead of the corresponding elements of Core ANSI RBAC.

In the rest of this section, we explain how elements of Definition 4 are or can be supported by CORBASec. Because the notion of user accounts is missing from CORBASec, the set *USERS*, relation *UA*, and functions *assigned_users* and *session_users* have to be implementation-dependant. The enumeration of elements from *SESSIONS*, which we have defined as a set of CORBA principals (Definition 4), was found to be not supported by CORBA specifications either. However, CORBASec's SecurityLevel1.Current interface does define the operation get_attributes, which returns a list of security attributes of the principal responsible for the current invocation. For the purposes of our analysis, the returned list of security attributes was sufficient to represent the current principal and therefore the current session.

An implementation of function *session_roles* is straightforward because an ANSI RBAC session is a principal in CORBA, and a principal is a set of security attributes, each of a particular type. Thus, all *session_roles* needs to do is to return those principal's attributes whose type is *role*.

As we explained at the end of Section 2.4, the CORBA specification does not define operations sufficient for enumerating all CORBA objects in a given CORBA deployment. An affirmative answer is necessary in order to enumerate the elements of the *PRMS* set, on which our operational definition of the *assigned_permissions* function depends (specifically, line 3). Thus, the functionality necessary for enumerating the *PRMS* set would have to be implementation-dependent.

The interfaces and data structures defined by CORBASec enable, however, the construction of individual elements of set *PRMS*. To demonstrate, consider data used for making access control decisions in CORBA. For any given request on a CORBA object, the CORBA security subsystem intercepts the request and invokes access_allowed (Algorithm 1) with the following parameters: subject's credentials, object's reference, the operation to be invoked, and the name of the interface on which the operation is defined. In its turn, access_allowed obtains the access policy domain that object *o* is a member of, before invoking *is_authorized* (defined by Algorithm 2). Thus, at the time when *is_authorized* is invoked, all data necessary for constructing a corresponding permission, as specified in Definition 4, are available to the CORBASec subsystem.

Due to the structure of permission, (op, i, d), valid implementations of the functions *Op*, *Ob*, *domain*, as well as *interface*, could just return the corresponding parts of the permission argument. For example, *Ob* needs to return the (i, d) tuple.

The function *avail_session_perms* is operationally defined by Algorithm 5. Also, see the caveat about the related function *SessionRoles* in Section 2.8.

avail_session_perms$(s : SESSIONS) \rightarrow 2^{PRMS}$

1: $AP \leftarrow \emptyset$ {Initialize the set of available permissions to return}
2: **for all** $r \in session\_roles(s)$ **do**
3: $\quad AP \leftarrow AP \cup assigned\_permissions(r)$
4: **end for**
5: **return** AP

Algorithm 5: Operational definition of the function *avail_session_perms*, which determines permissions available to a given session.

As can be seen from the above analysis of Definition 4, most elements of ANSI Core RBAC can

be provided by any implementation compliant with CORBA Security Main Functionality Level 2. However, support for user-specific elements of the Core RBAC, as well as for enumerating such sets as *SESSIONS* and *PRMS*, must be implementation-specific.

2.6.2 Hierarchical RBAC

In order to implement ANSI RBAC role hierarchies, a system—in addition to Core RBAC—has to provide support for modifying and reviewing a partial-order relation on roles, RH, and the functions *authorized_users* and *authorized_permissions* that are defined on RH. CORBASec does not provide direct support for RH and the two functions. An implementation, however, can emulate the support for role hierarchies—either *general* or *limited*—in three different ways.

First, PrincipalAuthenticator can activate not only those roles that can be activated through direct user-to-role assignment but also the roles junior to those activated. For example if $r' \succeq r$, and r' has been activated, then r is also activated. Proposals by [SA98, AS01] follow a similar path. In this case, the RH logic can be encapsulated into PrincipalAuthenticator, whereas the target security service and other CORBASec components provide no special support for role hierarchies. A valid implementation of Hierarchical RBAC using PrincipalAuthenticator could be one (a) that allows a user to specify any role junior to those the user is a member of; and (b) in which PrincipalAuthenticator activates the specified role(s) as well as all roles junior to the specified one(s).

The second choice is to shift support for role hierarchies to the TSS. Specifically, get_all_effective_rights would be required to return not only effective rights for the activated roles, but also for all roles junior to the activated ones, as in [Giu99]. Using the above example, a call to the modified version of get_all_effective_rights$(d, \{r'\})$, would be equivalent to get_all_effective_rights$(d, \{r', r\})$. This option requires maintenance of RH and run-time access to it by the TSS. Since in CORBASec, the credentials of the principal are always "pushed" from the client to the server, we found no opportunity to support Hierarchical RBAC by adding role attributes to the client's credentials by TSS, as proposals [Bar97, FBK99, Ahn00, PSA01, CO02, ZM04] do.

The third option is to modify the administrative tools—similarly to [PSA01]—to ensure that the CORBASec rights that are granted to every role include the rights this role inherits from the

junior roles. No special run-time support for role hierarchies would then be needed. However, this option requires not only maintaining RH but also keeping track of the reason(s) a right was assigned to a role, i.e., because of direct assignment or through inheritance from a particular role. Such assignment details would be necessary in order to perform right revocation and RH administration properly.

No matter which of the three options is selected, support for RH, *authorized_users*, and *authorized_permissions* would be implementation-specific.

2.6.3 Constrained RBAC

The Constrained RBAC component of ANSI RBAC [ANS04] introduces static and dynamic separation of duty relations to the RBAC reference model. In essence, SSD constrains user-to-role assignment (UA set and *assigned_users* function) and the role hierarchy (RH set and *authorized_users* function). DSD, on the other hand, constrains the role activation (*SESSIONS* set and *session_roles* function). Since user accounts, role hierarchies, and role activation are beyond the scope of CORBASec, the Constrained RBAC component, if supported, would have to be implementation-dependant.

2.7 Translating RBAC Policies to CORBA

An interesting and practical question is the translation of an arbitrary ANSI RBAC policy into a CORBA protection state. The key elements of an RBAC policy are the user-to-role (UA) and permission-to-role assignment (PA) relations. Given that the management of user accounts and their security attributes is beyond the scope of CORBA standards, the question boils down to the PA relation. In Algorithm 6, we define a sequence of steps that allows a given PA defined in ANSI RBAC terms to translate into the required (RR) and granted (GR) rights assignments and the assignment ($DOBS$) of CORBA objects ($Inst$) to policy domains (D).

For this purpose and without loss of generality, we assume that sets and relations I, OPS, $IOPS$, and $Inst$ are defined by the virtue of developers creating a CORBA system. Our approach requires that (1) all elements of PRMS are "atomic", i.e., each permission comprises only one object-operation pair, and (2) operations and objects of PRMS match the CORBA system at

hand. The former requirement can be lifted by translating compound permissions into atomic ones before executing Algorithm 6. The second requirement is necessary for eliminating manual mapping between RBAC operations and objects and those of the target CORBA system.

1: {Initialize CORBASec sets and relations.}
2: $D \leftarrow \emptyset$
3: $DOBS \leftarrow \emptyset$
4: $RIGHTS \leftarrow \emptyset$
5: $RR \leftarrow \emptyset$
6: $A \leftarrow \emptyset$
7: $GR \leftarrow \emptyset$
8: **for all** $p \in PRMS$ **do**
9: $o \leftarrow Ob(p)$
10: **if** $\neg\exists(o, d_o) \in DOBS$ **then**
11: create a separate access policy domain d_o for o
12: $D \leftarrow D \cup \{d_o\}$
13: $DOBS \leftarrow DOBS \cup \{(o, d_o)\}$
14: **end if**
15: create new right $right_p$
16: $RIGHTS \leftarrow RIGHTS \cup \{right_p\}$
17: $RR \leftarrow RR \cup \{(interface(p), Op(p), \{right_p\})\}$
18: **for all** $(p, r) \in PA$ **do**
19: $A \leftarrow A \cup \{r\}$
20: $GR \leftarrow GR \cup \{(r, d_o, right_p)\}$
21: **end for**
22: **for all** $(r' \succeq r) \in RH$ **do**
23: $GR \leftarrow GR \cup \{(r', d_o, right_p)\}$
24: **end for**
25: **end for**
26: **return** AP

Algorithm 6: Operational definition of translating from an ANSI RBAC system state to the one of CORBA. Lines 22-24 are only necessary for the third option of supporting Hierarchical RBAC in CORBA, as discussed in Section 2.6.2

The main advantage of the above approach is the straightforwardness of the initial translation and the simplicity of future incremental modifications to the policy. Granting/revoking a permission to/from a role requires adding/removing an association between a right and a role in the object's domain in the GR relation. Adding/removing an object results in creation/deletion of a policy domain (and possibly several rights, if the object implements a unique interface). The main disadvantage of this approach is the proliferation of rights and domains. The above steps result in the creation of as many rights as the number of unique interface-operation pairs and as many access policy domains as interface instances. Optimizations of this approach to policy translation,

although conceivable, are not discussed further due to space limitations. Another complementary question—which could be a subject of future research—is how to determine if a given CORBA protection state enforces a given ANSI RBAC policy. To illustrate the results of our analysis of the ANSI RBAC reference model support in CORBA systems, we show in [BD07, Section 5.3] how a CORBA-based distributed system could be configured to enforce a sample RBAC policy using single and multiple policy domains.

Having analyzed the support for the reference model of the ANSI RBAC in CORBA, we move on to presenting results of our analysis with regard to the support of ANSI RBAC functions.

2.8 Functional Specification

This section reports on the the results of our analysis of CORBASec support for system and administrative functional specifications of ANSI RBAC. We examined each ANSI RBAC function defined in Section 6 of [ANS04] on the subject of its support by a CORBASec implementation conforming to Security Functionality Level 2. That is, we did not assume any CORBASec functionality other than that required for Level 2 conformance [OMG02, p.374]. Particular implementations of CORBASec might provide additional functionality, and, as a result, support more ANSI RBAC functions. Examining support for ANSI RBAC on an implementation-by-implementation basis was, however, beyond the scope of this thesis.

Results of our examination suggest that the CORBASec functionality, as defined through the data structures and interfaces in Version 1.8, is largely insufficient for implementing ANSI RBAC functions. Specifically, Hierarchical and Constrained RBAC functions cannot be supported without extending an implementation beyond what CORBASec defines.

Even for Core RBAC, we found that most functions cannot be supported as is, as the summary of our analysis in Table 2.3 indicates. Because the CORBASec specification is not concerned with the administrative and run-time management of user accounts, user attributes, and principals (which are sessions in RBAC terms), the following functions prescribed for Core RBAC implementations cannot be supported without implementation-specific extensions: *AddUser*, *DeleteUser*, *AssignUser*, *DeassignUser*, *AddRole*, *DeleteRole*. The rest of this section discusses implementation of the other Core RBAC functions using CORBASec and its application programming interfaces

(APIs), and identifies the functionality necessary for supporting these functions that is missing from CORBASec.

GrantPermission$((i, d) : OBS, m : OPS, role : ROLES)$

1: $(rr, combinator) \leftarrow$ get_required_rights(i, m)
2: **if** *combinator* is "any" **then**
3: $R \leftarrow$ any right from rr
4: **else**
5: {Combinator is "all"}
6: $R \leftarrow rr$
7: **end if**
8: grant_rights(d, *role*, R)

Algorithm 7: Operational definition of the function *GrantPermission*, which grants a role the permission to perform an operation on an ANSI RBAC object.

GrantPermission, RevokePermission functions enable changes to the permission assignment (PA) set. The CORBASec operations set_required_rights, grant_rights, revoke_rights, and replace_rights (described in Section 2.3) allow modifications to RR and GR, and therefore PA, leading us to conclude that these CORBASec operations are sufficient for implementing *GrantPermission* and *RevokePermission* functions. As an illustration, we provide in Algorithm 7 an operational definition of the function *GrantPermission*, using CORBASec API. *RevokePermission* can be defined likewise, except that more care needs to be exercised in making sure that granted rights are revoked in all domains associated with the given permission.

CreateSession function creates a given session with a given user account as the owner. CORBASec utilizes the notion of the PrincipalAuthenticator—described in Section 2.3—whose functionality is expected to encompass the authentication of the CORBA principal, and create the principal's credentials, the equivalent of the ANSI RBAC session. Thus, even though CORBASec does not define an operation for creating sessions, a functional implementation of CORBASec would either rely on the underlying security infrastructure or implement an equivalent of *CreateSession* utilized by PrincipalAuthenticator. Since the notion of user accounts is missing from CORBA, this function cannot be completely supported without an implementation-specific extension.

DeleteSession function deletes a given session with a given user account as the owner. Even

Core RBAC Functions		**Functionality that needs to be defined to support this function**						
	completely supported	user management	user representation	attribute management	credentials deletion	enumeration of policy domains	attribute to credential translation	retrieving principal's attributes
Administrative Commands								
AddUser		√						
DeleteUser		√						
AssignUser		√						
DeassignUser		√						
AddRole				√				
DeleteRole				√				
GrantPermission	√							
RevokePermission	√							
Supporting System Functions								
CreateSession		√						
DeleteSession		√			√			
AddActiveRole		√						
DropActiveRole		√						
CheckAccess	√							
Review Functions								
AssignedUsers			√					
AssignedRoles			√					
Advanced Review Functions								
RolePermissions						√		
SessionPermissions						√		
UserPermissions		√				√		
SessionRoles								√
RoleOperationsOnObject							√	
UserOperationsOnObject			√					

Table 2.3: Functions defined by ANSI Core RBAC and their support in CORBA

though CORBASec's Credentials interface defines the operation destroy, this and other operations on Credentials can be invoked only within the operating system process where the Credentials object resides. Another limitation stems from the fact that there can be multiple copies of the same Credentials object, making their complete deletion difficult to implement; the CORBA specification does not provide a means for enumerating all copies of a given Credentials object. For the above reasons, and because the notion of user accounts is missing

from CORBA, we concluded that *DeleteSession* would have to be implementation-specific.

AddActiveRole, DropActiveRole functions add/delete a role as an active role of a session whose owner is a given user account. Even though CORBASec does not define a function with semantics compatible to *AddActiveRole*/*DropActiveRole* according to Liskov's *substitution principle* [LW94], it does specify the set_attributes operation on the SecurityLevel2.Credentials interface, enabling a privileged caller to modify attributes on a credential associated with a particular principal. However, the logic for checking the preconditions $session \in user\_sessions(user)$ and $(user \mapsto role) \in UA$ would have to be implementation-specific due to the lack of standardized support for user-account management in CORBA deployments.

CheckAccess returns a Boolean value indicating whether the subject of a given session is allowed, or not, to perform a given operation on a given object. This function is equivalent to is_authorized, which is defined by Algorithm 2.

AssignedUsers, AssignedRoles return the set of users/roles assigned to a given role/user, respectively. Both functions require the notion of user, which is missing from CORBASec, making these functions implementation-specific.

RolePermissions, SessionPermissions return the set of permissions (op, obj) granted to a given role or session, respectively. Implementations of these functions would require querying each instance of the DomainAccessPolicy interface in the given CORBA deployment in order to determine the content of the role's row in the granted rights matrix (see Table 2.1c). However, due to the lack of standard mechanisms for enumerating all objects in a CORBA deployment in general and all DomainAccessPolicy objects in particular, the querying would have to be implementation-specific.

UserPermissions returns the permissions a given user gets through his/her assigned roles. This function would be implementation-specific due to the lack of both standard mechanisms for enumerating all access-policy domains and a standardized support for managing user accounts.

SessionRoles returns the active roles associated with a session. This function is partially supported by the CORBASec. Any compliant implementation of CORBASec must implement

the Current.get_attributes operation, which allows retrieving security attributes of a specific type (e.g., role) for the principal associated with the current execution thread. However, the standard does not define an operation for retrieving attributes for an arbitrary principal or associating it with the current execution thread.

RoleOperationsOnObject returns the set of operations a given role is permitted to perform on a given RBAC object. Unlike the case of *RolePermissions*, support for this function does not require enumerating all DomainAccessPolicy objects. A reference of the corresponding CORBA object in question is sufficient for employing the AccessDecision.access_allowed operation for determining if a given role is allowed to invoke a particular operation on a given object. In order to determine the rights of a given role on all operations of the RBAC object, the CORBA Reflection facility [OMG06] can be used for enumerating operations implemented by the object. CORBASec, however, does not define operations for creating a valid credential—a required format of the input parameter for access_allowed—out of a security attribute (e.g., particular role). Therefore, the translation would have to be implementation-specific.

UserOperationsOnObject returns the set of operations a given user is permitted to perform on a given RBAC object. This function would be implementation-specific due to the lack of a standardized notion of user.

2.9 Discussion

The results of our analysis suggest that the CORBASec functionality—as defined through the data structures and interfaces in Version 1.8—is largely inadequate for implementing ANSI RBAC functions without resorting to vendor-specific extensions of the CORBAsec implementation. Even in the case of Core RBAC alone—the mandatory part of any compliant implementation of ANSI RBAC—there are three major causes of this inadequacy.

One is the lack of a standard mechanism for enumerating all objects that implement the DomainAccessPolicy interface in a CORBA deployment, which is necessary for enumerating all permissions granted to the corresponding user/principal/role. This limitation is due to the lack of support for enumerating all CORBA objects in a deployment. CORBA was originally positioned

as a generic middleware architecture scalable to Internet-wide deployments [Sie00]—where partial failures that are hard to detect and recover from are endemic—of potentially massive numbers of fine-grained objects. Such objects would range from intermittent (e.g., shopping cart for an online store customer) to long-lived and persistent (e.g., Parlay [3GP07]). Thus, its design intentionally avoids requirements for maintaining a view of the global state of a CORBA deployment—a prerequisite for a standardized capability to enumerate (and therefore register) all objects, or just all instances of DomainAccessPolicy. Although maintaining a view of the global state of an Internet-wide deployment for any application is clearly unfeasible, the reality is that CORBA deployments enjoy small-to-moderate scale [Hen06], are confined by enterprise boundaries—with the notable exception of Parlay—and commonly feature only course-grained [YD96], persistent objects. It seems reasonable to expect a standardized capability for enumerating all instances of the DomainAccessPolicy interface given that recent results demonstrate the feasibility of distributed lock [Bur06], table [CDG+06], and hash table [ZHS+04] data structures capable of holding tens of thousands of records and serving similar-sized populations of active clients. Such a capability might be featured, for example, only in enterprise-scale deployments of CORBA.

However, even with scalable data structures, *strict consistency* among multiple views of the system's global state is commonly believed to be essentially impossible [TS01], leaving only weaker consistency models to choose from. The semantics of these models, however, varies widely, from data-centric *linearizability* [HW90]—which requires a globally available clock—to client-centric *eventual consistency* [TS01]—which guarantees that all views eventually become consistent, but only if no updates take place for a long time. The choice of the acceptable consistency model(s) has to be explicit in ANSI RBAC in order for it to be applicable to those distributed systems where the protection state is distributed, as is the case with CORBA.

Another caveat is that other commercial-grade distributed technologies—e.g., COM+ [Obe00], EJB [DYK01], Grid [JBFT05], Web Services, and the HTTP-based Web—also lack a standardized capability for enumerating all resources (or just resources of a particular type) in a deployment. If most mainstream distributed technologies do not define this capability, is the reliance of ANSI RBAC on it realistic? Can the ANSI RBAC standard be revised to avoid the assumption that it is possible to enumerate all resources (and therefore permissions) in a system?

The second major limitation of CORBASec is its lack of the notion of user accounts and

support for their management (i.e., adding, deleting, (un)assigning to/from roles), as well as the lack of user representation. According to our analysis, which is summarized in Table 2.3, this limitation results in over one-half of Core RBAC functions being dependent on vendor-specific extensions. The architects of CORBASec intentionally left the notion of user and support for user management beyond the scope of the specification. The abstraction of PrincipalAuthenticator serves as an implementation-specific and technology-specific bridge between CORBASec run-time, which is concerned with principal credentials, and users, on behalf of which CORBA clients invoke operations on objects. PrincipalAuthenticator also performs user authentication and, if successful, activates roles at run-time. In order to provide standard support for administering and reviewing user accounts, their roles and their sessions, the corresponding administrative interfaces would need to be added to CORBASec. However, such a revision would be contrary to the emerging state of practice for application systems.

The notable trend in IT systems design is to re-allocate functionality for administering user accounts, and in some cases permissions, to single sign-on (SSO) [PM03] solutions for new applications [Got05] and to identity management (IDM) solutions for existing applications [BS03]. As a result, user accounts, and sometimes permissions, are administered across multiple application instances and types "outside" of the applications themselves. Therefore, an application system can only be successfully evaluated for compliance with ANSI RBAC when the application is considered together with the corresponding SSO or IDM solution. This condition makes evaluation of support for ANSI RBAC prohibitively expensive for systems designed to work in conjunction with multiple SSO or IDM solutions, as the evaluation would have to be performed for every combination of the system and the supporting SSO/IDM. Defining a separate ANSI RBAC profile for SSO/IDM solutions is a possible alternative to explore.

Even if CORBASec supported user accounts and their management, the inability to enumerate all CORBA principals related to a specific user (e.g., those with the same value of the auditId or accessId attributes) would still prevent CORBASec from providing complete support for such Core RBAC functions as *AddUser*, *CreateSession*, *DeleteSession*, *AddActiveRole*, *DropActiveRole*. All of their definitions use the helper function *user_sessions*, which can only be implemented if a CORBA deployment keeps track of all principals—CORBASec surrogates of sessions—for every user. However, principal tracking is prohibitively expensive for CORBA, and we believe for other

distributed systems, as well, due to the need for maintaining a view of the global state in the presence of partial failures. Thus, we echo the suggestion made by Li et al. [LBB06] to remove the notion of sessions from Core RBAC and introduce it in a separate, optional ANSI RBAC component.

Results of our analysis discussed in Sections 2.6.2 and 2.6.3 indicate that most functions for Hierarchical and Constrained RBAC options of the ANSI RBAC standard cannot be supported without extending a CORBASec implementation with additional operations. Even though there are at least three options for supporting role hierarchies, additional operations would have to be added to CORBASec in order to provide standard support for modification and review of the role hierarchy and for the functions *authorized_users* and *authorized_permissions*. Since support for Constrained RBAC is contingent on the support for user accounts, role hierarchies, and role activation, the standardization of support for these three is a prerequisite for standardization of Constrained RBAC in CORBASec.

In summary, while generic and versatile, the access control architecture of CORBASec does not define standard functionality for enumerating all DomainAccessPolicy objects in a deployment and all sessions for a given user. It also lacks the notion of user accounts and their run-time representation, as well as support for their management. These are three major roadblocks on the path of CORBASec conforming to ANSI RBAC. Our results are not conclusive, however, as to whether this mismatch between CORBASec and ANSI RBAC is exclusively due to the shortcomings of the former or also involves the failure of the latter to be sufficiently general.

2.10 Conclusion

Understanding middleware access control mechanisms is critical for protecting the resources of enterprise applications. In this chapter, we described in detail the architecture of access control mechanisms in CORBA Security and defined a configuration of the CORBA protection system in precise and unambiguous terms of set theory. Using the configuration definition, we suggested an algorithm that formally specifies the semantics of authorization decisions in CORBASec.

We analyzed CORBASec in relation to its support for ANSI RBAC components and discussed what functionality needs to be implemented, besides compliance with the CORBASec standard,

in order to support Core and Hierarchical RBAC. We suggested steps for translating an arbitrary ANSI RBAC policy into CORBA protection state. Finally, we analyzed CORBASec support for the functional specification of ANSI RBAC.

The results indicate that CORBASec falls short of supporting even functional Core RBAC due to (1) the lack of a standard mechanism for enumerating all DomainAccessPolicy objects in a CORBA deployment, (2) the lack of explicit user representation as well as the notion of user accounts and support for their management, and (3) the inability to enumerate all CORBA principals related to a specific user. Custom extensions are necessary in order for implementations compliant with CORBASec to support ANSI RBAC required or optional components. These results can be interpreted as either a demonstration of the inadequacy of CORBASec in supporting ANSI RBAC, or as an indication of ANSI RBAC not being sufficiently general. Examination of other representative systems on the subject of their support for ANSI RBAC may clarify this question.

The work presented in this chapter establishes a framework for assessing implementations of ANSI RBAC using CORBA Security. The results provide directions for CORBA Security developers implementing ANSI RBAC in their systems, and offer criteria to users for selecting such CORBA Security implementations that support required and optional components of ANSI RBAC.

Chapter 3

Analysis of ANSI RBAC Support in EJB

In this chapter we provide an overview of the access control architecture of EJB. We analyze access control mechanisms of EJB and define a configuration of the EJB protection system in a more precise and less ambiguous language than the corresponding EJB specification. Using this configuration, we suggest an algorithm that formally specifies the semantics of authorization decisions in EJB. We then analyze the level of support for the ANSI RBAC components and functional specification in EJB. We then conclude with a discussion of our results.

3.1 Overview of EJB Security

In this section we provide an overview of the EJB architecture, the main parts comprising an EJB system, as well as the declarative and runtime aspects of EJB systems. We also provide a description of the EJB security architecture.

3.1.1 EJB

This section provides a brief and informal overview of EJB. More information can be found in the corresponding EJB specification. Readers familiar with EJB are advised to proceed to Section 3.1.2.

The Enterprise Java Beans standard [DK06] defines an architecture for developing and deploying server-side components written in the Java programming language. The EJB architecture specifies the contracts that ensure the interoperability between various EJB components, clients, and deployment environments. These contracts ensure that an EJB product developed by one vendor is compatible with an EJB product provided by another vendor.

The EJB architecture, similar to other middleware technologies, allows application developers to implement their business logic without having to handle transactions, state management, multi-threading, connection pooling, and other deployment platform-dependent issues.

The EJB architecture consists of the following basic parts. These parts are also shown in Figure 3.1 for ProductBean, an example Enterprise Java Bean.

Enterprise Java Bean A server-side software component that is composed of one or more Java objects. The enterprise bean exposes certain interfaces that allow clients to communicate with the bean in compliance with the EJB specification. This is shown in Figure 3.1 as ProductBean. The EJB specification [DK06] defines three main types of enterprise beans: *entity*, *session* (which include stateful and stateless session beans), and *message-driven* beans. Depending on the type of the enterprise bean, its functionality ranges from a mere object-oriented abstraction of an entity that exists in persistent storage (such as a record in a database), to a web service implementing certain business logic.

EJB Container Provides services—such as persistence, concurrency, bean lifecycle, resource pooling, and security—to the enterprise beans it hosts. Multiple enterprise beans typically exist inside a single container. The container vendor provides necessary tools, which are specific to their container, to help in the deployment of enterprise beans, as well as runtime support for the deployed bean instances.

EJB Server Provides runtime environment to one or more containers. Since EJB specification does not explicitly define the separation of roles between containers and servers, they usually come inseparable as one system.

EJB Client A software component that invokes methods on the Enterprise Java Bean. The EJB architecture allows a variety of client applications to utilize the business logic that the beans provide. Servlets or Java Server Pages (JSP), Java stand-alone applications or applets are common types of EJB clients. EJBs can also be clients of other EJBs. CORBA-based applications, which are not necessarily developed in Java, may also be clients of EJBs. All EJB clients access enterprise beans logic through predefined protocols and software interfaces. These interfaces define the methods that can be invoked on the bean.

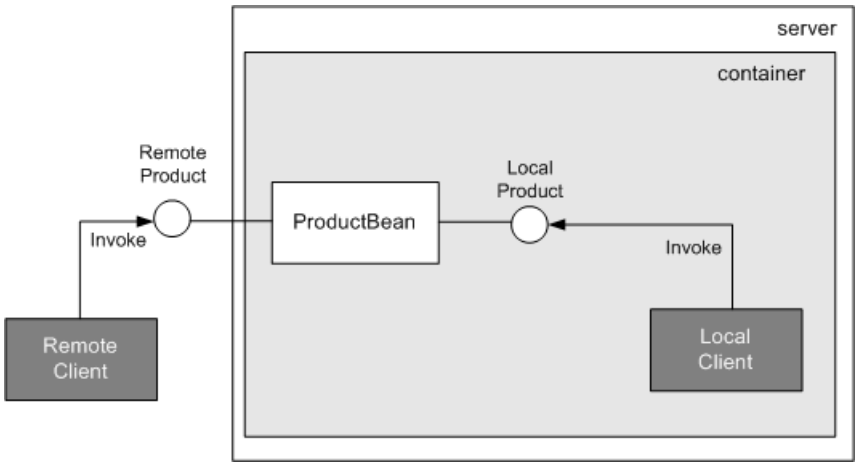


Figure 3.1: Basic parts of EJB architecture for an example Enterprise Java Bean Product

Remote Business Interface Java interfaces that are provided by the Enterprise Java Bean and marked with the @Remote Java language metadata annotation. [DK06] The EJB container tools handle the generation of required logic in order to support remote access to methods defined by this interface.

Local Business Interface A Java interface that is provided by the Enterprise Java Bean and that supports local access. Clients that utilize this type of interface have to be collocated in the same Java Virtual Machine (JVM) as the Enterprise Java Bean.

Although Enterprise Java Beans are written in the Java programming language, fully compliant EJB deployment environments support the Internet Inter-ORB Protocol (IIOP) [OMG04], leveraging IIOP and the Common Secure Interoperability Protocol Version 2 (CSIv2) [OMG04] capabilities which allow CORBA clients (which can be written in languages other than Java) to access enterprise bean objects.

Declarative Part Defining remote, and local interfaces as well as implementing the business logic in EJB is as easy as in standard Java. Figure 3.2 shows an example of an enterprise bean remote interface definition, and Figure 3.3 illustrates an example of the corresponding implementation for that interface.

In EJB 3.0, the metadata annotations defined in Java Development Kit (JDK) 5.0 and later are used to create annotated Enterprise Java Beans. The tools provided by the EJB Container

```
import javax.ejb.Remote;

@Remote public interface Product {
    public float getPrice();
    public void setPrice( float newPrice )
        throws InvalidPriceException;
};
```

Figure 3.2: Defining a remote interface for the Product enterprise bean (Product.java)

```
import javax.ejb.Stateful;

@Stateful public class ProductBean implements Product {
    private float price = 0;

    public float getPrice() {
        return price;
    }

    public void setPrice( float newPrice ) {
        if ( price < 0 ) {
            throw new InvalidPriceException();
        }

        price = newPrice;
    }
}
```

Figure 3.3: Implementing the remote interface for the Product enterprise bean (ProductBean.java)

vendors utilize these annotations to automatically generate proper Java classes as well as other required EJB interfaces.

As an alternative to metadata annotations, a bean developer can also specify transactional, security, and other requirements for the application using the *deployment descriptor*—an XML file with predefined syntax that holds all the explicit metadata for the assembly. The descriptor can be later augmented and altered by an application *assembler* and *deployer*, who play specific roles in the life cycle of enterprise beans predefined by the EJB specification.

Runtime Part While the remote object model for EJB components is based on the Remote Method Invocation (RMI) API [Sun07], all invocations between J2EE components are performed using IIOP. The use of the RMI remote invocation model over the IIOP protocol is usually referred to as RMI-IIOP. When EJB components use the RMI-IIOP (mandatory for EJB 2.0 and higher),

the standard mapping of the EJB architecture to CORBA enables interoperability with multi-vendor ORBs, other EJB servers, and CORBA clients written in languages other than Java.

Because of the IIOP, the same object reference used for CORBA is used in the EJB. The similarities between CORBA and EJB lie in their use of a secure channel, as well as their client and server security layer architectures. For more detailed explanation of EJB technology please refer to [RSB05].

3.1.2 EJB Security Subsystem

The EJB protection architecture is conceptually simple: When the client program invokes a method on a target EJB object, the identity of the subject associated with the calling client is transmitted to the EJB object's container. The container checks whether the calling subject has a right to invoke the requested method. If so, the container permits the invocation on the method.

Client Security Service

Because of the use of IIOP and CSIv2, the responsibilities of an EJB client security service (CSS) are similar to those of a CORBA CSS:

1. creating a secure channel with the target security service (TSS), and
2. obtaining the user's authenticated credentials or passing username and password over the CSIv2 context to TSS, as well as
3. protecting request messages and verifying response messages.

Treated by the EJB specification as an integral part of the server container, a TSS establishes and maintains a secure channel with the clients, verifies authenticated credentials or performs client authentication itself, implements message protection policies, and performs access checks before an invocation is dispatched to an enterprise bean. Depending on the application configuration, which is done through the deployment descriptor, the container associates the runtime security context of the dispatched method either with the identity of the calling client or with some other subject. Other security-related responsibilities of a container include the following:

- Isolating the enterprise bean instances from each other and from other application components running on the server,
- Preventing enterprise bean instances from gaining unauthorized access to the system information of the server and its resources,
- Ensuring the security of the persistent state of the enterprise beans,
- Managing the mapping of principals on calls to other enterprise beans, or on access to resource managers, according to the defined security policy,
- Allowing the same enterprise bean to be deployed independently multiple times, each time with a different security policy.

Implementation of Security Functions

The security parts of EJB specification focus largely on authentication and access control. The specification relies on CSIv2 level 0 for message protection, and it leaves support for security auditing to the discretion of container vendors. We describe EJB access control architecture in Section 3.2.1.

Authentication User authentication is either performed by the client's infrastructure (such as Kerberos), or by the EJB server itself. In the latter case, the EJB server receives user authentication data (only username and password for CSIv2 level 0) or credentials from a client and authenticates the client using local authentication service, which is not predefined by the specification. Once the container authenticated the client (or verified its credentials), it enforces access control policies. The notion of a *principal* is used in the EJB specification to refer to authenticated clients.

Administration Some security administration tasks of EJB servers are performed through changes in deployment descriptors. This includes definition of security roles, method permissions, and specification of security identity, either delegated or predetermined, for dispatching calls to bean methods. Other tasks, such as mapping users to roles, specifying message protection, admin-

istering an audit, and authentication mechanisms, are beyond the scope of the EJB specification and are therefore left up to the vendors of container products and deployment tools.

3.2 EJB Protection State

In this section, we first introduce the EJB access control architecture. Then, we formally define a configuration of the EJB protection state.

3.2.1 EJB Access Controls

An EJB container controls access to its beans at the level of an individual method on the bean class, not the bean object instances. That is, if different instances of the same bean have different access control requirements, they should be placed in different application assemblies, which are defined by JAR files. This means that the scope of the EJB's policy domain is the application assembly.

The EJB access control architecture provides two ways for enforcing access control decisions. One approach, known in EJB terminology as *declarative security*, is to configure the container to enforce authorization policy. The other is achieved by coding authorization decision and enforcement logic into the bean methods. In the former case, access permissions of principals are defined either using deployment descriptors, or through code annotations. The declarative approach decouples business logic from security logic. In the latter approach, also known as *programmatic security*, the application developers utilize methods such as IsCallerInRole and getCallerPrincipal to obtain necessary information about the caller, which can be used to enforce access control policies that cannot be expressed using the declarative approach.

Authorization to invoke enterprise bean's methods is enforced by the container. It grants or denies clients' requests to execute the methods in conformance with access control policies described in the *deployment descriptor* and/or through bean's metadata annotations. Since bean's metadata annotations are equivalent in the expressiveness to the policies supported by the deployment descriptor, we use only the latter in the rest of the chapter.

Access control decisions are based on the *security roles* (or just "roles" for short) of the principal that represents the calling client. The security role is defined in the EJB specification as "a

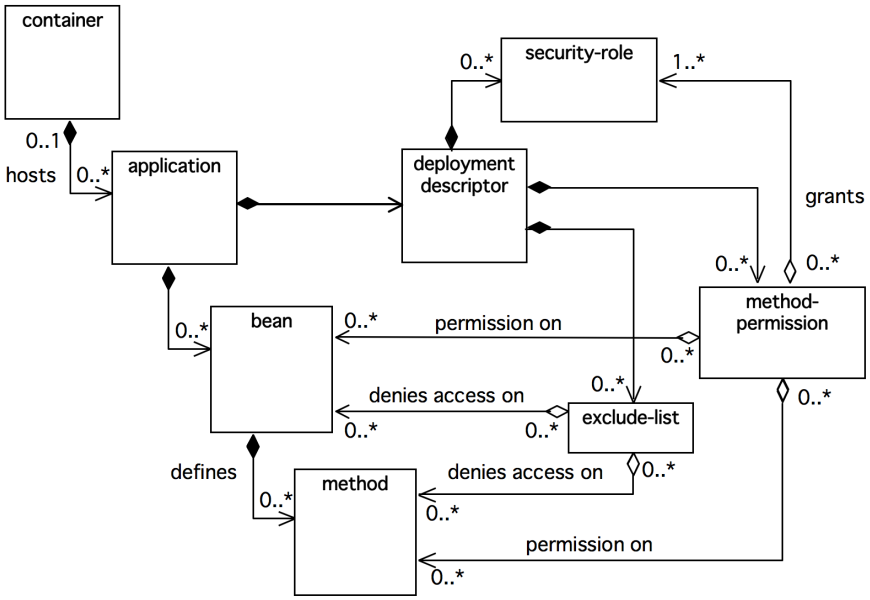


Figure 3.4: Relationships among the sections of deployment descriptor used for expressing access control policy and the elements of an EJB application

semantic grouping of permissions that a given type of users of the application must have in order to successfully use the application" [DK06, page 456]. As defined by the specification, there are three types of deployment descriptor sections relevant to the declarative access control: `security-role`, `method-permission`, and `exclude-list`. The `exclude-list` section specifies the methods that cannot be called by any principal, no matter which roles are assigned to the principal. Figure 3.4 uses Unified Modeling Language (UML) [OMG07a, OMG07b] notation to summarize the relationships among authorization-related sections of the deployment descriptor and the elements of an EJB application. In the rest of this section, we describe syntax and semantics of the two other sections.

Each `security-role` section lists a role with optional human-readable unstructured description of the role. This role can be referenced in other sections of the deployment descriptor. In essence, these sections define a set of roles for an EJB application.

Assignment of permissions to roles is done in `method-permission` sections. Each such section lists roles permitted to invoke one or more methods. Special role name "unchecked" can be used to

indicate that all roles are permitted to invoke the listed method(s). Each method is defined by the name of the bean class, method name, and, optionally, the formal parameter types to distinguish methods with overloaded names. Special method name "*" refers to all methods on a given bean.

An example of an assignment done through `method-permission` sections is shown in Table 3.1. First row illustrates an assignment of a permission to invoke method m_1 on bean b_1 ($b_1 \cdot m_1$) to role r_1. The second row shows how several roles (r_1 and r_2) can be granted permissions to invoke any of the listed methods ($b_1 \cdot m_2$ and $b_1 \cdot m_4$). This means that any principal that has any of theses two roles can invoke any of these two methods. Last row provides an example of using "unchecked" and "*" keywords. It states that any principal can invoke method $b_2 m_1$ as well as any method on bean b_3. The overall set of methods a principal can invoke on a given EJB application is the union of all the methods the principal's roles are permitted to invoke. For example, if a deployment descriptor contains only the three method-permission sections listed in Table 3.1, then a principal with role r_2 is granted permission to invoke methods $b_1 \cdot m_2$, $b_1 \cdot m_4$, $b_2 \cdot m_1$, and any method on bean b_3.

If a method (1) is not listed in any of the `method-permission` and `exclude-list` sections of a deployment descriptor, and (2) has no @DenyAll annotation in the code, then it is accessible by any principal.[5] For instance, if $b_1 \cdot m_3$ is such a method then any principal would be able to invoke it.

Even though the syntax of `method-permission` section allows listing more than one role and method, we will assume without the loss of generality that each section contains only one role and one method, as shown in the first row of Table 3.1. It is easy to define an algorithm for converting any number of `method-permission` sections in a deployment descriptor to this form. This assumption will simplify the definition of the protection state and the algorithm for making access control decisions in Section 3.2.2.

In addition to the above deployment descriptor sections, EJB server vendors (or container providers) define container-specific sections of deployment descriptors that map users and/or groups to roles. Table 3.2 shows additional deployment descriptor sections for major commercial EJB servers. Since the notions of users, groups, and the mapping from them to roles are

[5] According to Section 17.3.2.3 of the EJB specification [DK06], methods with unspecified permissions must be treated by the container as "unchecked."

Roles	Methods
r_1	$b_1 \cdot m_1$
r_1, r_2	$b_1 \cdot m_2$, $b_1 \cdot m_4$
"unchecked"	$b_2 \cdot m_1$, $b_3 \cdot *$

Table 3.1: Examples of method-permission sections of EJB deployment descriptor. For clarity sake, the data representation is converted from XML notation to human-understandable form, with each row corresponding to an individual section.

App. Server	Section(s)	Comments
Oracle	users, groups	A *security-role-mapping* XML tag maps logical roles defined in the application deployment descriptor to entities defined in the *users* and *groups* sections
Sun ONE	principal-name, group-name	A *security-role-mapping* tag defines mapping between *principal-name*s and roles, and/or between *group-name*s and roles
BEA WebLogic	principal-name	A *security-role-assignment* XML tag declares mapping between *principal-name*s and roles
IBM WebSphere	users, groups	Tools establish user-group memberships and mapping between groups and roles

Table 3.2: Additional authorization-related sections used in deployment descriptors of commercial EJB servers

lacking from the EJB v3.0 specification, these vendor-specific additions to the EJB system will not be used for defining the EJB protection state.

3.2.2 Formalization of the Protection State

In this section, we formalize the semantics of the EJB access control architecture.

Definition 5 [EJB Protection State] *A configuration of an EJB system protection state is a tuple* (R, B, OPS, M, MP, X) *interpreted as follows:*

- R *is a set of roles defined in the* `assembly-descriptor` *part of of the deployment descriptor provided with the EJB application. These roles are defined using the* `security-role` *tags. This set also includes special role "unchecked".*

- *B is a set of enterprise beans listed in the* `enterprise-beans` *section of the deployment descriptor.*

- *OPS is a set of methods defined by the enterprise beans of the application. Members of this set are denoted as m_i. The set also includes special method "*" for any bean defined by the application and signifying any method on that bean; for example, $OPS = \{m_1, m_2, ...\} \bigcup \{*\}$.*

- *$M \subseteq B \times OPS$ is the set of available uniquely identifiable methods. Members of this set are denoted $b_i \cdot m_j$.*

- *$MP \subseteq R \times M$ is a many-to-many permission assignment of EJB application roles to invoke methods, as specified in* `method-permission` *sections of the application's deployment descriptor.*

- *$X \subseteq M$ is a subset of methods—defined by* `exclude-list` *sections of the deployment descriptor—invocation of which is denied to any role.*

Note that the implementations of EJB containers and servers commonly have extensions to the deployment descriptors, which enable defining sets of users and groups, as well as assigning them to roles. Such vendor-specific extensions result in additional elements of the protection state. However, all elements defined in Definition 5 are present in any EJB implementation compliant with the specification. When analyzing in Section 3.3 EJB support for RBAC, we will identify additional elements of EJB protection state that are necessary for the support.

Given the protection state of an EJB application, Algorithm 8 defines the outcome of an access control decision. First, a check is performed on the membership of the requested method in the list of blocked methods. If the method is found in the list, then the access is denied. If not, then the method permissions are checked for every role of the principal and the special role "unchecked." If no appropriate element is in MP, then access is denied.

3.3 Analysis of Support for ANSI RBAC

As described in Section 1.2, the ANSI RBAC Reference Model defines four major components. In order for a system to conform to ANSI RBAC, Core RBAC must be implemented at a minimum.

```
Authorize(p : 2^R, b_i · m_j : M) → {allow, deny}
if b_i · m_j ∈ X then
  return deny
end if
for all r ∈ p ∪ {"unchecked"} do
  if (r, b_i · m_j) ∈ MP ∨ (r, b_i*) ∈ MP then
    return allow
  end if
end for
return deny
```

Algorithm 8: Authorization decision in EJB. Decide authorization for principal $p \equiv \{r_1, r_2, \ldots r_n\}$ invoking method m_j on bean b_i, where $r_1, r_2, \ldots r_n \in R$, and $b_i m_j \in M$

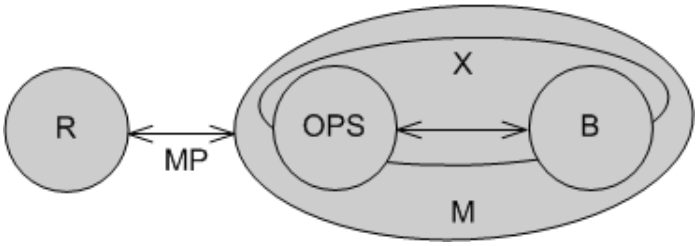
ANSI compliant RBAC system can also implement Hierarchical RBAC, which defines hierarchies of roles in addition to everything Core RBAC has. The other two optional components of the standard, Static Separation of Duty (SSD) and Dynamic Separation of Duty (DDS), define relations among roles with respect to user assignments as well as role activation in user sessions.

In Section 3.3.1, we first examine the extent to which an EJB protection state—as formalized in Definition 5—can support each of the four ANSI RBAC model components. In Section 3.3.3 we provide an example that illustrates the abilities of an EJB system to support ANSI RBAC. In Section 3.3.4 we then analyze the degree to which the structures defined in EJB specification support the functional specification of ANSI RBAC. In Section 4.4 we discuss results of our analysis.

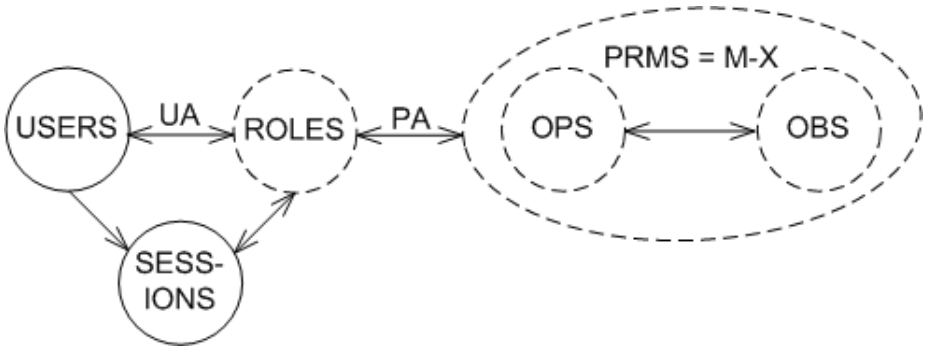
3.3.1 Reference Model

Core RBAC

Various Core RBAC data elements are mapped readily into EJB using the sets defined in Section 3.2. For example, the $ROLES$ set in RBAC maps directly to R, which defines the EJB security roles; RBAC objects (OBJ) are equivalent to EJB beans (B); RBAC operations (OPS) are represented by EJB OPS. The representation of other relations defined in Core RBAC is outside the scope of the EJB standard, as we will discuss later in this section. To aid with the understanding of the correspondence between elements in the RBAC model and EJB, we present in Figure 4.3 the sets and relations of EJB (with light grey background) and RBAC (with white background). Shapes with white background and dashed lines show mapped RBAC sets. The reader is encouraged to compare it to the diagram in Figure 1.1. We first define Core RBAC in



(a) EJB protection state sets



(b) Mapped and unmapped RBAC sets

Figure 3.5: EJB (with light grey background) and RBAC (with white backround) sets and relations.

the language of the EJB protection system more formally as follows:

Definition 6 [Core RBAC in EJB] *Core RBAC in the language of EJB is defined by the EJB system protection state outlined in Definition 5, as well as the following additional elements:*

- $USERS$ *is the set of users, where members of this set are defined in the operational environment of the EJB system.*
- $ROLES = R$*, is the set of roles as defined in Definition 5.*
- $OBS = B$ *is a set of enterprise beans.*
- $UA = USERS \times ROLES$*, is a many-to-many assignment of users to roles.*
- $assigned\_users(r : ROLES) = \{u \in USERS | (u, r) \in UA\}$*, is a function that returns the set of users in* $USERS$ *that are assigned to the given role* r*.*
- $PRMS \subseteq M - X$*, is a set of permissions to invoke EJB methods provided that these methods do not exist in the exclusion set* X*. The existence of* $b_i \cdot m_j$*, or* $b_i \cdot *$ *in* $PRMS$ *provides permission to invoke a specific method* m_j*, or all methods on bean* b_i*, respectively.*

- $PA \subseteq PRMS \times ROLES$, *a many-to-many assignment of permissions to roles.*
- $assigned\_permissions(r : ROLES) = \{p \in PRMS | (p, r) \in PA\}$, *is a function that returns the set of permissions in* $PRMS$ *that are assigned to the given role* r.
- $Op(p : PERMS) \rightarrow \{op \in OPS\}$, *a function that returns a set of operations that are associated with the given permission* p.
- $Ob(p : PERMS) \rightarrow \{ob \in OBS\}$, *a function that returns a set of objects that are associated with the given permission* p.
- $SESSIONS$ *is a set of sessions for a specific application. Members of this set are mappings between authenticated users and their activated roles for a specific EJB application.*
- $session\_users(s : SESSIONS) \rightarrow USERS$, *the mapping of session* s *onto the corresponding user.*
- $session\_roles(s : SESSIONS) \rightarrow 2^{ROLES}$, *the mapping of session* s *onto a set of roles. Formally:* $session\_roles(s_i) \subseteq \{r \in ROLES | (session\_users(s_i), r) \in UA\}$.
- $avail\_session\_perms(s : SESSIONS) \rightarrow 2^{PRMS}$, *the permissions available to a user in a session* $= \bigcup_{r \in session\_roles(s)} assigned\_permissions(r)$.

In order to support Core RBAC in EJB systems, Definition 6 identifies additional elements to those identified in Definition 5. These additional elements are related to users and sessions. In the rest of this section we discuss how elements of Definition 6 are or can be supported in an EJB system.

Although the EJB standard [DK06] does not mandate how users must be supported in an EJB system, various implementations of EJB servers and containers implement extensions to deployment descriptors, as described to in Section 3.2.2. These extensions provide support for adding users to the system, as well as mapping those users to roles. The $USERS$ set in Definition 6 abstracts this support; however, this support is implementation-dependent. By the same token, support for UA and $assigned\_users$ is also implementation-dependent.

The $SESSIONS$ set is another element of Definition 6. In relation to support for users, the EJB standard does not specify a mapping of authenticated users to roles, or more precisely,

role activation. Hence, EJB server's support for sessions is outside the scope of the EJB standard and is implementation-dependent. Similarly, in order to fully support Core RBAC, EJB implementations' support for session-related functions such as *session_users*, *session_roles*, and *avail_session_perms* are outside the scope of the EJB standard.

On the other hand, the sets $ROLES$, OPS, and OBS; the relations $PRMS$ and PA; and the functions *Op* and *Ob* are all supported by the EJB standard as these can be readily obtained from the deployment-descriptor.

To summarize, about half the elements of ANSI Core RBAC can be provided by any implementation compliant with the EJB standard; however, support for $USERS$, UA, *assigned_users*, $SESSIONS$, *session_users*, *session_roles*, and *avail_session_perms*, which relate to users and sessions, if provided, can only be implementation-dependent.

Hierarchical RBAC

The Hierarchical RBAC component specifies two types of role hierarchies: *general* and *limited*. Both types are formally defined using elements of Core RBAC. In addition to role hierarchies, Hierarchical RBAC defines two functions: *authorized_users* and *authorized_permissions*, as described in Section 3.3.1. Although the EJB standard does not provide direct support for Hierarchical RBAC, an EJB implementation can still emulate both types of role hierarchies. The rest of this section discusses ways of emulating Hierarchical RBAC in EJB.

EJB server administrative tools can be modified in order to support role hierarchy. First, the administrative tools must maintain hierarchy relationships between roles in a repository. Second, the tools must ensure that when method permissions are granted to a certain role in a deployment descriptor, those method permissions are also appropriately and consistently granted to all junior roles. Finally, the administrative tools must also keep track of whether a permission has been directly assigned to a role or the role inherited this permission through a role hierarchy. No special run-time support for role hierarchies would then be needed. This approach is similar to the ones used in [AS01] and [SA98] in order to support role hierarchy in various operating systems.

An alternative is an approach in which inherited permissions are determined at run-time. This approach would require the EJB server—or more specifically the Target Security Service (TSS) described in Section 3.1.2—to examine the role hierarchy repository during run-time. A

certain role is then granted permission to invoke a specific method not only based on direct permission-to-role assignment, but also based on permissions granted to a junior role. In addition to a repository that maintains role hierarchy relationship, a run-time computation of inherited permissions would be required. A similar approach is adopted in [FBK99] for Common Gateway Interface (CGI) based Web applications, and in [Giu99] for Java Authentication and Authorization Service (JAAS) [Sun01] based access control.

With either of the above approaches, support for this role hierarchy—and the *authorized_users* and *authorized_permissions* functions required for Hierarchical RBAC—is implementation-dependent and is not specified by the EJB standard.

Constrained RBAC

The Constrained RBAC component introduces separation of duty relations to the RBAC reference model. As with Hierarchical RBAC, these relations are defined in terms of Core RBAC constructs. In essence, SSD constrains user-to-role assignment (UA set and *assigned_users* function) and the role hierarchy (RH set and *authorized_users* function). DSD, on the other hand, constrains the role activation ($SESSIONS$ set and *session_roles* function). Since user accounts, role hierarchies, and role activation are beyond the scope of EJB, the Constrained RBAC component, if supported, would have to be implementation-dependent.

3.3.2 Translating RBAC Policies to EJB

In Definition 5 and Definition 6 we presented a protection state for EJB systems, and how Core RBAC can be modeled in the language of the EJB protection state, respectively. In this section, we present an algorithm that translates an arbitrary RBAC policy into an EJB protection state.

Algorithm 9 formalizes the translation from an RBAC policy to the EJB protection state defined in Definition 5. For clarity, we identify the RBAC sets in the algorithm with an RBAC subscript.

3.3.3 Example

In this section we present an example that illustrates the abilities of an EJB system to support ANSI RBAC. As discussed in Section 3.3.1, the EJB standard does not provide direct support

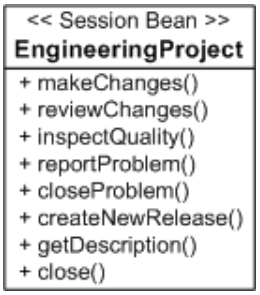
1: {Initialize EJB sets and relations.}
2: $R \leftarrow ROLES_{RBAC}$
3: $B \leftarrow OBS_{RBAC}$
4: $OPS \leftarrow OPS_{RBAC}$
5: $M \leftarrow \emptyset$
6: $MP \leftarrow \emptyset$
7: $X \leftarrow \emptyset$
8: **for all** $p \in PRMS_{RBAC}$ **do**
9: **for all** $(opr, obj) \in p$ **do**
10: $M \leftarrow M \cup \{(obj \dot{o} pr)\}$
11: **end for**
12: **end for**
13: **for all** $pa \in PA_{RBAC}$ **do**
14: **for all** $((opr, obj), r) \in pa$ **do**
15: $MP \leftarrow MP \cup \{(r, (obj \dot{o} pr))\}$
16: **end for**
17: **end for**

Algorithm 9: Operational definition of translating from an ANSI RBAC system state to the one of EJB.

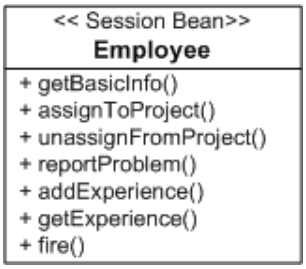
for role hierarchy; however, emulation of such support is possible as discussed earlier, and is straightforward. Hence, role hierarchy is not illustrated in this example.

The example in this section consists of a simple system that maintains employee and engineering project records in an engineering company. The system allows different users to perform various operations on the project and employee records, based on the users' roles in the company. The system handles the manipulation of various records through enterprise beans of two types: EngineeringProject and Employee. These enterprise beans are depicted in Figure 3.6. The figure shows the methods that can be invoked on the beans. The system also defines seven different user roles. Based on these roles and according to the policies listed in Figure 3.7, users are allowed to invoke various methods on a specific EJB. These roles are defined as follows:

- *Employee* represents a company employee.
- *Engineering Department* represents an employee of the engineering department.
- *Engineer* performs various engineering tasks in the company.
- *Product Engineer* is responsible for managing a product line.
- *Quality Engineer* is a quality assurance engineer.



(a) EngineeringProject bean



(b) Employee bean

Figure 3.6: Example session beans

Roles	**Methods**													
	EngineeringProject Bean								*Employee Bean*					
	makeChanges()	reviewChanges()	inspectQuality()	reportProblem()	closeProblem()	createNewRelease()	getDescription()	close()	getBasicInfo()	assignToProject()	unassignFromProject()	addExperience()	getExperience()	fire()
Employee									√				√	
Engineering Department				√			√		√				√	
Engineer	√	√							√				√	
Product Engineer						√			√				√	
Quality Engineer			√						√				√	
Project Lead					√				√				√	
Director								√	√	√	√	√	√	√

Table 3.3: Permission-to-role assignment for the example

- *Project Lead* overseas and leads the development of a project.
- *Director* is an engineering department director.

The access control policy that defines what actions each role is allowed to perform are summarized in Table 3.3, where a check mark ("√") denotes a granted permission for a specific EJB role to execute the corresponding enterprise bean method. Table 3.4 shows an example of system users, and their group memberships. Tables in Figure 3.8 show examples of user-to-role and group-to-role assignments. The following is a formalization of this example system's protection state as in Definition 5.

- R = {Employee, Engineering Department, Engineer, Product Engineer,

1. Anyone in the organization can look up an employee's basic information, such as their name, department, phone number, and office location.
2. Everyone in the engineering department can get a description of and report problems regarding any project and look up experience of any employee.
3. Engineers, assigned to projects, can make changes and review changes related to their projects.
4. Quality engineers, in addition to being granted engineers' rights, can inspect the quality of projects they are assigned to.
5. Product engineers, in addition to possessing engineers' rights, can create new releases.
6. The project lead, in addition to possessing the rights granted to product and quality engineers, can also close problems.
7. The director, in addition to being granted the rights of project leads, can manage employees (assign them to projects, un-assign them from projects, look up experience, add new records to their experience, and fire them) and close projects.

Figure 3.7: Authorization policy for the example EJB system describing what actions are allowed. All other actions are denied.

User	Group
Alice	accounting
Bob	hardware
Carol	software
Dave	software
Eve	software
Fred	management

Table 3.4: Example users, groups, and group memberships

User	Role
Alice	Employee
Bob	Engineer
Carol	Quality Engineer
Dave	Product Engineer
Eve	Project Lead
Fred	Director

(a) User-to-role assignment

Group	Role
hardware	Engineering Department
software	Engineering Department

(b) Group-to-role mapping

Figure 3.8: Example EJB system role mappings

Quality Engineer, Project Lead, Director}

- B = { EngineeringProject, Employee }
- OPS = { makeChanges, reviewChanges, inspectQuality, reportProblem, closeProblem, createNewRelease, getDescription, close, getBasicInfo, assignToProject, unassignFromProject, addExperience, getExperience, fire}
- M = {EngineeringProject.makeChanges, EngineeringProject.reviewChanges, EngineeringProject.inspectQuality, EngineeringProject.reportProblem, EngineeringProject.closeProblem, EngineeringProject.createNewRelease, EngineeringProject.getDescription, EngineeringProject.close Employee.getBasicInfo, Employee.assignToProject, Employee.unassignFromProject, Employee.addExperience, Employee.getExperience, Employee.fire }
- MP = { (Employee, Employee.getBasicInfo), (Employee, Employee.getExperience), (Engineering Department, EngineeringProject.reportProblem), (Engineering Department, EngineeringProject.getDescription), (Engineering Department, Employee.getBasicInfo), (Engineering Department, Employee.getExperience), (Engineer, EngineeringProject.makeChanges), (Engineer, EngineeringProject.reviewChanges), (Engineer, Employee.getBasicInfo), (Engineer, Employee.getExperience), (Product Engineer, EngineeringProject.createNewRelease), (Engineering Department, Employee.getBasicInfo), (Engineering Department, Employee.getExperience), (Quality Engineer, EngineeringProject.inspectQuality), (Quality Engineer, Employee.getBasicInfo), (Quality Engineer, Employee.getExperience),

(Project Lead, EngineeringProject.closeProblem),

(Project Lead, Employee.getBasicInfo),

(Project Lead, Employee.getExperience),

(Director, EngineeringProject.close),

(Director, Employee.getBasicInfo),

(Director, Employee.assignToProject),

(Director, Employee.unassignFromProject),

(Director, Employee.addExperience),

(Director, Employee.getExperience),

(Director, Employee.fire) }

- $X = \Phi$

The R and Bsets contain the roles and beans defined in the system. OPS defines all operations available to various roles. These methods are further qualified by the M set, where each method is qualified with the name of the bean for which it is defined. The MP set represents Table 3.3, and as described in Section 3.2.2, MP is a many-to-many permission assignment of EJB application roles to invoke defined methods. These permissions are listed in the `method-permission` sections of the application's deployment descriptor. This example does not require any methods to be in the `exclude-list` sections of the deployment descriptor for the application;hence, set X is empty.

We use the above formalization of the example system's protection state in order to support the ANSI Core RBAC reference model. Considering Definition 6, the content of $ROLES$, OPS, and OBS is straightforward. The rest of the sets are defined as follows.

- $USERS$ = {Alice, Bob, Carol, Dave, Eve, Fred, accounting, hardware, software, management}

- UA = { (Alice, Employee), (Bob, Engineer), (Carol, Quality Engineer), (Dave, Product Engineer), (Eve, Project Lead), (Fred, Director), (hardware, Engineering Department), (software, Engineering Department), (Bob, Engineering Department), (Carol, Engineering Department), (Dave, Engineering Department), (Eve, Engineering Department) }

- $PRMS = M$

- $PA = MP$

The EJB 3.0 standard does not specify how EJB roles should be mapped to the user groups and accounts that exist in the bean's operational environment. This makes the $USERS$ and UA sets dependent solely on the EJB container's operational environment, and the way users are managed there. For example, the UA set contains assignments that exist only due to user-group memberships. In this example, Carol is assigned to the *Engineering Department* role through her software group membership.

3.3.4 Functional Specification

This section reports on the results of our analysis of the support that the EJB standard [DK06] can provide for ANSI RBAC system and administrative functional specifications. For the purpose of this analysis, we examined every function specified in Section 6 of [ANS04] on the subject of its support by an EJB container conforming to the EJB standard.

Results of our examination suggest that the software interfaces that the EJB standard mandates are insufficient for implementing most of ANSI RBAC functions as is. Furthermore, the XML data structures defined in the EJB deployment descriptor, are incapable of fully supporting an ANSI RBAC compliant system. These data structures can provide support for implementing a limited number of Core RBAC functions. Other system and administrative Core RBAC functions, as well as all additional functions for Hierarchical and Constrained RBAC, cannot be supported without extending an EJB system implementation beyond what the EJB standard defines.

The following is an examination of various Core RBAC functions and their level of support in the EJB standard.

AddUser, DeleteUser operations allow users to be added to the $USERS$ set and to be removed from it. In an EJB environment, these are realized in a implementation-dependent manner. For example, the IBM WebSphere Application Server [SCH+04] allows EJB application deployers to use various user registries to maintain the $USERS$ set. WebSphere can be configured to use the local operating system user accounts, an LDAP [WHK97] server, or a custom user registry.

AddRole, DeleteRole add roles to and delete roles from the RBAC system. EJB data structures provide direct support for implementing these functions. They can be implemented by adding or removing a role definition using the `security-role` tags in the `assembly-descriptor`section of the deployment descriptor file.

AssignUser, DeassignUser allow assignment relationships to be established between roles and users. Similar to the *AddUser* and *DeleteUser*, these operations need to be implemented in an implementation-dependent manner.

GrantPermission, RevokePermission allow invocation permissions to be granted to or revoked for a certain role. These operations can be implemented by adding or removing the corresponding `method-permission` section of the deployment descriptor.

CreateSession, DeleteSession, AddActiveRole, DropActiveRole allow for the creation and deletion of sessions, as well as activation of user roles. In an EJB environment, these operations are likely to be implemented in a proprietary manner and would differ from one EJB application server to another.

CheckAccess make an access control decision. The Authorize method in Algorithm 8 can be used to implement CheckAccess.

AssignedUsers, AssignedRoles return users assigned to a given role, and roles assigned to a given user, respectively. Since these functions are not supported in EJB 3.0, they need to be provided by the EJB application server.

Advanced Review Functions for Core RBAC

RolePermissions returns the permissions granted to a given role. This function can be implemented by examining the `method-permission` sections, where method permissions are granted to roles.

UserPermissions returns permissions assigned to users. Given the permissions assigned to roles (using the *RolePermissions* function), and knowing the roles the user is assigned to (using *AssignedUsers*), the implementation of this function is straightforward.

SessionRoles, SessionPermissions return the roles and permissions associated with a specific user session. These can be provided by the EJB application server assuming that the server implementation alreadysupport the notion of sessions.

RoleOperationsOnObject, UserOperationsOnObject return a set of operations that can be invoked on an object given a certain role or a certain user, respectively. The operations that a certain role is permitted to invoke can be obtained directly from the `method-permission` sections of the deployment descriptor. The operations that a user is permitted to invoke, on the other hand, can be obtained given the implementation of the *RoleOperationsOnObject* as well as the *AssignedRoles* functions.

Table 3.5 provides a summary of the above results. The table classifies support for ANSI Core RBAC functions in two main categories. The first category contains functions that are supported directly by EJB data structures, whereas the second category identifies the supplemental components that must be implemented in an EJB system—outside the scope of the EJB specifications—in order to support the specified ANSI Core RBAC functions. These components are identified as related to user management, session and role activation. The user management related components are required to handle the addition/deletion of users from the system, as well as user-to-role assignments. On the other hand, the session and role activation related components are required to handle the management of user sessions and activation of permissions.

3.4 Discussion

The results of our analysis suggest that the EJB functionality—as defined through the data structures and interfaces—falls short of fully supporting ANSI RBAC without resorting to vendor-specific extensions. Even in the case of Core RBAC alone—the mandatory part of any compliant implementation of ANSI RBAC—there are two major causes of this inadequacy.

The two major limitations of EJB are its lack of the notion of user accounts and support for their management (i.e., adding, deleting, (un)assigning to/from roles), as well as the lack of support for user sessions and role activation. According to our analysis, which is summarized in Table 3.5, this limitation results in two thirds of Core RBAC functions being dependent on vendor-specific extensions (see column “Additional Required Components”). The architects of EJB might

Core RBAC Functions	EJB Data Structures Support	**Additional Required Components**	
		User Management	Sessions and Role Activation
Administrative Commands			
AddUser		√	
DeleteUser		√	
AssignUser		√	
DeassignUser		√	
AddRole	√		
DeleteRole	√		
GrantPermission	√		
RevokePermission	√		
Supporting System Functions			
CreateSession			√
DeleteSession			√
AddActiveRole			√
DropActiveRole			√
CheckAccess	√		
Review Functions			
AssignedUsers		√	
AssignedRoles		√	
Advanced Review Functions			
RolePermissions	√		
SessionPermissions			√
UserPermissions		√	
SessionRoles			√
RoleOperationsOnObject	√		
UserOperationsOnObject		√	

Table 3.5: Functions defined by ANSI Core RBAC and their support by EJB data structures

have intentionally left the notion of user and support for user management as well as session and role activation beyond the scope of the specification. In order to provide standard support for administering and reviewing user accounts, their roles and their sessions, the corresponding administrative interfaces would need to be added to EJB. However, such a revision would be contrary to the emerging state of practice for application systems.

The notable trend in IT systems design is to "outsource" the functionality for administering

user accounts, and in some cases permissions, to single sign-on (SSO) [PM03] solutions for new applications [Got05] and to identity management (IdM) solutions for existing applications [BS03]. As a result, user accounts, and sometimes permissions, are administered across multiple application instances and types "outside" of the applications themselves. Therefore, an application system can only be successfully evaluated for compliance with ANSI RBAC when the application is considered together with the corresponding SSO or IdM solution. This condition makes evaluation of support for ANSI RBAC prohibitively expensive for systems designed to work in conjunction with multiple SSO or IdM solutions, as the evaluation would have to be performed for every combination of the system and the supporting SSO/IdM. Defining a separate ANSI RBAC profile for SSO/IdM solutions is a possible alternative to explore.

The other limitations of the EJB specification relate to Hierarchical and Constrained RBAC components of ANSI RBAC. The EJB specification does not define support for either role hierarchies or separation of duty. In Sections 3.3.1 and 3.3.1, we sketch approaches for supporting the two components. However, additional data must be maintained outside of the standard deployment descriptor in order to implement role hierarchies.

3.5 Conclusion

In this chapter, we analyzed support for ANSI RBAC in EJB 3.0 compliant systems. Specifically, we defined a configuration of the EJB protection system in precise and unambiguous terms using set theory. Based on this configuration definition, we formally specified the semantics of authorization decisions in EJB. We analyzed support for various ANSI RBAC components in EJB, and illustrated our discussion with an example.

Our analysis shows a mismatch between the access control architectures of EJB and ANSI RBAC. Although, the specification of access controls in EJB does employ roles, it does not fully support even Core ANSI RBAC. The limitations are mainly due to the lack of support for (1) user accounts and their management, (2) user sessions, and (3) role activation. While these limitations can be easily worked around through vendor-specific and implementation dependent extensions, each EJB implementation would have to be evaluated for ANSI RBAC separately. In order to provide standard support for administering and reviewing user accounts, their roles and their

sessions, the corresponding administrative interfaces would need to be added to EJB, which would be contrary to the emerging practice of "outsourcing" such functions to enterprise-wide single sign-on and identity management solutions.

This chapter establishes a framework for analyzing support for ANSI RBAC in EJB implementations. The results provide directions for EJB developers implementing ANSI RBAC in their systems, and criteria application owners in selecting such EJB implementations that support required and optional components of ANSI RBAC.

Chapter 4

Analysis of ANSI RBAC Support in COM+

In this chapter, we provide an overview of, and analyze the access control architecture of COM+. We formalize the protection system state for COM+ in a less ambiguous language than the corresponding COM+ documentation. Using this formalization, we suggest an algorithm that specifies the semantics of authorization decisions in COM+. This is followed by analysis of the level of support for the ANSI RBAC components and functional specification in COM+. Finally, we discuss our findings.

4.1 Overview of COM+ Security

The following sub-section provides a brief informal overview of COM+. More information can be found on the Microsoft MSDN Library web site [Mic08], or various COM+ books, such as [Edd99].

4.1.1 COM+

The Microsoft Component Object Model (COM), Distributed COM (DCOM), and COM+ are all programming frameworks that provide a model for creating component based software.

The Component Object Model (COM) [Box97] provides an architecture for simple interprocess communication. COM is an object-based programming model and a binary standard that enables components written in different languages to interoperate. Like CORBA, EJB, and Web Services, COM is based on the principles of information hiding and design by contract [Som06]. This enables software components to be reused without any dependencies on the way a component interface is implemented, as long as the implementation satisfies the component's specification. The reuse of these components is based on compiled, binary code. This allows COM components to be upgraded

in already deployed systems without having to recompile the applications that use them. Various languages such as C++, Visual Basic and others can be used to develop COM components.

DCOM [BK98] extends COM with the support for distributed interprocess communication between COM applications; that is, the DCOM architecture enables components or processes to communicate across a computer network. The DCOM communication protocol consists of a set of extensions, layered on the distributed computing environment (DCE) RPC specification [TOG97], providing object-oriented remote procedure call (ORPC).

Built on top of COM, and using the DCOM communication protocol, COM+ provides services that handle object and connection pooling, thread synchronization, security, and other resource management tasks. The goal of COM+, like other middleware technologies, is to facilitate application development and deployment without requiring application developers to deal with low-level tasks such as load balancing, distributed transactions, remote method invocation, and so forth. The following are definitions of various COM+ terms.

Interface defines a set of public operations (a.k.a. methods) that can be invoked by client applications. The interface does not provide any implementation for those methods. In other words, the interface defines a specific way for using the COM+ component. Each interface is identified by a globally unique identifier (GUID).

Class is a software construct that provides a concrete implementation of one or more interfaces. These classes are compiled into binary files called servers.

Object An executable instance of a COM class.

Component is a software unit of composition with specified interfaces [SGM02]. In COM, a component is compiled code (usually in the form of a library) that complies with the COM standard, and can create COM objects.

Server is a collection of one or more classes that provide services to clients. These services are provided through the methods of the COM classes. In addition to containing the implementation logic for the classes, servers also support standard COM methods for object activation.

Client is software code that requests creation of server objects, invokes methods on those objects in order to utilize the services offered by the server, and releases those objects.

Application is a group of of one or more components. An application can be a server, a client, or a collection of both. There are two types of COM+ applications: *Server Applications* and *Library Applications*. Server Applications run in their own processes whereas Library Applications run in the same process as their clients.

Given these definitions, we proceed to describe two aspects of a COM+ system: the declarative part, and the runtime part.

Declarative Part Since various programming languages are used to develop COM components, a means to describe COM classes and interfaces in a programming language independent manner is required. The Microsoft Interface Definition Language (MIDL) [Mic05a] is used for this purpose. Once COM+ interfaces are defined in MIDL, an MIDL compiler is used to generate the software code required to implement each interface.

Figure 4.1 illustrates the definition of an example interface IEmployee and class CEmployee using MIDL. The first three lines include IDL definitions for the base interface of all COM interfaces, IUnknown; and two custom interfaces used in the example IProject and IExperience . Lines 5-10 define a custom structure for the EmployeeInfo data type. The *BSTR* keyword defines a string. Lines 12-15 define all attributes for the IEmployee interface. The *object* keyword informs the MIDL compiler to generate C++ code to be used to implement COM objects; when this keyword is not used, the MIDL compiler generates code suitable for DCE RPC programs. Since each interface should be uniquely identified, a Universally Unique Identifier (UUID) [TOG97] is used on line 4 to identify this interface.[6]

Lines 16-23 in the example contain the actual definition of the IEmployee interface, which inherits from IUnknown, as required for all COM interfaces. In addition to the methods specified on lines 17-22, the IEmployee interface also inherits the AddRef, Release, and QueryInterface methods defined in the base interface IUnknown. The former two methods are used for maintaining the life cycle of COM+ objects, and since a single class may implement more than one interface, QueryInterface() method is used to obtain a reference to a specific interface implementation. For each method parameter, *in* or *out* attributes define parameters to be set by the caller or returned to

[6] A UUID is equivalent to a GUID. Although the latter is more commonly used in COM+, the keyword uuid is used in IDL files. GUIDs are either created using the guidgen.exe utility, or programmatically using the CoCreateGuid function.

```
1  import "unknwn.idl"
2  import "project.idl"
3  import "experience.idl"
4
5  typedef struct EmployeeInfoStruct
6  {
7    BSTR familyName;
8    BSTR* middleNames;
9    BSTR givenName;
10 } EmployeeInfo;
11
12 [
13     object,
14     uuid(72d797d5-9f5d-4673-bf7b-ba1955ccb343),
15 ]
16 interface IEmployee:IUnknown {
17    HRESULT GetBasicInfo([out] EmployeeInfo* pInfo);
18    HRESULT AssignToProject([in] IProject project);
19    HRESULT UnassignFromProject([in] IProject project);
20    HRESULT AddExperience([in] IExperience experience);
21    HRESULT GetExperience([out] IExperience* pExperience);
22    HRESULT Fire();
23 }
24
25 [ uuid(82f19809-e2c4-4ac3-a7f7-b22da586f906) ]
26 library EmployeeLib {
27    [ uuid(4a317abe-f759-4b5b-81e1-a34a3a7a927c) ]
28    coclass CEmployee
29    {
30       interface IEmployee;
31    }
32 };
```

Figure 4.1: An example employee.idl file

the caller, respectively. The *HRESULT* is a type that encapsulates a 32-bit return value indicating either successful method execution or a specific error. Lines 27-31 define the class that will be included in the EmployeeLib library and what interfaces the class implements, along with the class attribute(s), such as its *uuid*.

Runtime Part COM+ objects have certain attributes that specify their runtime requirements for using various COM+ services, such as synchronization, transactions, security, and so on. These attributes are maintained in a repository referred to as the COM+ catalog. When a client application creates an instance of a COM+ server object, the COM+ catalog is consulted for information required to instantiate the server.

Each COM+ component has a set of attributes that defines the component's run-time needs, such as transactional, threading, security, and other requirements. A context is a set of runtime constraints associated with one or more COM objects. Each object is associated with only a single context for the duration of the object's life cycle; and each context is associated with exactly one apartment. If the caller and the target object are located in the same context, no constraint checks, including those related to security, are performed; however, if they are running in different contexts, the incoming call goes through an interceptor. The interceptor can do whatever is necessary to satisfy the runtime constraints.

Some of the runtime constraints are related to application's security. COM+ security controls the invocation of object methods in order to allow only authorized users to execute those methods. Several COM+ security features can be used to protect applications. The following section provides an informal description of various COM+ security aspects.

4.1.2 Security Subsystem

The security model in COM+ employs roles for expressing access control policies. A COM+ role identifies a group of users that share the same permissions to access services provided by a COM+ application. Once roles are defined for an application, the administrator assigns individual users or user groups to roles. Permissions are granted to roles to access certain components, interfaces, or methods in the application.

COM+ security functions are enforced outside of the application through security interceptors,

which are lightweight object proxies. These interceptors ensure that the role attempting to access the server component is authorized to do so. And before clients are authorized to invoke server methods, they may have to be authenticated. COM+ provides various levels of authentication that can be used to secure calls into an application.

Similar to other middleware technologies, a client-side layer (we refer to this layer as the Client Security Service, or CSS) and server side layer (Target Security Service, or TSS) are responsible for enforcing COM+ security. The following is a list of various functions provided by these layers.

Client Security Service

The CSS is responsible for providing an interface for clients to examine or modify the security settings of a particular connection with an out-of-process COM+ object. For example, the CSS informs the client application what authentication levels are acceptable by the server. The CSS is also responsible for passing client credentials to TSS, when required. On the other hand, if the client application requires the server identity to be authenticated, CSS will enforce this requirement. When a connection is established between the client and the server, CSS protects request messages and verifies response messages.

Target Security Service

In addition to participating in the authentication protocol negotiation with the client, the TSS supports administration of the server security. TSS can also be given a security descriptor containing a discretionary access control list (DACL) and perform process-wide coarse-grained access checks against the DACL and security tokens of the clients on all incoming calls. TSS interceptor enforces access policies whenever a call is to be dispatched to the application.

Implementation of Security Functions

COM+ provides applications with security features, such as authorization and authentication. In order to secure COM+ applications, authorization and authentication features of COM+ are required at a minimum. COM+ also offers other security features, such as auditing. Based on the security requirements for each application, various COM+ security features can be utilized. The

following is a brief description of the minimum requirements to secure a COM+ application; we elaborate more on access control in Section 4.2.1.

Authentication In COM+, Security Service Providers (SSPs) offer authentication services to both clients and servers. SSPs are implemented as DLLs, and can support a variety of authentication protocols: Kerberos [MNSS87], Windows NT LAN Manager (NTLM) challenge-response authentication protocol [Mic05b], public/private key [AL02] based authentication protocols, and some other authentication protocols.

COM+ allows server applications to be configured to require different levels of client authentication. The names of these levels and their descriptions are as follows:

- None: no client authentication is required
- Connect: authentication is required when a connection between the client and server applications is established
- Call: authentication is required on every method invocation
- Packet: authentication is required for each network packet
- Packet Integrity: authentication is required for each packet, and data integrity is also checked; and
- Packet Privacy: authentication is required for each packet; data encryption and integrity checking are also required

Once the client is authenticated, the COM+ roles to which the client's principal or group are assigned become all activated.

Administration The Microsoft Component Services administrative GUI can be used to deploy and configure COM+ applications. The GUI allows administrators to do the following:

- create application specific roles,
- assign users and groups to roles,

- assign permissions to roles,
- specify the minimum level of authentication and message protection a COM+ application would accept,
- enable authorization checks.

The Component Services GUI is built on top of the Component Services Administration Library (COMAdmin) [Mic06a]. This means that the functionality provided by the GUI can be also achieved programmatically, allowing administrative tasks to be automated through, for example, scripting.

4.2 COM+ Protection State

In this section, we informally describe access control architecture for COM+. Then, we formally define a configuration of the COM+ protection state. The COM+ concepts presented here are common to both COM+ versions 1.0 [Obe00] and 1.5 [Low01].

4.2.1 COM+ Access Control

Authorizations in COM+ can be specified at the granularity of the component (all class instances), interface, or method. If a client is permitted to access a component as a whole, then that client can invoke any of the component's methods. If the client is permitted to access only certain interfaces in the component, the client will be able to invoke only the methods in those interfaces. The scope of rights on interfaces is limited to the components implementing them, which means that different clients could have different access rights to the same interface implemented by different components. Furthermore, the client can be permitted to invoke only certain methods in an interface.

The built-in security of COM+ provides several features that can be used to protect COM+ applications. COM+ provides two methods of controlling access to resources: *declarative* and *programmatic*. The declarative method can be used to control access to components, interfaces, or even methods. Using the declarative approach, access control can be achieved without having to write code. As various application attributes are stored in the COM+ catalog, administrative tools

can be used to manipulate the COM+ catalog and configure access control for various application components. This approach facilitates the decoupling between application logic and security logic.

On the other hand, the programmatic approach can be used to achieve finer granularity of control. Interface methods can be implemented to check role memberships of clients using functions such as IsCallerInRole(). In addition, the following interfaces provide extra information pertaining to security as follows:

`ISecurityCallContext` provides access to information on the current method invocation.

`ISecurityCallersColl` provides access to information about individual callers in the collection of callers.

`ISecurityIdentityColl` provides access to the collection of information pertaining to the caller's identity.

The TSS controls client access to COM+ server applications. Based on the server application's access policy, security checks are performed before a client's call is successfully dispatched to a server object. For example, if the COM+ server is not running, the client needs to have sufficient permissions to launch the server application before any method can be invoked. TSS checks client permissions to activate the server process. In addition, when a call enters the running server process, further access checks are performed.

Access permissions are enforced using roles. The Component Services GUI allows administrators to create roles for a specific application when deploying it, and to map users to those roles. Once the roles are created, the administrator can choose which components, interfaces and methods in the COM+ application can be accessed by the users assigned to those roles. Figure 4.2 uses the Unified Modeling Language (UML) notation to summarize the relationships among authorization-related elements of the COM+ access control architecture, where *account* and *group* in the figure refer to Microsoft Windows based user account and users group, respectively.

An example of role-permission assignments in a COM+ application is shown in Table 4.1. The first row illustrates assignments of permissions to invoke method m_1 on interface i_1 in component c_1 to roles r_1 and r_2. This indicates that only principals with roles r_1 and/or r_2 are allowed to invoke method m_1 on c_1. The second row in the table illustrates an assignment of permission

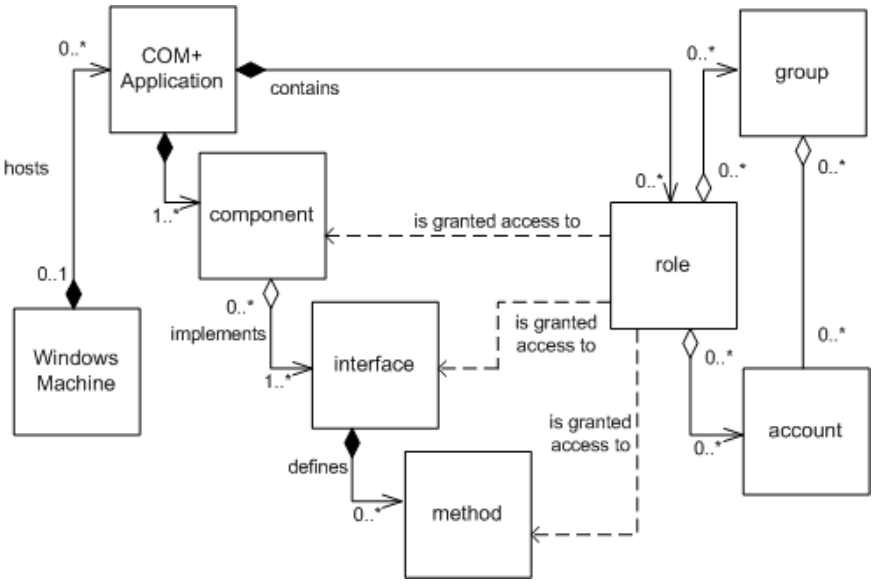


Figure 4.2: UML model of COM+ access control architecture

Roles	Methods
r_1, r_2	$c_1 i_1 m_1$
r_3	$c_2 i_1 *$, $c_2 i_2 *$
r_1, r_2, r_3	$c_3 *$

Table 4.1: Examples of role-permission assignment in a COM+ Application

to invoke all methods on interfaces i_1 and i_2 in component c_2 to role r_3. The last row shows an example of allowing roles r_1, r_2, and r_3 to invoke all methods provided by component c_3.

4.2.2 Formalization of the Protection State

In this section, we formalize the protection state of a COM+ system. In this formalization, we attempt to preserve the COM+ terminology.

Definition 7 [COM+ Protection State] *A configuration of a COM+ system protection state for a given application is a tuple* $(R, U, G, UGA, C, I, M, UA, GA, PA, isSecurityEnabled, user\_roles)$ *interpreted as follows:*

- *R is a set of the COM+ security roles as defined in the COM+ catalog for a specific application.*
- *U is a set of users.*

- *G is a set of user groups.*
- *$UGA \subseteq U \times G$ is a many-to-many relation of users to groups.*
- *C is a set of COM+ components for a specific application.*
- *I is a set of COM+ interfaces provided by the COM+ components in a specific application.*
- *M' is a set of COM+ method signatures, $\{m_1, m_2, ...\}$*
- *$M \subseteq C \times I \times M'$ is a set of COM+ methods implemented for the provided interfaces. Members of this set are denoted $c_j.i_k.m_l$, where $c_j \in C$, $i_k \in I$, and $m_l \in M$. The set also includes the elements $c_j.i_k.m_*$, which are all methods in interface i_k provided by component c_j; and the elements $c_j.i_*.m_*$, which are all methods in all interfaces provided by component c_j.*
- *$UA \subseteq R \times U$ is a relation of COM+ security roles to users.*
- *$GA \subseteq R \times G$ is a relation of COM+ security roles to groups.*
- *$PA \subseteq R \times M$ is a role-to-method relation.*
- *isSecurityEnabled is a boolean indicating whether access control should be enforced.*
- *$direct\_user\_roles(u : U) : U \rightarrow 2^R$ is a function mapping each user u to a set of roles that u is directly assigned to. Formally, $direct\_user\_roles(u : U) \subseteq \{r|(r, u) \in UA\}$.*
- *$group\_roles(g : G) : G \rightarrow 2^R$ is a function mapping each group g to a set of roles that g is directly assigned to. Formally, $group\_roles(g : G) \subseteq \{r|(r, g) \in GA\}$.*
- *$user\_groups(u : U) : U \rightarrow 2^G$ is a function mapping each user u to a set of groups that u is a member of. Formally, $user\_groups(u : U) \subseteq \{g|(u, g) \in UGA\}$.*
- *$indirect\_user\_roles(u : U)$ is a function mapping user groups to a set of roles. Formally, $indirect\_user\_roles(u : U) \subseteq \bigcup_{g \in user\_groups(u)} \{r|(r, g) \in GA\}$, where these roles are indirectly assigned to the user because the roles are (directly) assigned to the groups to which the user belongs.*
- *$user\_roles(u : U) \equiv direct\_user\_roles(u) \cup indirect\_user\_roles(u)$ is a set of all user roles.*

Given the protection state of a COM+application, Algorithm 10 defines the outcome of an access control decision. If isSecurityEnabled is true, it means that the application deployer or administrator explicitly enabled component level access checks. In such case, the algorithm proceeds to check the calling user's role membership. If any of the roles the user is assigned to has explicit permission to invoke the method m_l in interface i_k in component c_j, the algorithm will authorize the user to invoke the method in question. Furthermore, if any of the roles the user is assigned to has implicit permission to invoke the method in question, the algorithm will also authorize the user to invoke the method. By implicit permission we mean a permission that is inferred from either allowing the role to invoke all methods in the specific interface i_k that m_l belongs to, or allowing the role to invoke all methods in all interfaces in a specific component c_j that m_l is part of. If none of these conditions (either explicit or implicit permissions) is true, the authorization algorithm denies the user its request to invoke the method.

Authorize$(p : 2^R, c.i.m : M) \rightarrow \{allow, deny\}$
if $isSecurityEnabled \neq true$ **then**
  return allow
else
  for all $r \in p$ **do**
    if $(r, c.i.m) \in PA \vee (r, c.i.m_*) \in PA \vee (r, c.i_*.m_*) \in PA$ **then**
      return allow
    end if
  end for
  return deny
end if

Algorithm 10: Authorization decision in COM+

4.3 Analysis of ANSI RBAC Support in COM+

For a system to comply with ANSI RBAC, Core RBAC must be implemented at a minimum; the other three RBAC components (Hierarchical RBAC, Static Separation of Duty Relations, and Dynamic Separation of Duty) as defined in Section 1.2 are optional. In Section 4.3.1 we first examine the extent to which a COM+ protection state—as formalized in Definition 7—can support each of the four ANSI RBAC model components. In Section 4.3.3 we illustrate our formalization with an example. In Section 4.3.4, we then analyze the degree to which COM+ supports the

functional specification of ANSI RBAC.

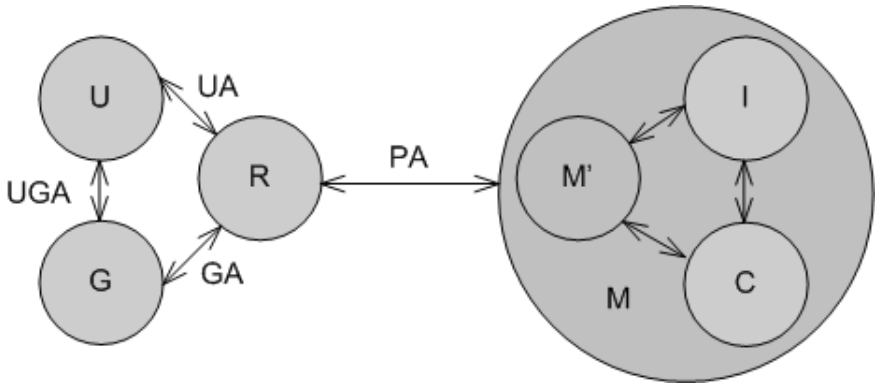
4.3.1 Reference Model

Core RBAC

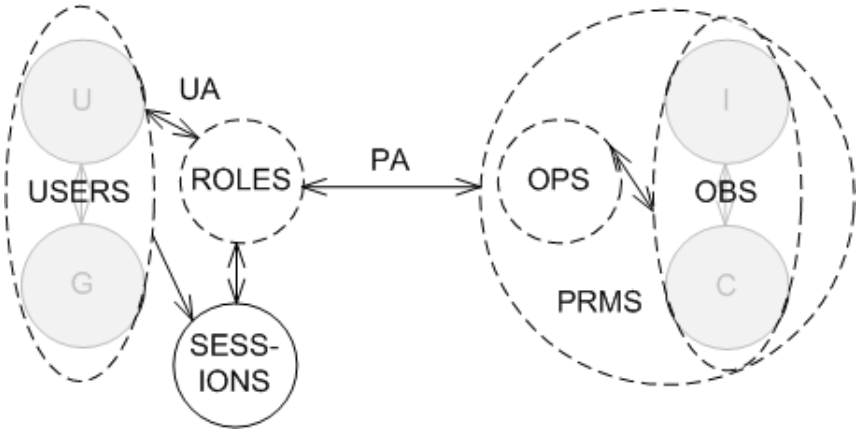
The COM+ protection state configuration provided in Section 4.2 can realize security policies that are based on Core RBAC as follows. Core RBAC *ROLES* map directly to COM+ security roles (R). Since permissions in COM+ can be assigned to users as well as individual groups, the $USERS$ set in RBAC readily maps to the union of U and G. RBAC permission assignments (PA) are equivalent to those in COM+ (PA). UA in RBAC is equivalent to UA in COM+. To aid with the understanding of the correspondence between elements in the RBAC model and EJB, we present in Figure **??** the sets and relations of COM+ (with light grey background) and RBAC (with white background). Shapes with white background and dashed lines show mapped RBAC sets. The reader is encouraged to compare it to the diagram in Figure 1.1. More formally, we define Core RBAC in the language of the COM+ protection system as follows:

Definition 8 [Core RBAC in COM+] *Core RBAC in the language of COM+ is defined by the COM+ system protection state outlined in Definition 7, as well as the following additional elements:*

- $USERS = U \bigcup G$ *is a set of users and groups, where members of* $USERS$ *are MS Windows based user accounts and groups.*
- $ROLES = R$ *is a set of roles, where members of* $ROLES$ *are the roles defined for a specific COM+ application.*
- OPS *is a set of operations, where members of this set are operations that can be invoked on COM+ components; for example,* $OPS = \{m_x, m_y, m_*, ...\}$.
- $OBS \subseteq C \times I$ *is a set of objects, where these objects are defined to be specific interfaces on certain components, or all interfaces on a certain component; for example,* $OBS = \{c_1i_y, c_1i_z, c_1i_*, c_2i_x, c_2i_*, ...\}$.
- $UA = USERS \times ROLES$, *is a many-to-many assignment of users to roles.*



(a) COM+ protection state sets



(b) Mapped and unmapped RBAC sets

Figure 4.3: COM+ (with light grey background) and RBAC (with white backround) sets and relations.

- *$assigned\_users(r : ROLES) = \{u \in USERS | (u, r) \in UA\}$, is a function that returns the set of users in $USERS$ that are assigned to the given role r.*

- *$PRMS \subseteq OPS \times OBS$ is a set of permissions to invoke COM+ interface methods for certain components. The existence of $c_j i_k m_l$, $c_j i_k m_*$ or $c_j i_* m_*$ in $PRMS$ provides permission to invoke a specific method m_l, all methods in interface i_k or all methods in all interfaces in component c_j, respectively; for example, $PRMS = \{c_1 i_y m_x, c_1 i_z m_*, c_2 i_x m_y, ...\}$*

- *$PA \subseteq PRMS \times ROLES$, a many-to-many assignment of permissions to COM+ roles.*

- *$assigned\_permissions(r : ROLES) = \{p \in PRMS | (p, r) \in PA\}$, is a function that returns the set of permissions in $PRMS$ that are assigned to the given role r.*

- *$Op(p : PERMS) \rightarrow \{op \in OPS\}$, a function that returns a set of operations that are associated with the given permission p. For example, $Op(c_j i_k m_l) = m_l$.*

- *$Ob(p : PERMS) \rightarrow \{ob \in OBS\}$, a function that returns a set of objects that are associated with the given permission p. For example, $Ob(c_j i_k m_l) = \{c_j i_k\}$.*

- *$SESSIONS$ is a set of sessions for a specific application. Members of this set are mappings between authenticated users and their activated roles for a specific COM+ application. Like many other systems, in a COM+ application environment, all roles and permissions are activated or turned on for a user once the user is authenticated.*

- *$session\_users(s : SESSIONS) \rightarrow USERS$, the mapping of session s onto the corresponding user.*

- *$session\_roles(s : SESSIONS) \rightarrow 2^{ROLES}$, the mapping of session s onto a set of roles. Formally: $session\_roles(s_i) \subseteq \{r \in ROLES | (session\_users(s_i), r) \in UA\}$.*

- *$avail\_session\_perms(s : SESSIONS) \rightarrow 2^{PRMS}$, the permissions available to a user in a session $= \bigcup_{r \in session\_roles(s)} assigned\_permissions(r)$.*

As shown in Definition 8, most of the elements required to support Core RBAC are already provided in the COM+ protection state (Definition 7). However, the elements related to $SESSIONS$ are not addressed in Definition 7. This is due to the fact that user sessions are specific to the MS

Windows platform and are not specific to COM+. Since all operating system processes on an MS Windows platform must be associated with a logon session, these sessions are handled by the operating system and are transparent to the COM+ application.

Hierarchical RBAC

General Role Hierarchies and Limited Role Hierarchies comprise the Hierarchical RBAC component of the ANSI RBAC Reference Model. Both role hierarchies are formally defined in terms of the sets and relations of the Core RBAC component. These components are described in Section 3.3.1. Neither the COM+ catalog nor the administrative tool that is part of the COM+ environment directly support creating hierarchical relationships between roles.

Constrained RBAC

Static separation of duty (SSD) and dynamic separation of duty (DSD) relations are part of the Constrained RBAC component of ANSI RBAC [ANS04]. Similar to the Hierarchical RBAC component, these relations are defined in terms of Core RBAC elements. Essentially, SSD constrains user-to-role assignment (UA set and $assigned\_users$ function) and the role hierarchy (RH set and $authorized\_users$ function). DSD, on the other hand, constrains the role activation ($SESSIONS$ set and $session\_roles$ function). The COM+ catalog does not allow for specifying any constraints on user-to-role assignments, whether static or dynamic; neither does it allow for specifying any constraints on role activation. As such, SSD and DSD are not supported in COM+.

4.3.2 Translating RBAC Policies to COM+

In Definition 7 and Definition 8 we presented a protection state for COM+ systems, and how Core RBAC can be modeled in the language of the COM+ protection state, respectively. In this section, we present an algorithm that translates an arbitrary RBAC policy into an COM+ protection state.

Algorithm 11 formalizes the translation from an RBAC policy to the COM+ protection state defined in Definition 7. For clarity, we identify the RBAC sets in the algorithm with an RBAC subscript. The algorithm requires the following two functions defined as follows:

- $component(o : OBS) \rightarrow c$: returns the component corresponding to a given object o.

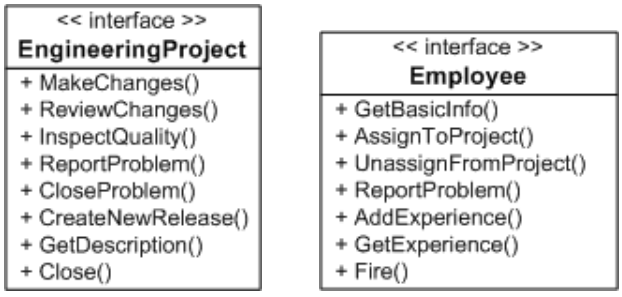
- $interface(o : OBS) \rightarrow i$: returns the interface corresponding to a given object o.

1: {Initialize COM+ sets and relations.}
2: $R \leftarrow ROLES_{RBAC}$
3: $U \leftarrow USERS_{RBAC}$
4: $G \leftarrow \emptyset$
5: $UGA \leftarrow U$
6: $C \leftarrow \emptyset$
7: $I \leftarrow \emptyset$
8: $M \leftarrow \emptyset$
9: $UA \leftarrow \emptyset$
10: $GA \leftarrow \emptyset$
11: $PA \leftarrow \emptyset$
12: $isSecurityEnabled \leftarrow$ **true**
13: **for all** $p \in PRMS_{RBAC}$ **do**
14: **for all** $(opr, obj) \in p$ **do**
15: $c \leftarrow component(obj)$
16: $i \leftarrow interface(obj)$
17: $C \leftarrow C \cup \{c\}$
18: $I \leftarrow C \cup \{i\}$
19: $M \leftarrow M \cup \{opr\}$
20: **end for**
21: **end for**
22: **for all** $pa \in PA_{RBAC}$ **do**
23: **for all** $((opr, obj), r) \in pa$ **do**
24: $UA \leftarrow UA \cup \{r\} \times assigned\_users(r)$
25: $PA \leftarrow PA \cup \{(r, opr)\}$
26: **end for**
27: **end for**

Algorithm 11: Operational definition of translating from an ANSI RBAC system state to the one of COM+.

4.3.3 Example

In this section we present an example that illustrates how ANSI RBAC can be supported in a COM+ system as discussed earlier. This example is a simple COM+ application that maintains employee and engineering project records in an engineering company. The application allows users to perform various operations on the project and employee records, based on the users' roles in the company. The application consists of a single component, EngineeringProjectService (EPS), which supports the following COM+ interfaces: EngineeringProject, and Employee. These interfaces are shown in Figure 4.4. In this example, we define seven different user roles. Based on these roles



(a) EngineeringProject interface

(b) Employee interface

Figure 4.4: Example COM+ interfaces

and according to the policies listed in Figure 4.6, users are allowed to invoke various methods in this application. These roles are defined as follows:

- *Employee* represents a company employee.
- *Engineering Department* represents an employee of the engineering department.
- *Engineer* performs various engineering tasks in the company.
- *Product Engineer* is responsible for managing a product line.
- *Quality Engineer* is a quality assurance engineer.
- *Project Lead* overseas and leads the development of a project.
- *Director* is an engineering department director.
- *Administrator* represents all employees who belong to upper management as well as operations.

System access control policy that defines what actions each role is allowed to perform is summarized in Table 4.2, where a check mark ("$\surd$") denotes a granted permission for a specific COM+ role to execute the corresponding method. Tables in Figure 4.5 show examples of users-to-roles assignments, groups-to-roles assignments, and an example of system users, and their group memberships from the underlying MS Windows operating system environment. The following is a formalization of this example system's protection state as in Definition 7.

User	Role
Alice	Employee
Bob	Engineer
Carol	Quality Engineer
Dave	Product Engineer
Eve	Project Lead
Fred	Director

(a) User-to-role mappings

Group	Role
hardware	Engineering Department
software	Engineering Department
accounting	Administrator
management	Administrator

(b) Group-to-role mappings

User	Group
Alice	accounting
Bob	hardware
Carol	software
Dave	software
Eve	software
Fred	management

(c) User-to-group mappings

Figure 4.5: Example COM+ application user, group, and role mappings

1. Anyone in the organization can look up an employee's name.
2. Everyone in the engineering department can get a description of and report problems regarding any project and look up experience of any employee.
3. Engineers, assigned to projects, can make changes and review changes related to their projects.
4. Quality engineers, in addition to being granted engineers' rights, can inspect the quality of projects they are assigned to.
5. Product engineers, in addition to possessing engineers' rights, can create new releases.
6. The project lead, in addition to possessing the rights granted to production and quality engineers, can also close problems.
7. The director, in addition to being granted the rights of project leads, can manage employees (assign them to projects, un-assign them from projects, look up experience, add new records to their experience, and fire them) and close engineering projects.
8. Everyone who is an administrator can get the description of a project, and look up the name and experience of any employee.

Figure 4.6: Sample authorization policy for the example COM+ application describing what actions are allowed. All other actions are denied.

COM+ Roles	**COM+ Methods**													
	EngineeringProject								*Employee*					
	MakeChanges()	ReviewChanges()	InspectQuality()	ReportProblem()	CloseProblem()	CreateNewRelease()	GetDescription()	Close()	get_basic_info()	AssignToProject()	UnassignFromProject()	AddExperience()	GetExperience()	Fire()
Employee									√				√	
Engineering Department				√			√		√				√	
Engineer	√	√							√				√	
Product Engineer						√			√				√	
Quality Engineer			√						√				√	
Project Lead					√				√				√	
Director								√	√	√	√	√	√	√
Administrator							√		√				√	

Table 4.2: Example COM+ role-method permissions

- R = {Employee, Engineering Department, Engineer, Product Engineer, Quality Engineer, Project Lead, Director, Administrator}
- U = {Alice, Bob, Carol, Dave, Eve, Fred}
- G = {hardware, software, accounting, management}
- UGA = {(Alice, accounting), (Bob, hardware), (Carol, software), (Dave, software), (Eve, software), (Fred, management)}
- C = {EPS}
- I = {EngineeringProject, Employee}
- M = {EPS.EngineeringProject.MakeChanges, EPS.EngineeringProject.ReviewChanges, EPS.EngineeringProject.InspectQuality, EPS.EngineeringProject.ReportProblem, EPS.EngineeringProjcet.CloseProblem, EPS.EngineeringProject.CreateNewRelease, EPS.EngineeringProject.GetDescription, EPS.EngineeringProject.Close, EPS.Employee.GetBasicInfo, EPS.Employee.AssignToProject, EPS.Employee.UnassignFromProject, EPS.Employee.AddExperience, EPS.Employee.GetExperience, EPS.Employee.Fire }

- UA = {(Employee, Alice), (Engineer, Rob), (Quality Engineer, Carol), (Product Engineer, Dave), (Project Lead, Eve), (Director, Fred)}

- GA = {(Engineering Department, hardware), (Engineering Department, hardware), (Administrator, accounting), (Administrator, management)}

- PA = { (Employee, EPS.Employee.GetBasicInfo), (Employee, EPS.Employee.GetExperience), (Engineering Department, EPS.EngineeringProject.ReportProblem), (Engineering Department, EPS.Employee.GetBasicInfo), (Engineering Department, EPS.Employee.GetExperience) (Engineer, EPS.EngineeringProject.MakeChanges), (Engineer, EPS.EngineeringProject.ReviewChanges), (Engineer, EPS.Employee.GetBasicInfo), (Engineer, EPS.Employee.GetExperience), (Product Engineer, EPS.EngineeringProject.CreateNewRelease), (Product Engineer, EPS.Employee.GetBasicInfo), (Product Engineer, EPS.Employee.GetExperience), (Quality Engineer, EPS.EngineeringProject.InspectQuality), (Quality Engineer, EPS.Employee.GetBasicInfo), (Quality Engineer, EPS.Employee.GetExperience), (Project Lead, EPS.EngineeringProject.CloseProblem), (Project Lead, EPS.Employee.GetBasicInfo), (Project Lead, EPS.Employee.GetExperience), (Director, EPS.EngineeringProject.Close), (Director, EPS.Employee.*), (Administrator, EPS.EngineeringProject.GetDescription), (Administrator, EPS.Employee.GetBasicInfo), (Administrator, EPS.Employee.GetExperience)}

- $isSecurityEnabled$ = true

In this example, the $user\_roles(u : U)$ function, as formalized in Definition 7, returns the roles assigned to a specific user whether this assignment is direct as specified in UA, or by inference using the information from the user's group assignment as specified in UGA *and* the user's group's role assignment as specified in GA. For example, $user\_roles(Fred) = \{Director, Administrator\}$, where the $(Director, Fred) \in UA$, and $(Fred, management) \in UGA$ *and* $(Administrator, management) \in GA$. In accordance with Definition 8, we also identify the following sets in order to support Core RBAC.

- $USERS$ = { Alice, Bob, Carol, Dave, Eve, Fred, hardware, software, accounting, management }

- OPS = {MakeChanges, ReviewChanges, InspectQuality, ReportProblem, CloseProblem, CreateNewRelease, GetDescription, Close, GetBasicInfo, AssignToProject, UnassignFromProject, AddExperience, GetExperience, Employee.Fire }
- $PRMS = M$

4.3.4 Functional Specification

In this section, we examine the COM+ middleware ability to support ANSI RBAC administrative operations for the creation and maintenance of RBAC element sets and relations, administrative review functions for performing administrative queries, and system functions for creating and managing RBAC attributes on user sessions and making access control decisions.

The COM+ Component Services Administration Library (COMAdmin) [Mic06a] provides a variety of classes and interfaces for managing COM+ applications, as well as for manipulating various attributes stored in the COM+ catalog.

The COMAdminCatalog class is one of the classes used to access COM+ configuration data stored in the COM+ catalog. The class implements two interfaces: ICOMAdminCatalog and ICOMAdminCatalog2; the latter is available only in COM+ version 1.5. The COMAdminCatalog provides the GetCollection method which can be used to retrieve COMAdminCatalogCollection objects that represent COM+ applications, COM+ components, and so on. Furthermore, each COM+ COMAdminCatalogCollection can be further queried for "sub-collections," more information, or manipulated as required.

Figure 4.7, illustrates the relationships amongst various collections. The arrows indicate the ability to navigate from one collection to another using the GetCollection method of the COMAdminCatalogCollection object .

In addition to the GetCollection method, the COMAdminCatalogCollection class also provides Add and Remove methods, which can be used to add or remove objects from a certain collection. For example, to add a new COM+ role to a certain application, an algorithm similar to the one outlined in Figure 4.8 can be used. The example omits common steps using custom methods such as GetCatalogObject, FindApplication, and CreateRoleObject.

In addition to COMAdmin, the Microsoft Windows operating system provides Network Man-

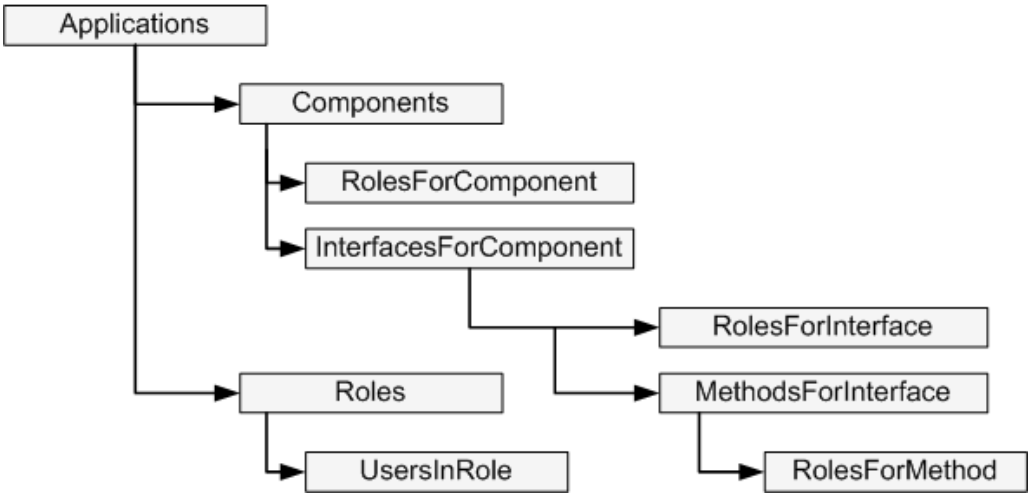


Figure 4.7: COM+ Administration Collections

agement APIs [Mic06b] that provide the ability to manage user accounts and network resources.

In the rest of this section we explore how the COMAdmin library APIs and the Network Management APIs can be used to implement administrative functional specifications for an ANSI RBAC system.

Administrative Commands for Core RBAC

AddUser, DeleteUser create a new user and delete an existing user from the system. These commands can be implemented using the NetUserAdd, and the NetUserDel methods of the Win32 Network Management APIs. The NetUserAdd method adds a new user account to a system given the system's Domain Name Service (DNS) [Moc87] name, or its NetBIOS [Gro87] name. On the other hand, the NetUserDel method deletes a user account from the system.

Considering Definition 7, the NetUserAdd and NetDelUser methods can be used to manipulate U by adding or deleting elements. In addition to these methods, the Win32 Network Management APIs NetGroupAdd, NetGroupAddUser, NetGroupDel, and NetGroupDelUser can be used to manipulate G, and UGA.

AddRole, DeleteRole methods are defined in ANSI RBAC to allow roles to be created and deleted from the RBAC system. These methods can be implemented by manipulating the

```
AddRole(in role, in application)
{
  COMAdminCatalog comAdminObj;
  COMAdminCatalogCollection appsColl, rolesColl;
  COMAdminCatalogObject roleObj, appObj;

  /* get the com admin obj */
  comAdminObj = GetCatalogObject();

  /* get a reference to the applications collection */
  appsColl = comAdminObj.GetCollection(``Applications'');

  /* find the specific application in the collection */
  app = FindApplication(appsColl, application);

  /* get the roles collection for that application */
  rolesColl = app.GetCollection(``Roles'');

  /* create a role object and add it to
   * the Roles collection for that application */
  roleObj = CreateRoleObject(role);
  rolesColl.Add(roleObj);
}
```

Figure 4.8: Psudo-code for adding a COM+ role to an application

COM+ catalog using the Add and Remove methods on the Roles COM+ collection. As shown in Figure 4.7, this collection objects can be accessed by first getting a reference to the Applications collection, then obtaining a reference to the Roles collection.

AssignUser, DeassignUser allow user-to-role assignments to be created and deleted. These commands can be implemented by manipulating the UsersInRole COM+ collection. A reference to this collection can be obtained through the Roles collection, which can be obtained from the Applications collection, which in turn can be obtained from the COM+ catalog. Once the reference to this collection is obtained, the Add or Remove method can be used to assign a user to a role, or to remove a user to role assignment.

GrantPermission, RevokePermission are used to grant or revoke the permission to invoke an operation on an object to a role. These methods can be implemented by manipulating the RolesForComponent, RolesForInterface, and RolesForMethod COM+ collections. Figure 4.7 shows how to navigate from the Applications collection to one of these role related collections. Using the Add and Remove methods provided by each one of these collections, a role object can be added to or removed from the collection. By adding a role to the RolesForComponent collection, for example, the role is granted permission to invoke all of the COM+ component's methods. On the other hand, adding a role to the RolesForMethod collection allows the role to invoke only a specific method (assuming the role is not added to the RolesForComponent or RolesForInterface collections).

CreateSession, DeleteSession, AddActiveRole, DropActiveRole are used to create and delete a session for a user, and activate or deactivate a role for a user in a given session, respectively. Sessions are handled by the Windows operating system, and are created upon user authentication. Role activation and deactivation are also handled by the Windows operating system, and are transparent to the user application. However, once the session is established, and throughout the session, roles cannot be deactivated and other roles cannot be activated for that session.

CheckAccess indicates whether a user is allowed or is not allowed to perform a given operation on a given object. Algorithm 10 defined in Section 4.2.2 can be used to implement CheckAccess.

However, since user sessions are handled transparently by the Windows operating system, the Authorize function in Algorithm 10 doesn't take a user session as an input parameter.

AssignedUsers, AssignedRoles return the set of users assigned to a specific role, and the set of roles assigned to a specific user, respectively. These functions can be implemented by querying various COM+ collections for their items. Besides the Add and Remove methods provided by the COM+ collections, the Count and Item methods are also provided, allowing administrative applications to query the collection for its items. AssignedUsers can be implemented directly by querying the UsersInRole collection; AssignedRoles can be implemented by searching for all roles assigned to a given user also using the UsersInRole collection.

Advanced Review Functions for Core RBAC

RolePermissions, UserPermissions return the set of permissions granted to a given role and user, respectively. RolePermissions can be implemented by identifying which roles are assigned to which component, interface, or method. This is done by querying the RolesForComponent, RolesForInterface, and RolesForMethod collections. UserPermissions, on the other hand, can be implemented by identifying which users are assigned to which roles using the UsersInRole collection; then RolePermissions can be used for each one of the user's roles to identify all permissions assigned to the user by knowing which roles the user is assigned.

SessionRoles, SessionPermissions return the active roles and permissions associated with a session. Since sessions are handled by the Windows operating system, and once a user is authenticated, a session is created for them, the AssignedRoles and UserPermissions methods discussed previously can be used to implement SessionRoles and SessionPermissions, respectively.

RoleOperationsOnObject, UserOperationsOnObject return the operations that a given role or user can perform on an object. These functions can be implemented by querying the RolesForComponent, RolesForInterface and RolesForMethod collections to identify which methods the given role is allowed to invoke. Similarly, the UserOperationsOnObject can be implemented by first identifying which roles are assigned to the given user using the

UsersInRole collection, then identifying which methods each one of the user's roles is allowed to invoke.

Table 4.3 provides a summary of the above discussion. Support for ANSI Core RBAC functions are classified in two categories as follows: the first category identifies the functions that can be supported using APIs built into the COM+ operating environment; the second category contains the functions that are not supported. Even if the operating environment provides APIs that are capable of implementing sessions and role activation related functionality, such implementation would be proprietary and completely outside the scope of the COM+ standard. As such, the ANSI Core RBAC functions in the last column of the table are flagged as *unsupported*.

4.4 Discussion

Results of our investigation suggest that COM+ falls short of fully supporting all functions required by the ANSI RBAC standard. As summarized in Table 4.3, COM+ supports 60% of ANSI Core RBAC functions. Fundamentally, all functions that relate to session management or role activation are not supported. This is mainly due to the fact that COM+ is tightly integrated with the underlying operating system. Based on the COM+ application access control policy, the operating system user accounts and groups are assigned to appropriate roles. If authentication is enabled for the COM+ application, the authentication of these users is performed in a transparent manner to the COM+ clients or servers. The ANSI RBAC functions that are not supported in COM+ are mainly a function of the underlying operating system. The following is a discussion of the consequences of the lack of this support.

As explained in Section 4.1.2, the authentication of users is the responsibility of the Security Service Provider (SSP). From a practical aspect, we believe that the concept of a *session* as discussed in the ANSI RBAC standard is really the login session with some extra attributes. These attributes would describe the roles of the authenticated users. In order to achieve this, we suggest a "role-aware" SSP that would implement the *CreateSession* and *DeleteSession* ANSI RBAC functions. These functions would now be part of the authentication system, and not the access control system. We suggest that the concept of session creation, maintenance, and deletion be part of a role-aware authentication system, and not the access control system.

Core RBAC Functions	Built-in API Support		
	Supported by COMAdmin Library APIs	Supported by Win32 Network Management APIs	Unsupported
Administrative Commands			
AddUser		√	
DeleteUser		√	
AssignUser	√		
DeassignUser	√		
AddRole	√		
DeleteRole	√		
GrantPermission	√		
RevokePermission	√		
Supporting System Functions			
CreateSession			√
DeleteSession			√
AddActiveRole			√
DropActiveRole			√
CheckAccess	√		
Review Functions			
AssignedUsers	√		
AssignedRoles	√		
Advanced Review Functions			
RolePermissions	√		
SessionPermissions			√
UserPermissions	√		
SessionRoles			√
RoleOperationsOnObject	√		
UserOperationsOnObject	√		

Table 4.3: Functions defined by ANSI Core RBAC and their support in COM+

Beside the session creation and deletion functions, COM+ does not support the *AddActiveRole* or *DropActiveRole* ANSI RBAC functions, either. The activation of roles, just like authentication, is handled transparently to the application. Upon authentication, all user roles are activated and

can be accessed programmatically using the IsCallerInRole method of the ISecurityCallContext COM+ interface. Ferraiolo et al argue that the fact that all roles are activated in a system potentially violates the principle of least privilege [FKS07]. However, how practical is it to achieve this principle in real systems? Activation of roles can be achieved in three ways: user-driven activation, system-driven activation, or perhaps a hybrid of both. In the first option, which is user-driven activation, the user is responsible for choosing which roles to activate in order to access various system functions. We believe this approach is impractical and has usability issues especially if the user is assigned multiple roles, and also requires the user to know the access control policy for the system. In the second option, a system-driven role activation scenario comes with its own challenges. This automation of role activation requires an algorithm that addresses many issues, such as when to activate a role, which role to activate if the same permissions are assigned to multiple roles, when should roles be deactivated during a session, how does the algorithm guarantee the prevention of information leak, and many other non-trivial issues. Some research that tackles system-driven role activation exists [AC06]; however, a formal approach to evaluate such algorithms is required. And finally, the third option of having a hybrid system also poses non-trivial issues of when the system should be responsible for activating roles, and when control is passed to the user.

Given the fact that all roles for a user are activated upon authentication and throughout their session, one can argue that some level of support for the *SessionPermissions* and *SessionRoles* functions is available, since the already supported *AssignedRoles* and *UserPermissions* functions return equivalent sets. However, since the session concept is not supported in COM+, support for the semantics of those functions is not available.

In addition to the lack of support for the aforementioned functions, role hierarchy is not addressed in COM+, and the Microsoft Component Services GUI described in Section 4.1.2 does not support role hierarchy. Nonetheless, role hierarchy is still possible though programmatic extensibility using the flexibility of the COMAdmin [Mic06a] library. A custom administrative application can be created to replace the Component Services GUI and provide support for role hierarchy. This application is required to maintain role hierarchy relations and manipulate various COM+ collections, such as the *UsersInRole* collection (see Figure 4.7). For example, when a role hierarchy is introduced, the custom administrative application is required to manipulate the *RolesForCom-*

ponent, *RolesForInterface*, and the *RolesForMethod* COM+ collections to ensure that appropriate roles are added to these collections to reflect the fact that roles would inherit permissions based on the role hierarchy.

Static separation of duty constraints are not supported by in a COM+; however, the custom administrative application that we discussed earlier would allow for this support. The application would implement functions such as *AssignUser*, for example, in a manner that would ensure that static separation of duty constraints are met before a user is assigned to a certain role. On the other hand, dynamic separation of duty constraints may not be implementable since role activation and sessions are handled by the operating system.

In summary, the role-based access control provided by COM+ falls short of supporting all of the functions mandated by the ANSI RBAC standard. The limitations that prevent full support of ANSI RBAC in COM+ are mainly related to the underlying operating system. A suggestion to improve the practicality of the ANSI RBAC standard would be to move the concept of the session to a "role-aware" authentication system.

4.5 Conclusion

In this chapter, we analyzed support for ANSI RBAC in COM+. Using set theory, we defined a configuration of the COM+ protection system in precise and unambiguous terms using set theory. Based on this definition, we formalized the semantics of authorization decisions in COM+. We analyzed support for various ANSI RBAC components in COM+, and illustrated our discussion with an example. Our result indicate that 40% of the ANSI RBAC functions are not supported due to the tight integration of the COM+ architecture with the underlying operating system, and the lack of the support for these functions in the operating system.

Chapter 5

Conclusions

In this chapter, we discuss the significance and contributions of the thesis, as well as a discussion of the results. We then state potential applications of our research, followed by a discussion on the limitations of our approach. We finally suggest directions for future research.

5.1 Contributions

This thesis provides analysis of support for the ANSI RBAC components and functional specification [ANS04] in three different commercial middleware technologies: CORBA [OMG02], EJB [DK06], and COM+ [Obe00]. The access control architecture of all these technologies is defined in different forms and formats. For example, CORBA is specified in the form of open application programming interfaces (APIs), whereas EJB is defined through APIs as well as the syntax and semantics of the accompanying eXtensible Markup Language (XML) files used for configuring an EJB container. COM+, on the other hand, is defined through an implementation of APIs as well as graphical user interfaces (GUI) for configuring the behavior of a COM+ server on Windows NT, 2000, 2003, XP, and Vista operating systems. Despite this disparity in the definitions for the security subsystem for the middleware technologies under study, we provide a unified approach for assessing implementations and support for ANSI RBAC for each middleware. Using this approach, we analyze the access control mechanisms of each middleware technology, and define a configuration of that middleware's protection system in a more precise and less ambiguous language than the corresponding middleware specification. Using these configurations, we suggest algorithms that formally specify the semantics of authorization decisions in each middleware technology.

The results of our analysis indicate that all three middleware technologies fall short of supporting even Core RBAC. Custom extensions are necessary in order for implementations compliant with CORBA Security to support ANSI RBAC required or optional components. EJB extensions

dependent on the operational environment are required in order to support ANSI RBAC required components. Other vendor-specific extensions are necessary in order to support ANSI RBAC optional components. Fundamental limitations exist, however, due to impracticality of some aspects in the ANSI RBAC standard itself. COM+ also falls short of supporting even Core RBAC. The main limitations exist due to the tight integration of the COM+ architecture with the underlying operating system. Other limitations exist due to impracticality of some aspects in the ANSI RBAC standard itself.

5.2 Discussion

Role-based access control (RBAC) has been studied for more than two decades. The ANSI RBAC standard, however, is relatively recent. There has also been growing interest in enterprise adoption of RBAC due to improved security, more efficient administration, and easier enforcement of business policies [Kam06]. In this thesis, we analyzed support for ANSI RBAC in different middleware technologies.

We started our analysis studying the access control architecture of CORBA. CORBA Security provides a very flexible framework for various different access control security policies, not just RBAC based policies. Secion 2.10, and Table 2.3 summarize our findings. Despite the flexibility of the CORBA Security architecture, it fell short of supporting even the functional model of Core ANSI RBAC. This lack of support was attributed in our analysis to the absence of support for user accounts and their management in CORBASec, and to the inability to enumerate DomainAccessPolicy objects and user sessions. These factors merely indicated a mismatch between CORBASec and ANSI RBAC, and were not conclusive in terms of whether the ANSI RBAC standard was sufficiently general.

Next we analyzed the access control architecture of a middleware that employs roles to make authorization decisions, namely EJB. The authorization architecture of EJB was analyzed as it is less general than that of CORBASec and only allows role-based access control. Section 3.5, and Table 3.5 summarize our findings. With EJB, the reasons for the lack of support for even Core ANSI RBAC were due to the lack of support in EJB specification for user accounts[7] and their

[7] Although various EJB server and component vendors provide support for mapping user accounts to security roles, the EJB specification does not address how support for user accounts should be accomplished.

management, user sessions, and role activation. The common factors between CORBASec and EJB that prevented support for ANSI RBAC, were related to user accounts, their management and the lack of support for sessions.

Based on these observations, we studied the access control architecture for a middleware that does provide support for user accounts and their management, namely COM+. Even though COM+ came the closest to supporting ANSI RBAC functions, it still failed to fully support Core ANSI RBAC, as indicated in Section 4.5 and Table 4.3. This failure was due to the lack of support for ANSI RBAC sessions and role activation.

To better understand this lack of full support for ANSI RBAC in all middleware technologies, it helps to categorize the functions of ANSI RBAC into static and dynamic functions. We can define the functions that do not change ANSI RBAC sets frequently during runtime of the system as static. Examples of static functions are *AddRole* and *AddUser*. On the other hand, the dynamic functions change ANSI RBAC sets frequently during runtime of the system. The dynamic functions are all those related to sessions and role activation, which are the unsupported functions in COM+, as indicated in Table 4.3; the static functions are the supported ones. It's apparent that the lack of support is always related to the dynamic functions.

When addressing role engineering in enterprise environments, [Kam06] indicated that *the challenge with the NIST/ANSI RBAC standard is that it is theoretical in nature and provides little guidance about how to design and implement a roles-based approach in organizations.* Our analysis also suggest that the dynamic functions of ANSI RBAC are theoretical in nature and do not take into account the dynamic aspects of real systems. We suggest that the sessions and role activation aspects of the ANSI RBAC standard should be mandated as part of a role-aware authentication system, with a defined interface to the authorization system.

The suggested role-aware authentication system component must have proper support for role activation and deactivation in order to adhere to the principle of least privilege [SS75]. On one extreme, an implementation of this component would allow one role to be active at a time. In this type of implementation [Fad99], once a role is activated, further requests to activate additional roles are done in a noncumulative fashion. That is, the activation of a new role, would automatically deactivate the previously activated role. This type of implementation does not provide proper role activation as in the absence of role hierarchy, for example, the user may need permissions assigned

to more than one role in order to invoke a certain operation. On another extreme, all roles of a user would be activated upon establishing a login session [AS01]. However, the proper implementation of role activation, should allow users to have one or more roles activated in a cumulative fashion.

The suggested role-aware authentication system may activate roles with, or without user intervention. Furthermore, the roles may be activated either upon login session establishment, or upon method invocation. In the former approach, a set of roles that are assigned to the user, or a default set of roles [FBK99] can be activated with or without user intervention. In the latter approach, roles are activated as necessary.

In summary, for practical purposes, we suggest that the dynamic functions of ANSI RBAC be mandated in a separate role-aware authentication system component, with a defined interface that can be used with the RBAC authorization system.

5.3 Applications

Our thesis findings and contributions provide more profound understanding of the ability of various commercial middleware technology to support the ANSI RBAC standard. It describes the limitations in both the middelware access control architecture under study, as well as the ANSI RBAC standard itself. The following are direct applications of our research.

- First, our research serves as a basis for implementing and assessing various implementations of middelware technologies from different vendors—where applicable—for support of ANSI RBAC.
- Second, the approach developed in our research can be used as a general tool to assess other technologies for their support for ANSI RBAC
- Third, our research serves as set of recommendations for improving the ANSI RBAC standard.

5.4 Limitations

Our formal approach using set theory [Ros06] provides a more precise and less ambiguous method to specify access control elements of various commercial middleware technologies. To help in the

analysis of ANSI RBAC support in different middleware access control architectures, the thesis also provides algorithms for translating an ANSI RBAC policy to the formalized protection system state of the middleware under study.

The specifications for each one of the middleware technologies under study is mostly written in natural language. A limitation of our formalization approach exists due to the fact that our translation from middleware specifications to models based on set theory is based on our understanding of the middleware specifications. This limitation can be overcome, however, by verifying our understanding of the access control specifications of the middleware technology against other researchers or domain experts.

Another limitation exists due to the fact that our formal model using set theory can only model the static aspects of the access control architecture under study. Dynamic aspects and interaction between various components of the access control architecture require a different modeling framework. However, this limitation does not affect the correctness and completeness of modeling the protection system state for the access control architecture under study.

5.5 Future Work

Our research provides ideas for possible directions for analysis-based and design-based research. Our work can be extended to further study support for ANSI RBAC in environments employing Web Services [HFBK03], or using an Enterprise Service Bus [Cha04] architecture. The ability of middleware technologies to support other access control models, such as UCON$_{ABC}$ [PS04], remains to be studied, as well. Furthermore, instead of using a pure theoretical approach to defining access control, such as the approach used in ANSI RBAC, studying the use of a more practical and software design-centric approach using UML [OMG07a] modeling, for example, or Design Patterns [GHJV95] can be pursued. Finally, the usability of various methods and approaches to role activation can also be investigated, including the recycling of role activation decisions, based on authorization recycling concepts [WCBR08].

Bibliography

[3GP07] 3GPP. *TS 29.198-01, Open Service Access (OSA); Application Programming Interface (API); Part 1: Overview*, v7.0.0 edition, March 2007.

[AC06] R. Adaikkalavan and S. Chakravarthy. Discovery-based role activations in role-based access control. *Performance, Computing, and Communications Conference, 2006. IPCCC 2006. 25th IEEE International*, pages 8 pp.–462, April 2006.

[Ahn00] Gail-Joon Ahn. Role-based access control in DCOM. *Journal of Systems Architecture*, 46(13):1175–1184, 2000.

[AL02] Carlisle Adams and Steve Lloyd. *Understanding PKI: Concepts, Standards, and Deployment Considerations*. Addison Wesley Professional, second edition, November 2002.

[ANS04] ANSI. ANSI INCITS 359-2004 for role based access control, 2004.

[AS01] Gail-Joon Ahn and Ravi Sandhu. Decentralized user group assignment in Windows NT. *The Journal of Systems and Software*, 56(1):39–49, 2001.

[Bar97] Larry S. Bartz. hyperDRIVE: leveraging LDAP to implement RBAC on the web. In *Proceedings of the Workshop on Role-based Access Control*, pages 69–74, New York, NY, 1997. ACM Press.

[BD99] Konstantin Beznosov and Yi Deng. A framework for implementing role-based access control using CORBA security service. In *Fourth ACM Workshop on Role-Based Access Control*, pages 19–30, Fairfax, Virginia, USA, 1999.

[BD07] Konstantin Beznosov and Wesam Darwish. Support for ANSI RBAC in CORBA. Technical Report LERSSE-TR-2007-01, accessible from http://lersse-

dl.ece.ubc.ca/search.py?recid=129, Laboratory for Education and Research in Secure Systems Engineering, University of British Columbia, July 27 2007.

[BK98] Nat Brown and Charlie Kindel. Distributed component object model protocol (DCOM/1.0). Technical Report draft-brown-dcom-v1-spec-03.txt, Microsoft Corporation, January 1998.

[BL75] D. E. Bell and L. J. LaPadula. Secure computer systems: Unified exposition and multics interpretation. Technical Report ESD-TR-75-306, MITRE, March 1975.

[BLL03] Ruth Baylis, Paul Lane, and Diana Lorentz. Oracle database administrator's guide, December 2003. 10g Release 1 (10.1).

[Box97] Don Box. *Essential COM*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997. Foreword By-Grady Booch and Foreword By-Charlie Kindel.

[BR02] David Basin and Frank Rittinger. A formal analysis of the CORBA security service. In *ZB 2002: Formal Specification and Development in Z and B, LNCS 2272*, pages 330–349. Springer, 2002.

[BS03] D.A. Buell and R. Sandhu. Identity management. *IEEE Internet Computing*, 7(6):26–28, Nov.-Dec. 2003.

[Bur06] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the Seventh Symposium on Operating System Design and Implementation*, pages 335–350, Seattle, WA, USA, November 6-8 2006.

[CDG+06] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *The 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI06)*, pages 205–218, Seattle, WA, USA, November 6-8 2006.

[Cha03] Thomas M. Chalfant. Role based access control and secure shell - a closer look at two Solaris™operating environment security features. Technical report, Sun BluePrints™OnLine, June 2003.

[Cha04] David A. Chappell. *Enterprise Service Bus*. O'Reilly Media, Inc., illustrated edition, June 2004.

[CO02] David W. Chadwick and Alexander Otenko. The PERMIS X.509 role based privilege management infrastructure. In *SACMAT '02: Proceedings of the Seventh ACM Symposium on Access Control Models and Technologies*, pages 135–140, New York, NY, USA, 2002. ACM Press.

[DK06] Linda DeMichiel and Michael Keith. JSR-220: Enterprise JavaBeans specification, version 3.0: EJB core contracts and requirements. Specification v.3.0 Final Release, Java Community Program, May 2006.

[DYK01] Linda G. DeMichiel, L. Ümit Yalçinalp, and Sanjeev Krishnan. *Enterprise JavaBeans Specification, Version 2.0*. Sun Microsystems, 2001.

[Edd99] Guy Eddon. *Inside COM+ Base Services*. Microsoft Programming Series. Microsoft Press, 1999.

[Fad99] Glenn Faden. RBAC in UNIX administration. In *RBAC '99: Proceedings of the fourth ACM workshop on Role-based access control*, pages 95–101, New York, NY, USA, 1999. ACM Press.

[FBK99] David F. Ferraiolo, John F. Barkley, and D. Richard Kuhn. A role-based access control model and reference implementation within a corporate intranet. *ACM Transactions on Information and System Security (TISSEC)*, 2(1):34–64, February 1999.

[FK92] D. Ferraiolo and R. Kuhn. Role-based access controls. In *Proceedings of the 15th NIST-NCSC National Computer Security Conference*, pages 554–563, Baltimore, MD, USA, 1992. National Institute of Standards and Technology/National Computer Security Center.

[FKS07] D. Ferraiolo, R. Kuhn, and R. Sandhu. RBAC standard rationale: Comments on "A critique of the ANSI standard on role-based access control". *Security & Privacy, IEEE*, 5(6):51–53, Nov.-Dec. 2007.

[FSG+01] David F. Ferraiolo, Ravi Sandhu, Serban Gavrila, D. Richard Kuhn, and Ramaswamy Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security*, 4(3):224–274, 2001.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Design*. Addison-Wesley Professional Computing Series. Addison-Wesley, Reading, 1995.

[Giu99] Luigi Giuri. Role-based access control on the Web using Java. In *Proceedings of the Fourth ACM Workshop on Role-based Access Control*, pages 11–18, New York, NY, USA, 1999. ACM Press.

[Got05] G. Goth. Identity management, access specs are rolling along. *IEEE Internet Computing*, 9(1):9–11, Jan.-Feb. 2005.

[Gro87] NetBIOS Working Group. Protocol standard for a NetBIOS service on a TCP/UDP transport: Concepts and methods, 1987.

[Gut01] Kurt Gutzmann. Access control and session management in the HTTP environment. *IEEE Internet Computing*, 5(1):26–35, 2001.

[Hen06] Michi Henning. The rise and fall of CORBA. *ACM Queue*, 4(5):28–34, June 2006.

[HFBK03] Bret Hartman, Donald J. Flinn, Konstantin Beznosov, and Shirley Kawamoto. *Mastering Web Services Security*. John Wiley & Sons, Inc., New York, 1st edition, 2003.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[IBM05] IBM. IBM informix dynamic server administrator's guide, December 2005. Informix Dynamic Server 10.0; Document ID: G251-2267-02.

[JBFT05] Bart Jacob, Michael Brown, Kentaro Fukui, and Nihar Trivedi. *Introduction to Grid Computing*. IBM Press, 2005.

[Kam06] Kevin Kampman. The business of roles. Technical report, Burton Group, February 2006.

[Kar00] Gunter Karjoth. Authorization in CORBA security. *Journal of Computer Security*, 8(2/3):89–108, 2000.

[Lam71] Butler W. Lampson. Protection. In *5th Princeton Conference on Information Sciences and Systems*, page 437, New York, NY, USA, 1971. ACM Press.

[LBB06] Ninghui Li, Ji-Won Byun, and Elisa Bertino. A critique of the ansi standard on role based access control. CERIAS and Department of Computer Science, March 3 2006.

[Low01] Juval Lowy. Windows xp: Make your components more robust with COM+ 1.5 innovations, August 2001.

[LW94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[Mic98] Microsoft. DCOM architecture, 1998.

[Mic05a] Microsoft. Microsoft interface definition language, 2005.

[Mic05b] Microsoft. Microsoft NTLM, 2005.

[Mic06a] Microsoft. Automating COM+ administration, 2006.

[Mic06b] Microsoft. Network management, 2006.

[Mic08] Microsoft. COM+ Administration Collections, 2008.

[MNSS87] S. P. Miller, B. C. Neuman, J. I. Schiller, and J. H. Saltzer. Kerberos authentication and authorization system. Technical report, Massachusetts Institute of Technology, 1987.

[Moc87] P. Mockapetris. Domain names - concepts and facilities, 1987.

[MyS07] MySQL AB. MySQL. http://www.mysql.com, 2007.

[NT94] B. Clifford Neuman and Theodore Ts'o. Kerberos: an authentication service for computer networks. *IEEE Communications Magazine*, 32(9):33–38, 1994.

[Obe00] Robert J. Oberg. *Understanding & programming COM+: a practical guide to Windows 2000 DNA*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[OF02] Rafael R. Obelheiro and Joni S. Fraga. Role-based access control for CORBA distributed object systems. In *Proceedings of the The IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*, page 53, Washington, DC, USA, 2002. IEEE.

[OMG99] OMG. The common object request broker: Architecture and specification. Specification formal/99-10-08, Object Management Group, 1999.

[OMG02] OMG. Common object services specification, security service specification v1.8, 2002.

[OMG04] OMG. Common object request broker architecture: Core specification v3.0.3, 2004.

[OMG06] OMG. CORBA Reflection v.1.0. OMG document# formal/06-05-03, May 2006.

[OMG07a] OMG. Unified Modeling Language: Infrastructure, v2.1.1. OMG document# formal/07-02-06, February 2007.

[OMG07b] OMG. Unified Modeling Language: Superstructure, v2.1.1. OMG document# formal/07-02-05, February 2007.

[OSM00] Sylvia Osborn, Ravi Sandhu, and Qamar Munawer. Configuring role-based access control to enforce mandatory and discretionary access control policies. *ACM Transactions on Information and System Security (TISSEC)*, 3(2):85–106, 2000.

[PM03] Andreas Pashalidis and Chris J. Mitchell. A taxonomy of single sign-on systems. In *Information Security and Privacy, 8th Australasian Conference, ACISP 2003, Wollongong, Australia*, volume 2727 of *Lecture Notes in Computer Science*, pages 249–264. Springer, July 9-11 2003.

[PP95] Tom Parker and Denis Pinkas. Sesame v4 - overview. Technical report, SESAME, December 1995.

[PS04] Jaehong Park and Ravi Sandhu. The UCONabc usage control model. *ACM Transactions on Information and System Security*, 7(1):128–174, 2004.

[PSA01] Joon S. Park, Ravi Sandhu, and Gail-Joon Ahn. Role-based access control on the web. *ACM Transactions on Information and System Security (TISSEC)*, 4(1):37–71, February 2001.

[Ros06] Kenneth H. Rosen. *Discrete Mathematics and its Applications*. McGraw-Hill, 6th edition, July 2006.

[RS98] C. Ramaswamy and R. Sandhu. Role-based access control features in commercial database management systems. In *Proc. 21st NIST-NCSC National Information Systems Security Conference*, pages 503–511, Arlington, VA, USA, 1998. National Institute of Standards and Technology/National Computer Security Center.

[RSB05] Ed Roman, Rima Patel Sriganesh, and Gerald Brose. *Mastering Enterprise JavaBeans*. Wiley Publishing, 10475 Crosspoint Boulevard, Indianapolis, IN 46256, USA, third edition, 2005.

[SA98] Ravi Sandhu and Gail-Joon Ahn. Decentralized group hierarchies in UNIX: An experiment and lessons learned. In *Proc. 21st NIST-NCSC National Information Systems Security Conference*, pages 486–502, Arlington, Virginia, USA, 1998. National Institute of Standards and Technology/National Computer Security Center.

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

[SCH+04] Carla Sadtler, Lee Clifford, Jeff Heyward, Arihiro Iwamoto, Noelle Jakusz, Lars Bek Laursen, WonYoung Lee, Isabelle Mauny, Shafkat Rabbi, and Ascension Sanchez. *IBM WebSphere Application Server V5.1 System Management and Configuration WebSphere Handbook Series*. IBM International Technical Support Organizat, October 2004.

[SFK00] R. Sandhu, D. Ferraiolo, and R. Kuhn. The NIST model for role-based access control: Towards a unified standard. In *Workshop on Role-Based Access Control*, Berlin, 2000. ACM.

[SGM02] Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Professional, 2002.

[Sie00] Jon Siegel. *CORBA 3 Fundamentals and Programming*. John Wiley & Sons, 2000.

[Som06] Ian Sommerville. *Software Engineering*. Addison Wesley, 8th edition, June 2006.

[SS75] J.H. Saltzer and M.D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(6):1278–1308, 1975.

[SS96] Richard Mark Soley and Christopher M. Stone. *Object Management Architecture Guide*. John Wiley & Sons, 492 Old Connecticut Path, Framingham, MA 01701 USA, 3 edition, 1996.

[Sun00] Sun Microsystems Inc. RBAC in the Solaris^{TM} operating environment. http://www.sun.com/software/whitepapers/wp-rbac/wp-rbac.pdf, 2000. White Paper.

[Sun01] Sun. Java authentication and authorization service (JAAS). http://java.sun.com/products/jaas/, 2001.

[Sun07] Sun. Remote method invocation, 2007.

[Syb05] Sybase Inc. System administration guide: Volume 1 – Adaptive Server Enterprise 15.0, October 2005. Document ID: DC31654-01-1500-02.

[TM06] Samantha Tran and Manoj Mohan. Security information management challenges and solutions. http://www.ibm.com/developerworks/db2/library/techarticle/dm-0607tran/index.html, 2006.

[TOG97] TOG. *DCE 1.1: Remote Procedure Call*. The Open Group, catalog number c706 edition, August 1997.

[TS01] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[WCBR08] Qiang Wei, Jason Crampton, Konstantin Beznosov, and Matei Ripeanu. Authorization recycling in RBAC systems. In *Proceedings of the thirteenth ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 63–72, Estes Park, Colorado, USA, June 11–13 2008. ACM.

[WdSFW+02] C. M. Westphall, Joni da Silva Fraga, M. S. Wangham, R. R. Obelheiro, and Lau Cheuk Lung. PoliCap—proposal, development and evaluation of a policy service and capabilities for CORBA Security. In *SEC '02: Proceedings of the IFIP TC11 17th International Conference on Information Security*, pages 263–274, Deventer, The Netherlands, The Netherlands, 2002. Kluwer, B.V.

[WF99] C. Westphall and J. Fraga. A large-scale system authorization scheme proposal integrating Java, CORBA and web security models and a discretionary prototype. In *Latin American Network Operations and Management Symposium*, pages 14–25, Rio de Janeiro, Brazil, December 1999. IEEE Press.

[WHK97] M. Wahl, T. Howes, and S. Kille. RFC 2251: Lightweight directory access protocol (v3), 1997.

[YD96] Zhonghua Yang and Keith Duddy. CORBA: a platform for distributed object computing. *SIGOPS Oper. Syst. Rev.*, 30(2):4–31, 1996.

[ZHS+04] BY Zhao, L. Huang, J. Stribling, SC Rhea, AD Joseph, and JD Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on selected areas in communications*, 22(1):41–53, 2004.

[ZM04] Wei Zhou and Christoph Meinel. Implement role based access control with attribute certificates. In *The 6th International Conference on Advanced Communication Technology (ICACT2004)*, volume 1, pages 536–541, Korea, Feb 2004. National Computerization Agency, Electronics and Telecommunications Research Institute, Korea.