SESSION-AWARE RBAC ADMINISTRATION, DELEGATION, AND ENFORCEMENT WITH XACML

by

Min Xu
A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
In Partial fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Information Technology

Committee:

________________________ Duminda Wijesekera, Dissertation Director

________________________ Ravi S. Sandhu, Committee Member

________________________ Daniel A. Menascé, Committee Member

________________________ Songqing Chen, Committee Member

________________________ Angelos Stavrou, Committee Member

________________________ Daniel A. Menascé, Senior Associate Dean

________________________ Lloyd J. Griffiths, Dean, The Volgenau School of Information Technology and Engineering

Date: ________________________ Spring Semester 2010
George Mason University
Fairfax, VA

Session-aware RBAC administration, delegation, and enforcement with XACML

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

By

Min Xu
Master of Science
University of Nevada, Las Vegas, 2002
Bachelor of Engineering
Huazhong University of Science and Technology, 2000

Director: Duminda Wijesekera, Associate Professor
Department of Computer Science

Spring Semester 2010
George Mason University
Fairfax, VA

Copyright © 2010 by Min Xu
All Rights Reserved

Dedication

To my parents, who have always inspired and encouraged me to continue my lifelong dream.

My special dedication goes to my lovely wife, Haiyin Hua, who has been always supporting me and sharing all the difficult times with great love and sacrifices.

To my lovely daughter, Jennifer, who is the constant sources of my joy and pride in my life.

Acknowledgments

I would like to express my sincere appreciation and gratitude to my dissertation director, Professor Duminda Wijesekera, who has enlightened and guided me throughout my doctoral studies. Great thanks to Dr. Wijesekera who made this work possible, and encouraged me during my difficult times.

Special appreciation and thanks to Professor Ravi Sandhu, who has brought me into the Information Security and Assurance field, advised and supported me during my earlier years of doctoral study and continuing serve on my committee even after leaving George Mason University.

I am also grateful to my dissertation committee members, Professor Daniel A. Menascé Professor Songqing Chen, and Professor Angelos Starvrou, for their valuable comments and suggestions.

I thank the administrative staff at GMU for their support. Special thanks go to Lisa Nolder who helped me go through some administrative obstacles.

My appreciation also goes to many friends at George Mason University for their help and to my co-authors for their collaboration. Special thanks go to Dr. Xinwen Zhang who I have collaborated on three paper. His comments and suggestions are very constructive, and his English writing is really good.

Table of Contents

Page

List of Tables vii
List of Figures viii
Abstract ix
1 Introduction 1
  1.1 RBAC and XACML 1
  1.2 Problem Statement 2
  1.3 Summary of Contributions 5
  1.4 Organization of the Dissertation 6
2 Background 7
  2.1 XACML 7
    2.1.1 Syntax 7
    2.1.2 Sun's Reference Implementation 11
  2.2 RBAC and ARBAC 15
  2.3 XACML v3.0 Administration and Delegation Profile 15
    2.3.1 An Example 16
  2.4 Role-based Delegation 20
3 Session Administrative Model 23
  3.1 RBAC Session Administration 23
    3.1.1 Formal Specification of Administrative Operations 28
  3.2 Concurrency Control 31
    3.2.1 Entities Affected by Enforcing Administrative Operations 33
  3.3 Related Work 35
  3.4 Summary 38
4 XACML-ARBAC Profile and the Enforcement Architecture 39
  4.1 The XACML-RBAC Profile 39
  4.2 The XACML-ARBAC Profile 40
    4.2.1 Administrative Operation 51

4.3 Enforcing The XACML-ARBAC Profile 52
4.3.1 Concurrency Control 53
4.3.2 The Lock Manager 56
4.4 The Birth and Death Processes 58
4.5 Related Work 60
4.6 Summary 61
5 XACML-ADRBAC Profile and Its Enforcement Architecture 63
5.1 Introduction 63
5.2 Administration and Delegation Model 65
5.3 The XACML-ADRBAC Profile 68
5.3.1 An Example 73
5.3.2 Role Reduction 79
5.4 Enforcement Architecture 80
5.4.1 Concurrency Control 81
5.5 Related Work 83
5.6 Summary 85
6 Prototype Implementation and Evaluation 87
6.1 Implementation 87
6.1.1 Implementing the Birth and Death Process 87
6.1.2 Implementing Condition Functions and Administrative Operations . 88
6.1.3 Implementing the Lock Manager 89
6.2 Performance Evaluation 91
6.2.1 Simple Administrative Operations 92
6.2.2 Complex Administrative Operations 93
6.3 Related Work 98
6.4 Summary 99
7 Conclusion and Future Work 100
7.1 Conclusion 100
7.2 Future Work 101
Bibliography 104

List of Tables

Table Page

3.1 Administrative operations 27

4.1 Extended functions applied in <Condition> in XACML-ARBAC profile . . 42

5.1 Delegation operations 68

5.2 Extended functions applied in <Condition> in XACML-ADRBAC profile . 71

5.3 Contrasting elements in XACML-ARBAC with XACML-ADRBAC 72

5.4 Comparison between XACML-ARBAC, XACML-Admin, and XACML-ADRBAC services 80

6.1 Accessor and mutator methods used in the `PolicyManager` 89

6.2 Execution time for RevokePermission (msecs). 94

List of Figures

Figure	Page
2.1 XACML data flow diagram.	14
2.2 PBDM1 model	21
3.1 An example role hierarchy.	34
3.2 Compute entities affected due to an administrative action.	34
4.1 Extended XACML architecture for XACML-ARBAC enforcement.	53
4.2 PDP evaluation algorithm.	55
4.3 Enforcing administrative operations.	56
5.1 Compute affected entities of a delegation operation.	69
5.2 Extended XACML architecture for XACML-ADRBAC enforcement.	81
5.3 Enforcing delegation operations.	84
6.1 Total time taken to execute `AssignUser`.	93
6.2 Total time taken to execute `RevokePermission`.	94
6.3 Effect of # edges on the time to remove a role.	96
6.4 Effect of # users on the time to remove a role.	96
6.5 Effect of # sessions on the time to remove a role.	97

Abstract

SESSION-AWARE RBAC ADMINISTRATION, DELEGATION, AND ENFORCEMENT WITH XACML

Min Xu, PhD

George Mason University, 2010

Dissertation Director: Duminda Wijesekera

An administrative role-based access control (ARBAC) model specifies administrative policies over a role-based access control (RBAC) system, where an administrative permission has the capability to modify an RBAC policy by updating permissions assigned to roles, or assigning/revoking users to/from roles. Enforcing ARBAC policies over an active access controller while some users are using protected resources may result in conflicts: a policy may be in effect in the RBAC system while being modified by an administrative operation. Towards solving this concurrency problem, this dissertation proposes a session-aware administrative model for RBAC to manage the interactions and potential conflicts between access control evaluation and the administrative operations. Based on this model, this dissertation specifies the concurrency requirements of an ARBAC model: (1) revoke an activated role or delete an active session immediately, and (2) delay administrative operations. This dissertation introduces the concept of lock scope for a role. This captures the affected roles when the permissions granted to this role are modified due to administrative operations.

Consider that eXtensible Access Control Markup Language (XACML) is the *de facto*

language to specify access control policies for Web Services; this dissertation proposes the XACML profile for administrative RBAC (XACML-ARBAC) which is the extension of the XACML-RBAC profile with the proposed session-aware administrative model. One of the advantages of doing so is to use XACML policies to administrate XACML-RBAC policies. The XACML policy evaluation runtime is enhanced by introducing a locking manager and a special administrative policy enforcement point (A-PEP). The lock manager handles concurrency control issues that arise when enforcing the XACML-ARBAC profile. The A-PEP competes read-write locks for RBAC and ARBAC policies in conjunction with the evaluation engine of the access controller.

Along with the administrative model, the fine-grained and flexible permission delegation capability of the RBAC system has obtained considerable adoption in the last decade. The OASIS technical committee published the XACML v3.0 administration and delegation profile (XACML-Admin) working draft on April 16, 2009 in order to provide policy administration and dynamic delegation services to the XACML runtime. To capture the concurrency control requirements for delegation, this dissertation further proposes that the XACML-ARBAC profile is augmented with role-based delegation, named role-based administration and delegation XACML profile (XACML-ADRBAC). The XACML-ADRBAC profile has two novel properties: scalability–it facilitates delegated permissions to a large number of users with the same permission assignment, and flexibility–it allows a delegator to delegate any subsets of permissions assigned to him/her and modify the delegated permission whenever required. Correspondingly, the proposed XACML-ARBAC enforcement mechanism is also enhanced to enforce the XACML-ADRBAC. To the author's best knowledge, this proposal is the first method to enforce the XACML-Admin profile proposed by OASIS.

To demonstrate the feasibility and performance of the framework, a prototype is implemented to enforce the XACML-ARBAC profile by augmenting Sun Microsystems's

XACML reference implementation. Experimental studies show that the system has reconcilable performance characteristics.

Chapter 1: Introduction

Web Services are emerging as a strong candidate for service oriented architecture (SOA) to build distributed application on the Web as basic building blocks. As Web Services are loosely coupled applications, the security administration of a single application involves potentially many distributed policy decision and enforcement points. Although many security protocols and languages have been proposed and deployed in existing Web Services infrastructures, efficient and secure policy management is still an open issue.

1.1 RBAC and XACML

Role Based Access Control (RBAC) has received considerable adoption as an applicable access control model during the last decade [34, 64]. RBAC is based on the basic principle that every *role* is granted a proper set of permissions necessary and sufficient to perform the *job functions* of any individual performing in that *role*. The RBAC research community has extended RBAC models to use RBAC itself to administrate the RBAC systems, commonly referred to as *administrative role-based access control (ARBAC)* models [27, 29, 30, 53, 63, 67]. ARBAC models specify the access permissions required to perform the job function of access control administrators such as updating role hierarchy, modifying permissions granted to roles, and assigning/revoking users to/from roles. These permissions are associated with *administrative* roles in ARBAC models.

Delegation of authority is another important business rule related to access control policies. Delegation would take places at (1) back up role, (2) decentralization of authority, and

(3) collaboration. Along with administrative model, the fine-grained and flexible permission delegation capability of RBAC system has obtained considerable attention and usage during the last decade [18, 19, 79, 80]. The delegators are assigned to a *delegable* role granted with a set of delegation permissions. The delegation permissions are different from the permissions to access resources. The delegation permissions are semi-administrative in nature such as creating/deleting a *delegated* role (e.g., a role that may be assigned to a delegatee), granting/revoking permission to/from a delegated role, and assigning/removing a user to/from a delegated role.

In a parallel, the eXtensible Access Control Markup Language (XACML) [2] is the *de facto* standard to specify access control policies for Web Services. Many policies that conform to traditional access control models such as discretionary [40], mandatory [21], and role-based (RBAC) have been specified in XACML syntax over the years. Recognizing that RBAC models are gaining popularity, the XACML technical committee has published an RBAC profile to the original XACML specification [1], which this dissertation refers to as the XACML-RBAC profile.

The OASIS technical committee published the XACML v3.0 administration and delegation profile (XACML-Admin) working draft on April 16, 2009 [6] to support two use cases: (1) policy administration and (2) dynamic delegation. The former controls the types of policies that individuals can create and modify, whereas the latter permits some users to create policies of limited duration in order to delegate selected capabilities to others. The administration and delegation in the XACML-Admin are not role-based.

1.2 Problem Statement

The XACML-RBAC profile 2.0 has been approved as an OASIS standard [1] to specify core and hierarchical components of RBAC models. However there is no corresponding

XACML profile for ARBAC models. ARBAC models use RBAC itself to administrate the RBAC models. Following this path, a demonstrated need exists to develop an XACML profile for ARBAC models to manage the XACML-RBAC profile.

RBAC [34] models include the concept of a session, which provides a context for a user to have multiple simultaneous interactions with resources. Therefore, a user may activate different roles within different sessions concurrently. Consequently, every activated role belongs to one session, and a session could have multiple roles activated, where each session belongs to a unique user. Some primitive session management functions are specified in the NIST RBAC model [34]. However, they are not included in existing ARBAC modes [27, 29, 30, 53, 63, 67]. What is needed is a session administrative model for RBAC systems to manage the interactions and potential conflicts between the access control evaluation and the administrative operations.

ARBAC models grant *administrative* roles with administrative permissions: such as updating role hierarchy, modifying permissions granted to roles, and assigning/revoking users to/from roles. One of the main challenges in enforcing these *administrative* permissions is that it requires changing access control policies–in case of XACML–those are compliant with XACML-RBAC profiles. This raises two issues. First, when an administrator exercises any administrative privilege (e.g. those permissions given under administrative roles) granted under an XACML-ARBAC policy, it could result in altering the permissions of a user. For example a user may lose an already granted privilege by a XACML-RBAC policy. Therefore, enforcing an ARBAC policy could entail immediately changing the permissions granted to a user for a resource while the same user may be accessing the resource. Second, an administrative operation usually updates an RBAC policy, which results in read-write conflicts when the access controller attempts to evaluate a user's access request. The underlying reason for these problems lies in the fact that all existing ARBAC models focus

on defining policies to assign different administrative permissions to different administrative roles, while in practice, enforcing these policies affects the runtime state of the RBAC system which may result in an unexpected change of permissions within ongoing sessions. The concurrency issues of XACML administration must be addressed.

Another auxiliary issue that has not yet been adequately addressed previously is the birth and death processes of the access controller itself, known as the XACML policy evaluation runtime in this dissertation. When the access controller is initialized, there is no default role or mechanism to properly activate the stored policies. When the access controller is directed to end, there must be a mechanism to *clean up* the system in order to ensure the safety property of the access controller.

The delegation model used in the XACML-Admin profile [6] is a discretionary access control (DAC) model. This XACML-Admin profile only allows the owner of a permission to delegate it to a specific user, which is not scalable when permissions are required to be delegated to a large number of users with the same job function. In cases in which the delegator is not available, or is unable to perform the delegation, it is more convenient to have a third party, such as the administrator, initiates the delegation on behalf of the user. This profile can not keep pace with scalability requirement of a large distributed networks. This is called the *automation* principle in [62]. This profile also lacks the support to allow delegators to delegate any subset of permissions assigned to him/her. This profile does not have an enforcement mechanism. Enforcing administrative or delegation operations will update relevant policies which results in read-write conflicts while the access controller attempts to evaluate a user's access request. When an administrator or delegator attempts to revoke a permission granted to a user, the same user could be exercising the permission to access a resource, which violates system the safety property. A more scalable and flexible delegation profile is needed along with a corresponding enforcement mechanism.

1.3 Summary of Contributions

This dissertation contains the following contributions.

- A session-aware framework is developed to enforce ARBAC policy with XACML. Within this framework:
  - a session-aware administrative model for RBAC is used to manage interactions and potential conflicts between access control evaluation and administrative operations; and
  - two concurrency requirements are specified for the session administrative model and the ARBAC model for an RBAC system; and
  - an XACML-ARBAC profile is developed to specify ARBAC policies; and
  - an XACML enforcement architecture is extended by introducing an administrative policy enforcement point (A-PEP) and a *Lock Manager*, the former competes for read-write locks for RBAC and ARBAC policies along with PDP, and the latter handles concurrency control issues arising in enforcing the XACML-ARBAC profile; and
  - a formal specification of the birth and death process of the access controller and a default XACML-ARBAC policy which boots up the access controller.
- A prototype to enforce the extended XACML profile for ARBAC is implemented and the following performance evaluation results are presented:
  - Simple administration operations execute very fast because they do not affect the activities of the users; and
  - The complex operations take more time because they require executing a series of administrative operations; and

    - The overall effect to the access controller due to administrative operations is minimal and acceptable.

  These results were initially presented in [76, 77].

- A role-based XACML administration and delegation profile and its enforcement architecture are developed:
    - By adding role-based delegation to the XACML-ARBAC profile, the scalability and flexibility of the delegation mechanism is improved;
    - The enforcement architecture for XACML-ARBAC is enhanced to enforce the role-based administration and delegation XACML profile (XACML-ADRBAC).

These results were initially presented in [75].

1.4 Organization of the Dissertation

Chapter 2 first briefly describes XACML essentials, then describes the preliminary information about RBAC and ARBAC, and the role-based delegation models. Chapter 3 develops a session-administrative model to manage the interactions and potential conflicts between access control evaluation and administrative operations, and specifies the concurrency control requirements. Chapter4 presents the XACML-ARBAC profile and the architecture to enforce this profile. Chapter 5 proposes a role-based XACML administration and delegation profile and its enforcement architecture. Chapter 6 describes the prototype by extending Sun's reference implementation and presents some performance characteristics. Chapter 7 summarizes this dissertation and presents directions for future work.

Chapter 2: Background

This chapter presents background knowledge and work relevant to this dissertation. XACML syntax and data flow diagram are introduced, followed by RBAC and ARBAC essentials. Delegation models are introduced at the end of this chapter.

2.1 XACML

2.1.1 Syntax

The eXtensible Access Control Markup Language (XACML) is an XML-based language which specifies access control policies, requests, and responses in distributed computing environments such as Web Services. A request originates from a <Subject> (e.g., a user or a process) to perform an <Action> (e.g., read or write) on a <Resource> (e.g., a file or a disk block) within an environment (e.g., from a secure machine). XACML is designed to be extendable to define new functions, data types, access control rules, and policy combining algorithms for applications or systems.

Standard XACML uses three basic elements in constructing access control policies: <Rule>, <Policy>, and <PolicySet>, and allows hierarchical nesting of them. A XACML <Rule> has two elements, a <Condition> and a <Target>, and an *Effect* attribute. The intuitive reading of a XACML rule is that, if the condition of the rule evaluates to be *true*, then the access control decision to perform <Actions> by the <Subjects> on the <Resources> are given by the *Effect* attribute. A <Policy> can consist of a set of <Rule>s. A <PolicySet> holds <Policy>s and other <PolicySet>s. The XACML policy evaluation algorithm uses

the so called “rule and policy combining algorithms” [2] to recursively compute the decision of a nested rule/policy. The return value of such an evaluation must be one of {*permit*, *deny*, *nonApplicable*, *indeterminate*}. The OASIS specification [2] identifies four standard combining algorithms: *deny-override*, *permit-override*, *first-applicable*, and *only-one-applicable*. For a *deny-override* policy (or policy set), a deny is returned if any rule (or policy) evaluation returns deny; permit is returned if all rule (or policy) evaluations return permit. For a *permit-override* policy (or policy set), permit is returned if any rule (or policy) evaluation returns permit; deny is returned if all rule (or policy) evaluations return deny. For a *first-applicable* policy (or policy set), the decision of the first applicable rule (or policy) is returned. For an *only-one-applicable* policy (or policy set), the decision of the only applicable rule (or policy) is returned; *indeterminate* indicates an error and is returned if there are more than one applicable rule (or policy). For all of these combining algorithms, *notApplicable* is returned if no rule (or policy) is applicable.

<Target> specifies a set of predicates which are constructed from <Subject>, <Resource>, and <Action> attributes that must be met for a <PolicySet>, <Policy>, or <Rule> to apply to an access request. The attribute values in a request are compared with those included in the <Target>, and if all the attributes match then the request is *applicable*. If the request and the <Target> attributes do not match, then the request is *notApplicable*, and if the evaluation results in an error, then the request is *indeterminate*. If a request satisfies the <Target> of a policy, then the request is further checked against the rule set of the policy; otherwise, the policy is skipped without further examination.

The <Condition> element further restricts the applicability of the <Rule> already matching by the <Target> in the rule. <Condition>s can be nested using boolean combinators over other <Condition>s.

Any <PolicySet> can include one or more <PolicyIdReference> or <PolicySetIdReference> elements which are pointers to the referenced <Policy>s

or <PolicySet>s. The intended semantics of including a <PolicySetIdReference> in a <PolicySet> is that the content of the referenced <PolicySet> replaces the <PolicySetIdReference> verbatim in the referring <PolicySet>. This feature is used in the XACML-RBAC profile [1] to specify role-to-permission assignments and role hierarchies.

```
1  <Policy PolicyId="add:a:role"
2    RuleCombiningAlgId="permit-overrides">
3   <Target>
4     <Subjects><AnySubject/></Subjects>
5     <Resources> <AnyResource/></Resources>
6     <Actions><AnyAction/></Actions>
7   </Target>
8   <Rule RuleId="Permission:to:add:a:role" Effect="Permit">
9     <Target>
10      <Subjects><AnySubject/></Subjects>
11     <Resources>
12      <Resource>
13       <ResourceMatch MatchId="string-equal">
14       <AttributeValue DataType="string">role
15         </AttributeValue>
16       <ResourceAttributeDesignator
17         AttributeId="resource-id" DataType="string"/>
18       </ResourceMatch>
19      </Resource>
20     </Resources>
21     <Actions>
22       <Action>
```

```
23          <ActionMatch MatchId="string-equal">
24             <AttributeValue DataType="string">AddRole
25             </AttributeValue>
26           <ActionAttributeDesignator AttributeId="action-id"
27           DataType="string"/>
28           </ActionMatch>
29         </Action>
30       </Actions>
31      </Target>
32      <Condition FunctionId="not">
33        <Apply FunctionId="role-exist">
34        <ResourceAttributeDesignator AttributeId="new-role-id"
35          DataType="role"/>
36       </Apply>
37      </Condition>
38    </Rule>
39    <Rule RuleID="2" Effect="Deny"> </Rule>
40</Policy>
```

Policy 1: An XACML example policy.

Policy1 is an example XACML policy that specifies a permission to add a role. This policy has one <Policy> element containing two rules, *Rule "Permission:to:add:a:role"* (lines 8-38), and *Rule2* (line 39). Line 2 of the policy indicates that the rule combining algorithm to be used is *permit-overrides*. Lines 3-7 define the policy's target, which indicates that this policy is applicable to any subject requesting permission to execute any action on any resource. The target of *Rule "Permission:to:add:a:role"* (lines 9-31) narrows the scope of applicable requests to those requesting accesses to the resource *role* with

the action *AddRole*. The condition of *Rule "Permission:to:add:a:role"* (lines 32-37) indicates that if the role does not exist (computed using our extended function *role-exist* (to be explained shortly)), the request should be permitted. Otherwise, according to *Rule2* (line 39) and the "rule combining algorithm" of the policy (line 2), the request should be denied.

2.1.2 Sun's Reference Implementation

Sun Microsystems's XACML reference implementation [8], more commonly referred to as Sun's reference implementation, is one of the first implementations of the XACML evaluation engines and it is by far the most widely used XACML evaluation engine both in commercial applications and in research projects. Figure 2.1 shows the high-level architecture of Sun's reference implementation with the following main components:

Policy Administration Point (PAP) is the entity that creates policies and policy sets.

Policy Decision Point (PDP) is the entity that evaluates policies and renders one of *permit, deny, indeterminate, notApplicable* as the authorization decision.

Policy Enforcement Point (PEP) is the entity that enforces the access control decision.

The Context Handler is the entity that converts native request to one that is in the XACML canonical format (to be explained shortly) and converts authorization decisions in the XACML canonical format to native formats.

The canonical form is called the XACML "Context". The XACML context is defined in XML schema [11], describing a canonical representation for the inputs and outputs of the PDP. Policy 2 shows a canonical representation of a request which contains attribute of `Subject`(lines 2-6), `Resource` (lines 7-19), and `Action` (lines 11-13).

```
1 <Request>
```

```
2  <Subject SubjectCategory="urn:oasis:names:tc:xacml:1.0:subject-
     category:access-subject">
3    <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0
       :subject:subject-role-id"
4    DataType="role"><AttributeValue>SSO</AttributeValue></
       Attribute>
5    <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0
       :subject:subject-role-id"
6    DataType="role"><AttributeValue>SeniorManager</AttributeValue
       ></Attribute>
7    <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0
       :subject:subject-id"
8    DataType="http://www.w3.org/2001/XMLSchema#string"><
       AttributeValue>Alice</AttributeValue></Attribute>
9  </Subject>
10 <Resource>
11   <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0
       :resource:resource-id"
12   DataType="http://www.w3.org/2001/XMLSchema#string"><
       AttributeValue>role</AttributeValue></Attribute>
13   <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0
       :resource:new-role-id"
14   DataType="role"><AttributeValue>Manager</AttributeValue></
       Attribute>
15 </Resource>
16 <Action>
```

```
17      <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0
           :action:action-id"
18      DataType="http://www.w3.org/2001/XMLSchema#string"><
           AttributeValue>add</AttributeValue></Attribute>
19    </Action>
20</Request>
```

Policy 2: Alice, who has been assigned to the SSO role, requests to add a Manager role

Policy 2 is an XACML request context which comprises a <Request>(line 1) element, which includes <Subject>(lines 2-6), <Resource>(lines 7-10), <Action>(lines 11-13) and <Environment> (which is optional) element. An XACML response context comprises a <Response> element, which includes one or more <Result> elements. A <Result> comprises a <Decision> element, some status information, and (optionally) one or more <Obligations>. The <Decision> element contains one of the four possible results of policy evaluation: *permit*, *deny*, *notApplicable* (no policies or rules are applicable to the access request) or *indeterminate* (unforseen errors occurred during evaluation). The status information could be the reasons for the evaluation failure. The <Obligations> element includes processing directives to be performed by the PEP (in addition to enforcing the PDP's decision).

The PAP creates policies at authoring time, e.g., by security administrators using an acceptable text editor. At an access control request time, a subject sends an access request to the PEP as shown in flow 2 of Figure 2.1. The PEP then forwards this request to the context handler (flow 3) and obtains all the values of the attributes passed in the request. The context handler forms the access control request based on the attributes of the requester, action, resource, and environment, and forwards the request to the PDP (flows 4, 5, 6, 7, 8). PDP uses this information to find the access control policy applicable to the request, which

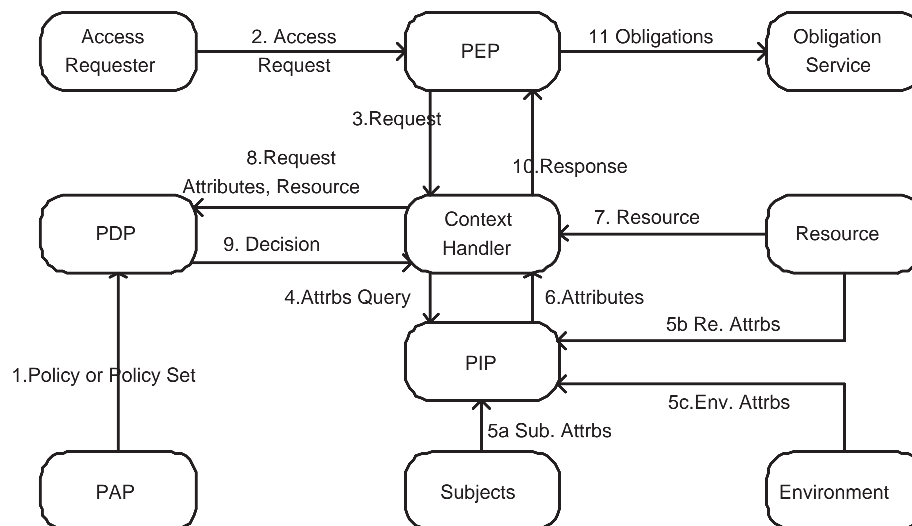

Figure 2.1: XACML data flow diagram.

is defined in terms of the attributes of the requestor, action, and the resource. The policy can also include functions defined on these attributes. The PDP uses two steps to evaluate the request: it first attempts to find all the policies applicable to the request by using the target matching (flow 1) algorithm, and then it evaluates the rules of the applicable policies and returns its decision back to the PEP via the context handler (flows 9, 10). Finally, the PEP enforces the authorization decision.

Sun's reference implementation provides a set of APIs that understand the XACML syntax, and rules to process requests and manage attributes. This implementation only provides a PDP for policy evaluation that can read, but not modify, any policies, which needs to be enhanced in order to enforce the administrative operations specified in ARBAC policies.

2.2 RBAC and ARBAC

This dissertation uses the notation $RBAC = (U, O, A, R, P, \leq, U2R, R2P)$ to model an RBAC system, where the first four entities are the sets of users, objects, actions, and roles, respectively. P is a subset of $O \times A$, representing the set of permissions. The partial ordering $\leq \subseteq R \times R$ is the role hierarchy. $U2R : U \mapsto 2^R$ and $R2P : R \mapsto 2^P$ are relations that are functional in their first coordinate, modeling user-to-role and role-to-permission assignments. That is, $U2R(u, M)$ and $R2P(r, N)$ are true iff user u is allowed to play the set of roles M and role r can execute the permission set N respectively. This dissertation uses function $assignPerms(u) = \cup_{r \in U2R(u), r \geq r_1} R2P(r_1)$ to return the set of all possible permissions that a given user can obtain by invoking all roles assigned to him or her.

This dissertation bases the work partially on ARBAC97 [63] and SARBAC [29], which suggest having a set of *administrative roles* (AR) distinct from *user roles*, and permit these administrative roles to create and remove users, roles, assign and revoke users to (user) roles, and grant and revoke permissions to (user and administrative) roles. ARBAC97 has three sub-models referred as URA97, PRA97, and RRA97, which represent controls over user-to-role assignment (U2R), role-to-permission assignment (R2P), and the role hierarchy ($\leq$), respectively.

2.3 XACML v3.0 Administration and Delegation Profile

The OASIS technical committee published an XACML v3.0 administration and delegation profile (XACML-Admin) working draft on April 16, 2009 [6] to provide policy administration and dynamic delegation. The former controls the types of policies that individuals can create and modify, and the latter permits some users to create policies to delegate his/her permissions to another user.

The delegation model proposed in the XACML-Admin profile specifies the permissions

to create policies and methods to account for created policies against these permissions by attributing their lineage to a delegation chain. In order to do so, the proposed XACML-Admin profile adds a new key word <PolicyIssuer> element that identifies the source of the policy, where a missing <PolicyIssuer> element implies that the policy is *trusted*. A trusted policy is considered valid and its origin is assumed not to require verification by PDP. Policies that have an issuer must have their authorities accounted for by using a delegation chain. If the authority of the policy issuer can be traced back to a trusted policy, the policy is used by the PDP; otherwise, it is discarded. The authority of the issuer depends on the context of the access request; therefore, a policy can be both valid and invalid depending on the request context. Two access situations exist: current attributes mode and historic attributes mode. In current attributes mode, when a delegate attribute is dynamic, the attribute value used is at the access request time. In the historic attribute mode, when a delegate attribute is dynamic, the attribute value at the policy creation time must be used. Steps in the validation process are performed using a special XACML request, called the *administrative request*, [1] which contains information about the policy issuers and the access mode.

2.3.1 An Example

Consider the example where Mary is the manager of a company and approves expense reports for her department. When she is on vacation, Jack approves expense reports. The example in Policy 3 is used to express the policy using the XACML-Admin profile. As shown in Policy 3, this policy set contains two policies. PolicyA is a trusted policy because it has no issuer. PolicyA allows Mary to create any policy which allows all requests allowed by PolicyA; that is, Mary can give “approve expense reports” permission to Jack. PolicyB

[1] The policy administration request in [6] is different from the administrative request in [77] which requests an administrative operation to change the configuration of an RBAC system.

is issued by Mary as indicated by the <PolicyIssuer> element. There are no delegated categories and thus it is an access policy that grants “approve expense reports” permission to Jack.

```
1 <PolicySet PolicySetId="PolicySet1"
2  PolicyCombiningAlgId="policy-combining-algorithm:permit-
      overrides">
3   <Policy PolicyId="PolicyA"
4   RuleCombiningAlgId="rule-combining-algorithm:permit-overrides">
5     <Target>
6       <Match MatchId="function:string-equal">
7         <AttributeValue
8           DataType="http://www.w3.org/2001/XMLSchema#string">
9           Jack</AttributeValue>
10        <AttributeDesignator Category="attribute-
            category:delegated:subject-category:access-subject "
11          DataType="http://www.w3.org/2001/XMLSchema#string"/>
12      </Match>
13      <Match MatchId="function:string-equal">
14        <AttributeValue
15          DataType="http://www.w3.org/2001/XMLSchema#string">
              expense reports</AttributeValue>
16        <AttributeDesignator Category="attribute-
            category:delegated:attribute-category:resource"
17          AttributeId="resource:resource-id"
18          DataType="http://www.w3.org/2001/XMLSchema#string"/>
19      </Match>
20      <Match MatchId="function:string-equal">
```

```
21        <AttributeValue
22          DataType="http://www.w3.org/2001/XMLSchema#string">
              approve</AttributeValue>
23       <AttributeDesignator Category="attribute-
            category:delegated:attribute-category:action"
24          AttributeId="action:action-id"
25          ataType="http://www.w3.org/2001/XMLSchema#string"/>
26      </Match>
27      <Match MatchId="function:string-equal">
28        <AttributeValue
29          DataType="http://www.w3.org/2001/XMLSchema#string">Mary
              </AttributeValue>
30        <AttributeDesignator
31          Category="attribute-category:delegate"
32          AttributeId="subject:subject-id"
33          DataType="http://www.w3.org/2001/XMLSchema#string"/>
34      </Match>
35    </Target>
36    <Rule RuleId="Rule1" Effect="Permit">
37      <Target/>
38    </Rule>
39  </Policy>
40  <Policy PolicyId="PolicyB"
41  RuleCombiningAlgId="rule-combining-algorithm:permit-overrides">
42   <PolicyIssuer>
43    <Attribute
44      AttributeId="subject:subject-id"
```

```
45      DataType="http://www.w3.org/2001/XMLSchema#string">
46     <AttributeValue>Mary</AttributeValue>
47    </Attribute>
48   </PolicyIssuer>
49    <Target>
50      <Match MatchId="function:string-equal">
51        <AttributeValue
52         DataType="http://www.w3.org/2001/XMLSchema#string">Jack<
            /AttributeValue>
53        <AttributeDesignator Category="subject-category:access-
           subject"
54         AttributeId="subject:subject-id"
55         DataType="http://www.w3.org/2001/XMLSchema#string"/>
56      </Match>
57      <Match MatchId="function:string-equal">
58       <AttributeValue
59        DataType="http://www.w3.org/2001/XMLSchema#string">
           expense reports</AttributeValue>
60       <AttributeDesignator Category="attribute-
          category:delegated:attribute-category:resource"
61        AttributeId="resource:resource-id"
62        DataType="http://www.w3.org/2001/XMLSchema#string"/>
63      </Match>
64      <Match MatchId="function:string-equal">
65       <AttributeValue
66        DataType="http://www.w3.org/2001/XMLSchema#string">
           approve</AttributeValue>
```

```
67        <AttributeDesignator Category="attribute-
             category:delegated:attribute-category:action"
68          AttributeId="action:action-id"
69          DataType="http://www.w3.org/2001/XMLSchema#string"/>
70        </Match>
71      </Target>
72    <Rule RuleId="Rule2" Effect="Permit">
73        <Target/>
74      </Rule>
75  </Policy>
76  </PolicySet>
```

Policy 3: XACML policy for the delegation use case.

2.4 Role-based Delegation

A number of models address various aspects of delegation, including [17,37,57,58]. There is a series of role-based delegation models [18,19,79,80]. RBDM0 [19] and RDM2000 [79] in particular are primarily based on roles. RBDM0 addresses human-to-human delegation, whereby a user in a role (delegator role) delegates his role membership to another user in another role (delegatee role). RDM2000 is an extension of RBDM0 by adding role hierarchies and multi-step delegation.

This dissertation bases the work partially on PBDM [80], which builds on RBAC [64] to include user-to-user delegations and role-to-role delegations. PBDM summarizes three scenarios that delegation takes places: (1) back up role, (2) decentralization of authority, and (3) collaboration. The first and third cases require temporary delegation, while the second case requires durable delegation.

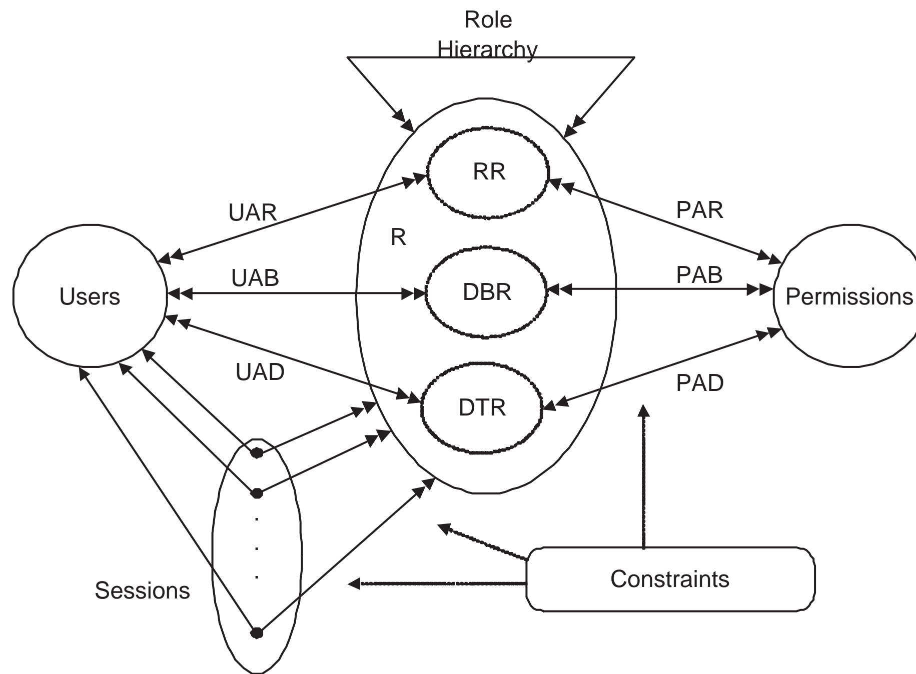

Figure 2.2: PBDM1 model

PBDM has three sub-models: PBDM0, PBDM1 and PBDM2. PBDM0 and PBDM1 support user-to-user delegations and PBDM2 supports role-to-role delegations. Fig 2.2 shows the basic components of PBDM1. In PBDM1, there are three different types of roles: regular role (RR), delegable roles (DBR), and delegation roles (DTR). Permissions assigned to regular roles cannot be delegated to other roles or users. A delegable role is one whose permission can be delegated to other roles or users by creating delegation roles. Each delegable role has exact one base regular role. The users assigned to a delegable role are the same as those assigned to a regular role it based on. A pair of (regular role, delegable) role are used as a single role as user-to-role assignment. The security administrator creates each delegable role and manages the permission-to-delegable role assignment (PAB). RR and DBR are durable roles while DTRs are temporary at the discretion of the owner . The permission-to-regular role assignment (PAR), user-to-regular role assignment (UAR), permission-to-delegable role assignment (PAB), and user-to-delegable role assignment

(UAB) are managed by security administrators, while the permission-to-delegation role assignment (PAD) and the user-to-delegation role assignment (UAD) are managed by individual users.

PBDM provides for single-step as well as multi-step delegation. PBDM supports revocation by a user(1-3) or by an administrator (4-5) as follows.

1. Revoke the user-delegation role assignment.
2. Remove permissions from the delegation role.
3. Remove the delegation role.
4. Remove permissions from the delegable role.
5. Remove a user from a regular and its delegable role.

Chapter 3: Session Administrative Model

This chapter first presents the session administrative model for an RBAC system, then specifies the concurrency control requirements between the session administrative model and the system administrative model for an RBAC system.

3.1 RBAC Session Administration

The RBAC96 [64] and NIST RBAC [34] models include the concept of a session, which provides a context for a user to have multiple simultaneous interactions with resources. Therefore, a user may activate different roles within different sessions concurrently. Every activated role belongs to one session, and a session could have multiple roles activated where each session belongs to a unique user. Some primitive session management functions are specified in the NIST RBAC model [34]. However, they are not included in the existing ARBAC modes [27, 29, 30, 53, 63, 67]. These sessions in RBAC are different from transactions in database management systems [3]. A database transaction must be atomic, consistent, isolated and durable, e.g., satisfy so called ACID properties. Towards the deployment of an administrative RBAC model for a running RBAC system, the author believes that considering session management in ARBAC is mandatory. On one hand, executing an administrative permission may change the configuration of the RBAC system, such as user-to-role or role-to-permission assignment. This configuration change for the running system is problematic if its influence on system state is not carefully designed, e.g., it can result in unexpected usage of permissions for a user who has activated the role in a session before the permissions granted to the role or the role hierarchy is modified. On

the other hand, system or organization requirements may demand that when a user activates particular roles in a session, the associated permissions should not be revoked by any administrative operation during the life cycle of the session, e.g., to preserve the integrity of an object in some applications. By way of example, if the user session is performing some activity that must be completed, such as waiting for a reply from a remote database or rolling back a transaction, then this user session should not be revoked. Now suppose that an administrator wants to revoke the permission granted to the user which requires to terminate the user session. In this case, the enforcement of administrative operation should be delayed until the user session ends to avoid inconsistency. To maintain system safety, different instances of the same role should be granted the same set of permissions at any given state.

This dissertation assumes that the session management is handled by the PEP, which is responsible for creating/deleting a session for a user, or activating/deactivating a role within a session, etc. In order to specify appropriate ARBAC policies for an RBAC system, an administrative model for session management is first defined as follows.

Definition 1 (Session Administration)**.** *Let* $(U, O, A, R, P, \leq, U2R, R2P)$ *be the model of an RBAC system. A session administrative model is a tuple* $SAM = (S, ACTIVE-S, S-ACTION, U2S, S2R, actRole, actPerms)$*, where*

- S *is the set of sessions;*
- $ACTIVE-S$ *is the set of all active sessions at a given system state;*
- $S - ACTION$ $=\{CreateSession(u, s), DeleteSession(u, s), ActivateRole(u, s, r), DeactivateRole(u, s, r)\}$ *is the set of session administrative actions, where* $u \in U$*,* $r \in R$ *and* $s \in ACTIVE-S$*.*
- $U2S : U \mapsto 2^{ACTIVE-S}$ *is a function mapping a user to a set of active sessions at a system state;*

- $S2R : ACTIVE - S \mapsto 2^R$ *is a function mapping an active session to a set of activated roles at a system state;*

- $U2S \circ S2R(u) \subseteq U2R(u)$ *is the constraint that at a system state, all activated roles of a user is a subset of the set of his or her assigned roles, where* $U2S \circ S2R(u) = \cup_{s \in U2S(u)} S2R(s)$*;*

- $activeRoles(u) = \cup_{s \in U2S(u)} S2R(s)$ *is a function mapping a user to a set of activated roles in all active sessions at a system state;*

- $activePerms(u) = \cup_{s \in U2S(u), r \in S2R(s), r \geq r_1} R2P(r_1)$ *is a function mapping a user to a set of activated permissions at a system state.*

$U2S, S2R, activeRoles(u), activePerms(u)$ are the "book keeping" functions to view the states of the system. Invoking any session administrative action changes the system to a new state, by creating/deleting a session for a user, or activating/deactivating a role within a session, etc. The formal semantics of these actions are defined as follows.

- *CreateSession*(u, s), creates a new session *s* for user *u*.
  - *Pre-condition:* *u* is already a member of the user data set and *s* is not a member of ACTIVE-S.

    Formal Specification: $u \in U \wedge s \notin ACTIVE - S$
  - *Post-condition:* U2S is updated.

    Formal Specification: $U2S' = U2S \setminus \{u \mapsto U2S(u)\} \cup \{u \mapsto (U2S(u) \cup \{s\})\} \wedge s \in ACTIVE - S$

- *DeleteSession*(u, s), deletes a given session *s* of user *u*.
  - *Pre-condition:* (u,s) is an entry of U2S.

    Formal Specification: $(u, s) \in U2S \wedge s \in ACTIVE - S$

- *Post-condition:* U2S is updated.

  Formal Specification: $U2S'$=$U2S \setminus \{u \mapsto U2S(u)\} \cup \{u \mapsto (U2S(u) \setminus \{s\})\}$

- *ActivateRole*(u, s, r), activates role *r* in session *s* of user *u*.

  - *Pre-condition:* (u,s) is a member of U2S and (u,r) is a member of U2R.

    Formal Specification: $(u, s) \in U2S \wedge (u, r) \in U2R \wedge s \in ACTIVE - S$

  - *Post-condition:* S2R is updated.

    Formal Specification: $S2R'$= $S2R \setminus \{s \mapsto S2R(s)\} \cup \{s \mapsto (S2R(s) \cup \{r\})\}$.

- *DeactivateRole*(u, s, r), deactivates role *r* from session *s* of user *u*.

  - *Pre-condition:* (s,r) is a member of S2R.

    Formal Specification: $(s, r) \in S2R$

  - *Post-condition:* S2R is updated.

    $S2R'$=$S2R \setminus \{s \mapsto S2R(s)\} \cup \{s \mapsto (S2R(s) \setminus \{r\})\}$.

After defining the session administration model, an ARBAC model is defined as follows.

Definition 2 (ARBAC)**.** *Let* $(U,\ O,\ A,\ R,\ P,\ \leq,\ U2R,\ R2P)$ *be an RBAC model. An administrative RBAC model is a tuple* $ARBAC$ = $(U$, AO, AA, AR, AP, $\leq_A$, $U2AR$, $AR2AP)$*, where*

- $AO = U \cup R \cup U2R \cup R2P \cup \leq$ *is the set of administrative objects;*

- AA *is the set of administrative actions given in Table 3.1 including "+" and "−" operations;*

- AR *is a set of administrative roles;*

Table 3.1: Administrative operations

Positive (+) Operations	**Negative (-) Operations**
AddUser(u)	DeleteUser(u)
AddRole(r)	DeleteRole(r)
AssignUser(u,r)	DeassignUser(u,r)
GrantPermission(r,P)	RevokePermission(r,P)
AddEdge(r^c, r^p)	DeleteEdge(r^c, r^p)

- $AP \subseteq (AO \times AA) \cup (AO \times AO \times AA)$ *is the set of administrative permissions which is an application of an administrative action on one or two appropriate administrative objects;*

- $\leq_A \subseteq AR \times AR$ *is the administrative role hierarchy;*

- $U2AR : U \mapsto 2^{AR}$ *is the user-to-administrative role assignment;*

- $AR2AP : AR \mapsto 2^{AP}$ *is the administrative role-to-administrative permission assignment.*

As defined, the administrative objects (AO) in ARBAC include the set of users (U), roles (R), user-to-role ($U2R$), role-to-permission ($R2P$) mapping, the role inheritance relation ($\leq$) from an RBAC model, and administrative actions defined in Table 3.1. For example, the $AssignUser$ and $DeassignUser$ operations create and remove entries from the user-to-role mapping ($U2R$), respectively. Each execution of an administrative action changes the RBAC system to a new state. The pre-conditions and post-conditions of these operations are specified in Section 3.1.1.

All administrative operations can be classified into "+" operations and "-" operations. A "+" operation adds elements to existing administrative objects from administrative objects, such as assigning a user or granting a permission to a role, while a "-" operation deletes elements, such as revoking a user or permission from a role. Different administrative operations invoke different session administrative actions in the session-aware administrative

model defined in Definition 1.

3.1.1 Formal Specification of Administrative Operations

This dissertation formally specifies suggested administrative operations in terms of pre-conditions and post-conditions using the Z-notation [68]. As per Z-notation, a value of a data item before the execution of a command (so called pre-state of a data structure) is denoted by a symbol, and its value after the execution of the operation (e.g., the so called post state) is denoted by the same symbol followed by a *prime* (').

- *AddUser(u)*: creates an RBAC user *u*.
  - *Pre-condition:* *u* is not already a member of the user data set.
    Formal Specification: $u \notin U$
  - *Post-condition:* The user data set is updated. Initially, *u* is not assigned to any role.
    Formal Specification: $U' = U \cup \{u\} \wedge U2R' = U2R$
- *DeleteUser(u):* deletes an existing user *u* from the user data set.
  - *Pre-condition:* *u* is already a member of the user data set and no roles are assigned to *u*.
    Formal Specification: $u \in U \wedge \nexists r \in R, M \subseteq R : U2R(u, M) \wedge r \in U$
  - *Post-condition:* The user data set is updated.
    Formal Specification: $U' = U \setminus \{u\}$
- *AddRole(r):* creates a new role *r*.
  - *Pre-condition:* *r* is not already a member of roles.
    Formal Specification: $r \notin R$

- *Post-condition:* The new role is added to the roles set R. $U2R$ and $R2P$ remain unchanged.

  Formal Specification: $R' = R \cup \{r\} \wedge U2R' = U2R \wedge R2P' = R2P$

- *DeleteRole(r):* deletes an existing role *r* from the roles data set.

  - *Pre-condition:* The role *r* is a member of the set roles, no user is assigned to *r* and *r* is not a part of the role hierarchy.

    Formal Specification: $r \in R \wedge \nexists u \in U, M \subseteq R : U2R(u, M) \wedge r \in M \wedge \nexists r_1 \in R : (r \leq r \vee r_1 \leq r)$

  - *Post-condition:* *r* is removed from the roles data set.

    Formal Specification: $R' = R \setminus \{r\}$.

- *AssignUser(u,r):* assigns a user *u* to a role *r*.

  - *Pre-condition:* The user *u* is a member of the users data set. The role *r* is a member of roles data set, and the role *r* is not assigned to *u* and is not a child of another role r' assigned to *u*.

    Formal Specification: $[u \in U \wedge r \in R] \wedge \nexists M \subseteq R[r \in M : U2R(u, M)] \wedge \nexists r_1 \in R[r_1 \geq r \wedge r_1 \in M \wedge U2R(u, M)]$.

  - *Post-condition:* $U2R$ is updated.

    Formal Specification: $[U2R(u, M) \rightarrow U2R' = U2R \setminus (u, M) \cup (u, M \cup \{r\})] \wedge [\nexists M \subseteq RU2R(u, M) \rightarrow U2R'(u, \{r\})]$.

- *DeassignUser(u,r):* de-assigns the user *u* from the role *r*.

  - *Pre-condition:* The user *u* is a member of the users data set, the role *r* is a member of roles data set and *u* is assigned to *r*.

    Formal Specification: $u \in U \wedge r \in R, \exists M \subseteq R : r \in M \wedge U2R(u, M)$

- *Post-condition:* The $U2R$ is updated.

  Formal Specification: $\exists\, M \subseteq R, U2R(u, M) \rightarrow U2R'(u, M \setminus \{r\})$

- *GrantPermssion(r,(a,o)):* grants the permission to perform an action *a* on an object *o* to a role *r*.

  - *Pre-condition:* The role *r* is a member of the roles data set and (a,o) is a permission

    Formal Specification: $r \in R \wedge (a, o) \in P$

  - *Post-condition:* The $R2P$ is updated.

    Formal Specification: $\exists\, N \subseteq P : R2P(r, N) \rightarrow R2P(r, N \cup \{(a, o)\})$

- *RevokePermission(r,(a,o)):* revokes the permission to perform action *a* on an object *o* from the set of permissions granted to *r*.

  - *Pre-condition:* The role *r* is a member of the roles data set and (a,o) is assigned to *r*.

    Formal Specification: $r \in R \wedge \exists\, N \subseteq P : R2P(r, N\{(a, o)\})$

  - *Post-condition:* The $R2P$ is updated.

    Formal Specification: $\exists\, N \subseteq P : R2P(r, N) \rightarrow R2P' = [R2P \setminus (r, N)] \cup \{(r, N \setminus \{(a, o)\})\}$.

- *AddEdge*(r^c, r^p)*:* makes the role r^c a child role of r^p.

  - *Pre-condition:* r^c and r^p are members of the roles data set, not related yet and adding does not create cycles in the inheritance hierarchy. $SRole$ is neither a parent nor a child of any role.

    Formal Specification: $r^c, r^p \in R \wedge r^p \not\leq r^c \wedge r^c \not\leq r^p \wedge r^p \neq SRole \wedge r^c \neq SRole \wedge [\nexists r, s \in R(r^c < r < r^P \wedge r^p < s < r^c)]$.

- *Post-condition:* r^p is the parent of r^c.

  Formal Specification: $<' = < \cup \{(r^c, r^p)\}$.

- *DeleteEdge*(r^c, r^p)*:* deletes an existing child-parent relationship $r^c < r^p$.

  - *Pre-condition:* r^c and r^p are members of the roles data set and r^p is a parent of r^c.

    Formal Specification: $r^c, r^p \in R \wedge [r^c < r^p]$.

  - *Post-condition:* The relationship $r^c < r^p$ is deleted.

    Formal Specification:$<' = < \setminus \{(r^c, r^P)\}$.

3.2 Concurrency Control

Similar to an RBAC model, an ARBAC model defines the configuration of the administrative functions of an RBAC system. However, as stated previously, any configuration change affects the running system state, and may require session administrative actions. The interaction between session administrative actions and system administrative operations (e.g., the ARBAC operations defined in Section 3.1.1) needs to be specified for a safe and complete ARBAC model. As one of the major contributions of this dissertation, the author identifies the following two concurrency control requirements between the session administrative model and the system administrative model for an RBAC system.

Revoke an activated role or delete an active session immediately: Suppose an administrative action $aact \in AA$ changes an $RBAC$ model to $RBAC'$, according to the semantics specified in Section 3.1.1. Then, in order to retain the consistency, the affected session is either deleted, or the users from the affected role are de-assigned. This is formally stated as follows:

if $\exists\, u \in U, p \in P, p \in activePerms(u) \wedge p \notin assignPerms(u)'$,

then $[\forall\, s \in U2S(u), \exists\, r \in R, p \in R2P(r) \wedge r \in S2R(r)] \mapsto DeleteSession'(u,s) \vee DeactivateRole'(u,s,r)$,

where $assignPerms(u)'$ is the set of permissions that user u can activate under $RBAC'$, and $DeleteSession(u,s)'$ and $DeactivateRole(u,s,r)'$ are session administrative actions at system state $RBAC'$. This requirement specifies that, when $aact$ removes one or more activated permissions of a user in a session at a system state, either the active session of the user should be terminated, or all corresponding roles with the given permissions should be revoked within their sessions. Obviously, only "-" administrative operations cause these changes in a system.

Delay administrative operations: At a given system state of $RBAC$, when a permission is activated by a user in an active session, any revocation of this permission from the user by an administrative operation is delayed until the role corresponding to the permission is deactivated, or the active session is terminated. Formally, when $aact \in AA$ changes an $RBAC$ model to an $RBAC'$,

if $\exists\, u \in U, p \in P, s \in U2S(u)$, and $p \in activePerms(u) \wedge p \notin assignPerms(u)'$,

then $aact'$ when $p \notin activePerms(u)'$. That is, the administrative operation $aact'$ is executed at a later stage when the permission is not activated anymore.

Note that these two requirements can be individually or jointly specified in a particular system, e.g., some permissions are required to be immediately deactivated in an active session when they are revoked by an administrative action, while other permissions may delay the execution of an administrative operation.

When an administrative operation modifies a role, the access controller not only needs to manage current active sessions but also any newly created sessions. This is especially necessary in delayed administrative actions. Specifically, when an administrative operation is delayed, although the affected permissions or roles are not deactivated immediately, the access controller needs to prevent users from activating them in new sessions. To do this,

the access controller will lock the affected roles to ensure the safety property. The administrative operation places write locks on the affected roles to prevent the PDP from "reading" the roles and other administrative operation from "writing" the roles. This dissertation defines the affected roles in Definition 3.

Definition 3 (Lock Scope)**.** *Let* $(U, O, A, R, P, \leq, U2R, R2P)$ *be the model of a RBAC system and* $r \in R$ *be a role. I define the read scope and write scope of* r *respectively as* $rScope(r) = \{r_1 \in R | r_1 \leq r\}$ *and* $wScope(r) = \{r_1 \in R | r_1 \geq r\}$.

As stated in Definition 3, the read scope of a role r includes all its junior roles and itself, and the write scope of r includes all its senior roles and itself. This is because, a role r may lose permissions if any junior role r_1 loses its permissions because of inheritance, and therefore needs to ensure that if r_1 is to lose permissions, then r needs to be deactivated to ensure consistency. Therefore, when the PDP is evaluating role r, all the roles junior to r and r itself (that is, the *read scope* of r), must not be allowed to be modified. Conversely, if role r is to lose permissions due to an administrative operation, then all roles senior to r and r itself, that is the *write scope* of r must not be allowed to be activated. For example in Figure 3.1, the read lock scope for R3 is {R6, R5, R3}. The write lock scope for R3 is {R0, R1, R2, R3}. Note that the lock scopes of a role could be changed because of an administrative operation. For example, the write lock scope for R4 is {R0, R1, R4}. If an administrative role executes the administrative operation *AddEdge(R4, R2)*, the write lock scope for R4 becomes {R0, R1, R2, R4}.

3.2.1 Entities Affected by Enforcing Administrative Operations

Each administrative operation could affect many entities. The affected entities can be defined because of invoking an administrative operation using lock scope. Algorithm 1 in Figure 3.2 shows this information for every administrative operation listed in Table 1.

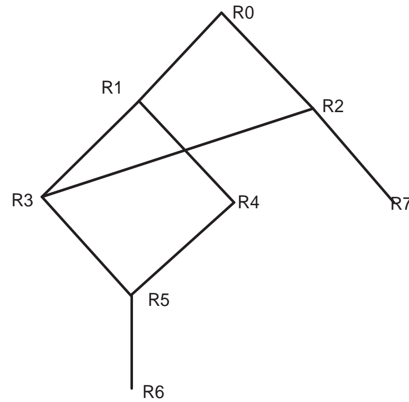

Figure 3.1: An example role hierarchy.

Algorithm 1: Compute affected entities
Input: adminOp
Output: Return affected to A-PEP

```
1   switch adminOp do
2     case DeleteUser(u)
3       affected:=u;
4     case DeleteRole(r)
5       affected:=wScope(r);
6     case DeassignUser(u,r)
7       affected:=(rScope(r),u);
8     case RevokePermission(r,P)
9       affected:=wScope(r);
10    case DeleteEdge(r^c, r^p)
11      affected:=wScope(r^p);
12    otherwise
13      affected:=NULL;
14  return affected;
```

Figure 3.2: Compute entities affected due to an administrative action.

DeleteUser(u) deletes user u which affects all the sessions u has activated. Consequently, the affected entity is u as computed in lines 2-3. *DeleteRole(r)* deletes role r which affects all the roles senior to *r* and *r* itself, and that is wScope(r) as computed in lines 4-5. *DeassignUser(u,r)* prevents user *u* from activating role *r* which affects all the roles subordinate

to *r*, computed as *rScope(r)* in lines 6-7. *RevokePermission(U,P)* revokes the permission set P from the role r which affects all the roles senior to *r* and *r* itself. Consequently, the affected entities are computed as *wScope(r)* in lines 8-9. *DeleteEdge*(r^c, r^p) deletes the relation $r^c < r^p$ which makes all the roles senior to (r^p) and (r^p) lose the permissions granted to (r^c). Therefore, the affected entities are wScope(r^p) as computed in lines 10-11. For example, deleting the role R3 from the role hierarchy in Figure 3.1 affects all the sessions where R0, R1, R2 and/or R3 are activated. The affected entities are those in *wScope(R3)*.

3.3 Related Work

The RBAC research community has extended RBAC models to use RBAC itself to administrate the RBAC models, commonly referred to as *administrative role-based access control (ARBAC)* models [27,29,30,53,60,63,67]. The main focus of these ARBAC models is how to configure the components of an RBAC system such as user-to-role assignment, role-to-permission assignment, and the role hierarchy. There is no context of session management and concurrency control in these existing ARBAC models.

ARBAC97 [63] is the first attempt to specify a comprehensive administrative model for RBAC. ARBAC97 is based on the RBAC96 models [64]. ARBAC97 assumes that there is a set of administrative roles, *AR*, which is disjoint from the set of normal roles. Only members of these roles can perform administrative operations. ARBAC97 consists of three sub-models: URA97 for administration of user-to-role assignment, PRA97 for administration of permission-to-role assignment, and RRA97 for administration of role-to-role assignment, that is administration of the role hierarchy. All three sub-models rely on the concept of role ranges, which are used as administrative domains. Creation of users and permissions are not included in ARBAC97, which ARBAC97 believes that its the responsibility of personal management department and system/application administrators. In this

dissertation, the treatment of them is the same as other administrative operation to maintain simplicity and uniformity.

ARBAC02 [60, 67] introduces the concept of organization structure for defining user and permission pools independent of roles and role hierarchies, while a user pool in ARBAC97 is implemented using a prerequisite role. ARBAC02 retains the main features of ARBAC97 and adds new components of organization structures as user and permission pools to overcome some shortcomings in URA97 and PRA97. ARBAC02 presents a bottom-up approach of permission-to-role administration in contrast to the top-down approach in ARBAC97.

SARBAC [29, 30] extends RBAC administration by adding the concept of *administration scope*s. Administrative scope is defined using the role hierarchy, and is used for defining administrative domains. The administrative scope of a role *(*r) consists of all roles that are descendants of *r* and are not descendants of any role that is incomparable with *r*. This definition of scopes works best when the role hierarchy is a tree with an all-powerful root role. In this case, each role's administrative scope is the subtree rooted at that role. When an operation may affect existing administrative domains, ARBAC97 forbids these operations, while SARBAC allows them and handles them by changing existing administrative domains. One feature of SARBAC is that one simple operation may affect administrative domains of many roles. SARBAC overcomes some shortcomings of RRA97.

The administrative model for role hierarchy in SARBAC is later refined and extended to RBAT, a template for role-based administrative models [27]. RBAT formalizes the interaction between the role hierarchy operations and the administrative scopes by having the operations preserve certain aspects of administrative scopes. The role hierarchy administrative model in both ARBAC97 and SARBAC can be expressed in terms of the RBAT framework.

UARBAC [53] takes a principled approach in designing and analyzing administrative

models for RBAC motivated by scalability, flexibility, psychological acceptability, and economy of mechanisms. The six principles (or requirements) in UARBAC: (1) support decentralized administration and scale well to large RBAC systems; (2) are policy neutral in defining administrative domains; (3) should provide that apparently equivalent sequences of operations should have the same effect; (4) support reversibility; (5) support predictability; and (6) use RBAC to administrate RBAC. UARBAC consists of a basic model and one extension: $UARBAC^P$. The basic model adopts the approach of administrating RBAC using RBAC. $UARBAC^P$ adds parameterized objects and constraint-based administrative domains.

NIST [20, 33, 38] has implemented RBAC with an *administrative tool* and an RBAC database to store instances of U2R, R2P, and $\leq$ relationships. The administrative tool determines if updatea to the three relations stored in the database are permitted by checking the consistency rules, and, if so, updates the relationships in the database. This implementation is built for the Intranet, which is not suitable for distributed computing environment such as Web Services.

There are practical studies in [35, 47, 48, 56, 65] using RBAC in enterprise administration and settings, these papers report priceless experiences from deploying large RBAC systems in practice; even though they do not provide formal models for RBAC administration. The session administrative model in this dissertation is largely inspired by these experiences. RBAC administration is also studied in [71–73]. This dissertation differs from them in that this dissertation adopts session management and specified concurrency control requirements in the administrative model.

3.4 Summary

In this chapter, I have developed a session administrative model for an RBAC system. A model is given of a set of system states, a set of session administrative actions, and a set of "book keeping" functions mapping the attributes at a system state. To keep consistency of an RBAC system, two concurrency control requirements are identified: (1) revoke an activated role or delete an active session immediately, (2) delay administrative operations. These two requirements can be individually or jointly specified in a particular system. The concept of lock scope for a role is introduced to capture the affected role when the permissions granted to this role are updated due to administrative operations. I have developed an algorithm to calculate the affected entities because of invoking an administrative operation using lock scope.

Chapter 4: XACML-ARBAC Profile and the Enforcement Architecture

This chapter presents an XACML profile for ARBAC and the architecture to enforce this profile. Because ARBAC is an RBAC model with administrative roles having specialized permissions to administrate an underlying RBAC system, the XACML-ARBAC profile is also an XACML-RBAC profile.

4.1 The XACML-RBAC Profile

The XACML-RBAC profile 2.0 has been approved as an OASIS standard [1] to specify core and hierarchical components of RBAC models. In this profile, objects, actions, and users are expressed as XACML <Resource>s, <Action>s, and <Subject>s. However, roles are expressed as <Subject> attributes or <Resource> attributes. This profile further defines three generic XACML policies: a *Permission <PolicySet>*, a *Role* <PolicySet>, and a *Role Assignment <Policy>* or <PolicySet>. These are used to express the remaining entities of an RBAC model (e.g., permissions, $U2R$ and $R2P$ mappings, and role hierarchy $\leq$), which are briefly explained as follows:

A Permission <PolicySet> is a <PolicySet> used to define a set of permissions associated with a role. It may contain <PolicySetIdReference>s to other Permission <PolicySet>s. Stated <PolicySetIdReference> can be used to inherit permissions of a junior role. Currently, this is the only way to specify the role hierarchy in the XACML-RBAC profile.

A Role <PolicySet> binds a set of attributes defining a role in a <Target> to a <PolicySetIdReference> outside of that <Target>. The latter points to the Permission <PolicySet> of this role.

A Role Assignment <Policy> or <PolicySet> does not have a standard specification. The objective of the role assignment <Policy> or <PolicySet> is to specify the user-to-role ($U2R$) assignment. This part of an RBAC policy is supposed to be specified by an entity external to the XACML policy framework, referred to as the *Role Enabling Authority (REA)*. The XACML-RBAC profile does not specify any more requirements of the REA.

4.2 The XACML-ARBAC Profile

In the OASIS XACML-RBAC profile, roles are defined as attributes of subjects and resources. This dissertation enhances the XACML syntax by introducing a new data type *Role*. As the *administrative roles* needs to be distinguished from *user roles*, this dissertation introduces a *roleType* attribute that can take value from $\{$*userRole*,*adminRole*$\}$. This dissertation uses all other primitive entities from the XACML-RBAC profile. In particular, the role hierarchy and role-to-permission assignments are expressed in the same way as in the XACML-RBAC profile. This dissertation uses an XML file to maintain all user-to-role assignments in the policy repository as the follows:

```
1        <Subjects>
2          <Subject SubjectId="Alice">
3            <Roles> <Role>SSO </Role>
4                    <Role>Manager</Role>
5            </Roles>
6          </Subject>
7          <Subject SubjectId="Bob">
```

```
8            <Roles> <Role>Manager</Role></Roles>
9          </Subject>
10       </Subjects>
```

The PDP gets all the roles that a user can invoke by querying this XML file. Although this dissertation could have maintained the user-to-role assignment as a Role Assignment <PolicySet>, but this dissertation does not do so because the current XACML reference implementation does not answer a query such as *What are the roles assigned to Alice?*. Using this extra XML file, this dissertation specifies administrative policies using the same machinery as the XACML-RBAC profile, albeit with the following constraints.

Constraining the Permission <PolicySet>: All permissions listed in a <PolicySet> of an administrative role must be administrative permissions. By enforcing the following constraints on the syntax used in a permission <PolicySet>, this dissertation ensures that it is an *administrative* Permission <PolicySet>.

1. The <Condition>s are created from applying boolean operations to existing XACML condition functions and an enlarged set of condition functions listed in Table 4.2.

2. The (<Action>, <Resource>) pair listed in <Rule> must be an administrative permission (AP); that is, the actions must be chosen from operations listed in Table 3.1.

Policy 5 gives an example of a Permission <PolicySet> of SSO, which is an administrative role. For simplify, I only list the permission “AddRole” (line 37) and permission “AssignUser” (line 65) in Policy 5. Other permissions listed in Table 3.1 can also be specified similarly.

Constraining the Role <PolicySet>: The Role <PolicySet> of an administrative role must be an administrative <PolicySet> with the following additional constraints:

Table 4.1: Extended functions applied in <Condition> in XACML-ARBAC profile

Function	**Intuitive Meaning**
role-exist(r)	check the presence of the role r
inherited-by-assigned-role(r)	check if the role r is inherited by a role already assigned to the subject
inherit-assigned-role(r)	check if the role r inherits a role already assigned to the subject
role-assigned-exist(s,r)	check if the subject s is already assigned to the role r
permission-exist(r,p)	check if the role r has been already granted the permission p
role-has-children(r)	check if the role r has any children
role-has-parent(r)	check if the role r has any parent
role-is-assigned(r)	check if the role r is assigned or not
role-is-inherited-by(r1,r2)	check if r1 is inherited by r2
role-is-parent-of(r1,r2)	check if r1 is parent of r2

1. All role names that appear in the <Target> of the Role <PolicySet> should be administrative roles; that is, their *roleType* should be set to “adminRole” as shown in line 19 of Policy 5.

2. The <PolicySetIdReference> contained in the Role <PolicySet> should point to an administrative Permission <PolicySet> where all permissions must be chosen from the administrative permissions listed in Table 3.1.

```
1<PolicySet PolicySetId="Admin::Role:Policy" PolicyCombiningAlgId=
   "urn:oasis:names:tc:xacml:1.0:policy-combining-
   algorithm:permit-overrides">
2  <Target>
3    <Subjects>
4      <AnySubject/>
5    </Subjects>
6    <Resources>
7      <AnyResource/>
8    </Resources>
9    <Actions>
```

```
10      <AnyAction/>
11    </Actions>
12  </Target>
13  <PolicySet PolicySetId="RPS:SSO:role" PolicyCombiningAlgId="
      urn:oasis:names:tc:xacml:1.0:policy-combining-
      algorithm:permit-overrides">
14    <Target>
15      <Subjects>
16        <Subject>
17          <SubjectMatch MatchId="role-equals">
18            <AttributeValue DataType="role">SSO</AttributeValue>
19            <SubjectAttributeDesignator SubjectCategory="
                urn:oasis:names:tc:xacml:1.0:subject-
                category:access-subject" AttributeId="
                urn:oasis:names:tc:xacml:1.0:subject:subject-role-
                id" DataType="adminRole"/>
20          </SubjectMatch>
21        </Subject>
22      </Subjects>
23      <Resources>
24        <AnyResource/>
25      </Resources>
26      <Actions>
27        <AnyAction/>
28      </Actions>
29    </Target>
30    <PolicySetIdReference>PPS:SSO:role</PolicySetIdReference>
```

```
31   </PolicySet>
```

Policy 4: Example of a Role <PolicySet> for SSO.

```
1 <PolicySet PolicySetId="Admin:Permission:Policy"
     PolicyCombiningAlgId="urn:oasis:names:tc:xacml:1.0:policy-
     combining-algorithm:permit-overrides">
2    <Target>
3      <Subjects>
4        <AnySubject/>
5      </Subjects>
6      <Resources>
7        <AnyResource/>
8      </Resources>
9      <Actions>
10       <AnyAction/>
11     </Actions>
12   </Target>
13   <PolicySet PolicySetId="PPS:SSO:role" PolicyCombiningAlgId="
       urn:oasis:names:tc:xacml:1.0:policy-combining-
       algorithm:permit-overrides">
14     <Target>
15       <Subjects>
16         <AnySubject/>
17       </Subjects>
18       <Resources>
19         <AnyResource/>
20       </Resources>
```

```
21      <Actions>
22        <AnyAction/>
23      </Actions>
24    </Target>
25    <Policy PolicyId="Permissions:specifically:for:the:SSO:role"
         RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-
         combining-algorithm:permit-overrides">
26      <Target>
27        <Subjects>
28          <AnySubject/>
29        </Subjects>
30        <Resources>
31          <AnyResource/>
32        </Resources>
33        <Actions>
34          <AnyAction/>
35        </Actions>
36      </Target>
37      <Rule RuleId="Permission:to:add:a:role" Effect="Permit">
38        <Target>
39          <Subjects>
40            <AnySubject/>
41          </Subjects>
42          <Resources>
43            <Resource>
44              <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1
                   .0:function:string-equal">
```

```
45                <AttributeValue DataType="http://www.w3.org/2001/
                     XMLSchema#string">role</AttributeValue>
46                <ResourceAttributeDesignator AttributeId="
                     urn:oasis:names:tc:xacml:1.0:resource:resource
                     -id" DataType="http://www.w3.org/2001/
                     XMLSchema#string"/>
47              </ResourceMatch>
48            </Resource>
49          </Resources>
50          <Actions>
51            <Action>
52              <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0
                   :function:string-equal">
53                <AttributeValue DataType="http://www.w3.org/2001/
                     XMLSchema#string">add</AttributeValue>
54                <ActionAttributeDesignator AttributeId="
                     urn:oasis:names:tc:xacml:1.0:action:action-id"
                      DataType="http://www.w3.org/2001/XMLSchema#
                     string"/>
55              </ActionMatch>
56            </Action>
57          </Actions>
58        </Target>
59        <Condition FunctionId="urn:oasis:names:tc:xacml:1.0
             :function:not">
60          <Apply FunctionId="role-exist">
```

```
61            <ResourceAttributeDesignator AttributeId="
                 urn:oasis:names:tc:xacml:1.0:resource:new-role-id"
                  DataType="role"/>
62          </Apply>
63        </Condition>
64      </Rule>
65     <Rule RuleId="Permission:to:assign:a:user:to:a:role" Effect=
          "Permit">
66        <Target>
67          <Subjects>
68            <AnySubject/>
69          </Subjects>
70          <Resources>
71            <Resource>
72              <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1
                   .0:function:string-equal">
73                <AttributeValue DataType="http://www.w3.org/2001/
                     XMLSchema#string">user</AttributeValue>
74                <ResourceAttributeDesignator AttributeId="
                     urn:oasis:names:tc:xacml:1.0:resource:resource
                     -id" DataType="http://www.w3.org/2001/
                     XMLSchema#string"/>
75              </ResourceMatch>
76            </Resource>
77          </Resources>
78          <Actions>
79            <Action>
```

```
80          <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0
               :function:string-equal">
81            <AttributeValue DataType="http://www.w3.org/2001/
                XMLSchema#string">assign</AttributeValue>
82            <ActionAttributeDesignator AttributeId="
                urn:oasis:names:tc:xacml:1.0:action:action-id"
                 DataType="http://www.w3.org/2001/XMLSchema#
                string"/>
83          </ActionMatch>
84        </Action>
85      </Actions>
86    </Target>
87    <Condition FunctionId="urn:oasis:names:tc:xacml:1.0
         :function:and">
88      <Apply FunctionId="role-exist">
89        <ResourceAttributeDesignator AttributeId="
             urn:oasis:names:tc:xacml:1.0:resource:role-id"
             DataType="role"/>
90      </Apply>
91      <Apply FunctionId="urn:oasis:names:tc:xacml:1.0
           :function:not">
92        <Apply FunctionId="role-assignment-exist">
```

```
93          <SubjectAttributeDesignator SubjectCategory="
              urn:oasis:names:tc:xacml:1.0:subject-
              category:access-subject" AttributeId="
              urn:oasis:names:tc:xacml:1.0:subject:assignee-
              subject-id" DataType="http://www.w3.org/2001/
              XMLSchema#string"/>
94          <ResourceAttributeDesignator AttributeId="
              urn:oasis:names:tc:xacml:1.0:resource:role-id"
              DataType="role"/>
95        </Apply>
96      </Apply>
97      <Apply FunctionId="urn:oasis:names:tc:xacml:1.0
          :function:not">
98        <Apply FunctionId="inherits-assigned-role">
99          <ResourceAttributeDesignator AttributeId="
              urn:oasis:names:tc:xacml:1.0:resource:role-id"
              DataType="role"/>
100         <SubjectAttributeDesignator SubjectCategory="
              urn:oasis:names:tc:xacml:1.0:subject-
              category:access-subject" AttributeId="
              urn:oasis:names:tc:xacml:1.0:subject:assignee-
              subject-id" DataType="http://www.w3.org/2001/
              XMLSchema#string"/>
101       </Apply>
102     </Apply>
103     <Apply FunctionId="urn:oasis:names:tc:xacml:1.0
          :function:not">
```

```
104              <Apply FunctionId="inherited-by-assigned-role">
105                <ResourceAttributeDesignator AttributeId="
                     urn:oasis:names:tc:xacml:1.0:resource:role-id"
                     DataType="role"/>
106                <SubjectAttributeDesignator SubjectCategory="
                     urn:oasis:names:tc:xacml:1.0:subject-
                     category:access-subject" AttributeId="
                     urn:oasis:names:tc:xacml:1.0:subject:assignee-
                     subject-id" DataType="http://www.w3.org/2001/
                     XMLSchema#string"/>
107              </Apply>
108            </Apply>
109          </Condition>
110        </Rule>
111      </Policy>
112    </PolicySet>
```

Policy 5: Example of the Permission <PolicySet> for SSO.

Sun's reference implementation [8] uses a set of methods, referred as *condition functions*, to compare retrieved attributes' values with expected values in order to make access decisions. An example condition function provided by the reference implementation is the form "[type]-one-and-only", that accepts a bag of values of the specified type and returns the single value if there is exactly one item in the bag, or an error if there are zero or multiple values in the bag. The condition functions provided by Sun's reference implementation are not capable of checking the conditions for most administrative operations. For example, to add a role r into the system, the access controller needs to check if r is already defined. To support all possible conditional checks for administrative operations formally specified

in Section 3.1.1, this dissertation adds a new set of condition functions listed in Table 4.2. For example, Function “role-exist” is used in line 59 of Policy 5 to check a role existence prior to adding it to system. Function “role-assignment-exist” is used in line 93 of Policy 5 to check a user-to-role assignment instance existence prior to assigning a user to the role. These condition functions are internal auxiliary functions which do not affect the system state.

4.2.1 Administrative Operation

Each administrative operation, such as *AddRole(r)*, has two steps. First, the administrator who is assigned to the administrative role needs to get the permission to add a role r, then add the role r into the RBAC system by updating the corresponding policies. Next the author specifies how to update the corresponding policies for each administrative operation.

- *AddUser(u):* adds the new user *u* entry into the user-to-role assignment XML file with no roles assigned to it.

- *DeleteUser(u):* deletes the user *u* entry with all the roles assigned to it from the user-to-role assignment XML file.

- *AddRole(r):* adds role *r* to the system, by creating a new Role <PolicySet> with *r* in the <Target>. It also creates Permission <PolicySet> with empty permissions, and inserts the <PolicySetIdReference> reference into the Role <PolicySet< pointing to the Permission <PolicySet>.

- *DeleteRole(r):* removes role *r* from the system by deleting the Role <PolicySet> which has *r* in the <Target> and the Permission <PolicySet> which the Role <PolicySet> points to.

- *AssignUser(u, r):* updates the user-to-role assignment XML file by adding role *r* under the user *u* entry.
- *DeassignUser(u, r):* updates the user-to-role assignment XML file by removing role *r* from the user *u* entry.
- *GrantPermission(r, P):* adds permission *P* as a rule into the Permission <PolicySet> of role *r*.
- *RevokePermission(r, P):* removes the rule specified for permission *P* from Permission <PolicySet> of role *r*.
- *AddEdge(r^c, r^p):* adds a <PolicySetIdReference> reference into the Permission <PolicySet> of role r^p pointing to the the Permission <PolicySet> of role r^c.
- *DeleteEdge(r^c, r^p):* deletes a <PolicySetIdReference> reference from the Permission <PolicySet> of role r^p pointing to the Permission <PolicySet> of role r^c.

4.3 Enforcing The XACML-ARBAC Profile

In order to enforce this XACML-ARBAC profile, this dissertation enhances the existing XACML reference implementation with the two entities shown with bold borders in Figure 4.1 and explained as follows.

The *Administrative PEP (A-PEP)* receives an administrative access control request, returns a response to the administrator, and, if needed, updates relevant policies as a consequence of enforcing the requested administrative operation. The A-PEP also functions as a *Role Enabling Authority*. When a subject is assigned to a role and revoked from a role, the A-PEP acts as an enabler/ disabler by invoking the appropriate administrative operation

and updates the U2R mapping in an XML file. When needed by the PDP or the context handler, the A-PEP provides appropriate instances of the U2R mapping.

The *Lock Manager* provides the concurrency control necessary to maintain the transactional consistency between simultaneous operations that the PDP requires to read policies in order to evaluate them and the A-PEP needs to modify policies to enforce administrative operations.

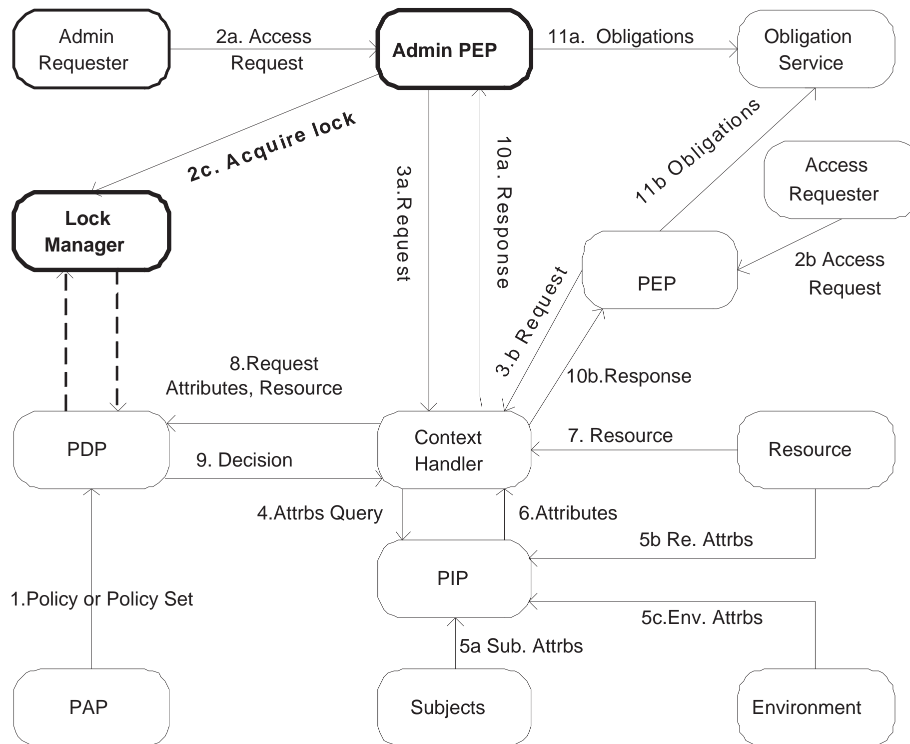

Figure 4.1: Extended XACML architecture for XACML-ARBAC enforcement.

4.3.1 Concurrency Control

When a non-administrative request arrives at the PDP, the PDP requests a read lock on the policy that is found using the *target matching* algorithm (described in Section 2.1).

In the case of an administrative request, the policy evaluation part is similar to the non-administrative request, where the PDP acquires a read lock on the policy for evaluation. If the administrative request is granted, the PDP sends a *permit* decision to the A-PEP. After receiving a *permit* decision from the PDP, the A-PEP acquires a write lock on the policy (recall that administrative operations update XACML policies) that is to be updated. This dissertation now describes the details of these steps.

Evaluating Authorization Requests Sun's reference implementation does not alter any XACML policies, and it uses the policy evaluation algorithm explained in [2]. As the administration operations update policies, this evaluation algorithm needs to be protected by a semaphore. When a non-administrative request arrives at the PDP, the PDP first requests a read lock (from the *Lock Manager*) on the policy that is found using the *target matching* algorithm (line 2), evaluates the request using the existing XACML policy evaluation algorithm (line 3), updates the runtime PEPList (the list of active PEPs) (line 4), and finally releases the read lock on the policy (line 5) and returns the response back tho the requesting PEP (line 8), which in turn returns the response back to the user and invokes application dependent activity to enforce the decision. If the PDP fails to acquire the read lock, it returns *indeterminate* as a response to the requesting PEP. The PDP goes through the steps outlined in Figure 4.2.

Enforcing Administrative Operations When an administrative request is submitted to the A-PEP, the A-PEP forwards the request to the PDP for evaluation. The PDP uses the same evaluation algorithm used to evaluate the non-administrative request (see Figure 4.2) and returns its decision to the A-PEP. If the returned value received at the A-PEP is not a *permit*, the A-PEP conveys the decision to the administrator. Otherwise (the return value is *permit*), the A-PEP uses the algorithm shown in Figure 4.3 to enforce that decision. As the algorithm states, if the decision is not a *permit*, the A-PEP returns that decision to the administrator (line 19). Otherwise, it acquires a write lock on the policy to be updated

```
Algorithm 2: PDP evaluating request
Input: Request, PEPID
Data: PEPList
Output: access control decision
   /*PDP maintains the PEP-List accessible to A-PEP*/
1  policy:=targetMatching(request);
   /*find the policy to be evaluated using target matching*/
2  if AcquireLock(policy,read) then
3     decision:=evaluate(Request,policy);
4     PEPList:=+PEPID;
5     ReleaseLock(policy,read);
6  else
7     decision:=Intermediate;
8  return decision;
```

Figure 4.2: PDP evaluation algorithm.

(line 3), and calls the method `getAffected(adminOp)` using the algorithm shown in Figure 3.2 in Section 3.2 to determine the parameters that are *affected* by the administrative operation (line 5). Then, the A-PEP sends a request to all PEPs to terminate user sessions that may be affected by enforcing the administrative operation (lines 6-8). Because the access controller cannot wait forever for those PEPs to confirm that the requested sessions have been terminated, the A-PEP sets up a timer (line 7). If all PEPs returned successful answers (lines 12-14), then the A-PEP will update the policy to reflect the administrative operation, release the write lock on the policy (line 16), and finally inform the administrator that the administrative operation is enforced (the *permit* decision). Conversely, if any PEP fails to return a positive answer when the timer expires, the administrative request is denied.

```
Algorithm 3: Enforcing administrative operations
Input: adminOp, PDPdecision
 /*PDP returns policy decision to A-PEP*/
Data: PEPList
Output: Return decision to administrator
1   if PDPdecision==permit then
2     decision:=deny;
3     if AcquireLock(policy,write) then
4       if adminOp is a (-) operation then
5         Affected:=getAffected(adminOp);
6         forall PEP ∈ PEPList do
7           set(timer, value);
8           sendRequest(PEP,(Affected,killSession));
9         if expires(timer) then
10          acceptFlag:=ok;
11      forall PEP ∈ PEPList do
12        recv(PEP,(Affected, killsSession, NotOK));
13        acceptFlag:=reject;
14      if acceptFlag=ok then
15        modifyPolicy(policy, adminOp);
16        ReleaseLock(policy,write);
17        decision:=permit;
18  else
19    decision:=PDPdecision;
20  return (admin, decision);
```

Figure 4.3: Enforcing administrative operations.

4.3.2 The Lock Manager

The PDP and A-PEP competition is similar to the Readers and Writers problems described in [51]. The Readers and Writers problems have a common basis but differ in matters of policy governing control invoking Readers versus Writers. Multiple requests from the PDP to read a policy simultaneously for policy evaluation maybe be allowed but the A-PEP must have exclusive access to modify a policy. Different scenarios must be considered as follows:

- In situations where one or more PDP instances are active, is it permissible to allow a newly-arriving PDP instance to join immediately even if there are also A-PEP instances waiting? If so, a continuous stream of entering PDP instances will cause A-PEP to starve. If not, the throughput of PDP instances decreases.

- If both some PDP instances and some A-PEP instances are waiting for an active A-PEP instance to finish, should you bias the policy toward allowing PDP instances? a A-PEP instance? Earliest first? Random? Alternate? Similar choices are available after termination of PDP instances.

- Is a policy needed to allow an A-PEP instance to downgrade access to become a PDP instance without having to give up locks?

As there are no generally recognized "right" answers to these policy questions, the author proposes a standard solution with a set of choices: PDP instances are blocked if there are waiting A-PEP instances, waiting A-PEP instances are chosen arbitrarily (just relying on the order in which the underlying Java Virtual Machine (JVM) scheduler happens to resume unblocked threads), and there are no down-grade mechanisms.

The *Lock Manager* (see Figure 4.1) maintains read/write locks on policies. Because these policies are role-based, the locks are placed on the roles. The functionality of the Lock Manager that provides and maintains locks is summarized below:

- AcquireLock (role, read/write) : used to obtain a lock on a role.

- ReleaseLock (role, read/write): used to relinquish active locks.

- AttemptLock (role, ReadLock, WriteLock): Attempt lock method is used to wait if there is already lock on a role.

Lock acquisition and release are atomic operations. For example, to acquire a write lock on R3 in Figure 3.1, first the A-PEP need acquire a write lock on R0, if succeed, then acquire a write lock on R1, R2 finally, acquire a write lock on R3. If failing acquiring any write lock, acquiring a write lock on R3 is failed. This can prevent dead lock, for example: when two concurrent requests to acquire write locks on R3 and R7. The lock scope for R3 is R0, R1, R2, R3 and the lock scope for R7 is R0, R1, R7. If there is no order, R3 might get lock for R0 first then attempt to get a lock for R2, and the same time, R7 gets lock R2 first then attempts to get a lock for R0, it will have a dead lock. Because the lock scope is an ordered list, it is by design and inherently dead lock free in [49].

4.4 The Birth and Death Processes

In this dissertation, when the access controller becomes alive, it follows the *initialization sequence* of creating a *super user (SU)* and a *super role (SRole)*, where the *SRole* is the administrative role. This dissertation simplifies the administrative RBAC system with only a single administrative role. Consequently, the resulting U2R and R2P updates are precisely specified in the following pre-conditions and post-conditions.

- *Pre-condition:* U, R, AR, U2R, and R2P is empty.

  Formal Specification: $U = \{\} \wedge R = \{\} \wedge AR = \{\} \wedge U2R = \{\} \wedge R2P = \{\}$.

- *Post-condition:* SU is the only member of the users data set and $SRole$ is the only member of the administrative role (AR) data set. These and the appropriate permissions are created during the bootstrapping procedure of the access controller from a file which contains the default administrative policy loaded into the system data structures.

  Formal Specification: $U' = \{SU\} \wedge AR' = \{SRole\} \wedge U2R' = \{(SU, SRole)\} \wedge$

$R2P' = \{(SRole, p)\} \wedge p \neq (deleteRole, SRole) \wedge p \neq (deleteUser, SU) \wedge u \neq SU$, where p is any administrative operation described in Table 3.1.

As seen from the post-condition, after the initialization phase finishes, the super user SU is the only user in the system endowed with $SRole$'s permissions - the administrative permissions described in Table 3.1. Also as specified, the $SRole$ does not have permissions to delete SU, nor de-assign SU from the $SRole$. Consequently, permissions granted to the $SRole$ remain un-alterable and the $SRole$ has no relation with other roles through $\leq$, as formally specified in the $AddEdge$ administrative operation in the Section 3.1.1.

The access controller does not entertain any user requests during the initialization phase. After the RBAC system boots up, the $SRole$ may perform other administrative operations such as creating user roles, creating users and assigning users to roles, etc.

When the access controller is ready to stop services, the SU notifies all the active PEPs that the access controller is going to stop services and requests the PEPs to terminate any user sessions authorized by this access controller. After getting the acknowledgement messages from the PEPs or the timer expires, SU signals the operating system to shutdown the access controller. Here this dissertation assumes that all PEPs are cooperative. The access controller does not entertain any user requests after sending messages to all PEPs. Consequently, the resulting $ACTIVE-S$, $U2S$, $S2R$, $actRole$, and $actPerms$ are specified in the following post-conditions.

Post-condition: $ACTIVE-S$, $U2S$, $S2R$, $actRole$, and $actPerms$ are empty.

Formal Specification: $ACTIVE-S = \{\} \wedge U2S = \{\} \wedge S2R = \{\} \wedge actRole = \{\} \wedge actPerms = \{\}$.

4.5 Related Work

There have been many works in the area of access control for Web Services [22,31,74,78]. Most of these focus on how to express access control policies and the architectures to implement the model. However, these solutions have not addressed the concurrency issues between policy evaluation and administrating the access control policies.

Crampton and Chen [28] have proposed an approach to implement the RBAC model using XACML. They attempt to implement the ANSI RBAC standard [34] using a suit of XACML policies. They use attribute-based role assignment for the U2R assignment, define an XML-based language for specifying separation of duty constraints and propose an extension to the XACML reference architecture in order to enforce these constraints. To the best knowledge of the author, these proposals have not been fully implemented.

Seitz et al. [66] present a system permitting controlled policy administration and delegation using the XACML access control system. They use a second access control system, *Delegent*, which has delegation capabilities to supervise modifications of the XML-encoded XACML policies. Concurrent administration with authorization is not addressed in *Delegent*.

Karjoth et al. [46] implement ACL-based policies in XACML by mapping the IBM Tivoli Access Manager (AM) policy language [45] into XACML. They use novel XACML features to translated ACL policy, Traverse permission PolicySet, protected object policy, and authorization PolicySet. They have not completely translate the AM functions into XACML. Their experiences give hope that legacy access control systems could also be supported in XACML. Concurrent administration with authorization is not addressed in their system. As is the case with *Delegent*, however, concurrent administration with authorization is not addressed here as well.

Dhankhar et al. [32] enhance the XACML syntax for “locks” to support three types

of access control use cases: (1) ensuring exclusive access to globally available resources, (2) preventing access to a resource given a concurrent conflicting use of another resource (dynamic separation of duty constraints), and (3) preventing access to a resource given a history of conflicting access (Chinese Wall constraints) which current XACML does not support. They add a lock manager to the policy enforcement module and require that all globally accessible resource register with unique lock manager. Policy administration and session management is not addressed in their paper. Their locks are put in the XACML syntax while the locks are placed in the policy in this dissertation.

Concurrency control on XML data has been an active research area recently. Haustein et al. [41–43] introduce a data model called taDOM tree to allow fine-grained locking using a combination of node locks, navigation locks, and logical locks, which could be further investigated for future research.

Janicke et al. [39] propose a concurrent enforcement model for usage control (UCON) [61] policies. They model an enforcement mechanism as a reactive system in form of a Statechart which clearly separates between user, controller and system. While this technique enforces concurrency control based on static analysis of dependencies between policies, this dissertation resolves concurrency issues during the runtime of a system.

4.6 Summary

In this chapter I have presented an XACML profile the ARBAC. This XACML-ARBAC profile extends the OASIS XACML-RBAC profile in: introducing a new data type *Role*, using a XML file for use-to-role assignment, adding constraints for the administrative Role <PolicySet> and Permission <PolicySet>, and expanding the condition functions to check the pre-conditions of the administrative operations. To enforce the XACML-ARBAC profile and address concurrency issues, I have enhanced the existing XACML

reference implementation with two entities: the *Administrative PEP (A-PEP)* and the *Lock Manager*. The A-PEP competes for read-write locks for RBAC and ARBAC policies along with PDP of the access controller. The *Lock Manager* maintains the transactional consistency between simultaneous operations that the PDP requires to read policies and the A-PEP needs to modify policies. The birth and death processes of the access controller are formally specified and a default XACML-ARBAC profile is defined which contains a persistent *Super Role (SRole)* that may be invoked by a so called *Super User (SU)*. The *SU* is assigned to the *SRole*. The *SU* then is used to instantiate the stored policies and enforce the XACML-ARBAC profile.

Chapter 5: XACML-ADRBAC Profile and Its Enforcement Architecture

This chapter first introduces the administration and delegation model for an RBAC system; then proposes the role-based administration and delegation XACML profile (XACML-ADRBAC); finally, extends the enforcement architecture for XACML-ARBAC proposed in Chapter 4 to enforce the XACML-ADRBAC profile.

5.1 Introduction

The OASIS technical committee published the XACML v3.0 administration and delegation profile (XACML-Admin) working draft on April 16, 2009 [6] to support two use cases: (1) policy administration, and (2) dynamic delegation. The former controls the types of policies that individuals can create and modify, whereas the latter permits some users to create policies of limited duration to delegate selected capabilities to others. The delegation model used in the XACML-Admin profile is a discretionary access control (DAC) model. The profile only allows the owner of a permission to delegate it to a specific user, which is not scalable when permissions need to be delegated to a large number of users with the same job function. In many cases in which the delegator is not available, or is unable to perform the delegation, it is more convenient to have a third party, such as the administrator, initiating the delegation on behalf of the user. This profile also lacks the flexibility to allow delegators to delegate any subset of permissions assigned to him/her.

Separately, this profile does not have an enforcement mechanism. Enforcing administrative or delegation operations will update relevant policies that results in read-write

conflicts while the access controller attempts to evaluate a user's access request. Further, when an administrator or delegator attempts to revoke a permission granted to a user, the same user might still be exercising the permission to access a resource, which violates the system safety. In this chapter, I extend the XACML-Admin profile to include the use cases of: (1) role-based delegation extending the delegation framework of the XACML-admin profile, and (2) policy administration with or without delegation extending the use cases proposed in Chapter 4. Furthermore, I show how these extended use cases can be realized by extending the architecture implemented in Chapter 4 that retains the system safety by revoking permissions invalidated by policy updates. To provide the extra use cases and enforce delegation, I divide the access requests into three categories as follows:

1. Regular User Access Request: User requests access permission to a resource. This is the most common type of request made to an access control system.

2. Administrative Request: Administrator requests to modify a component of the system, such as changing privileges granted to a role, etc.

3. Delegation Request: Delegator (user or administrator) requests delegating one user's permissions to another.

The first two category requests have been successfully specified and enforced in Chapter 4. In this chapter, I propose a role-based administration and delegation model, in which the delegators (user or administrator) are assigned to a *delegable* role granted with a set of delegation permissions. Our delegation permissions are different from the permissions to access resources. The delegation permissions are semi-administrative in nature such as creating/deleting a *delegated* role (e.g., a role that can be assigned to a delegatee), granting/revoking permission to/from a delegated role, and assigning/removing a user to/from a delegated role. To provide multi-step delegation (e.g., to be able to delegate delegation

permission), the delegable role also can create another delegable role. This role-based approach is scalable because it facilitates permissions to be delegated to a large number of users that may want to be delegatees of the same permission set. It is flexible because it allows the delegators to delegate any subset of the permissions assigned to him/her and modify the delegated permissions in case of needed.

5.2 Administration and Delegation Model

In order to cover the XACML-Admin profile [6], this dissertation adds the PBDM [80] delegation model to the ARBAC model. This dissertation partitions roles into regular roles (RR), delegable roles (DBR), delegated roles (DR) (which are called delegation roles in PBDM), and administrative roles (AR). A delegable role can be delegated to other roles or users by creating delegated roles or delegable roles (for multi-step delegation purposes). To support discretionary access control (DAC), users that are assigned to a regular role are also assigned to the delegable role based on it. Administrators can be assigned to the delegable roles. The delegated roles are the roles assigned to the delegatees created by the delegable roles. This separation induces a partition of $U2R$ and $R2P$. $U2R$ is separated into user-to-regular role assignment (U2RR), user-to-delegable role assignment (U2DBR), user-to-delegated role assignment (U2DR), and user-to-administrative role assignment (U2AR). Similarly $R2P$ is separated into regular role-to-permission assignment (RR2P), delegable role-to-delegation permission assignment (DBR2DP), administrative role-to-administrative permission assignment (AR2AP) and delegated role-to-permission assignment (DR2P). Administrative permissions are different from the regular permissions. Delegation permissions are also different from regular permissions.

A delegable role cannot have any senior regular role if it is placed into the role hierarchy. In general, definition of RR, DBR, $U2RR$, $RR2P$, $U2DBR$, $DBR2P$, $\leq$, and making

changes to the delegable role hierarchy $\leq_B$ is the responsibility of security administrators. Definition of DR, $U2DR$, $DR2P$ and delegated role hierarchy ($\leq_D$) is the responsibility of users that can add a form of discretionary access control (DAC) to the ARBAC model. $DBR \mapsto (DR \times DR)$ is the local delegated role hierarchy created by the DBR.

Definition 4 (ADRBAC)**.** *Let $(U, O, A, R, P, \leq, U2R, R2P)$ be an RBAC model. An administrative and delegation RBAC model is a tuple $ADRBAC = (U, RR, DBR, DR, AR, AO, AA, DP, AP, P, U2RR, U2DBR, U2DR, U2AR, RR2P, DBR2DP, DR2P, AR2AP, \leq, \leq_B, \leq_D, \leq_A)$, where*

- *R=$RR \cup DBR \cup DR \cup AR$, where RR is a set of regular roles, DBR is a set of delegable roles, DR is a set of delegated roles, and AR is a set of administrative roles, with constraints: $RR \cap DR = \{\}$, $RR \cap DBR = \{\}$, $RR \cap AR = \{\}$, $DBR \cap DR = \{\}$, and $AR \cap DR = \{\}$;*

- *$AO = U \cup RR \cup DBR \cup U2RR \cup U2DBR \cup RR2P \cup DBR2DP \cup \leq \cup \leq_B$ is the set of administrative objects;*

- *AA is the set of administrative actions given in Table 3.1;*

- *$AP \subseteq (AO \times AA) \cup (AO \times AO \times AA)$ is the set of administrative permissions;*

- *DP is the set of delegation permissions give in Table 5.1;*

- *$P \subseteq (O \times A)$is the set of regular permissions ;*

- *$U2RR : U \mapsto 2^{RR}$ is the user-to-regular role assignment;*

- *$U2DBR : U \mapsto 2^{DBR}$ is the user-to-delegable role assignment;*

- *$U2DR : U \mapsto 2^{DR}$ is the user-to-delegated role assignment;*

- *$U2AR : U \mapsto 2^{AR}$ is the user-to-administrative role assignment;*

- $RR2P : RR \mapsto 2^P$ *is the regular role-to-permission assignment;*
- $DBR2DP : DBR \mapsto 2^{DP}$ *is the delegable role-to-permission assignment;*
- $DR2P : AR \mapsto 2^P$ *is the delegated role-to-permission assignment;*
- $AR2AP : AR \mapsto 2^{AP}$ *is the administrative role-to-administrative permission assignment;*
- $\leq \subseteq RR \times RR$ *is the regular role hierarchy;*
- $\leq_B \subseteq DBR \times DBR$ *is the delegable role hierarchy;*
- $DBR \mapsto (DR \times DR)$ *is the local delegated role hierarchy created by the* DBR*;*
- $\leq_A \subseteq AR \times AR$ *is the administrative role hierarchy;*
- $senior(r) : R \rightarrow 2^R$*: a function mapping a role to all its senior roles in role hierarchy;*
- $base(dbr) : DBR \rightarrow RR$*: a function mapping each delegable role to a single regular role on which it is based;*
- $\forall\, dbr \in DBR \cdot senior(dbr) \cap RR = \{\}$*: no delegable role has a senior regular role;*
- $\forall\, dr \in DR \cdot senior(dr) \cap RR = \{\} \wedge senior(dr) \cap DBR = \{\}$ *: no delegated role has a senior regular role or a delegable role;*

Similar to administrative operations, delegation operations change the configuration of an RBAC system. Any configuration change affects the running system state, which may demand session administrative actions (see Chapter 3) to be invoked. The interaction between session administrative actions and the delegation operations (defined in Table 5.1)

Table 5.1: Delegation operations

Operations	**Intuitive Meaning**
DelegateRole(u,dr)	Delegate dr to u
DeassignUser(u,dr)	Deassign u from dr
GrantPermission(dr,p)	Grant p to dr
RevokePermission(dr,p)	Revoke p from dr
AddRole(dr)	Add dr
DeleteRole(dr)	Delete dr
AddEdge(dr^c, dr^p)	Make dr^c as a child of dr^p
DeleteEdge(dr^c, dr^p)	Remove dr^c as a child of dr^p

needs to be specified in order to ensure the safety of the ADRBAC model. Because this dissertation treats delegation operations as semi-administrative operations, concurrency control requirements between the session administrative model and delegation model for an RBAC system extends the concurrency control requirements stated in Chapter 3: *(1) revoke activated role or delete active session immediately,* and *(2) delay administrative/delegation operations.* When a delegation operation modifies a role, it might affect other roles. For example, removing a permission from a delegated roles will affect the users granted to the delegated role and all roles senior to the delegated role. This dissertation can define the affected entities because of invoking a delegation operation using lock scope defined in Chapter 3. Algorithm 1 in Figure 5.1 shows this information for every delegation operation listed in Table 5.1, although this dissertation does not consider revoking delegations.

5.3 The XACML-ADRBAC Profile

For the XACML-ARDRBAC profile, this dissertation adds the values *delegatedRole, delegableRole* to the *roleType* attribute to distinguish *delegated roles, delegable roles* from *user roles (regular roles) and administrative roles*. This XACML-ARDRBAC profile uses all other primitive entities from the XACML-ARBAC profile. This dissertation specifies the

```
Algorithm 1: Compute affected entities
Input: delegateOp
Output: Return affected to PAP
1   switch delegateOp do
2     case DeleteUser(u)
3       affected:=u;
4     case DeleteRole(dr)
5       affected:=wScope(dr);
6     case DeassignUser(u,dr)
7       affected:=(rScope(dr),u);
8     case RevokePermission(dr,P)
9       affected:=wScope(dr);
10    case DeleteEdge(dr^c, dr^p)
11      affected:=wScope(dr^p);
12    otherwise
13      affected:=NULL;
14  return affected;
```

Figure 5.1: Compute affected entities of a delegation operation.

administration and delegation profile (XACML-ADRBAC) using the same machinery as the XACML-ARBAC profile, but with the following added constraints for delegable roles and delegated roles.

Constraining the Delegable Role <PolicySet>: The Role <PolicySet> of a delegable role must be a delegable Role <PolicySet> with the following constraints:

1. All role names that appear in the <Target> of the Role <PolicySet> should be delegable roles, with the *roleType* set to "delegableRole" as shown in line 8 of Policy 6.

2. The <PolicySetIdReference> contained in the Role <PolicySet> should point to a delegation Permission <PolicySet>, to be described shortly.

3. There should be exact one regular role matching this delegable role. This dissertation

adds a <BaseRole> element to link the delegable role to the regular role as shown in line 20 of Policy 6.

Constraining the Delegable Permission <PolicySet>: All permissions listed in a <PolicySet> of a delegable role must be delegation permissions as defined in Table 5.1. By enforcing the following constraints on the syntax used in a permission <PolicySet>, it can be ensured as a *delegable* Permission <PolicySet>.

1. The (<Action>, <Resource>) pair listed in <Rule> must form a delegation permission. That is, the <Action> attribute must be chosen from the operation names and the <Resource> attribute must be chosen from the operation parameters stated in Table 5.1.

2. This delegable role does not have any senior regular role.

Constraining the Delegated Role <PolicySet>: The Role <PolicySet> of a delegated role must be a delegated Role <PolicySet> with the following constraints:

1. All role names that appear in the <Target> of the Role <PolicySet> should be delegated roles. That is, their *roleType* should be set to "delegatedRole" as shown in line 30 of Policy 6 .

2. The <PolicySetIdReference> contained in the Role <PolicySet> should point to a delegated Permission <PolicySet> to be described shortly.

Constraining the Delegated Permission <PolicySet>: All permissions listed in a <PolicySet> of a delegated role must be a subset of permissions granted to the base role of the delegable role which created the delegated role. By enforcing the following constraints on the syntax used in a permission <PolicySet>, it can be ensured as a *delegated* Permission <PolicySet>.

Table 5.2: Extended functions applied in <Condition> in XACML-ADRBAC profile

Function	Intuitive Meaning
role-exist(r)	check the presence of the role r
inherited-by-assigned-role(r)	check if the role r is inherited by a role already assigned to the subject
inherit-assigned-role(r)	check if the role r inherits a role already assigned to the subject
role-assigned-exist(s,r)	check if the subject s is already assigned to the role r
permission-exist(r,p)	check if the role r has been already granted the permission p
role-has-children(r)	check if the role r has any children
role-has-parent(r)	check if the role r has any parent
role-is-assigned(r)	check if the role r is assigned or not
role-is-inherited-by(r1,r2)	check if r1 is inherited by r2
role-is-parent-of(r1,r2)	check if r1 is parent of r2
permission-in-permissionSet(p,P)	check if p is an element of permission set P

1. The permissions are a subset of permissions of the base role of the delegable role which creates the delegated role. This can be checked by using the extended function “permission-in-permissionSet(p,P)” stated in Table 5.3.

2. This delegated role has no senior regular or delegable role.

The XML file recoding the user-to-role assignment is changed as follows. For any delegable role, this dissertation adds a <BaseRole> element. For example, Mary is assigned to the “Manager” role and a delegable “ManagerDelegable” role which has a base “Manager” role. Alice, the administrator, is also assigned to the “ManagerDelegable” role. Notice that Alice is not assigned to the “Manager” role, implying that the delegable may not have all the permissions granted to the base role. Jack is assigned to the “ManagerDelegated” role. This assignment fragment is as follows:

```
1          <Subjects>
2           ...
3            <Subject SubjectId="Mary">
4               <Role> Manager</Role>
5              <Role>ManagerDelegable
6              <BaseRole>Manager</BaseRole></Role>
```

Table 5.3: Contrasting elements in XACML-ARBAC with XACML-ADRBAC

XACML-ARBAC	XACML-ADRBAC
user-role XML assignment constraints on administrative role PolicySet constraints on administrative permission PolicySet	user-role XML assignment constraints on administrative role PolicySet constraints on administrative permission PolicySet constraints on delegable role PolicySet constraints on delegable permission PolicySet constraints on delegated role PolicySet constraints on delegated permission PolicySet

```
7              </Roles>
8            </Subject>
9            <Subject SubjectId="Alice">
10             <Role>ManagerDelegable
11             <BaseRole>Manager</BaseRole></Role>
12             <Role>SSO</Role>
13             </Roles>
14           </Subject>
15            <Subject SubjectId="Jack">
16             <Roles> <Role>ManagerDelegated</Role>
17             </Roles>
18           </Subject>
19          ...
20         </Subjects>
```

This dissertation enlarges the set of XACML condition functions (listed in Table 5.3) in order to check the constraints added in this section. Specifically, I add "permission-in-permissionSet(p,P)" function to check if p is an element of permission set P. Table 5.3 contrasts the elements of the XACML-ARBAC profile with the XACML-ADRBAC profile.

5.3.1 An Example

This dissertation specifies proposed XACML-ADRBAC profiles in Policy 6 and Policy 7 for the example stated in Section 2.3. Policy 6 shows the Role <PolicySet> for "ManagerDelegable" and "ManagerDelegated" roles. Policy 7 shows the Permission <PolicySet> for "ManagerDelegable" and "ManagerDelegated" roles. For simplicity, this dissertation only lists "add a delegated role" permission in the "ManagerDelegable" <PolicySet>. Other permissions such as granting permission to the "ManagerDelegated" role can also be specified similarly. Because Mary is assigned to the "MangerDelegable" role, Mary can create the "ManagerDelegated" role, grant the "approve expense reports" permission to the "ManagerDelegated" role, and assign Jack to the "ManagerDelegated" role.

```
1 <PolicySet PolicySetId="RPS:ManagerDelegable:role"
     PolicyCombiningAlgId="policy-combining-algorithm:permit-
     overrides">
2      <Target>
3        <Subjects>
4          <Subject>
5            <SubjectMatch MatchId="role-equals">
6              <AttributeValue DataType="role">RPS:ManagerDelegable<
                 /AttributeValue>
7              <SubjectAttributeDesignator SubjectCategory="subject-
                 category:access-subject"
8              AttributeId="subject:subject-role-id" DataType="
                 delegableRole"/>
9            </SubjectMatch>
10         </Subject>
11       </Subjects>
```

```
12        <Resources>
13          <AnyResource/>
14        </Resources>
15        <Actions>
16          <AnyAction/>
17        </Actions>
18     </Target>
19      <PolicySetIdReference>PPS:MangerDelegable:role</
          PolicySetIdReference>
20     <BaseRole>Manager</BaseRole>
21     <RoleCreator>SSO</RoleCreator>
22</PolicySet>
23<PolicySet PolicySetId="RPS:ManagerDelegated:role"
     PolicyCombiningAlgId="policy-combining-algorithm:permit-
     overrides">
24      <Target>
25        <Subjects>
26         <Subject>
27            <SubjectMatch MatchId="role-equals">
28             <AttributeValue DataType="role">RPS:ManagerDelegated</
                  AttributeValue>
29              <SubjectAttributeDesignator SubjectCategory="subject-
                   category:access-subject"
30              AttributeId="subject:subject-role-id" DataType="
                   delegatedRole"/>
31            </SubjectMatch>
32          </Subject>
```

```
33        </Subjects>
34        <Resources>
35          <AnyResource/>
36        </Resources>
37       <Actions>
38          <AnyAction/>
39        </Actions>
40      </Target>
41      <PolicySetIdReference>PPS:ManagerDelegated:role</
          PolicySetIdReference>
42       <RoleCreator>ManagerDelegable</RoleCreator>
43</PolicySet>
```

Policy 6: An Example Role <PolicySet>.

```
1<PolicySet PolicySetId="PPS:ManagerDelegable:role"
   PolicyCombiningAlgId="policy-combining-algorithm:permit-
   overrides">
2   <Target>
3   ...
4   </Target>
5   <Policy PolicyId="Permissions:ManagerDelegable:role"
      RuleCombiningAlgId="rule-combining-algorithm:permit-
      overrides">
6     <Target>
7     ...
8     </Target>
```

```
9       <Rule RuleId="Permission:to:add:a:delegatedrole" Effect="
           Permit">
10        <Target>
11          <Subjects>
12            <AnySubject/>
13          </Subjects>
14         <Resources>
15            <Resource>
16              <ResourceMatch MatchId="function:string-equal">
17                <AttributeValue DataType="http://www.w3.org/2001/
                     XMLSchema#string">delegated role</
                     AttributeValue>
18                <ResourceAttributeDesignator AttributeId="
                     resource:resource-id" DataType="http://www.w3.
                     org/2001/XMLSchema#string"/>
19              </ResourceMatch>
20            </Resource>
21          </Resources>
22          <Actions>
23            <Action>
24              <ActionMatch MatchId="function:string-equal">
25                <AttributeValue DataType="http://www.w3.org/2001/
                     XMLSchema#string">add</AttributeValue>
26                <ActionAttributeDesignator AttributeId="
                     action:action-id" DataType="http://www.w3.org
                     /2001/XMLSchema#string"/>
27             </ActionMatch>
```

```
28             </Action>
29           </Actions>
30         </Target>
31         <Condition FunctionId=":function:not">
32           <Apply FunctionId="role-exist">
33             <ResourceAttributeDesignator AttributeId="
                  resource:new-role-id" DataType="delegatedRole"/>
34           </Apply>
35         </Condition>
36       </Rule>
37   </PolicySet>
38
39<PolicySet PolicySetId="PPS:ManagerDelegated:role"
     PolicyCombiningAlgId="policy-combining-algorithm:permit-
     overrides">
40     <Target>
41      ...
42     </Target>
43     <Policy PolicyId="Permissions:ManagerDelegated:role"
         RuleCombiningAlgId="rule-combining-algorithm:permit-
         overrides">
44       <Target>
45      ...
46       </Target>
47       <Rule RuleId="Permission:to:approve:expenseReports" Effect=
            "Permit">
48         <Target>
```

```
49          <Subjects>
50            <AnySubject/>
51          </Subjects>
52          <Resources>
53            <Resource>
54              <ResourceMatch MatchId="function:string-equal">
55                <AttributeValue DataType="http://www.w3.org/2001/
                     XMLSchema#string">expense reports</
                     AttributeValue>
56                <ResourceAttributeDesignator AttributeId="
                     resource:resource-id" DataType="http://www.w3.
                     org/2001/XMLSchema#string"/>
57              </ResourceMatch>
58            </Resource>
59          </Resources>
60          <Actions>
61            <Action>
62              <ActionMatch MatchId="function:string-equal">
63                <AttributeValue DataType="http://www.w3.org/2001/
                     XMLSchema#string">approve</AttributeValue>
64                <ActionAttributeDesignator AttributeId="
                     action:action-id" DataType="http://www.w3.org
                     /2001/XMLSchema#string"/>
65              </ActionMatch>
66            </Action>
67          </Actions>
68        </Target>
```

```
69        </Rule>
70</PolicySet>
```

Policy 7: An Example Permission <PolicySet>.

The above example not only allows Mary to delegate the “approve expense reports” permission to Jack, but also allows Mary or other users assigned to the “ManagerDelegable” role to delegate the “approve expense reports” permission to whomever is assigned to the “ManagerDelegated” role, thereby improving the scalability of delegating the same set of permissions to multiple users. Also, the XACML-ADRBAC profile allows users assigned to delegable roles to modify the permissions granted to a delegable role or a delegated role, thereby providing them with the flexibility of choosing the delegated permissions. For example, if Mary want to delegate “approve vacation requests” permission to Jack, Mary needs to grant the “approve vacation requests” permission to the “ManagerDelegated” role. The system only needs to update the “ManagerDelegated” permission <PolicySet>.

5.3.2 Role Reduction

Inspired by the OASIS XACML-admin profile, I add a <RoleCreator> element to insure the policy authority. As shown in Policy 6, the “ManagerDelegable” role is created by the SSO which is an administrative role, and the “ManagerDelegated” role is created by the “ManagerDelegable” role. In order to establish the lineage of policies, the role creator must be traced back to an administrative role. The role reduction only need apply to the Role <PolicySet>s. To summarize, the features of the XACML-ARBAC profile, OASIS XACML-Admin profile, and the XACML-ADRBAC profile are listed in Table 5.4.

Table 5.4: Comparison between XACML-ARBAC, XACML-Admin, and XACML-ADRBAC services

XACML-ARBAC	XACML-Admin	XACML-ADRBAC
role-base administration	policy administration discretionary delegation policy reduction	role-based administration discretionary delegation role based reduction role-based delegation flexible permission delegation

5.4 Enforcement Architecture

In order to enforce the proposed XACML-ADRBAC profile, this dissertation extends the XACML-ARBAC enforcement architecture as shown in Figure 5.2. For simplicity, this dissertation only draws the added interaction in Figure 4.1. The *n.a* shows the flow of the regular user access request, *n.b* shows the flow of the administrative request, and *n.c* shows the flow of the delegation request, where n is an integer that shows the stage of the control flow inside the enhanced XACML evaluation architecture.

The *PDP* evaluates the requests against the policies and renders one of {*permit, deny, indeterminate, notApplicable*} as the outcome of the authorization decision.

The *PEP* receives a regular request, and enforces the authorization decision from the PDP. If needed, terminates the user sessions required due to enforcing an administrative operation or delegation operation.

The *Administrative PEP (A-PEP)* receives an administrative request, returns a response to the administrator, and if needed, updates relevant policies as a consequence of enforcing the requested administrative operation. When a user is assigned to a role and revoked from a role, the A-PEP acts as a role enabler/disabler by invoking the appropriate administrative operation and updates the U2R mapping in an XML file. Details have been provided in Chapter 4.

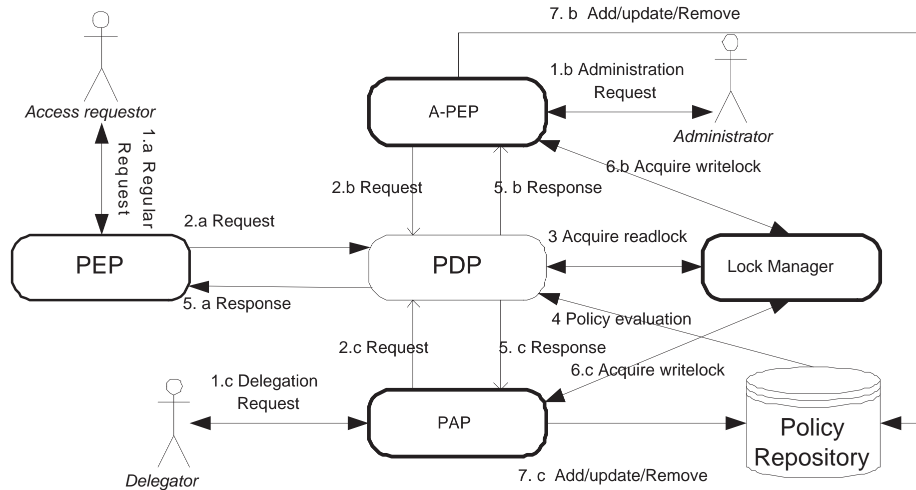

Figure 5.2: Extended XACML architecture for XACML-ADRBAC enforcement.

The *Policy administration point (PAP)* creates policies at authoring time and administrates (creates, updates, removes) policies during runtime. This dissertation mainly discusses the runtime enforcement of these operations. When a delegator sends a policy delegation request to the PAP, the PAP in turn send a request to the PDP , obtains the result and forwards the response to the delegator, and, if needed, updates relevant policies as a consequence of enforcing the requested delegation operation.

The *Lock Manager* provides the concurrency control necessary to maintain the transactional consistency between simultaneous operations that are generated by (1) the PDP reading policies in order to evaluate access request, (2) the A-PEP modifying policies to enforce administrative operations, and (3) the PAP modifying policies to enforce delegation operations.

5.4.1 Concurrency Control

Regular User Access Request: When an access request arrives at the PEP (flow 1.a), the PEP forwards the request to the PDP (flow 2.a). The PDP requests a read lock on the policy

that is found using the `target matching` algorithm (flow 3). Then the PDP evaluates the request (flow 4) and conveys the authorization decision to the PEP (flow 5.a). The details of evaluating authorization requests are described in Figure 4.2.

Administrative Request: When an administrative request arrives at the A-PEP (flow 1.b), the A-PEP forwards the request to the PDP (flow 2.b). The PDP requests a read lock on the policy that is found using the `target matching` algorithm (flow 3) from the lock manager. Then the PDP evaluates the request (flow 4) and conveys the authorization decision to the A-PEP (flow 5.b). The policy evaluation part (flows 3 and 4) is similar to that of evaluating a regular request. If the A-PEP gets a *permit* decision from the PDP, the A-PEP acquires a write lock on the policy (recall that administrative requests update XACML policies) that is to be updated (flow 6.b). After acquiring the write lock successfully, the A-PEP updates the policy to enforce the administrative operation (flow 7.b). The details of enforcement of administrative operations are described in Figure 4.3

Delegation Request: When a delegation request is submitted to the PAP (flow 1.c), the PAP forwards the request to the PDP for evaluation (flow 2.c). The PDP uses the same evaluation algorithm used to evaluate the access request (flows 3,4) (see Figure 4.2) and returns the decision to the PAP (flow 5.c). If the returned value received at the PAP is not a *permit*, the PAP conveys the decision to the delegator (user or administrator). Otherwise (e.g., the return value is *permit*), the PAP uses the algorithm shown in Figure 5.3 to enforce that decision.

As the algorithm states, if the decision is not a *permit*, the PAP returns that decision to the delegator (line 19). Otherwise, it acquires a write lock on the policy to be updated (line 3), calls the method `getAffected(delegateOp)` using the algorithm shown in Figure 5.1 to determine the parameters that are *affected* by the delegation operation (line 5). Then, the PAP sends a request to all PEPs to terminate user sessions that can be affected by enforcing the delegation operation (lines 6-8), so that updating a policy while these users

access permissions granted earlier do not render the access controller unsafe. Because the access controller cannot wait endlessly for those PEPs to confirm that the requested sessions have been terminated, the PAP sets up a timer (line 7). If all of these PEPs returned successful answers (lines 12-14), the PAP updates the policy/user-role assignment XML file to enforce the delegation, releases the write lock on the policy (line 16), and finally informs the delegator that the delegation is enforced (the *permit* decision). Conversely, if any PEP fails to return a positive answer when the timer expires, the delegation is denied, the requestor is informed, and the write lock is released.

5.5 Related Work

Recently, OASIS XACML v3.0 administration and delegation profile has been approved as an OASIS committee working draft [6]. It describes a profile to express administrative meta-policies which can control different types of policies that individuals can create and modify, and permits some users to create policies of limited duration to delegate selected capabilities to others. The delegation model used in this profile is a discretionary access control (DAC) model. Consequently, the profile constrains the owner of a permission to delegate it to a specific user, which is not scalable when permissions need to be delegated to a large number users with the same job function. In many cases in which the delegator is not available, or is unable to perform the delegation, it is more convenient to have a third party , such as the administrator, initiating the delegation on behalf of the user. This profile also lacks the support to allow delegators to delegate any subset of permissions assigned to him/her. This profile is also lack an enforcement mechanism.

PERMIS [24, 25] develops a role based access control infrastructure using X.509 [4] attribute certificates (ACs) to store the U2R relation. The PERMIS architecture includes a privilege allocator GUI tool and a bulk loader tool that allows administrators to construct

```
Algorithm 2: Enforcing Delegation Operations
Input: delegateRequest, PDPdecision
 /*PDP returns policy decision to PAP*/
Data: PEPList
Output: Return decision to the delegator
1   if PDPdecision==permit then
2     decision:=deny;
3     if AcquireLock(policy,write) then
4       if delegateOp is a (-) operation then
5         Affected:=getAffected(delegateOp);
6         forall PEP ∈ PEPList do
7           set(timer, value);
8           sendRequest(PEP,(Affected,killSession));
9         if expires(timer) then
10          acceptFlag:=ok;
11      forall PEP ∈ PEPList do
12        recv(PEP,(Affected, killsSession, NotOK));
13        acceptFlag:=reject;
14      if acceptFlag=ok then
15        modifyPolicy(policy, delegateRequest);
16        ReleaseLock(policy,write);
17        decision:=permit;
18  else
19    decision:=PDPdecision;
20  return (delegator, decision);
```

Figure 5.3: Enforcing delegation operations.

and sign ACs and store them in an LDAP directory to be used by the PERMIS decision engine. All access control decisions are driven by an authorization policy, which itself is stored in an X.509 attribute certificate. Authorization policies are written in DTDs. Later PERMIS uses an XML interface using Sun's reference implementation [8] and adds dynamic delegation of authority [26]. PERMIS assumes that privileges can be formulated as attributes and given to users. The dynamic delegation of authority is enacted via the issuing of credentials from one user to another. The delegation model used in this dissertation

is role-based which is more scalable and flexible. This dissertation uses an XML file for user-to-role assignment which can be directly updated by the A-PEP while PERMIS stores the ACs in an LDAP directory which cannot be directed updated by the access controller.

Many theoretical works on the subject of delegation exist. Delegation logic [52] is used to as a trust management engine with credentials proving that a request complies with a policy. Delegation logic lacks the explicit subject abstraction which is desired for attributed-based delegation and roble-based delegation. Li et al. [54] propose a role-based trust-manage framework (RT). In RT framework, role activation is delegated by issuing delegation credentials. Tamassia et al. [69] propose a model for delegation of authority in decentralized trust management systems which combines the RT framework [54] with cascading delegation. Bandmann et al. [17] introduce a constrained delegation model which controlled the possible *shapes* of delegation chains using constraints to restrict the capability at each step of delegation.

5.6 Summary

In this chapter, I have incorporated a delegation model into the ARBAC model, which adds *a delegable* role and a *delegated* role. The delegable role is granted a set of permissions to delegate any subset of permissions granted to the regular role which can be delegated in case of need. Second, I have extended the XACML-ARBAC profile to cover the XACML-Admin profile, which I refer to as the XACML-ADRBAC profile by adding appropriate syntax and constraints. Third, I have extended the XACML-ARBAC enforcement architecture described in Chapter 4 and specified the extra functionality required for the policy administration point (PAP) to enforce the extended XACML-ADRBAC profile. In order to achieve all these concurrently, I direct different types of access requests to different entities of the XACML runtime: the regular request to the policy enforcement point (PEP), the

administrative request to the administrative PEP (A-PEP), and the delegation request to the PAP. The *Lock Manager* functionalities are enhanced to enforce concurrency control necessary to maintain the transactional consistency between simultaneous operations among the policy decision point (PDP), the A-PEP and the PAP.

Chapter 6: Prototype Implementation and Evaluation

This chapter describes the prototype to enforce the extended XACML profile for ARBAC and concurrency control, and presents some experimental study of performance characteristics.

6.1 Implementation

To show the feasibility and performance of the proposed framework for enforcing the extended XACML profile for ARBAC and concurrency control, the author has implemented a prototype by augmenting Sun's reference implementation. In this prototype, the author choose to revoke all ongoing user sessions that conflict with the administrative operations immediately prior to enforcing the administrative operations.

6.1.1 Implementing the Birth and Death Process

The prototype boots up the access controller with a default administrative XACML policy, which permits the creation of *SU* and *SRole*, assigns *SU* to *SRole*, and grants the administrative permissions as shown in Table 3.1 to *SRole*. The access controller does not entertain any user requests during this initialization phase.

Immediately prior to the access controller's planned death, the *SU* sends a message to all active PEPs from the PEP-List maintained by the PDP (See Chapter 4) to notify that the access controller will stop services and request the PEPs to terminate all user sessions authorized by this access controller. After sending the messages, the access controller will

not process any more requests on behalf of any PEP including the A-PEP. After receiving the acknowledgement from all PEPs or the timer expires, the access controller signals the operating system to stop its services. In this prototype, the PEPs actions to terminate user sessions are similuated using method calls.

6.1.2 Implementing Condition Functions and Administrative Operations

As aforementioned, the condition functions in Sun's reference implementation are not sufficient for enforcing the XACML-ARBAC profile. I have made two enhancements in the implementation. In order to check for pre-conditions of every administrative operation, condition functions given in Table 4.2 are implemented by extending the function base provided by the existing reference implementation. In each function, I implement the `evaluate` method which is used to evaluate the condition. The input to the condition functions is provided using `attribute designators` that read information from the request context. In addition, the condition evaluation also requires access to policies, which is provided by initializing each function with a reference to the `policy finder` module of the PDP.

The second is a module used by the A-PEP to modify XACML policies when the PDP permits an administrative operation. This is achieved by using a `PolicyManager` that initializes and calls accessor and mutator methods to update the policies. The `AbstractPolicy` class in Sun's reference implementation has been extended with mutator methods as described in Table 6.1. To obtain and update user-to-role assignment, I use standard DOM APIs [9] to parse the XML file which contains the user-to-role assignment.

Table 6.1: Accessor and mutator methods used in the `PolicyManager`

Methods	Intuitive Meaning
getInstance(XMLNode)	create a instance of Policy or PolicySet object based on the DOM node
getChild(childId)	return a child of the instance of the Policy or PolicySet
addChild(childId)	add a child to the instance of the Policy or PolicySet
deleteChild(childId)	delete the child from the instance of the Policy or PolicySet
getChildren(XMLNode)	return all children of the Policy or PolicySet
setChildren(XMLNode)	set the child policy tree elements for this node
encode(outputStream)	encode the state of the Policy object to Policy Type XML reprentation

6.1.3 Implementing the Lock Manager

The *Lock Manager* implements a waiting queue with a vector, where index i indicates the ith access request, and serves all requests in the order of submission. The vector of a waiting process hold semaphores. When a process calls `AcquireLock()`, the semaphore has "memory" if a previous `ReleaseLock()` has been completed. The implementation uses a waiting thread that is awaken when its turn arises in the waiting queue. The semaphore data structure is as following:

```
class RClass extends Semaphore
// Semaphore for reader processes
{
  protected RClass(int i)
  { // Init a semaphore with i permits
    super(i);
  }
}
class WClass extends Semaphore
// Semaphore for writer processes
{
```

```
  protected WClass(int i)
  { // Init a semaphore with i permits
    super(i);
  }
}
```

The following code gives the general sketches for PDP evaluation and A-PEP updating:

```
int counter=0;
bool writing=false;
public void pdp() {
    if (!writing)
    readLock().acquire();
    try{
    /* PDP evaluates a policy */
      counter++;
     evaluate();
      }
    finally {
    readlock().release();
    counter--;
    }
   }
public void APEP(){
    if(counter==0 && !writing}
    writeLock().acquire();
    try{
```

```
    /* A-PEP updates a policy */
    writing=true;
    modify();
    }
    finally {
    writeLock().release();
    writing=false;
    }
}
```

Global variable *counter* records the number of PDP instances which are currently evaluating the policies; *writing* is true if and only if an A-PEP instance is updating the policies.

To get the lock scope of a role, I first build up the role hierarchy tree for all the roles. The hierarchy is built by traversing all the Permission <PolicySet> of the roles. If there is a "PolicySetIdReference" to another Role Permission <PolicySet>, then referred role is the child of the referring role. The lock scope for each role is computed based on the role hierarchy.

6.2 Performance Evaluation

The concurrency controller's *waiting queue* implementation slows down the access controller. If the number of administrative operations are small and the rate of each administrative request is low, then there is a minimal waiting time for the PDP to request and obtain read locks. However, when an administrative operation is submitted, the total service time becomes the sum of request generation time to the PDP, PDP evaluation time, response building time, lock acquisition time, communication time with affected PEPs, time to terminate sessions (optional), time to update a policy, and the time consumed to release the

locks. Thus, when an administrative request is submitted, it delays other user requests that have been submitted after that request. Hence the objective of this empirical study is to evaluate the overall effect on the access controller due to administrative requests.

In order to determine the timing overheads, this dissertation builds the role hierarchy given in Figure 3.1. As seen from Figure 3.1, the role hierarchy has eight (8) roles. Each role is granted with ten (10) permissions. Each role is assigned with fifty (50) users. After building this RBAC policy, the sizes of our disk resident Role <PolicySet>, Permission <PolicySet> and user-to-role assignment XML file became 12k, 122k, and 43k, respectively.

The current implementation does not have an elaborate PEP (although it has an A-PEP). Therefore, the PEP action is simulated using method calls where the PEPs take an equal time to terminate a session. The PDP, A-PEP, and all other (user) PEPs are placed on the same machine - a 3.4GHz Dual Core Windows XP machine with 1.5G memory. The elapse time of administrative operations is measured by calling the Java method *System.nanoTime()* [5]. Under the given conditions, I have experimented with executing the administrative operations. I have executed 8 out of the 10 administrative operations and measured their execution delays. Each experiment runs 20 times since the values converged with high confidence (95%) and the student-distribution [7] shows that. One simple operation is reported in Section 6.2.1. Two complex operations of removing some permissions from a role and removing a role from the role hierarchy, which requires executing a series of administrative operations. They are described in Section 6.2.2.

6.2.1 Simple Administrative Operations

I built the role hierarchy shown in Figure 3.1 using the simple "+" administrative operations. That activity took about 959 msecs to add 8 roles, 844 msecs to add 9 edges, and 711 msecs to grant 10 permissions per each of the 8 roles, and about 3384 msecs to assign 50

users to each role on average. The average time taken for each simple operation is between 68 to 120 msecs. Out of all these operations, Figure 6.1 shows the time taken for assigning different number of users to to a role. The time increases due to the growth of the U2R mapping. Further analysis shows that this is due to the fact that the time taken to parse the XML policy is proportional to the file size. This is a limitation because the DOM parser used by Sun's reference implementation acquires stack space as the XML file gets larger. A better parser would improve the performance.

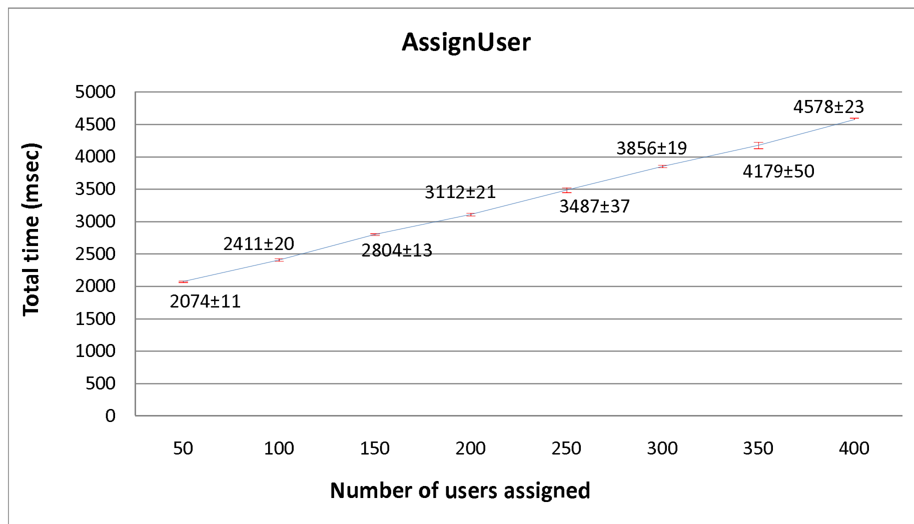

Figure 6.1: Total time taken to execute `AssignUser`.

6.2.2 Complex Administrative Operations

I further show the performance characteristics of removing some permissions from a role that has been activated by some users. The performance characteristics of removing a role from the role hierarchy while some users actively use that role are also studied.

Recall that the definition of *RevokePermission(r,(a,o))* removes the permission *(a,o)* from the role *r*, provided that no user actively uses *r*. Consequently, removing any permission, say *(a,o)* must be preceded by terminating all sessions that have activated any role in *wScope(r)*, locking all roles in *wScope(r)* so that no other session activates any of them,

and then finally revoking the permissions using the administrative operation *RevokePermission(r,(a,o))*.

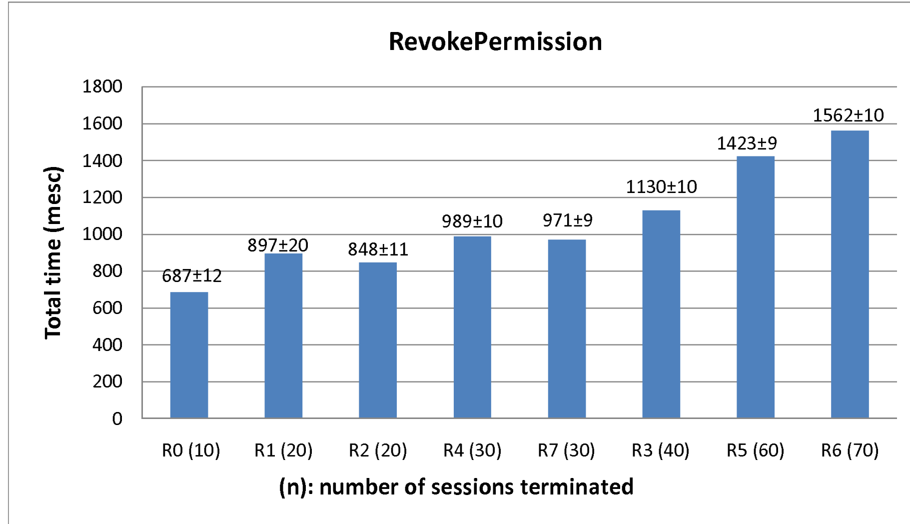

Figure 6.2: Total time taken to execute `RevokePermission`.

Table 6.2: Execution time for RevokePermission (msecs).

Role	# of sessions terminated	mean (mecs)	95% Confidence Interval
R0	10	687	12
R1	20	897	20
R2	20	848	11
R3	40	1130	10
R4	30	989	10
R5	60	1423	9
R6	70	1562	10
R7	30	971	9

In this experiment, I assume that there are ten (10) active sessions for each role. Because the role hierarchy, the number of sessions need to beterminated for each role are different (see Table 6.2.2). As Figure 6.2 shows, the time to remove a permission increases with the number of sessions that need to be terminated in order to lock all roles in *wScope(r)*. For example, revoking a permission from R1 requires terminating 20 sessions, taking a total of 897 msecs on average. Revoking a permission from R5 requires terminating 60 sessions,

taking 1562 msecs on average The observation is that revoking a permission from a role at the bottom of the hierarchy takes more time than at the top of the hierarchy because the permission may be used by more users assigned to the senior roles.

Recall the definition of the *DeleteRole(r)* which requires that for $r \in R$, no user has activated r in any session and the r is not related to any other roles in the role hierarchy. Therefore, before removing a role, it needs to ensure that these pre-conditions are satisfied by (1) terminating all sessions that have activated r, (2) removing all $(u, r) \in U2R$ for all $u \in U$, (3) removing all edges (r, r^p) or $(r^c, r) \in \leq$, and then (4) calling the administrative operation *DeleteRole(r)*. Therefore, the time to remove a role from the role hierarchy is the sum of time taken to do these individual operation. Accordingly, in order to determine the effect of the time taken to delete a role on the number of users permitted to use the role, the number of sessions activating the role and the number of edges connecting the role, I conducted three experiments.

In the first experiment, the number of users assigned to each role and the number of active sessions of each role is fixed. Figure 6.3 shows the total time taken to delete a role with a fixed number (50) of users permitted to use that role and fixed number of sessions (3) that activated the role given in Figure 3.1, with various number of edges to be deleted. Starting with Figure 3.1, deleting roles R6 and R7 requires deleting one edge, deleting roles R0 and R4 requires deleting 2 edges, deleting R1, R2, R3, and R5 requires deleting 3 edges. Figure 6.3 shows that the time taken to delete edges slightly increases with the number of edges that need to be deleted.

In the second experiment, the number of sessions activated is fixed by each user at 3, with various number of users permitted to activate the role. Here I assigned 10, 20, 30, 40, 50, 60, 70, and 80 users to R0, R1, R2, R3, R4, R5, R6, and R7, respectively. Figure 6.4 shows the total amount of time taken to delete each role. Figure 6.4 shows that the total time taken to delete a role slightly increases with the number of users that need to be de-assigned

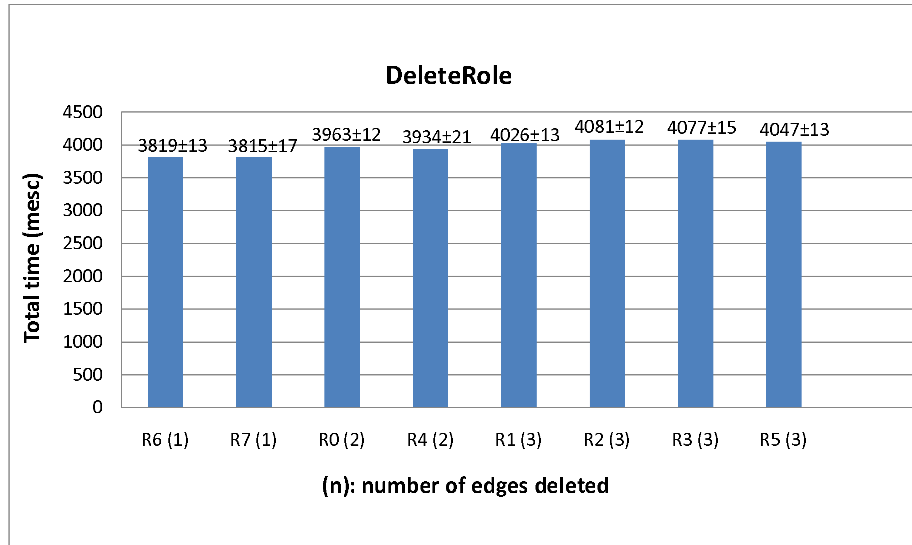

Figure 6.3: Effect of # edges on the time to remove a role.

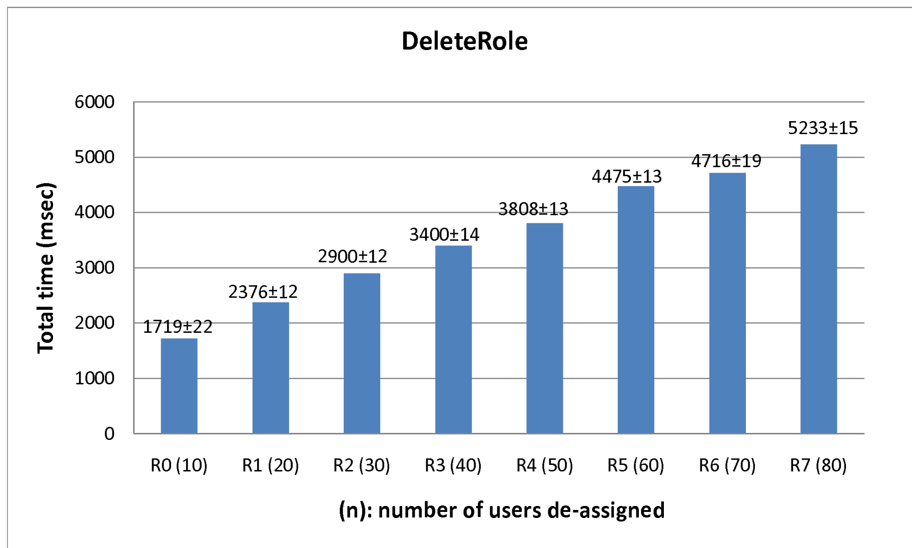

Figure 6.4: Effect of # users on the time to remove a role.

from the role.

In the last experiment, the number of users assigned to each role is fixed at 50, with various number of sessions where the role is activated. I activated 10, 20, 30, 40, 50, 60, 70, and 80 sessions by R0, R1, R2, R3, R4, R5, R6, and R7, respectively. As Figure 6.5 shows, the total time taken to remove a role slightly increases with the number of sessions

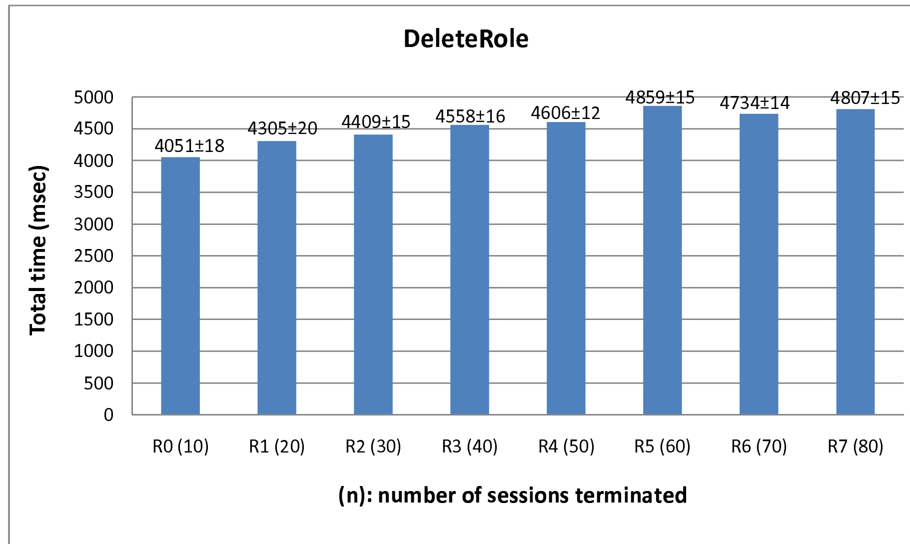

Figure 6.5: Effect of # sessions on the time to remove a role.

where the role is activated.

The results of these experiments show that the number of users permitted to use the role affects *DeleteRole* operation the most, followed by the number of sessions activating the role. The number of edges connecting the role has the lest effect on this operation.

The performance study indicates several facts. First, simple administrative operations execute very fast because they do not affect users' activities. Second, the complex operation, especially *DeleteRole* operation, takes more time because it requires executing a series of administrative operations. For example, in the last experiment, *DeleteRole(R3)* requires executing 50 *DeassignUser* operations, 3 *DeleteEdge* operations, 1 *DeleteRole* operation, and terminating 40 sessions. The amortized time for each operation is about 83 msecs which is reasonable. Fortunately, deleting a role in a system or organization does not happen often once the system is built.

6.3 Related Work

Up to now there has not been substantial research completed regarding performance study of access control policy evaluation. Bauer et al. [50] evaluate the efficiency of mobile access control policies deploying them on cell phones and PDAs. Their system specifies an access control policy as a statement in Intuitionistic predicate logic, and the access controller attempts to create a proof of the claim using known proof reduction techniques. They use two metrics to measure the efficiency of the access controller. The first is the amount of time it takes to either construct a complete proof, or if no complete proof can be found, to generate a list of choices to give to the user. These are used to refine the predicate by adding a user's input, and attempting to search for a proof again. Another metric they used is the number of subgoals investigated by the prover and the size of the knowledge base produced by forward chaining and path compression. Both of these metrics are related to the proof procedures, proof tactics and tacticals available to the prover, and are not directly related to the policy administration and concurrency control advocated in this dissertation.

Turkmen et al. [70] have tested three open source XACML implementations with different policy/request settings. The first implementation is Sun's reference implementation which is the one used in this dissertation. The second one is the XACMLight [12] which has been designed as a Web Service with the necessary components (e.g., the PDP and the PAP) for policy evaluation. It is a Java implementation and uses XMLBeans [14] to access the XML elements. The third implementation is the XACML Enterprise [10] which provides a simple API for PAP. It has been implemented in Java and uses Xerces [13] for XML operations. The cost of evaluation is given by two operations: loading the policy from disk to memory and then the actual evaluation of the submitted request against the loaded policies. In their reported results, XACML Enterprise performed best in terms of

evaluation time. It was the worst in policy loading. Sun's reference implementation had reasonable performance but it does not implement any of the optimizations of the other two engines. XACMLight had scalability problems with large number of policies.

XEngine [55] is a optimized XACML policy evaluation scheme. XEngine first converts a textual XACML policy to a numerical policy. Second, it normalizes the numerical policy with complex structures. Third, it converts the normalized numerical policy to tree data structures. Their experimental results show that XEngine is orders of magnitude more efficient than Sun's reference implementation. However, the performance experiments conducted in [55] are limited with the large number of rules case. Because two additional numericalization and normalization phases, the loading of policies is expected to take longer than the other engines.

6.4 Summary

In this chapter, I have implemented a prototype to enforce the XACML-ARBAC profile to show the feasibility and performance of the proposed framework for enforcing ARBAC policies with XACML. The prototype boots up with a default administrative XACML policy to initialize the access controller. The access controller does not entertain any requests during initialization phase. I have made two enhancements in the prototype. First, I have extended the function base to implement the expanded condition function list in Table 4.2. Second, I have implemented an administrative module for A-PEP by adding mutator methods to update the policies. I have conducted experiments to measure the execution delays of the administrative operations. The performance study shows that the solution has reconcilable performance characteristics and can be used for general policy management systems.

Chapter 7: Conclusion and Future Work

7.1 Conclusion

A session-aware enforcement framework is proposed in this dissertation to enforce administration and delegation RBAC policies with XACML in a distributed computing environment, such as Web Services. To address concurrency issues that arise between enforcing administrative policies and policy evaluation, a session-aware administrative model for RBAC is used to manage the interaction and potential conflicts between access control evaluation and administrative operations. Based on this model, this dissertation proposes using locks to handle concurrency control issues arising in enforcing the XACML-ARBAC profile. This dissertation defines the concept of a *lock scope* for a role, which captures the roles that would be adversely affected due to enforcing an administrative operation. To control such adverse affects, this dissertation makes some architectural enhancements to the current design of the XACML runtime. Specifically, an administrative policy enforcement point (A-PEP) is developed to compete for read-write locks for RBAC and ARBAC policies along with the policy decision point (PDP) of the access controller. A *Lock Manger* is developed to ensure the safety and integrity of policy management. This dissertation formally specifies the birth and death process of an access controller, and defines a default XACML-ARBAC profile that contains a persistent *Super Role (SRole)* that may be invoked by a so called *Super User (SU)*. The *SU* is assigned to the *SRole*. The *SU* then is used to instantiate the stored policies and enforce the XACML-ARBAC profile. For the planned death of the access controller, this dissertation has a special administrative *kill* method that

will request the active policy enforcement points (PEPs) to terminate all active user sessions authorized by this access controller. After obtaining agreement from the PEPs, the *SU* signals the operating system to shutdown the access controller. To demonstrate the feasibility of the framework, a prototype has been implemented to enforce the extended XACML-ARBAC profile. The experimental study shows that the solution exhibits only a small performance overhead and can be used for general policy management systems.

This dissertation further infues a role-based delegation model into the ARBAC model, which adds *a delegable* role and a *delegated* role. The delegable role is granted a set of permissions to delegate any subset of permissions granted to the regular role which can be delegated if required. The XACML-ARBAC profile is extended to cover the XACML-Admin profile, which is referenced here as the XACML-ADRBAC profile by adding appropriate syntax and constraints. This dissertation extends the XACML-ARBAC enforcement architecture and specifies the extra functionality required for the policy administration point (PAP) to enforce the extended XACML-ADRBAC profile. In order to achieve all of these simultaneously, this dissertation directs different types of access requests to different entities of the XACML runtime: the regular request to the PEP, the administrative request to the A-PEP, and the delegation request to the PAP. This dissertation uses a *Lock Manager* to enforce concurrency control necessary to maintain the transactional consistency between simultaneous operations among the PDP, the A-PEP and the PAP.

7.2 Future Work

This dissertation lays the groundwork for considerable future research and development of of integrating concurrency control and access control at runtime. First, the lock manager should be further investigated to provide more granular locking. The taDOM tree data

model in [41–43] allows fine-grained locking using a combination of node locks, navigation locks, and logical locks. The locks can be applied not only on the policy (role) level, but also on the rule level and even on any other components of the policy.
Second, constraints are an important aspect of role-based access control (RBAC) and are often argued to be one of the principal motivations behind RBAC [15]. The current XACML-RBAC profile lacks the syntax to specify constraints. The constraints are essential to capture envisioned security policies such as separation of duty, Chinese Wall [23] etc. Crampton et al. [28] enforce the separation of duty constraints by maintaining dynamic blacklists for each user. They propose eXtensible Access Control Constraint Language(XACCL), a new XML-based language designed to specify separation of duty constraints. Further, they propose to extend the XACML evaluation architecture (See Figure 2.1) with a constraint decision point, a constraint specification point and a blacklist repository to enforce the constraints. Adding constraints in XACML-RBAC profile and XACML-ARBAC profile is a candidate for future work.

Safety, liveness and fairness play important roles in system/software specification, development, and verification. Safety properties ensure that something undesirable never occurs. One example is that a PDP process and an A-PEP process should never access the same policy simultaneously. Liveness properties state that something desirable must eventually occur. One example is that the PDP process should eventually read the policy. Fairness properties state that if something is enabled often enough, then it must eventually occur. Fairness assumptions are often necessary to prove liveness properties. Verification of these properties in the access control model with concurrency control is a non-trivial topic. One of the very popular languages in the study of concurrency is Communicating Sequential Process (CSP) [44]. An option of future work is to use CSP and its model checker FDR [36] to study the interactions between PDP, A-PEP and PEP.

Emerging cloud computing services delivers new computing models for service providers and individual consumers. These models include infrastructure-as-a-service(IaaS), platform-as-a-service (PaaS), and software-as-a-service(SaaS) which enable novel IT business models such as resource-on-demand, pay-as-you-go, and utility-computing [16]. Clouds must leverage a unified identity and security infrastructure to enable flexible provisioning, yet enforce security policies through the cloud. The integration of federated identity management implemented in SAML [59] and federated policy management carried out in XACML becomes essential to provide necessary controls on the sensitive information. Federation serves a critical role for IaaS - compute cloud and storage cloud services, in terms of exchanging attributes around privacy, QoS, performance, resource usage and more, along with the respective policy enforcements. Study the emerging applications of XACML with these new service models is another option of future work.

Bibliography

Bibliography

[1] Core and hierarchical role based access control (RBAC) profile of XACML v2.0, http://docs.oasisopen.org/xacml/2.0/access_control-xacml-2.0-rbac-profile1-spec-os.pdf.

[2] Core specification: extensible access control markup language (XACML), http://www.oasis-open.org/committees/tc_home_php?wg_abbrev=xacml.

[3] Database trascation, http://en.wikipedia.org/wiki/database_transaction.

[4] *ISO 9594-8/ITU-T Rec. X. 509 (2001) The Directory: Public-key and attribute certificate frameworks*.

[5] Java 2 platform standard edition 5.0, http://java.sun.com/j2se/1.5.0/docs/api/.

[6] OASIS XACML v3.0 administration and delegation profile version 1.0, http://www.oasisopen.org.

[7] Student's t-distribution, http://en.wikipedia.org/wiki/student's_t-distribution.

[8] Sun's XACML implementation, http://sunxacml.sourceforge.net/.

[9] W3c recommendations, http://www.w3c.org/.

[10] XACML Enterprise, http://code.google.com/p/enterprise-java-xacml/.

[11] XACML v2.0 context schema, http://docs.oasisopen.org/xacml/2.0/access_control-xacml-2.0-context-schema-os.xsd.

[12] XACMLight, http://sourceforge.net/projects/xacmllight/.

[13] Xerces Java Parser, http://xerces.apache.org/xerces-j/.

[14] XMLBeans, http://xmlbeans.apache.org/.

[15] Gail-Joon Ahn and Ravi Sandhu. Role-based authorization constraints specification. *ACM Transactions on Information and Systems Security*, 3(4):207–226, 2000.

[16] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report 28, UC Berkeley EECS, 2009.

[17] O. Bandmann, M. Dam, and B. Sadighi. Constrained delegation. In *Proceedings of IEEE Symposium on Security and Privacy*, May 2002.

[18] E. Barka and R. Sandhu. Framework for role-based delegation models. In *Proceedings of 16th Annual Computer Security Application Conference (ACSAC 2000)*, December 2000.

[19] E. Barka and R. Sandhu. A role-based delegation model and some extensions. In *Proceedings of 23rd National Information Systems Security Conference (NISSC 2000)*, December 2000.

[20] J. Barkley, A. Cincotta, D. Ferraiolo, S. Gavrila, and D.R. Kuhn. Role based access control for the world wide web. In *Proceedings of 20th National Information System Security Conference*. NIST/NSA, 1997.

[21] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations and model. *Mitre Corp. Report No.M74-244, Bedford, Mass.*, 1975.

[22] E. Bertino, A. Squicciarini., I. Paloscia, and L. Martino. Ws-ac: A fine grained access control system for web services. *World Wide Web*, 9(2):143–171, 2006.

[23] David F.C. Brewer and Micheal J. Nash. The Chinese wall security policy. In *Proceedings of IEEE Symposium on Security and Privacy*, 1989.

[24] D.W. Chadwick and A. Otenko. The PERMIS X.509 Role Based Privilege Management Infrastructure. In *Proceedings of Seventh ACM Symposium On Access Control Models And Technologies (SACMAT 2002)*, June 2002.

[25] D.W. Chadwick, A. Otenko, and E. Ball. Implementing role based access controls using X.509 attribute certificates. *IEEE Internet Computing*, pages 62–69, March 2003.

[26] D.W. Chadwick, S. Otenko, and T.A. Nguyen. Adding Support to XACML for Dynamic Delegation of Authority in Multiple Domains. In *Proceedings of 10th IFIP TC-6 TC-11 Int Conf, CMS 2006*, pages 67–86, October 2006.

[27] J. Crampton. Understanding and developing role-based administrative models. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, November 2005.

[28] J. Crampton and L. Chen. Implementing RBAC and ABRA using XACML. In submission.

[29] J. Crampton and G. Loizou. Administrative scope and role hierarchy operations. In *Proceedings of Seventh ACM Symposium on Access Control Models and Technologies (SACMAT 2002)*, June 2002.

[30] J. Crampton and G. Loizou. Administrative scope: A foundation for role-based administrative models. *ACM Transactions on Information and Systems Security*, 6(2):201–231, 2003.

[31] E. Damiania, S. De Capitani di Vimeracti, X. Paraboschi, and P. Samrarti. Fine grained access control for SOAP e-services. In *Proceedings of 10th Internation Conference on World Wide Web (WWW)*, 2001.

[32] Vijayant Dhankhar, Saket Kaushik, and Duminda Wijesekera. XACML policies for exclusive resource usage. In *Proceedings of 21st Annual IFIP WG 11.3 Working Conference on Data and Applications Security (DBSec 07)*, July 2007.

[33] D. F. Ferraiolo, J. Barkley, and D.R. Kuhn. A role based access control model and reference implementation within a corporate intranet. *ACM Transactions on Information and System Security*, 1(2):201–231, February 1999.

[34] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. Richard Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security*, 4(3):224–274, August 2001.

[35] D. F. Ferraioloand, R. Chandramouli, Gail-Joon Ahn, and Serban I. Gavrila. The role control center: features and case studies. In *Proceedings of the Eighth ACM symposium on Access control models and technologies (SACMAT 2003)*, June 2003.

[36] Paul Gardiner, Michael Goldsmit, Jason Hulance, David Jackson, A. W. Roscoe, and Bryan Scattergood. *FDR2 User Manual*. Formal Systems Ltd., fifth edition, 2000.

[37] M. Gaseer and E. McDermott. An architecture for practical delegation in a distributed system. In *Proceedings of IEEE Symposium on Security and Privacy*, May 1990.

[38] S. Gavrila and J. Barkley. Formal specification for role based access control user/role and role/role relationship management. In *Proceeding of Third ACM Workshop on Role Based Access Control*, 1998.

[39] F. Siewe H. Janicke, A. Cau and H. Zedan. Concurrent enforcement of usage control polices. In *Proceedings of IEEE Workshop on Policies for Distributed Systems and Networks (POLICY)*, July 2008.

[40] M. H. Harrison, W. L. Ruzzo, and J. D. Ullman. Protection in operating systems. *Communication of ACM*, 19(8), 1976.

[41] M. Haustein and T. Härder. tadom: A tailored synchronization concept with tunable lock granularity for the dom api. In *Proceedings of ADBIS*, pages 88–102, 2003.

[42] M. Haustein and T. Härder. Optimizing lock protocols for native xml processing. *Data and Knowledge Engineering*, 65(1), 2008.

[43] M. Haustein, T. Härder, and K. Luttenberger. Contest of xml lock protocols. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 1069–1080, 2006.

[44] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International Series in Computer Science, UK, 1995.

[45] Günter Karjoth. Access control with IBM tivoli access manager. *Transactions on Information and Systems Security*, 6(2):232–257, 2003.

[46] Günter Karjoth, Andreas Schade, and Els Van Herreweghen. Implementing acl-based policies in xacml. In *ACSAC '08: Proceedings of the 2008 Annual Computer Security Applications Conference*, pages 183–192, December 2008.

[47] Alexl Kern. Advanced features for enterprise-wide role-based access control. In *Proceedings of the 18th Annual Computer Security Applications Conference*, December 2002.

[48] Axel Kern, Andreas Schaad, and Jonathan Moffett. An administration concept for the enterprise role-based access control model. In *Proceedings of the Eighth ACM symposium on Access control models and technologies(SACMAT 2003)*, June 2003.

[49] H. Korth and A. Silberschatz. *Database System Concepts*. McGraw-Hill, Inc, 1991.

[50] S. Garriss L. Bauer and M. K. Reiter. Efficient proving for practical distributed access-control systems. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, 2007.

[51] Doug Lea. *Concurrent Programming in Java Design Principles and Patterns*. Addiaon-Wesley, 2000.

[52] N. Li, B. Grosof, and J. Feigenbaum. A practiacally implementale and tractable delegation logic. In *Proceedings of IEEE Symposium on Security and Privacy*, 2000.

[53] N. Li and Z. Mao. Administration in role based access control. In *Proceedings of ACM Symposium on InformAtion, Computer and Communications Security (ASIACCS)*, March 2007.

[54] N. Li, J. Mitchell, and W. Winsborough. Design of a role-based trust-management framework. In *Proceedings of IEEE Symposium on Security and Privacy*, May 2002.

[55] Alex X. Liu, Fei Chen, JeeHyun Hwang, and Tao Xie. Xengine: a fast and scalable xacml policy evaluation engine. *SIGMETRICS Perform. Eval. Rev.*, 36(1):265–276, 2008.

[56] Andrew D. Marshall. A financial institution's legacy mainframe access control system in light of the proposed nist rbac standard. In *ACSAC '02: Proceedings of the 18th Annual Computer Security Applications Conference*, pages 382–390, December 2002.

[57] J. D. Moffett. *Delegation of Authority Using Domain Based Access Rules*. PhD thesis, Department of Computing, Imperial College, 1990.

[58] N. Nagaratnam and D. Lea. Secure delegation for distributed object environments. In *Proceedings of USENIX Conference on Objected Oriented Technology and Systems*, April 1998.

[59] OASIS. *Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML)*.

[60] S. OH and R. Sandhu. A model for role admininstration using organization structure. In *Proceedings of Seventh ACM Symposium on Access Control Models and Technologies (SACMAT 2002)*, June 2002.

[61] J. Park and R. Sandhu. The $\mathrm{UCON}_{\mathrm{abc}}$ usage control model. *ACM Transactions on Information and Systems Security*, 7(1):128–174, February 2004.

[62] R. Sandhu and V. Bhamidipati. The ASCAA principles for next-generation role-based access control. In *Proceedings of 3rd International Conference on Availability, Reliability and Security (ARES)*, June 2008.

[63] R. Sandhu, V. Bhamidipati, and Q. Munawer. The ARBAC97 model for role-based administration of roles. *ACM Transactions on Information and Systems Security*, 2(1):105–135, 1999.

[64] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[65] Andreas Schaad, Jonathan Moffett, and Jeremy Jacob. The role-based access control system of a european bank: a case study and discussion. In *Proceedings of the Sixth ACM symposium on Access control models and technologies (SACMAT 2001)*, pages 3–9, June 2001.

[66] L. Seitz, E. Rissanen, T. Sandholm, B. Sadighi, and O. Mulmo. Policy administration control and delegation using XACML and delegent. In *Proceedings of 6th IEEE/ACM International Workshop on Grid Computing*, November 2005.

[67] S.OH, R. Sandhu, and X. Zhang. An effective role administration model using organization structure. *ACM Transactions on Information and Systems Security*, 9(2):113–137, 2006.

[68] J. M. Spivey. *The Z Notation: a reference manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.

[69] R. Tamassia, D. Yao, and W. Winsborough. Role-based cascaded delegation. In *Proceedings of Ninth ACM Symposium On Access Control Models And Technologies (SACMAT 2004)*, June 2004.

[70] Fatih Turkmen and Bruno Crispo. Performance evaluation of xacml pdp implementations. In *SWS '08: Proceedings of the 2008 ACM workshop on Secure web services*, pages 37–44, Octorber 2008.

[71] Hai Wang and S. L. Osborn. An administrative model for role graphs. In *Proceedings of 17th Annual IFIP WG11.3 Working Conference on Database Security*, August 2003.

[72] Horst F. Wedde and Mario Lischka. Cooperative role-based administration. In *Proceedings of the eighth ACM symposium on Access control models and technologies (SACMAT 2003)*, pages 21–32, June 2003.

[73] Horst F. Wedde and Mario Lischka. Modular authorization and administration. *ACM Transactions on Information and Systems Security*, 7(3):363–391, August 2004.

[74] R. Wonohoesodo and Z. Tari. A role base access control for web services. In *Proceedings of IEEE International Conference on Service Computing (SCC'04)*, 2004.

[75] Min Xu and Duminda Wijesekera. A role-based xacml administration and delegation profile and its enforcement architecture. In *SWS '09: Proceedings of the 2009 ACM Workshop on Secure Web Services*, November 2009.

[76] Min Xu, Duminda Wijesekera, and Xinwen Zhang. Runtime adminiration of RBAC profile for XACML. *To appear in IEEE Transctions on Sevices Computing*, 2010.

[77] Min Xu, Duminda Wijesekera, Xinwen Zhang, and Deshan Cooray. Towards session-aware RBAC administration and enforcement with XACML. In *Proceedings of IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY)*, July 2009.

[78] E. Yuan and J. Tong. Attributed based access control (abac) for web services. In *ICWS '05: Proceedings of the IEEE International Conference on Web Services*, 2005.

[79] L. Zhang, G. Ahn, and B. Chu. A rule-based framework for role-based delegation. In *Proceedings of Sixth ACM Symposium On Access Control Models And Technologies (SACMAT 2001)*, June 2001.

[80] X. Zhang and R. Sandhu. PBDM: A flexible delegation model in RBAC. In *Proceedings of Eighth ACM Symposium On Access Control Models And Technologies (SACMAT 2003)*, June 2003.

Curriculum Vitae

Min Xu was born on April 14, 1978, in Jiangsu, P. R. China and is a citizen of P. R. China. He received the Bachelor of Engineering in Computer Science from Huazhong University of Science and Technology, Wuhan, China in 2000 and Master of Science in Computer Science from University of Nevada, Las Vegas in 2002. During 2002-2005, he was a software development engineer. Currently he is a Ph.D. candidate in the Department of Computer Science at George Mason University, Fairfax, Virginia, USA.