Runtime Administration of an RBAC Profile for XACML

Min Xu, Duminda Wijesekera, *Senior Member*, *IEEE*, and Xinwen Zhang, *Member*, *IEEE*

Abstract—The eXtensible Access Control Markup Language (XACML) is the *de facto* language to specify access control policies for web services. XACML has an RBAC profile (XACML-RBAC) to support role-based access control policies. We extend this profile with an administrative RBAC profile, which we refer to as the XACML-ARBAC profile. One of the advantages of doing so is to use policies based on RBAC model to administrate XACML-RBAC policies. Because using permissions granted by XACML-ARBAC policies alter XACML-RBAC policies, enforcing XACML-ARBAC polices requires some concurrency control within XACML access controller's runtime. In order to solve this concurrency problem, we propose a session-aware administrative model for RBAC, and enhance the XACML policy evaluation runtime using a locking mechanism. Experimental study shows reconcilable performance characteristics of our enhancements to Sun's XACML reference implementation.

Index Terms—RBAC, ARBAC, XACML, concurrency control, security.

1 Introduction

The eXtensible Access Control Markup Language (XACML) [3] is emerging as the *de facto* standard to specify access control policies for web services. Many policies that conform to traditional access control models such as discretionary [26], mandatory [10] and role-based (RBAC) [34], [21] 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 we refer to as the XACML-RBAC profile. 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 [33], [17], [18], [15], [31], [30]. Extending the XACML-RBAC profile to cover ARBAC models is the first fundamental contribution of this paper.

The RBAC model is based on the tenant that every role is granted a set of permissions necessary and sufficient to perform the *job functions* of any individual playing that role. Consequently, ARBAC models specify the access permissions required to perform the job function of the access control administrator such as creating/removing roles, changing permissions granted to roles, and assigning/revoking users to/from roles to a so called *administrative* role. One of the main issues in enforcing these *administrative* permissions is that they require changing access control policies—and in case of XACML, those that are compliant with the XACML-RBAC profile. That raises two issues. First, when an administrator exercises any administrative privilege (i.e., those 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 loose an already granted privilege by an XACML-RBAC policy. Therefore, enforcing an ARBAC policy would entail immediately changing the permissions granted to a user for a resource while the same user may still 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 that may result in unexpected change of permissions within ongoing sessions. Addressing concurrency issues of XACML administration is the second fundamental contribution of this paper.

Another auxiliary issue that has not been adequately addressed previously is the birth and death processes of the access controller itself—in our case, the XACML policy evaluation runtime. When the access controller is initialized, there is no default role or mechanism to properly activate the stored polices. When the access controller is asked to die, there must be a mechanism to *clean up* the system to ensure the safety property of the access controller. Formalizing the birth process is our third contribution of this paper.

In order to solve these problems, we propose a *session-aware* administrative model for RBAC. By this we mean that the policy enforcement point (PEP) maintains state information about user sessions. Based on this model, we propose using locks to handle concurrency control issues arising in enforcing the XACML-ARBAC profile. In using locks to enforce concurrency control, we define the concept of a *lock scope* for a role, that captures the roles that would be adversely affected due to enforcing an administrative

- *M. Xu and D. Wijesekera are with the Department of Computer Science, George Mason University, 4400 University Drive MSN 4A5, Fairfax, VA 22030. E-mail: {mxu, dwijesek}@gmu.edu.*
- *X. Zhang is with Samsung Information Systems America, Samsung R&D Center in Silicon Valley, 75 West Plumeria Drive, San Jose, CA 95134. E-mail: xinwen.z@samsung.com.*

Manuscript received 14 May 2009; revised 8 Oct. 2009; accepted 8 Feb. 2010; published online 29 Apr. 2010.
For information on obtaining reprints of this article, please send e-mail to: tsc@computer.org, and reference IEEECS Log Number TSCSI-2009-05-0126. Digital Object Identifier no. 10.1109/TSC.2010.27.

operation. To control such adverse effects, we make some architectural enhancements to the current design of the XACML runtime. Specifically, we develop an administrative policy enforcement point (A-PEP) that competes for read-write locks for RBAC and ARBAC polices along with the policy decision point (PDP) of the access controller. We also have a session administrator that terminates all user sessions that are affected due to a pending administrative policy change, immediately before its enforcement.

In order to standardize the birth and death processes of our enhanced XACML runtime, we define 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*. We then use this *SU* to instantiate the stored policies and enforce our XACML-ARBAC profile. For the planned death of the access controller, we have a special administrative *kill* method that will request active policy enforcement points (PEPs) to terminate all active user sessions authorized by this access controller. After obtaining agreement from the PEPs, *SU* signals the operating system to shut down the access controller. Finally, we demonstrate our solutions by extending Sun's XACML reference implementation engine [6] and report performance results of our implementation.

Preliminary results of this paper appeared in [39]. The current paper extends that work by formally specifying the administrative operations, enhancing concurrency control requirements of the session administration, developing the birth and death processes of the access controller, and conducting a more elaborate performance evaluation.

The rest of the paper is organized as follows: Section 2 briefly describes XACML and ARBAC essentials. Section 3 introduces our session administrative model for an RBAC system and concurrency control requirements. Section 4 presents our XACML-ARBAC profile and the architecture to enforce this profile in XACML. Section 5 describes our implementation. Section 6 presents some performance characteristics. Section 7 presents related work. Section 8 concludes this paper.

2 Preliminaries

2.1 XACML 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).

Standard XACML uses three basic elements in constructing access control policies: <Rule>, <Policy>, and <PolicySet>, and allows hierarchical nesting of them. An XACML <Rule> has two elements, a <Condition> and a <Target>, and an *Effect* attribute. The intuitive reading of an 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 [3] 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 [3] identifies four standard combining algorithms: *deny-override*, *permit-override*, *first-one-applicable*, and *only-one-applicable*. For example, the *deny-overrides* algorithm evaluates to *deny* if any applicable rule evaluates to *deny*.

<Target> specifies a set of predicates 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. We can check the pre-conditions of each administrative operation (so to be explained shortly) in the <Condition>.

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.

Fig. 1 shows 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 7-35), and *Rule2* (line 36). Line 1 of the policy indicates that the rule combining algorithm to be used is *permit-overrides*. Lines 2-6 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 8-28) 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 29-34) 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 *Rule 2* (line 36) and the rule combining algorithm of the policy (line 1), the request should be denied.

2.2 Sun's Reference Implementation

Fig. 2 shows the high-level architecture of Sun's XACML reference implementation [6]. In this architecture, the *Policy Administration Point (PAP)* is the entity that creates policies and policy sets; the *Policy Decision Point (PDP)* is the entity that evaluates policies and renders one of {*permit*, *deny*,

```
1 <Policy PolicyId="add:a:role"
   RuleCombiningAlgId="permit-overrides">
2  <Target>
3    <Subjects><AnySubject/></Subjects>
4    <Resources> <AnyResource/></Resources>
5    <Actions><AnyAction/></Actions>
6  </Target>
7  <Rule RuleId="Permission:to:add:a:role" Effect="Permit">
8    <Target>
9      <Subjects><AnySubject/></Subjects>
10     <Resources>
11      <Resource>
12       <ResourceMatch MatchId="string-equal">
13       <AttributeValue DataType="string">role
         </AttributeValue>
14       <ResourceAttributeDesignator
15         AttributeId="resource-id" DataType="string"/>
16       </ResourceMatch>
17      </Resource>
18     </Resources>
19     <Actions>
20       <Action>
21        <ActionMatch MatchId="string-equal">
           <AttributeValue DataType="string">AddRole
           </AttributeValue>
23         <ActionAttributeDesignator AttributeId="action-id"
24         DataType="string"/>
25         </ActionMatch>
26       </Action>
27    </Actions>
28   </Target>
29   <Condition FunctionId="not">
30     <Apply FunctionId="role-exist">
31     <ResourceAttributeDesignator AttributeId="new-role-id"
32       DataType="role"/>
33     </Apply>
34   </Condition>
35 </Rule>
36 <Rule RuleID="2" Effect="Deny"> </Rule>
37 </Policy>
```

Fig. 1. Example XACML policy.

indeterminate, notApplicable} as the authorization decision; the *Policy Enforcement Point (PEP)* is the entity that enforces the access control decision; and the *Context Handler* is the entity that converts native request to one that is in the XACML format (consisting of three components *Subject*, *Resource*, and *Action*) and converts authorization decisions in the XACML format to native formats.

The PAP creates policies at authoring time, e.g., by security administrators using some text editor. At an access control request time, a subject sends an access request to the PEP as shown in flow 2 of Fig. 2. 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). The PDP uses this information to find the access control policy applicable to the request, which 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 polices applicable to the request by using 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 [6] provides a set of APIs that understand the XACML syntax, and rules to process requests and manage attributes. But this implementation only provides a PDP for policy evaluation that can only read, but not modify any policies, which we need to enhance in order to enforce the administrative operations specified in ARBAC polices.

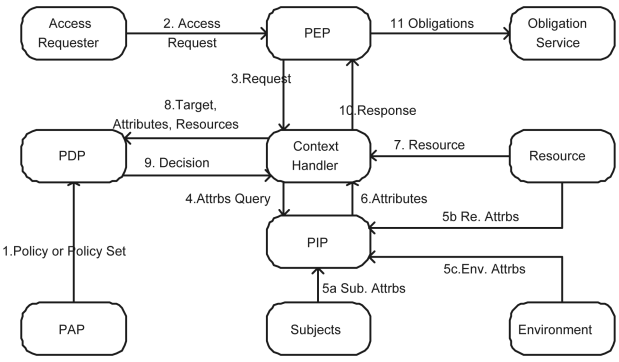

Fig. 2. XACML data flow diagram.

2.3 RBAC and ARBAC

We use 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. We use 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.

We base our work partially on ARBAC97 [33] and SARBAC [17], 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 submodels 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. An ARBAC model is defined as follows:

Definition 1 (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 1 including $+$ and $-$ operations.*
- *AR is a set of administrative roles.*
- *$AP \subseteq (AO \times AA) \cup (AO \times AO \times AA)$ 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.*

TABLE 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)

As defined, 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 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 preconditions and postconditions of these operations are specified in Section 2.4.

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.

2.4 Formal Specification of Administrative Operations

We formally specify suggested administrative operations in terms of preconditions and postconditions using the Z-notation [36]. As per Z-notation, a value of a data item before the execution of a command (so called prestate of a data structure) is denoted by a symbol, and its value after the execution of the operation (i.e., the so called poststate) is denoted by the same symbol followed by a *prime* (′).

- *AddUser(u)*: creates an RBAC user u.
  - *Precondition:* u is not already a member of the user data set.
    Formal Specification: $u \notin U$.
  - *Postcondition:* 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.
  - *Precondition:* u is already a member of the user data set and no roles are assigned to u.
    Formal Specification: $u \in U \wedge \nexists r \in R, M \subseteq R : U2R(u, M) \wedge r \in U$.
  - *Postcondition:* The user data set is updated.
    Formal Specification: $U' = U \setminus \{u\}$.
- *AddRole(r):* creates a new role r.
  - *Precondition:* r is not already a member of roles.
    Formal Specification: $r \notin R$.
  - *Postcondition:* The new role is added to the roles set R. $U2R$ and $R2P$ remain unchanged.
    Formal Specification: $RO' = R \cup \{r\} \wedge U2R' = U2R \wedge R2P' = R2P$.
- *DeleteRole(r):* deletes an existing role r from the roles data set.
  - *Precondition:* 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: $r \in R \wedge \nexists u \in U,\ M \subseteq R : U2R(u, M) \wedge r \in M \wedge \nexists r_1 \in R(r \leq r \vee r_1 \leq r)$.
  - *Postcondition:* 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.
  - *Precondition:* 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: $[u \in U \wedge r \in R] \wedge \nexists M \subseteq R[r \in M : U2R(u, M)] \wedge \nexists r_1 \in R[r_1 \geq r \wedge r_1 \in M \wedge U2R(u, M)]$.
  - *Postcondition:* $U2R$ is updated.
    Formal Specification: $[U2R(u, M) \rightarrow U2R' = U2R \setminus (u, M) \cup (u, M \cup \{r\})] \wedge [\nexists M \subseteq R\, U2R(u, M) \rightarrow U2R'(u, \{r\})]$.
- *DeassignUser(u,r):* deassigns the user u from the role r.
  - *Precondition:* 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: $u \in U \wedge r \in R, \exists M \subseteq R : r \in M \wedge U2R(u, M)$.
  - *Postcondition:* 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.
  - *Precondition:* The role r is a member of the roles data set and (a,o) is a permission.
    Formal Specification: $r \in R \wedge (a, o) \in P$.
  - *Postcondition:* 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.
  - *Precondition:* The role r is a member of the roles data set and (a,o) is assigned to r.
    Formal Specification: $r \in R \wedge \exists N \subseteq P : R2P(r, N\{(a, o)\})$
  - *Postcondition:* 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.
  - *Precondition:* 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: $r^c, r^p \in R \wedge r^p \not\leq r^c \wedge r^c \not\leq r^p \wedge r^p \neq SRole \wedge r^c \neq SRole \wedge [\neg \exists r, s \in R(r^c < r < r^P \wedge r^p < s < r^c)]$.

- *Postcondition:* 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 $r^c < r^p$.
  - *Precondition:* r^c and r^p are members of the roles data set and r^p is a parent of r^c.
    Formal Specification: $r^c, r^p \in R \wedge [r^c < r^p]$.
  - *Postcondition:* The relationship $r^c < r^p$ is deleted.
    Formal Specification: $<'=< \setminus\{(r^c, r^P)\}$.

3 Session Administrative Model

3.1 RBAC Session Administration

The RBAC96 [34] and NIST RBAC [21] 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 at the same time. 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 [21]. However, they are not included in existing ARBAC modes [33], [17], [18], [15], [31], [30]. Our sessions are different from transactions in database management systems [8]. A database transaction must be atomic, consistent, isolated and durable, i.e., satisfy so called ACID properties. We assume that the session management is handled by the PEP. When an administrator executes some administrative operation, the state of the RBAC system, as modeled by the U2R, R2P and $\leq$ relations may change. Consequently, in order to maintain consistency over all sessions, some permissions and roles granted to users may need to be terminated, or the enforcement of the administrative operation may have to be delayed. At any given state, different instances of the same role should be granted the same set of permissions. We choose to implement the former, realizing that this has limitations. For example, when a user may lose his/her role or permission to an operation that cannot be preempted. In order to specify appropriate ARBAC policies for an RBAC system, first we define an administrative model for session management as follows:

Definition 2 (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.*

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.
  - *Precondition:* u is already a member of the user data set and s is not a member of $ACTIVE-S$.
    Formal Specification: $u \in U \wedge s \notin ACTIVE-S$.
  - *Postcondition:* $U2S$ 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.
  - *Precondition:* (u, s) is an entry of $U2S$.
    Formal Specification: $(u, s) \in U2S \wedge s \in ACTIVE-S$.
  - *Postcondition:* $U2S$ 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.
  - *Precondition:* (u, s) is a member of $U2S$ and (u, r) is a member of $U2R$.
    Formal Specification: $(u, s) \in U2S \wedge (u, r) \in U2R \wedge s \in ACTIVE-S$.
  - *Postcondition:* $S2R$ 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.
  - *Precondition:* (s, r) is a member of $S2R$.
    Formal Specification: $(s, r) \in S2R$.
  - *Postcondition:* $S2R$ is updated.
    $S2R' = S2R \setminus \{s \mapsto S2R(s)\} \cup \{s \mapsto (S2R(s) \setminus \{r\})\}$.

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 mentioned, any configuration change affects the running system state, and may require session administrative actions. The interaction between session administrative actions and system administrative operations (i.e., the ARBAC operations defined in Section 2.3)

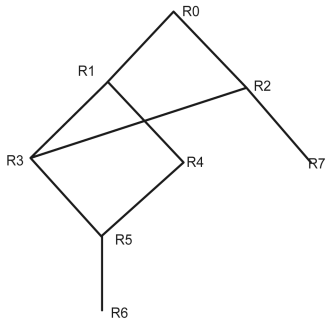

Fig. 3. Example role hierarchy.

needs to be specified for a safe and complete ARBAC model. As one of the major contributions of this paper, we identify 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 2.4. Then, in order to retain the consistency, we either delete the affected session, or deassign the user from the affected role. 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 user's active session 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 $RBAC$, 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 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 PDP not only needs to manage current active sessions, and 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 PDP needs to prevent users from activating them in new sessions. To do this, the PDP will lock the affected roles to ensure the safety property of the access controller. 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.

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;
```

Fig. 4. Compute entities affected due to an administrative action.

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. We 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. Conversely, if role r is to lose permissions due to an administrative operation, then all roles senior to r, that is the *write scope* of r must not be allowed to be active. For example in Fig. 3, 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}.

We can define the affected entities because of invoking an administrative operation using lock scope. Algorithm 1 in Fig. 4 shows this information for every administrative operation list in Table 1.

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 Fig. 3 affects all the sessions where R0, R1, R2 and/or R3 are activated. The affected entities are those in *wScope(R3)*.

4 XACML-ARBAC Profile and the Enforcement Architecture

In this section, we present 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, our XACML-ARBAC profile is also an XACML-RBAC profile. We first describe the XACML-RBAC profile and then present our extensions for ARBAC. We then show how the XACML-ARBAC compliant policies can be used to *administrate* the XACML-RBAC policies by executing administrative operations. Finally, we present the architecture to enforce the XACML-ARBAC profile.

4.1 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. But roles are expressed as <Subject> attributes or <Resource> attributes. This profile also 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 (i.e., permissions, $U2R$ and $R2P$ mappings, and role hierarchy $\leq$), and 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> to other Permission <PolicySet>s. Stated <PolicySetIdReference>s can be used to inherit permissions of a junior role. Currently, this is the only way to specify the role inheritance 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 the 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 XACML-ARBAC Profile

In the OASIS XACML-RBAC profile, roles are defined as attributes of subjects and resources. We enhance the XACML syntax by introducing a new data type *Role*. As our implementation needs to distinguish *administrative roles* from *user roles*, we introduce a *roleType* attribute that can take value from {*userRole*, *adminRole*}. We use 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. We use an XML file to maintain all user-to-role assignments in the policy repository as the follows:

```
<Subjects>
  <Subject SubjectId="Alice">
   <Roles> <Role>SSO </Role></Roles>
  </Subject>
  <Subject SubjectId="Bob">
   <Roles> <Role>ManagerABC</Role>
   </Roles>
  </Subject>
 </Subjects>
```

The PDP gets all the roles that a user can invoke by querying this XML file. Although we could have maintained the user-to-role assignment as a Role Assignment <PolicySet>, the reason we do not do so is that the current XACML reference implementation does not answer a query, such as *What are the roles assigned to Alice?*. Using this extra XML file, we specify administrative policies using the same machinery as the XACML-RBAC profile, but 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>, we ensure 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 2 (explained shortly).
2. The (<Action>, <Resource>) pair listed in <Rule> must be an AP. That is, the actions must be chosen from operations listed in Table 1.

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

1. All role names that appear in the <Target> of the Role <PolicySet> should be administrative roles.
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 1.

Sun's reference implementation 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 implementation [6] are not capable of checking the conditions for

TABLE 2
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 given role r is inherited by a role already assigned to the subject
inherit-assigned-role(r)	check if the given role 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 given role has any children
role-has-parent(r)	check if the given role has any parent
role-is-assigned(r)	check if the give role 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

most administrative operations. For example, to add a role r into the system, the access controller needs to check if r is already defined. Consequently, we add a new set of condition functions listed in Table 2 to support all possible conditional checks for administrative operations. These condition functions are internal auxiliary functions which do not affect the system state.

4.3 Enforcing the XACML-ARBAC Profile

In order to enforce our XACML-ARBAC profile, we enhance the existing XACML reference implementation with the two entities shown with bold borders in Fig. 5 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 polices as a consequence of enforcing the requested administrative operation. The A-PEP functions as a *Role Enabling Authority*. Consequently, 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, 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

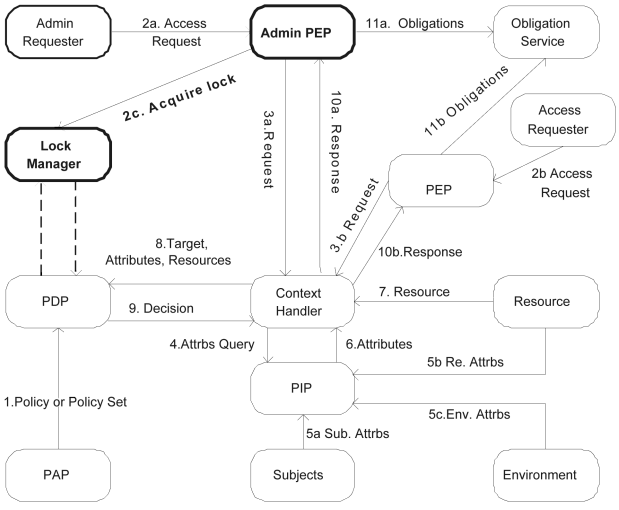

Fig. 5. Extended XACML architecture for XACML-ARBAC enforcement.

policies in order to evaluate them and the A-PEP needs to modify polices to enforce administrative operations.

4.4 Birth and Death Processes

In our new design, 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. Here we simplify the administrative RBAC system with only a single administrative role. Consequently, the resulting U2R and R2P updates are precisely specified in the following preconditions and postconditions.

- *Precondition:* U, R, AR, U2R, and R2P are empty. *Formal Specification:* $U = \emptyset \wedge R = \emptyset \wedge AR = \emptyset \wedge U2R = \emptyset \wedge R2P = \emptyset$.
- *Postcondition:* SU is the only member of the users data set and SROLE is the only member of the 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 1.

As seen from the postconditions, after the initialization phase finishes, the super user SU—the only user in the system endowed with $SRole$'s permissions—the administrative permissions described in Table 1. Also as specified, the $SRole$ does not have permissions to delete SU, nor deassign SU from the $SRole$. Consequently, permissions granted to the $SRole$ remain unalterable and the $SRole$ has no relation with other roles through $\leq$, as formally specified in the $AddEdge$ administrative operation in Section 2.4.

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 die, 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

```
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;
```

Fig. 6. PDP evaluation algorithm.

expires, SU signals the operating system to shutdown the access controller. Here we assume 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 postconditions.

Postcondition: $ACTIVE-S$, $U2S$, $S2R$, $actRole$, and $actPerms$ are empty. *Formal Specification:* $ACTIVE-S = \emptyset \wedge U2S = \emptyset \wedge S2R = \emptyset \wedge actRole = \emptyset \wedge actPerms = \emptyset$.

4.5 Concurrency Control

When a nonadministrative request arrives at the PDP, the PDP requests a read lock on the policy that is found using the *target matching* algorithm. In case of an administrative request, the policy evaluation part is similar to the nonadministrative 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 requests update XACML policies) that is to be updated. We now describe the details of these steps.

4.5.1 Evaluating Authorization Requests

Sun's reference implementation does not alter any XACML policies, and it uses the policy evaluation algorithm explained in [3]. As our enhancements update policies, this evaluation algorithm needs to be protected by a semaphore. Consequently, when a nonadministrative 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, evaluates the request using the existing XACML policy evaluation algorithm, updates the runtime PEP-List (the list of PEPs), and finally releases the read lock on the policy and sends the response back through the requesting PEP, 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 Fig. 6.

4.5.2 Enforcing Administrative Operations

When an administrative request is submitted to the A-PEP, the A-PEP forwards the request to the PDP for evaluation.

```
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);
```

Fig. 7. Enforcing administrative operations.

The PDP uses the same evaluation algorithm used to evaluate the nonadministrative request (see Fig. 6) 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 (i.e., the return value is *permit*), the A-PEP uses the algorithm shown in Fig. 7 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 (line 3), and calls the method `getAffected(adminOp)` using the algorithm shown in Fig. 4 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.

4.6 Lock Manager

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. The *Lock Manager* maintains read/write locks on policies. Because the polices are role-based, the locks are placed on the roles. We implement locking with two atomic operations `AcquireLock(role, read/write)`, `ReleaseLock (role, read/write)` and an `Attempt-Lock(role, ReadLock, WriteLock)` operation. The method prevents dead-locks and circular locks because all roles that we

TABLE 3
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

maintain are in an ordered list and locks are acquired in the same (increasing) order [28].

5 Prototype Implementation

To show the feasibility and performance of our framework, we have implemented a prototype to enforce the extended XACML profile for ARBAC and concurrency control by augmenting Sun's XACML reference implementation [6]. In this prototype, we revoke the all ongoing user sessions that conflict with administrative operations immediately prior to enforcing them.

5.1 Implementing the Birth and Death Process

Our 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 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 Section 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, we simulate the PEPs actions to terminate user sessions using method calls.

5.2 Implementing Condition Functions and Administrative Operations

As stated, the condition functions in Sun's reference implementation are not sufficient for enforcing the XACML-ARBAC profile. We have made two enhancements in our implementation. In order to check for preconditions of every administrative operation, condition functions given in Table 2 are implemented by extending the function base provided by the existing reference implementation. In each function, we implement the `evaluate` method that is used to evaluate the condition. The input to the condition 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 3. To obtain and update user-to-role assignment, we use standard DOM APIs [7] to parse the XML file containing user-to-role assignments.

5.3 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 submitted requests. The vector of a waiting process hold semaphores. When a process calls `AcquireLock()`, the semaphore has "memory" if a previous `ReleaseLock()` has been made. Our implementation uses a waiting thread that is awaken when its turn arises in the waiting queue.

6 Performance Evaluation

The concurrency controller's *waiting queue* implementation slows down the access controller. If the number of administrative operations are few and far between, 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, time to communicate with affected PEPs, time to terminate sessions (optional), time to update a policy, and the time taken to release the locks. Thus, when an administrative request is submitted, it delays other user requests that have been submitted after that request. Hence, our objective is to evaluate this overall effect on the access controller due to administrative requests.

In order to determine the timing overheads, we build the role hierarchy given in Fig. 3. As seen from Fig. 3, our role hierarchy has eight roles. We grant 10 permissions per each of these eight roles. We assign 50 users per role, and assume that there are 10 active user sessions per each role. After building this RBAC policy, the sizes of our disk resident Role <PolicySet>, Permission <PolicySet> and user-to-role assignment file became 12k, 122k, and 41k, respectively.

Our current implementation does not have an elaborate PEP (although we have an A-PEP). Therefore, we simulate the PEP action using method calls where the PEP takes an equal time to terminate a session. We also place the PDP, A-PEP, and all other (user) PEPs on the same machine—a

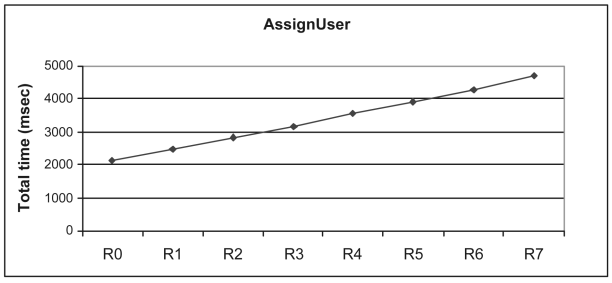

Fig. 8. Total time taken to execute `AssignUser`.

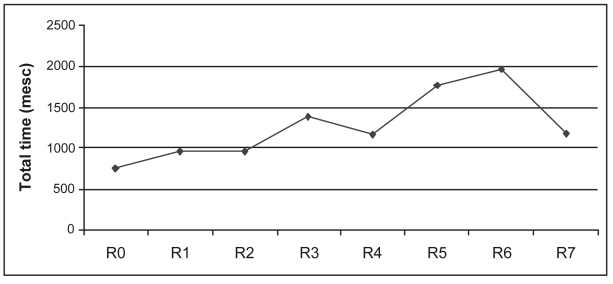

Fig. 9. Total time taken to execute `RevokePermission`.

3.4 GHz Dual Core Windows XP machine with 1.5 GB memory. We measure the elapse time of administrative operations by calling the Java method *System.nanoTime()* [5]. Under the given conditions, we have experimented with executing the administrative operations. We have executed eight out of the 10 administrative operations and measured their execution delays, of which we report one in Section 6.1. In addition, we have executed two other 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.

6.1 Simple Administrative Operations

We built the role hierarchy shown in Fig. 3, using our administrative operations. That activity took about 959 msecs to add eight roles, 844 msecs to add nine edges, and 711 msecs to grant 10 permissions per each of the eight roles, and about 3,384 msecs to assign 50 users to each role. The average time taken for each simple operation is between 68 to 120 msecs. Out of all these operations, Fig. 8 shows the individual time taken for assigning 50 users to each of the 10 roles. We notice that the time grows due to the growth of the U2R mapping. Further analysis shows that this is due to the fact that 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. We are looking for a better parser to improve the performance.

6.2 Complex Administrative Operations

We further show the performance characteristics of removing some permissions from a role that has been activated by some users. We also study the performance by removing a role from the role hierarchy while some subjects actively use that role, which invokes a series of administrative operations.

Recall that our 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))*.

As Fig. 9 shows, the time to remove a permission is proportional to 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 955 msecs. Revoking a permission from R5 requires terminating 60 sessions, taking 1,775 msecs. Our 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.

Recall that our definition of the *DeleteRole(r)* assumes 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, we need to ensure that these prerequisites 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)*.

Consequently, 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 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, we conducted three experiments.

In the first experiment, we fixed the number of users assigned to each role and the number of active sessions of each role. Fig. 10 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 (three) that activated the role given in Fig. 3, with various number of edges to be deleted. Starting with Fig. 3, deleting roles R6 and R7 requires deleting one edge, deleting roles R0 and R4 requires deleting two edges, deleting R1, R2, R3, and R5 requires deleting three edges. Fig. 10 shows that the time taken to delete edges is proportional to the number of edges that need to be deleted.

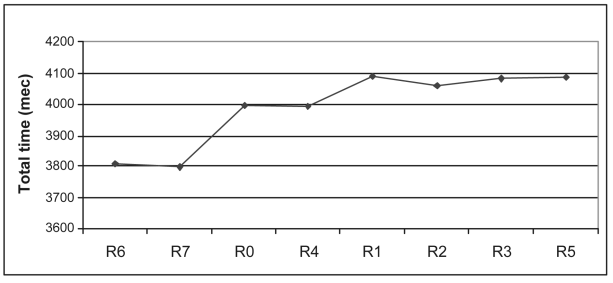

Fig. 10. Effect of # edges on time to remove a role.

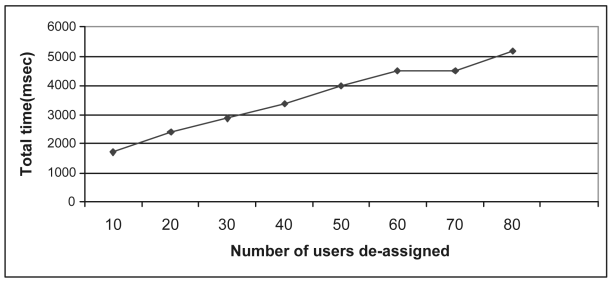

Fig. 11. Effect of # users on time to remove a role.

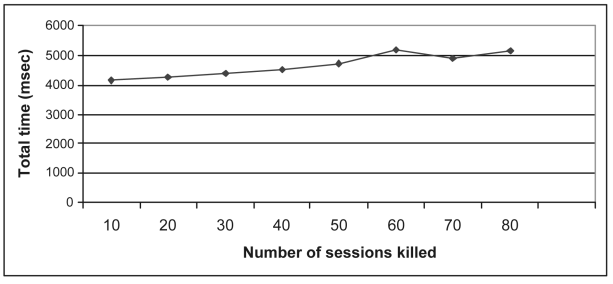

Fig. 12. Effect of # sessions on time to remove a role.

In the second experiment, we fixed the number of sessions activated by each user at three, with various number of users permitted to activate the role. Fig. 11 shows the total amount of time taken to delete each role in Fig. 3. Here we assigned 10, 20, 30, 40, 50, 60, 70, and 80 users to R0, R1, R2, R3, R4, R5, R6, and R7, respectively. Fig. 11 shows that the total time taken to delete a role is proportional to the number of users that need to be revoked from the role.

In the last experiment, we fixed the number of users assigned to each role at 50, with various number of sessions where the role is activated. We activated 10, 20, 30, 40, 50, 60, 70, and 80 sessions by R0, R1, R2, R3, R4, R5, R6, and R7, respectively. As Fig. 12 shows, the total time taken to remove a role increases with the number of sessions where the role is activated.

Our 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, three *DeleteEdge* operations, one *DeleteRole* operation, and killing 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.

7 Related Work

There have been many works in the area of access control for web services [19], [37], [38], [11]. Most of the work in the area forces on how to express access control polices and the architectures to implement the model. But these solutions have not addressed the concurrency issues between enforcing the access control decisions and administrating the access control polices.

UARBAC [30] proposes a principled approach in designing and analyzing administrative models for RBAC motivated by scalability, flexibility, psychological acceptability, and economy of mechanisms. 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. To the best of our knowledge, UARBAC has not been implemented and concurrency control has not been addressed.

SARBAC [17], [18] 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. To the best of our knowledge, SARBAC has not been implemented yet and concurrency control has not been addressed.

NIST [9], [20], [22] 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 an update to the three relations stored in the database is permitted by checking the consistency rules, and if so, updates the relationships in the database. This implementation is built for Intranet web servers, which is not suitable for distributed applications such as web services on the Internet.

PERMIS [12], [13] has developed a role-based access control infrastructure using X.509 [2] 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 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. A later version of PERMIS uses an XML interface using Sun's reference implementation [6] and adds dynamic delegation of authority [14]. Concurrency control has not been addressed in PERMIS.

Crampton and Chen [16] have proposed an approach to implement the RBAC model using XACML. They attempt to implement the ANSI RBAC standard [21] using a suit of XACML polices. 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 of our knowledge, these have not been fully implemented.

Seitz et al. [35] 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 their system.

Recently, OASIS XACML v3.0 Administration Standard has been approved as an OASIS committee working draft [4]. It describes a profile to express administrative metapolicies which can control different types of polices that individuals can create and modify, but does not use role-based administrative model to manage these XACML policies which is not scalable and flexible.

Concurrency control on XML data has been an active research recently. Haustein et al. [25] introduce a data model called taDOM tree to allow fine-grained locking using a combination of node locks, navigation locks, and logical locks, which we intend to use for our future research.

Janicke et al. [27] propose a concurrent enforcement model for usage control (UCON) [32] policies. Their model separates user, access controller, and system. While their technique enforces concurrency control based on static analysis of dependencies between polices, we resolve concurrency issues during the runtime of a system.

8 Conclusion and Future Work

An enforcement framework is proposed in this paper to enforce ARBAC policies with XACML in web services environment. To address concurrency issues that arise between enforcing administrative polices and policy evaluation, a session-aware administrative model for RBAC is used to manage the interactions and conflicts between session management and administrative operations. We specify concurrency requirements of an ARBAC model and introduce the concept of lock scope for a role, which captures the affected roles when the permissions granted to this role are updated due to administrative operations. We have developed an XACML-ARBAC profile to specify ARBAC polices and extended the Sun's XACML enforcement architecture by introducing an administrative policy enforcement point (A-PEP) and a*Lock Manger* to ensure the safety and integrity of policy management. We have developed the birth and death process of the access controller. We have implemented a prototype to enforce the extended XACML-ARBAC profile and demonstrated the feasibility of our framework. Our experimental study shows that our solution encounters a small performance overhead and can be used for general policy management systems.

One of our ongoing work is to refine the locking granularity for policies. We are also working toward enhancing the A-PEP functionality and creating a richer interface between the PEPs and A-PEP for session management in distributed environments.

References

[1] *Core and Hierarchical Role Based Access Control (RBAC) Profile of XACML v2.0,* OASIS Standard, http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-rbac-profile1-spec-os.pdf, 2005.

[2] *Information Technology - Open Systems Interconnection - The Directory: Public-Key and Attribute Certificate Frameworks,* ISO 9594-8/ITU-T Recommendation X. 509, 2011.

[3] OASIS XACML Technical Committee, *Core Specification: eXtensible Access Control Markup Language (XACML),* 2005.

[4] *OASIS XACML v3.0 Administration and Delegation Profile Version 1.0,* http://www.oasis-open.org, 2009.

[5] *Java 2 Platform Standard Edition 5.0,* http://download.oracle.com/javase/1,5.0/docs/api, 2011.

[6] "Sun's XACML Implementation," http://sunxacml.sourceforge.net, 2006.

[7] World Wide Web (W3C) Consortium, http://www.w3c.org, 2011.

[8] "Database Transaction," http://en.wikipedia.org/wiki/Database_ transaction, 2011.

[9] J. Barkley, A. Cincotta, D. Ferraiolo, S. Gavrila, and D.R. Kuhn, "Role Based Access Control for the World Wide Web," *Proc. 20th Nat'l Information System Security Conf.,* 1997.

[10] D.E. Bell and L.J. LaPadula, "Secure Computer Systems: Mathematical Foundations and Model," Technical Report No. M74-244, Mitre Corporation, 1975.

[11] E. Bertino, A. Squicciarini, I. Paloscia, and L. Martino, "Ws-AC: A Fine Grained Access Control System for Web Services," *World Wide Web: Internet and Web Information Systems,* vol. 9, no. 2, pp. 143-171, 2006.

[12] D.W. Chadwick and A. Otenko, "The PERMIS X.509 Role Based Privilege Management Infrastructure," *Proc. Seventh ACM Symp. Access Control Models and Technologies (SACMAT '02),* pp. 135-140, June 2002.

[13] D.W. Chadwick, A. Otenko, and E. Ball, "Implementing Role Based Access Controls Using X.509 Attribute Certificates," *Proc. IEEE Internet Computing Conf.,* pp. 62-69, Mar. 2003.

[14] D.W. Chadwick, S. Otenko, and T.A. Nguyen, "Adding Support to XACML for Dynamic Delegation of Authority in Multiple Domains," *Comm. and Multimedia Security,* pp. 67-86, Springer, Oct. 2006.

[15] J. Crampton, "Understanding and Developing Role-Based Administrative Models," *Proc. 12th ACM Conf. Computer and Comm. Security (CCS '05),* pp. 158-167, 2005.

[16] J. Crampton and L. Chen, "Implementing RBAC and ABRA Using XACML," submitted for publication, 2007.

[17] J. Crampton, "Administrative Scope and Role Hierarchy Operations," *Proc. Seventh ACM Symp. Access Control Models and Technologies (SACMAT '02),* pp. 145-154, June 2002.

[18] J. Crampton and G. Loizou, "Administrative Scope: A Foundation for Role-Based Administrative Models," *ACM Trans. Information and Systems Security,* vol. 6, no. 2, pp. 201-231, 2003.

[19] E. Damiania, S. De Capitani di Vimeracti, X. Paraboschi, and P. Samrarti, "Fine Grained Access Control for SOAP e-Services," *Proc. 10th Int'l Conf. World Wide Web (WWW '01),* pp. 504-513, 2001.

[20] D.F. Ferraiolo, J. Barkley, and D.R. Kuhn, "A Role Based Access Control Model and Reference Implementation within a Corporate Intranet," *ACM Trans. Information and System Security,* vol. 1, no. 2, pp. 34-64, 1999.

[21] D.F. Ferraiolo, R. Sandhu, S. Gavrila, D. Richard Kuhn, and R. Chandramouli, "Proposed NIST Standard for Role-Based Access Control," *ACM Trans. Information and System Security,* vol. 4, no. 3, pp. 224-274, 2001.

[22] S. Gavrila and J. Barkley, "Formal Specification for Role Based Access Control User/Role and Role/Role Relationship Management," *Proc. Third ACM Workshop Role Based Access Control,* pp. 81-90, 1998.

[23] M. Haustein and T. Härder, "taDOM: A Tailored Synchronization Concept with Tunable Lock Granularity for the DOM API," *Proc. Conf. Administrative Data Base Information System,* pp. 88-102, 2003.

[24] M. Haustein and T. Härder, "Optimizing Lock Protocols for Native XML Processing," *Data and Knowledge Eng.,* vol. 65, no. 1, pp. 147-173, 2008.

[25] M. Haustein, T. Härder, and K. Luttenberger, "Contest of XML Lock Protocols," *Proc. 32nd Int'l Conf. Very Large Data Bases (VLDB '06),* pp. 1069-1080, 2006.

[26] M. Harrison, W. Ruzzo, and J. Ullman, "Protection in Operating Systems," *Comm. ACM,* vol. 19, no. 8, pp. 461-471, 1976.

[27] H. Janicke, A. Cau, F. Siewe, and H. Zedan, "Concurrent Enforcement of Usage Control Polices," *Proc. IEEE Workshop Policies for Distributed Systems and Networks (POLICY '08),* pp. 111-118, July 2008.

[28] H. Korth and A. Silberschatz, *Database System Concepts.* McGraw-Hill, 1991.
[29] D. Lea, *Concurrent Programming in Java Design Principles and Patterns.* Addison-Wesley, 2000.
[30] N. Li and Z. Mao, "Administration in Role Based Access Control," *Proc. ACM Symp. Information, Computer and Comm. Security (ASIACCS '07),* pp. 127-138, Mar. 2007.
[31] S. OH, R. Sandhu, and X. Zhang, "An Effective Role Administration Model Using Organization Structure," *ACM Trans. Information and Systems Security,* vol. 9, no. 2, pp. 113-137, 2006.
[32] J. Park and R. Sandhu, "The $\mathrm{UCON}_{\mathrm{abc}}$ Usage Control Model," *ACM Trans. Information and Systems Security,* vol. 7, no. 1, pp. 128-174, Feb. 2004.
[33] R. Sandhu, V. Bhamidipati, and Q. Munawer, "The ARBAC97 Model for Role-Based Administration of Roles," *ACM Trans. Information and Systems Security,* vol. 2, pp. 105-135, 1999.
[34] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman, "Role Based Access Control Models," *Computer,* vol. 29, no. 2, pp. 38-47, 1996.
[35] L. Seitz, E. Rissanen, T. Sandholm, B. Sadighi, and O. Mulmo, "Policy Administration Control and Delegation Using XACML and Delegent," *Proc. Sixth IEEE/ACM Int'l Workshop Grid Computing,* pp. 49-54, 2005.
[36] J.M. Spivey, *The Z Notation: A Reference Manual,* second ed., Prentice Hall Int'l Series in Computer Science, 1992.
[37] R. Wonohoesodo and Z. Tari, "A Role Based Access Control for Web Services," *Proc. IEEE Int'l Conf. Service Computing (SCC '04),* pp. 49-56, 2004.
[38] E. Yuan and J. Tong, "Attributed Based Access Control (ABAC) for Web Services," *Proc. IEEE Int'l Conf. Web Services (ICWS '05),* pp. 561-569, 2005.
[39] M. Xu, D. Wijesekera, X. Zhang, and D. Corray, "Towards Session-Aware RBAC Administration and Enforcement with XACML," *Proc. IEEE Symp. Policies for Distributed Systems and Networks,* pp. 9-16, July 2009.

Min Xu received the BE degree from the Huazhong University of Science and Technology, Wuhan, China, in 2000, and the MS degree from the University of Nevada, Las Vegas, in 2002, both in computer science. He is currently pursuing the PhD degree in the Computer Science Department at George Mason University, Fairfax, Virginia. His research interests include computer systems and networking security policies, models, architectures and mechanisms, trusted computing, and virtualization technologies.

Duminda Wijesekera is an associate professor in the Department of Computer Science at George Mason University, Fairfax, Virginia. His research interests are in security, multimedia, networks, secure signaling (telecom, railway, and SCADA), avionics, missile systems, web, and theoretical computer science. He is a senior member of the IEEE.

Xinwen Zhang received the PhD degree in information technology from George Mason University, Fairfax, Virginia. He is currently a research scientist at Samsung Information Systems America in San Jose, California. His research interests include security policies, models, architectures, and mechanisms in general computing and networking systems. His recent research focuses on secure and trusted mobile platforms, applications, and services. He is a member of the IEEE.