XACML Policies for Exclusive Resource Usage

Vijayant Dhankhar, Saket Kaushik, and Duminda Wijesekera

Department of Information & Software Engineering
George Mason University
Fairfax, VA 22030, U.S.A
{vdhankha | skaushik | dwijesek}@gmu.edu

No Institute Given

Abstract. The *extensible access control markup language (XACML)* is the standard access control policy specification language of the World Wide Web. XACML does not provide exclusive accesses to globally resources. We do so by enhancing the policy execution framework with locks.

1 Introduction

The *extensible access control markup language (XACML)* [18] is the standard language to specify accesses to resources available on the world wide web. However, the XACML normative specifications lack necessary syntax to specify *exclusive access* to resources, and furthermore, publicly available XACML policy enforcement frameworks do not enforce them. Given that web orchestrations are composed from existing ones using languages such as BPEL [16] may pose concurrent request for exclusively usable resources (such as updating an XML schema), we enhance XACML syntax and enforcement mechanisms to do so.

Perils of not using a synchronization mechanism (such as the *dirty read* [21] in distributed systems) in exclusive accesses are well known. Consequently, we advocate to make a distinction in granting exclusive access and non-exclusive accesses by access controllers. Thus, we add appropriate syntax to XACML and an enforcement mechanism using locks. Consequently, if and when granted, the access control policy is aware that such permissions are granted in exclusion. This enrichment to XACML has no relationship to application level concurrency control, but not surprisingly, due to the enforced semantic distinction between exclusive and non exclusive acccesses, aids in enforcing *separation of duty* principles [12, 8, 9, 20].

To enforce enhanced XACML policies, we add a lock manager to the policy enforcement module and require that all globally accessible resources register with a unique lock manager. In order to ensure starvation avoidance, we assume that resource requesters give up such resource after their usage - although this latter aspect is being driven by policy in our ongoing work.

The rest of the paper is organized as follows. Section 2 has related work. Section 3 presents sample Use Cases and Misuse Cases of exclusive access and our design to realize the former. Syntactic extensions to XACML appear in Section 4 and Section 5 describes the architectural enhancements used to enforce locking. Section 6 describes

our implementation and Section 7 informally argues that our locking ensures safety and liveliness. Finally, Section 8 concludes the paper.

2 Related Work

Motivated by a desire to to introduce trust-based, context-aware access control framework for Web Service invocations, including support for RBAC sessions, Bhatti *et al.*, [4, 6, 5, 7] define X-RBAC and X-GTRBAC models for access control frameworks for Web Services, respectively based on RBAC [12] and GTRBAC [14] models of access control. However, they do not provide mechanisms to enforce dynamic separation of duty (DSoD) policies, as is the case with current XACML RBAC profile [17]. Cardea by Lepro *et al.* [15] offers a dynamic access control system for the Web, where the *dynamism* means that the request is not bound to local identities at runtime, but instead uses a remote requester's context instead. However, Cardea does not explicitly address concurrent access to exclusively used resources nor dynamic separation of duty policies.

3 Use Cases, Misuse Cases and Requirements

Although some existing work on Web Services orchestration argues the need to lock shared resources [3, 13], to the best of our knowledge they only reserving syntax for locks [3]. Following Use Cases show the need.
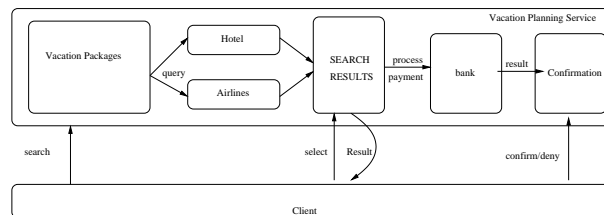

Fig. 1: *Vacation Planning Service*

3.1 Use Case 1: exclusive access

Consider an example, *vacation planning service (VPlanner)* that reserves hotel rooms and air tickets for its clients, whose work-flow is given in figure 1 [23]. As seen, this interactive service is used to first *searches* for available rooms and air tickets for specified dates and destinations and presents various alternatives to its clients, from which the latter *chooses* alternative for reservations. The service then *initiates* a monetary transfer request to the credit granting agency. On success, the room(s) and air tickets are reserved and aborted otherwise. An efficient implementation should invoke airline and hotel room searches concurrently, while, work-flow dependencies require that monetary transfer request should wait till other parts of the procedure are complete.

Now, suppose two clients are searching for reservations from the VPlanner and are shown the same tickets and hotel rooms. This is potentially dangerous because both can choose the same room or ticket, where simultaneous requests can deadlock two BPEL server processes. One way of avoiding this situation is to not offer a second

client the choices while a precedent client s in the process of reserving a package - thus requiring the VPlanner to lock rooms and tickets during ongoing reservations, referred to as *tentative locking of resources* in the Web Services literature [3].

3.2 Use Case 2: enforcing dynamic constraints

Example 1 (DSoD [8]). Consider a DSoD constraint *an employee cannot invoke role 1 in a session if another role, role 2, is already invoked in some other session. Assuming a data structure maintained by the system* 'sessions' with following XML schema:

```
<user id="ID">
  <sessions>
    <session id="123123">
        <role name="role1"/>
        <role name="role3"/>
    </session> ...
  </sessions>
</user>
```

An abbreviated DSoD XACML policy as follows:

```
1 <Rule RuleId="DSoD:role1-role2:requirements" Effect="Deny">
2 <!-- SoD Rule for Example 1 (begin) -->
3 <Target>
4   <Subjects>
5     <AnySubject/>
6   </Subjects>
7   <Resources>
8     <AnyResource/>
9   </Resources>
10  <Actions>
11    <Action>
12      <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
13        <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">activate-role</AttributeValue>
14        <ActionAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#string" AttributeId="urn:oasis:names:
              tc:xacml:1.0:action:action-id"/>
15      </ActionMatch>
16    </Action>
17  </Actions>
18 </Target>
19 <!-- SOD check -->
20 <Condition FunctionId="urn:oasis:names:tc:xacml:1.0:function:seperationOfDutyCheck">
21   <!-- sessions -->
22   <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:getSubjectSessions">
23     <!-- subject-id -->
24     <AttributeSelector RequestContextPath="//Request/Subject/Attribute[1]/AttributeValue/text()" DataType="http://
              www.w3.org/2001/XMLSchema#string"/>
25   </Apply>
26   <!-- role-id -->
27   <AttributeSelector RequestContextPath="//Request/Resource/Attribute[2]/AttributeValue/text()" DataType="http://
              www.w3.org/2001/XMLSchema#string"/>
28   <!-- comma delimited set of conflicting roles -->
29   <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">role1,role2</AttributeValue>
30 </Condition>
31 </Rule>
```

Policy 1: DSoD policy

Example 1 above is a DSoD policy expressed in terms of the XACML RBAC profile [17], where as stated in lines 20-30, role 1 and role 2 cannot be co-activated. However, this policy is not currently enforceable because XACML enforcement does not consider concurrent requests. To be fair, the XACML RBAC profile *outsources* the process of enabling roles to the *Role Enablement Authority* module.

Our design enables the follwing Use cases:

Secure registration of resources: A resource may register itself to a unique lock manager.

Secure deregistration of resources: Only a resource is able to securely deregister itself from the lock manager.

Exclusive access /relinquish resources: Exclusive use of a resource must be granted to a unique requester at any given time.
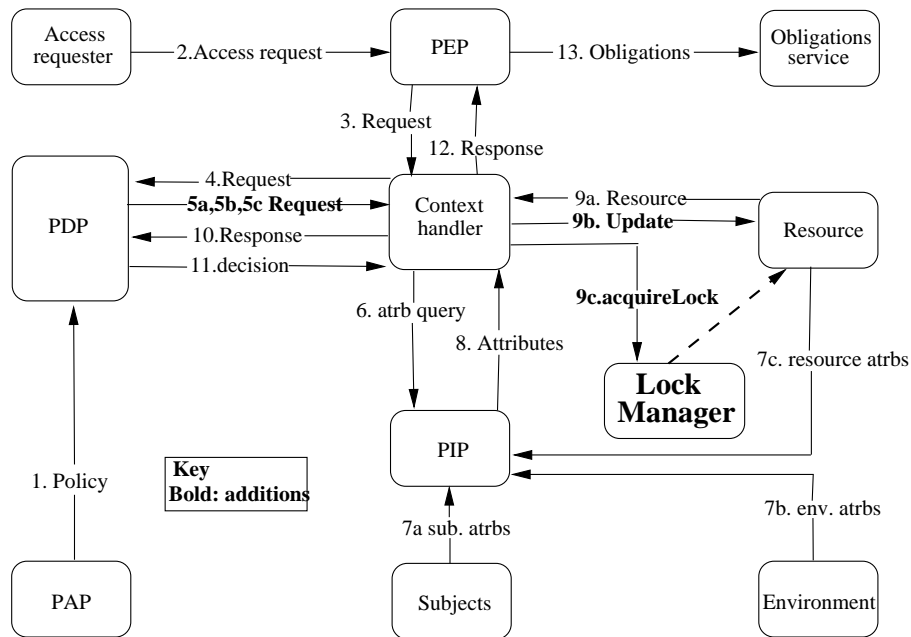
Fig. 2: *Extended XACML data flow diagram*

3.3 Preventing Misuse Cases

Our design prevents following Misuse Cases:

Registering a resource with multiple lock managers: An exclusively usable resource
being registered with multiple lock managers, referred to as *singular registration*.

Spoofing a resource: Others (de)registering an exclusively accessible resource.

Preventing simultaneous exclusive access: Multiple requesters simultaneous access-
ing an exclusively usable resource.

Starvation: Refusing exclusive access to resources when not in use.

4 Enhancing the XACML syntax

Because our solution use locks, we add them to XACML syntax. Each of the following
elements are specified within `<Rule/>`, `<Policy/>` and `<PolicySet/>` elements
of XACML.

– `<PreAction />` specifies a set of locks to be acquired before rule evaluation.
  `<AcquireLocks />` specifies a set of locks to be acquired and is a sub element of
      `<PreAction/>` element, where the `<AcquireLock>` sub element specifies an in-
      dividual lock.
– `<PostAction />` identifies a set of actions to be performed after (a) rule evaluation leads
  to a permitted request, (b) rule evaluation leads to a denied request. The set of actions may in-
  clude releasing locks or updating system resources. For different evaluation results multiple
  `<PostAction />` elements may be defined.

`Effect` attribute indicate the *effect* of a post action, as discussed above.

`<Updates />` specifies updates to be performed in a `<PostAction/>`. `<Update>` sub element specifies an individual change.

`<ReleaseLocks />` specifies a set of locks to release and is a sub element of `<PostAction/>` element. `<ReleaseLock>` sub element specifies an individual lock.

Introduced elements specify lock acquisition prerequisite for evaluating a rule and post evaluation steps to be taken. The following example policy extends policy 1 with proposed syntactic enhancements.

```xml
1<Rule RuleId="DSoD:role1-role2:requirements" Effect="Deny">
2 <PreAction>
3   <AquireLocks>
4     <AquireLock>
5       <!-- sessions -->
6       <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:getSubjectSessions">
7         <!-- subject-id -->
8         <AttributeSelector RequestContextPath="//Request/Subject/Attribute[1]/AttributeValue/text()" DataType="http
                ://www.w3.org/2001/XMLSchema#string"/>
9       </Apply>
10    </AquireLock>
11    .
12    .
13    .
14   </AquireLocks>
15 </PreAction>
16     .
17     . <!-- DSoD Rule in Example 1 -->
18     .
19 <PostAction Effect="Permit">
20  <Updates>
21    <Update>
22      <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:addRoleToSession">
23        <!-- role-id -->
24        <AttributeSelector RequestContextPath="//Request/Resource/Attribute[2]/AttributeValue/text()" DataType="
                http://www.w3.org/2001/XMLSchema#string"/>
25        <!-- session-id -->
26        <AttributeSelector RequestContextPath="//Request/Resource/Attribute[3]/AttributeValue/text()" DataType="
                http://www.w3.org/2001/XMLSchema#string"/>
27        <!-- sessions -->
28        <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:getSubjectSessions">
29          <!-- subject-id -->
30          <AttributeSelector RequestContextPath="//Request/Subject/Attribute[1]/AttributeValue/text()" DataType="
                http://www.w3.org/2001/XMLSchema#string"/>
31        </Apply>
32      </Apply>
33    </Update>
34    .
35    .
36    .
37  </Updates>
38  <ReleaseLocks>
39    <ReleaseLock>
40      <!-- sessions -->
41      <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:getSubjectSessions">
42        <!-- subject-id -->
43        <AttributeSelector RequestContextPath="//Request/Subject/Attribute[1]/AttributeValue/text()" DataType="http
                ://www.w3.org/2001/XMLSchema#string"/>
44      </Apply>
45    </ReleaseLock>
46    .
47    .
48    .
49  </ReleaseLocks>
50 </PostAction>
51</Rule>
```

Policy 3: *Enhancements to DSoD policy*

The `PreAction` element in lines (2–15) of policy 3 above states that before evaluating a rule in the DSoD policy, the user session must be locked (lines 5-10). Similarly, `PostAction` element in lines (19-50) requires that after the policy has been evaluated, the locks acquired earlier be released for future concurrency-free changes to user sessions. We assume here that resources are not created during XACML evaluation (they already exist and are registered with a lock manager), however their usage status, *i.e.*,

open for read/write, *etc.*, may be modified during a policy evaluation. For example, an XACML evaluation can modify a log file, *etc.*

4.1 Implemented semantics of syntactic extensions

Postaction elements are evaluated in the following manner.

- Post action only updates resources for locks obtained at the corresponding level or those obtained at the level of the container.
- If two rules within a policy require same lock then they must be acquired and released at Policy level. Such locks are visible within all embedded rules. Similarly, if a lock is acquired at the `PolicySet` level then it is visible to all embedded policies.
- If a resource must be updated in multiple rules, then corresponding lock must be acquired at their container level, *i.e.*, `Policy`.
- Rule evaluation within a `Policy` element is evaluated by a single thread of execution.
- If locks are required at only the rule level, they must be released at the rule level `<PostAction/>`, otherwise, they must be released at the `<Policy/>` level `<PostAction/>`

4.2 XACML functions

Extensions to policy syntax is achieved with the help of following functions:

function:getSubjectSessions($subject-id as string) Accepts a subject-id as an input and returns a bag of session objects used by this subject. If the subject's sessions have been acquired by PDP through exclusive access, *i.e.*, locked, then the sessions are cached till the lock to the sessions is released.

function:addRoleToSession($role-id as string, $session-id as string, $sessions as bag) Accepts role-id, session-id and a bag of sessions as input and it adds the passed role to the particular session. Sessions data structure contains all the sessions and session-id is used to locate the relevant session in the current implementation, although more efficient implementations are possible.

5 Architectureral Enhancements needed for Locks

Figure 2 shows the existing XACML execution model with data flow for policy control [18]. The data flow begins with the Policy Administration Point (PAP) that authors the policies evaluated by the XACML framework, shown in *Flow 1*. Next, access requests (*Flow 2*), initiated by resource requesters, are intercepted by the Policy Enforcement Point (PEP). PEP forwards them, *Flow 3*, to the Context Handler (CH) with optional requester attributes and environmental conditions required for processing. Context handler has following three functions:

Flow 4: Translate access requests into a format understood by the Policy Decision Point (PDP).

Flow 10: In response to Flow 5, generate the evaluation context by gathering resource, requester attributes and current system state of from the policy information point (PIP) (i.e. from, Flows 6,7,8 and 9), and pass them to the PDP.

Flow 12: Receive policy decisions from the PDP and translate them back to the PEP.

PDP evaluates an XACML policy applicable to the access request and accompanying context. If fails, the access is denied and granted otherwise. This decision is made available to the context handler of Flow 11 and is relayed to the PEP for enforcement.

Figure 2 shows the *extended* XACML data flow diagram that introduces a *lock manager (LM)* to augment XACML access control decisions. The *Lock Manager* grants and revokes locks for accessing resources registered with itself, requiring extra data flows as follows:

Flow 5b - Update System Request (USR): May be initiated by the PDP to update system resources for setting up an access. For example, enabling a role may require that the user session (a system resource) be updated.

Flow 5c - Create Lock Request (CRL:) Is initiated by the PDP on behalf of the requesting process, in response for exclusive access to an available resource. This is finally refined to the `acquireLock` operation (9c.), where the lock is *owned* by the requesting process.

Flow 10 - Response to PDP queries (overloaded): We reuse the response sent by the context handler to the PDP queries for sending USR and CRL responses in addition to resource query response.

Flow 9b - Resource update, 9b: Upon enforcing the access control decision, the PEP updates an internal log of accesses. This data flow enables enforcing history based resource usage.

Flow 9c - AcquireLock, 9c: Instructs the LM to invoke a lock on behalf of a requester. Based on the availability of a resource, this operation may succeed or fail.

As in the normative XACML specification, we assume that all attributes have been authenticated (using attribute certificate authenticity) prior to policy evaluation. We discuss the additional complexity due to these addition later in section 5.2. First, we describe the Lock Manager design.

5.1 Lock Manager (LM)

The Lock Manager (see figure 2) is a *privileged process* that, at any given time, has only a single instance running. The Lock Manager maintains and creates locks for resources it manages. The functionality of the Lock Manager that provides and maintains locks are as follows:

Lock acquisition Lock acquisition is an atomic operation `acquireLock`, implemented using an atomic operation, akin to the unix `test&set` operation [22], with two associated strings - `requesterId` and `resourceId` with the `acquireLock`. These strings can be qualified names or URIs of network entities. The actual call is made by the lock manager within a critical section, as shown in section 6.

Releasing a lock Lock release is an atomic process implemented as the method *releaseLock*, implemented atomically.
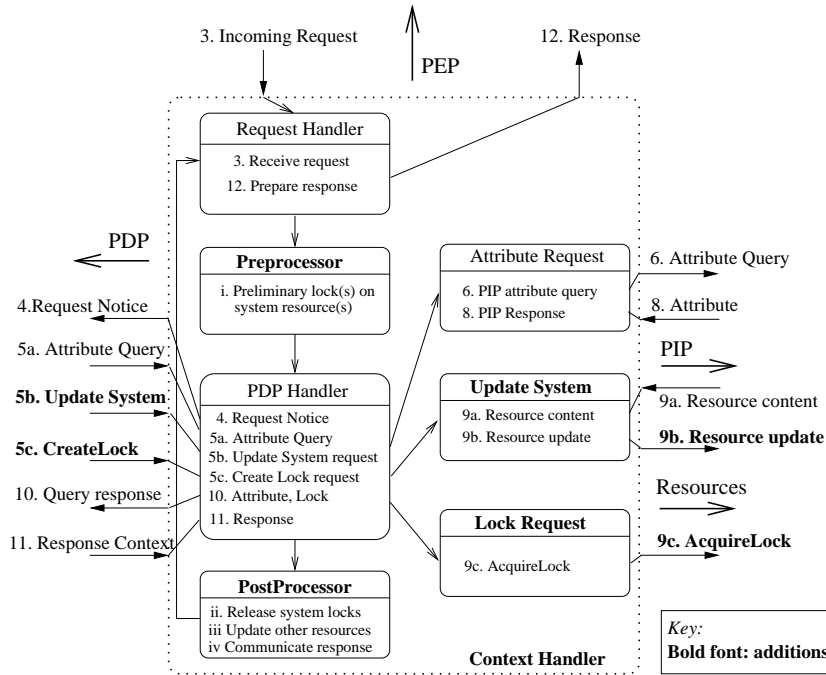
Fig. 3: *Context Handler*

Verifying a lock The `verifyLock` operation verifies the validity of a lock. The execution model states that for every resource update must be preceded by a call to *verify-Lock*, and is invoked by the resource manager for verifying the validity of a lock.

The execution model is that a requester gets the lock and at the time of resource usage and presents the locking permission to the resource manager that in turn verifies the validity of the presenter's claim with the lock manager and or PDP.

Registering with a single LM Registering a resource with a single lock manager is a basic design requirement enforced by requiring an attribute certificate where each resource is bound to a *single* lock manager (by a local certificate authority) using an X.509 attribute certificate [11].

5.2   Enhancing the context handler

We enhance the context handler with resource pool bootstrapping, termination and maintenance, (*i.e.*, performing singular registration, deregistration, *etc.*) and extending the business logic for additional functionality.

Resource pool maintenance We extend the context handler to support the lock manager bootstrapping and maintenance of the resource pool. Algorithms in this section are written in pseudo code, with '−>' symbol indicating a call to a sub-module within the context handler.

Securely registering resources To prevent multiple concurrent registration requests being invoked by a resource, we introduce the secure registration of resource procedure. Secure registration begins with a registration request by a resource, accompanied with a verifiable attribute certificate containing the resource and the lock manager. This request is serviced by Algorithm 1 below, as shown in figure 3.

```
1 register(R,C,LM)
2 Inputs: R(resource), C(certificate), LM(lock manager)
3 Output: Lock information, Exception
4
5 Translate request to XML document
6 Pass translated XML to PDP Handler
7 -->Invoke PDP to verify R,C and LM
8  if (decision == accept)
9    Lock Request for lock to global.lock
10   -->Acquire lock to global.lock
11   Update System Request for variable
12   -->Create variable lock.R
13    if (lock.R NOT IN global.lock)//
14      Assign lock.R.subjectID = ""
15      Insert lock.R in global.lock
16      Assign message = lock.R
17      Hand request to PostProcessor
18      -->Lock Request for release lock
19        -->Release lock
20        Return request+message to Request Handler
21    else
22      Assign exception = Already Registered
23      Hand Request to PostProcessor
24      -->Lock Request for release lock
25        -->Release lock
26        Return request+exception to Request Handler
27  else
28    Assign exception = Invalid Request
29    Hand the request to PostProcessor
30    -->Return request+exception to Request Handler
```

Algorithm 1: *Secure Registration*

The `register` method accepts three inputs – R, the registering resource; C, its attribute certificate, and LM, the Lock Manager. This request is handed to the 'Request handler' module of CH by the PEP (line 5). The request handler translates the request into XML and hands it to PDP handler for further processing (line 6). The PDP handler invokes the PDP and processes the response (lines 7,8). If the decision is accept, a lock is acquired (lines 9,10) and an update request is fired (line 11). Based on the success of this call, lock is release and the relevant message is returned to the request handler (lines 18-26). If PDP denies the register request, an appropriate response is constructed as well (line 27-30).

Securely deregistering a resource Similar to registration, deregistration requires locking support because in order to prevent being deregistered while in use. The process flow is as follows:

```
1 deregister(R,C,LM)
2 Inputs: R(resource), C(certificate), LM(lock manager)
3 Output: boolean, Exception
4
5 Translate request to XML document
6 Pass translated XML to PDP Handler
7 -->Invoke PDP to verify R,C and LM
8  if (decision == accept)
9    Lock Request for lock to global.lock
10   -->Acquire lock to global.lock
11   if (acquirelock()== false)
12     Assign exception = In use // no waiting!!
13     Hand request to PostProcessor
14     -->Send request+exception to Request Handler
15   else
16     if(lock.R IN global.lock && lock.R.subjectId="")
17       Update System Request
```

```
18       -->Assign global.lock = global.lock -lock.R
19         Assign message = true
20       Hand request to PostProcessor
21       -->Lock Request for release lock
22         -->Release Lock // will succeed
23         Return request+message to Request Handler
24     else
25       Assign exception = Does not exist/In use
26       Hand Request to PostProcessor
27     -->Lock Request for release lock
28       -->Release lock
29       Return request+exception to Request Handler
30   else
31     Assign exception = Invalid Request
32     Hand the request to PostProcessor
33     -->Return request+exception to Request Handler
```

Algorithm 2: *Secure Deregistration*

Similar to the `register` method `deregister` accepts the same three inputs and securely removes the resource from LM. The difference here is that the update request removes the lock from a global lock data structure in a critical section (line 18).

Extending the Business Logic

Gaining Exclusive Access to Resources Once a resource is registered, exclusive access to it can be guaranteed by a process very similar to the above processes, as follows:

```
1 exclusiveAccess(R,S)
2 Input: R (resource), S (Subject)
3 Output: boolean, Exception
4
5 Translate request to XML document
6 Pass translated XML to Pre processor
7 -->Lock Request for lock to global.lock
8   -->Acquire lock to global.lock
9   if (acquireLock() == false)
10    Assign exception = In use // no wait
11    Hand Request to PostProcessor
12    -->Send Request+exception to Request Handler
13  else
14    Hand request to PDP Handler
15    -->Invoke PDP for authorizing S to R
16    -->Attribute Query for lock.R
17      -->Read lock.R
18        if(lock.R IN global.lock)
19          send lock.R to PDP
20        else
21          exception = resource unknown
22          Hand request to PostProcessor
23          -->Return Request+exception // to RH
24    if (decision == accept)
25      Update System Request
26      -->Assign lock.R.subjectId = S
27        Assign message = true //
28      Hand request to PostProcessor
29      -->Lock Request for release lock
30        -->Release Lock
31        Return request+message to Request Handler
32      else
33        Assign exception = Invalid Request
34        Hand the request to PostProcessor
35        -->Return request+exception to Request Handler
```

Algorithm 3: *Exclusive access*

The `exclusiveAccess` method invokes a similar CH work-flow for granting access to externally usable resources. The main difference here is a call to the pre processor module that acquires locks before beginning any PDP evaluation (lines 7-12). In addition, the PDP may query for lock attributes (lines 16-23). Finally, if the request is granted, the lock manager updates `lock.R` to indicate the new owner (line 26).

Using Resource: Exercising exclusive access, once a requester has gained such an access to a resource, is a call to the resource (through the PEP). It is the resource's responsibility to ensure that the requester owns a valid lock to it, done by a *verifyLock* call to the lock manager (details are omitted due to lack of space).

Guaranteeing Dynamic/ History-based Access Constraints Steps for enforcing dynamic constraints are as follows:

```
1 accessResource(R,S)
2 Inputs: R (resource), S (Subject)
3 Output: boolean, Exception
4
5 Translate request to XML document
6 Pass translated XML to Pre processor
7 -->Locate internal resources for the request
8   Lock Request for locks to all internal resources
9   -->while(more resources)
10      { Acquire lock } // locking phase
11  if (all locks acquired == false)
12    Assign exception = Cannot process // no wait
13    Hand Request to PostProcessor
14    -->Send Request+exception to Request Handler
15  else
16    Invoke PDP for authorizing S to R
17    Attribute Queries (optionally)
18    -->Read attribute
19      if(attribute present)
20        send attribute to PDP
21      else
22        exception = resource unknown
23        Hand request to PostProcessor
24        -->Send Request+exception to Request Handler
25    if (decision == accept)
26      Update System Request
27      -->while(more resources need updation)
28         { update ith resource }
29         Assign message = true //
30      Hand request to PostProcessor
31      -->Lock Request for releasing all locks
32        -->Release Lock
33        Return request+message to Request Handler
34    else
35      Assign exception = Invalid Request
36      Hand the request to PostProcessor
37      -->Return request+exception to Request Handler
```

Algorithm 4: *Dynamic constraints*

Releasing Resource A resource requester holding a lock can *release* the resource. A simple modification to the algorithms presented above does this, where the details are omitted due to lack of space.

6 Implementing the Enhanced Design

In this section presents the salient features of our LM implementation. We begin our discussion with data structures to implement locks, followed by a sample Java snippet and WSDL interaction to acquire locks.

```
1 <xs:element name="Lock"
2  type="xacml-context:LockType"/>
3 <xs:complexType name="LockType">
4   <xs:sequence>
5     <xs:element name="resourceId" type="xs:string"
6       minOccurs="1" maxOccurs="1"/>
7     <xs:element name="ownerId" type="xs:string"
8       minOccurs="0" maxOccurs="1"/>
9   </xs:sequence>
10 </xs:complexType>
```

Listing 1: *The Lock Data Structure*

Listing 1 shows the *LockType* XML Schema (lines 3-10) that represents a lock for a *single* shared resource. The data structure identifies the lock through a `resourceID` and an `ownerId`. An available lock has an `ownerId` as an empty string, while a locked resource has a non-empty `ownerId`. Several such locks (one for each shared resource) are stored in a global LM-data structure called the *GlobalLock*.

```
1 <xs:element name="GlobalLock"
2 type="xacml-context:GlobalLockType"/>
3 <xs:complexType name="GlobalLockType">
4 <xs:sequence>
5   <xs:element ref="xacml-context:GlobalLockEntryType"
6     maxOccurs="unbounded"/>
7 </xs:sequence>
8 </xs:complexType>
```

Listing 2: *The Global Lock Data Structure*

```
1 <xs:complexType name="GlobalLockEntryType">
2 <xs:sequence>
3   <xs:element name="resource" type="xacml-context:Resource"
4     minOccurs="1" maxOccurs="1"/>
5   <xs:element name="lock" type="xacml-context:Lock"
6     minOccurs="0" maxOccurs="1"/>
7 </xs:sequence>
8 </xs:complexType>
```

Listing 3: *The Global Lock Entry Type*

The *Global Lock Entry Type* data structure stores (key,value) pairs including resource and its locks.

```
1 <wsdl:message name="aquireLockRequest">
2   <wsdl:part name="correlationSet" element=
3     "xacml-context:CorrelationSet" />
4   <wsdl:part name="lock" element=
5     "xacml-context:Lock" />
6 </wsdl:message>
7
8 <wsdl:message name="lockResult">
9   <wsdl:part name="correlationSet" element=
10    "xacml-context:CorrelationSet" />
11  <wsdl:part name="result" element=
12    "xacml-context:LockResult" />
13 </wsdl:message>
```

Listing 4: *Lock acquisition/response message*

Listing 3 shows WSDL message definitions for the `acquireLock` request and the *result* response.

```
1 public class LockManager {
2
3 // list of all the resources and locks
4 public static Map GlobalLock = new HashMap();
5
6 /**
7 * Aquire a lock
8 */
9 public static LockResult aquireLock (AquireLockRequest request) {
10
11 Lock lock = null;
12 boolean aquireSuccess = false;
13
14 // aquire global lock
15 synchronized (GlobalLock) {
16
17   // if lock already acquired or not registered then fail
18   if (GlobalLock.containsKey(request.getResourceId())) {
19     lock = GlobalLock.get(request.getResourceId());
20     if (lock==null) {
21       // if no locks exist on the resource
```

```
22      // then create a new resource
23      lock = new Lock();
24      lock.setOwnerId(request.getActorId());
25      lock.setResourceId(request.getResourceId());
26      GlobalLock.put(request.getResourceId(), lock);
27      aquireSuccess = true;
28    }
29  }
30
31 } // release the global lock
32
33 LockResult result = new LockResult();
34 if (!aquireSuccess) {
35  // couldnt aquire lock
36  result.setStatus("fail");
37 } else {
38  // lock aquired
39  result.setStatus("pass");
40  result.setLock(lock);
41 }
42 return result;
43}
```

Listing 5: *AcquireLock method*

Listing 5 shows a `Java` implementation of the `AcquireLock` method for acquiring a lock to an existing resource. Java allows *synchronization* through exclusive access to objects that we leverage upon in this implementation (line 17). In Line 16 we aquire exclusive access to the GlobalLock data structure till Line 33. In line 19 we check if the resource is registered with the Lock Manager. Line 21: We check if the lock has already been granted, if not then we create a new Lock with the owner and resource specified in request and add it to GlobalLock (Line 24-27). Finally after updating the GlobalLock we remove our exclusive access to it (Line 33) Line 35: we create a result data structure. Line 36-43: we construct the result for the request accordingly and return it in line 44.

7 Safety and Liveliness Properties

Because we allow concurrent requests and use locks to serialize access to critical sections of the security monitor, we ensure liveliness and safety properties. In this section, we informally argue for them.

Lemma 1. *Given a resource pool $\mathcal{R}$ and a set of lock managers $\mathcal{L}$, a resource $R_i \in \mathcal{R}$ can only be registered with a single lock manager $L_j \in \mathcal{L}$*

Proof Sketch: See [10] for proof.

Lemma 2. *Given a resource R_i and a lock manager L_K to which R_i can register itself,* `register` *method ensures that R_i can be registered only once with L_K.*

Proof Sketch: See [10] for proof.

Lemma 3. *Given a resource R_i and a lock manager L_K to which R_i is registered,* `deregister` *method ensures that only R_i can be deregister itself from L_K.*

Proof Sketch: See [10] for proof.

Theorem 1 (Safety of the exclusive access:)**.** *Given an XACML policy P for exclusive access to an available resource R and multiple concurrent access requests from subjects $S_i, i \in [1, n]$ (i.e.,* `exclusiveAccess(R, S_i)`*), only one request from the above set is authorized by P.*

Proof: See [10] for proof.

Theorem 2 (Safety of dynamic constraints:)**.** *Given an XACML policy P for dynamic constraint for access to a resource and multiple conflicting access requests (`accessResource`(R, S_i)), then P authorizes only one request.*

Proof: Similar to that of Theorem 1.

Theorem 3 (Liveliness1:)**.** *Given a resource R and an exclusive use access request by subject S `exclusiveAccess(R,S)`), then policy evaluation will release locks to all internal resources irrespective of the access control decision.*

Proof Sketch: See [10] for proof.

Theorem 4 (Liveliness2:)**.** *Given resources x, y and an exclusive use access requests by subject S `exclusiveAccess(x,S)` followed by `exclusiveAccess(y,S)`) and concurrent exclusive use access requests by subject T `exclusiveAccess(y,T)` followed by `exclusiveAccess(x,T)`), then at-least one of the exclusive access requests is denied by policy evaluation and all locks acquired for that policy evaluation are released.*

Proof Sketch: See [10] for proof.

8 Conclusions

XACML is the default access control specification language for the World Wide Web [1, 2, 19]. But XACML does not currently support three types of access control Use Cases, *viz.*, ensuring exclusive access to globally available resources, preventing access to a resource given a concurrent conflicting use of another resource (DSoD constraints), and preventing access to a resource given a history of conflicting access (such as in Chinese Wall constraints). We extend XACML syntax for supporting the above-mentioned use cases, by enhancing the XACML policy enforcement framework with a *lock manager*, to realize the additional Use Cases, and informally argue that safety and liveliness properties are ensured by our implementation.

References

1. Entrust. http://www.entrust.com/.
2. Vordel. http://www.vordel.com/.
3. B. Benatallah, F. Casasti, F. Toumani, and R. Hamadi. Conceptual modeling of web service conversations. Technical Report HPL-2003-60, HP Laboratories Palo Alto, March 2003.
4. R. Bhatti, E. Bertino, and A. Ghafoor. A trust-based context-aware access control model for web services. In *2nd IEEE International Conference on Web Services (ICWS)*, July 2004.
5. R. Bhatti, J. B. D. Joshi, E. Bertino, and A. Ghafoor. Access control in dynamic XML-based web services using X-RBAC. In *First International Conference on Web Services (ICWS)*, June 2003.
6. R. Bhatti, J. B. D. Joshi, E. Bertino, and A. Ghafoor. X-GTRBAC admin: A decentralized administration model for enterprise-wide access control. In *9th ACM Symposium on Access Control Models and Technologies (SACMAT)*, June 2005.
7. R. Bhatti, J. B. D. Joshi, E. Bertino, and A. Ghafoor. X-GTRBAC:an xml-based policy specification framework and architecture for enterprise-wide access control. *ACM Transactions on Information and System Security (TISSEC)*, 8(2), May 2005.

8. D. Clark and D. Wilson. A comparison of commercial and military computer security policies. In *IEEE Symposium on Security and Privacy*, pages 184–194, Oakland, April 1987.

9. D. Clark and D. Wilson. Evolution of a model for computer integrity. In *Eleventh National Computer Security Conference*, Baltimore, October 1988.

10. V. Dhankhar, S. Kaushik, and D. Wijesekera. XACML policies for exclusive resource usage. Technical Report ISE-TR-07-03, ISE Department, George Mason University, Fairfax, April 2007.

11. S. Farrell and R. Housley. RFC 3281- an internet attribute certificate, April 2002.

12. D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. 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.

13. S. Haddad, P. Moreaux, and S. Rampacek. Client synthesis for Web Services by way of a timed semantics. In *8th International Conference on Enterprise Information Systems*, May 2006. ICEIS 06.

14. J. B. Joshi, E. Bertino, U. Latif, and A. Ghafoor. A generalized temporal role-based access control model. *IEEE Transaction on Knowledge and Data Engineering*, 17(1), January 2005.

15. R. Lepro. Cardea: Dynamic access control in distributed systems. Technical Report NAS-03-020, NASA Advanced Supercomputing (NAS) Division, NASA Ames Research Center, Moffet Field, CA, Nov 2003.

16. OASIS. Business process execution language for web services, May 2003.

17. OASIS. Core and hierarchical role based access control (rbac) profile of xacml v2.0. http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-rbac-profile1-spec-os.pdf, Feb 2005.

18. OASIS. Extensible access control markup language, Feb 2005.

19. RFC 2753. A framework for policy-based admission control.

20. R. S. Sandhu. A lattice interpretation of the chinese wall policy. In *Proc. 15th NIST-NCSC National Computer Security Conference*, pages 329–339, 1992.

21. A. S. Tanenbaum and M. v. Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2002.

22. A. S. Tannenbaum. *Modern operating systems*. Prentice-Hall Inc., Englewood Cliffs, NJ, 1992.

23. F. Tartanoglu, V. Issarny, N. Levy, and A. Romanovsky. Dependability in the web service architecture. In *ICSE Workshop on Architecting Dependable Systems*, Orlando, FL, May 2002.