CHAPTER 5

XML ACCESS CONTROL: MAPPING XACML POLICIES AND RULES INTO TEMPORAL RELATIONS

Although XACML is an integral policy description language, the structure of the existing XACML policy is very complex and hard. Therefore, it is necessary for the users to understand XACML well, in order to write all the securing policy specifications. On the other hand, the Query languages of a RDBMS are easy and simple to use by all the users. Moreover, SQL-like query languages overcome the difficulties of XACML, by storing the XACML policies and rules in relations. Therefore, it is easy for the users to use and understand the XACML policies and rules. Moreover, storing the data and rules in tables provides more flexibility. Hence, it is necessary to provide the table based storage features of web databases. This chapter proposes new algorithms for mapping the XACML policies and rules into relational rules, and storing them in the form of relational rules in temporal relations, to ease the access control to the XML documents. The results of the new algorithms are temporal relations, because the XACML polices may have a time-stamp in the <Environment> or in the <Condition>. Moreover, these algorithms are capable for mapping the policies of the RBAC profile of XACML that may have a time-stamp in the <Environment> or in the <Condition>. The implementation of the proposed algorithms demonstrates a significant access decision time.
5.1 INTRODUCTION

XML is the de facto standard for data representation and exchange over the web, and it is adopted in communication networks. XML documents over the World Wide Web include sensitive data, such as financial and scientific data. To protect these critical data, it is necessary to apply protection techniques on the content of the XML documents. Hence, only users who are authorized to access the XML documents are granted access to sensitive data (Patel & Atay 2011).

According to Abassi et al (2010), an access control algorithm for XML documents must provide expressiveness, modularity, interoperability, and efficiency. Expressiveness helps to write a wide range of security policy specifications effectively. Modularity provides flexibility for policies composition, and interoperability provides the ability for the access control policies to interact. Finally, efficiency is useful to determine whether an access to an element can be granted or denied by the security policy.

The XML documents can be stored in either native XML databases, relational databases, or a hybrid of both relational and XML databases. In the past, many techniques were proposed by various researchers to secure native XML databases. However, there are only a few techniques, which are proposed to control the access to the XML documents stored in relational databases (Tan et al 2001, Lee et al 2003, Luo et al 2007, and Patel & Atay 2011). Lee et al (2003), Luo et al (2007) proposed techniques for storing schema-less XML documents in relational databases. Tan et al (2001) proposed a new technique, which is not pure relational but enforces the access control rules from outside the relational engine, thus leading to a decrease in performance. Patel & Atay (2011) proposed an access control model, named XML to Relational Authorization Rule (X2RAC), for the storage of the XML documents in relations. Their model allows the security administrators to
specify the authorization rules in XML, and store them in relations. However, their technique maps only rules but not policies. Moreover, they included several issues in the mapping of an object parameter, and its condition to their relational entities, which caused some performance overhead.

XACML is the result of OASIS, which is useful to express and interchange access control policies and requests/responses effectively (Rissanen 2013). The XACML standard defines a declarative access control policy language, which has been implemented in XML. It also provides a processing model, which describes the techniques on how to evaluate the authorization requests according to the rules defined in the policies. This language offers the functionalities present in most of the security policy languages. Moreover, the structure of the XACML policy language is more complex, and difficult to learn and use for users. To overcome all these problems, this thesis proposes new algorithms that hide the complexity of the XACML policy definition in a RDBMS. These new algorithms help to store the XACML policies in temporal relations effectively. Hence, the proposed algorithms store the XACML policies and rules, in the form of rules in a relational database system, and hence, can control the access to the XML documents effectively.

The major advantages of this proposed work are:

1. It relieves the users from the effort of learning and understanding the XACML policies and rules; hence, it leads to reduction in the users' time and effort.

2. It controls the access to the XML documents stored in either native or relational databases, using the XACML policies.

3. It applies the constraints of the rules, and obligations and provides the response to an access request effectively.
5.2 MAPPING XACML POLICIES AND RULES INTO TEMPORAL RELATIONS

XACML is one of the most efficient languages used for describing access control policies to web documents. However, it has a complicated representation structure, and is not simple to manipulate for non-expert users. Hence, this thesis proposes new algorithms to overcome this complexity, and store the XACML policies and rules, in the form of relational rules. Moreover, triggers are created for these temporal relations, to provide integrity and security. This section introduces the proposed work that maps the XACML policies and rules into temporal relations.

5.2.1 Proposed Mapping Algorithm

The procedure of the proposed mapping algorithm, which facilitates the mapping of the XACML policies into relational rules, and storing them in the form of relational rules in temporal relations, is presented in Figure 5.1:

<table>
<thead>
<tr>
<th>Input</th>
<th>An XACML rule, policy, or policy set document.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>Authorization relations and corresponding triggers.</td>
</tr>
</tbody>
</table>

BEGIN:

Step 1: Parse the XACML document.

Step 2: IF a <PolicySet> is the root

    Call PolicySet_as_root().

Step 3: ELSE IF a <Policy> is the root

    Call Policy_as_root().

Step 4: ELSE

    Call Rule_as_root().

END IF.

END.

Figure 5.1 Procedure of the Proposed Mapping Algorithm
This mapping algorithm begins by parsing the XACML document, and checking which the root of the document is. If the <PolicySet> is the root, it calls the method PolicySet_as_root (). Else, if the <Policy> is the root, it calls the method Policy_as_root (). Else, it calls the method Rule_as_root (). The three methods are presented as follows:

5.2.1.1 Policy Set is the Root

When the root of the XACML document is a <PolicySet>, the mapping algorithm calls the method PolicySet_as_root (). The steps of the algorithm of the method PolicySet_as_root () are as follows:

PolicySet_as_root ():

BEGIN:

Step 1: Create a PolicySet table:

\[ \text{PolicySet (PolicySetID, PCAlgId, Description).} \]

Step 2: IF the <Target> is non-empty, create a PolicySetTarget table:

\[ \text{PSTarget (PSTargetID, PolicySetID) and call Check_target_components (PSTargetID).} \]

END IF.

Step 3: IF the <PolicySet> has <Policy> elements:

a. Create a Policy table:

\[ \text{Policy (PolicyID, RCAlgId, Description, PolicySetID).} \]

b. FOR each <Policy> do:

i. IF the <Target> is non-empty, create a PolicyTarget table:

\[ \text{PTarget (PTargetID, PolicyID) and call Check_target_components (PTargetID).} \]
ii. END IF.

iii. Call Create_rule_table (PolicyID).

iv. IF the <Policy> has an <ObligationExpressions>, call
    Create_policy_obligation_table (PolicyID).

v. END IF.

c. END FOR.

END IF.

END.

The steps of the algorithm of the method Create_rule_table (PolicyID_fk) are as follows:

Create_rule_table (PolicyID_fk):

BEGIN:

Step 1: Create a Rule table:

Rule (RuleID, Effect, Description, PolicyID_fk).

Step 2: FOR each <Rule> do:

a. IF a <Target> exists and is non-empty, create a RuleTarget table: RTarget (RTargetID, RuleID) and call
    Check_target_components (RTargetID).

b. END IF.

c. IF the <Rule> has a <Condition>, call
    Create_rule_condition_table (RuleID).

d. END IF.

e. IF the <Rule> has an <ObligationExpressions>, call
Create_rule_obligation_table (RuleID).

f. END IF.

END FOR.

END.

The steps of the algorithm of the method Create_rule_condition_table (RuleID_fk) are as follows:

Create_rule_condition_table (RuleID_fk):

BEGIN:

Step 1: Create a Condition table: Condition (CondID, RuleID_fk).

Step 2: Create a CondApply table:

CondApply (AppID, CondID, FuncID, AppID_fk, CondID_fk).

Step 3: FOR each <Apply> do:

a. IF it has an <AttributeValue>, create an ApplyAttVal table:

ApplyAttVal (ID, AttDT, AttVal, AppID, CondID).

b. ELSE IF it has an <AttributeDesignator>, create an ApplyAttDes table: ApplyAttDes (ID, AttDesID, AttDesDT, AttDesCat, AppID, CondID).  
c. ELSE create an ApplyAttSel table: ApplyAttSel (ID, Path, AttSelDT, AttSelCat, AppID, CondID).

d. END IF.

END FOR.

END.

The steps of the algorithm of the method Create_rule_obligation_table (RuleID_fk) are as follows:
Create_rule_obligation_table (RuleID_fk):

BEGIN:

Step 1: Create a ROblExp table: ROblExp (OID, OblID, FulfillOn, RuleID_fk).

Step 2: FOR each <ObligationExpression> do:

a. FOR each <AttributeAssignmentExpression> do:

   i. IF it has an <AttributeDesignator>, create a ROblAttDes table: ROblAttDes (ID, AttDesID, AttDesDT, AttDesCat, OID).

   ii. ELSE IF it has an <AttributeSelector>, create a ROblAttSel table: ROblAttSel (ID, Path, AttSelDT, AttSelCat, OID).

   iii. ELSE create a ROblAttVal table: ROblAttVal (ID, AttDT, AttVal, OID).

   iv. END IF.

b. END FOR.

END FOR.

END.

The steps of the algorithm of the method Create_policy_obligation_table (PolicyID_fk) are as follows:

Create_policy_obligation_table (PolicyID_fk):

BEGIN:

Step 1: Create a POblExp table: POblExp (OID, OblID, FulfillOn, PolicyID_fk).
**Step 2:** FOR each <ObligationExpression> do:

a. FOR each <AttributeAssignmentExpression> do:

i. IF it has an <AttributeDesignator>, create a POblAttDes table: \texttt{POblAttDes (ID, AttDesID, AttDesDT, AttDesCat, OID)}.

ii. ELSE IF it has an <AttributeSelector>, create a POblAttSel table: \texttt{POblAttSel (ID, Path, AttSelDT, AttSelCat, OID)}.

iii. ELSE create a POblAttVal table: \texttt{POblAttVal (ID, AttDT, AttVal, OID)}.

iv. END IF.

b. END FOR.

END FOR.

END.

As stated above, the steps of the algorithm of the method PolicySet\_as\_root () are executed as follows:

1. **Step 1** creates a table, called a PolicySet table:

   \texttt{PolicySet (PolicySetID, PCAlgId, Description)}.

2. **Step 2** checks whether the <Target> of the <PolicySet> is empty or not. If it is non-empty, it creates a PSTarget table: \texttt{PSTarget (PSTargetID, PolicySetID)} and calls the method Check\_target\_components (PSTargetID).
3. In **step 3**, if the <PolicySet> has <Policy> elements, **step 3.a** creates a Policy table:

   **Policy** (PolicyID, RCAlgId, Description, PolicySetID).

4. **Step 3.b** loops over all the <Policy> elements and for each one, it calls the method Create_rule_table (PolicyID) and checks whether the <Policy> has a <Target> and an <ObligationExpressions> or not.

5. If the <Policy> has a non-empty <Target>, **step 3.b.i** creates a PTarget table: **PTarget** (PTargetID, PolicyID) and calls the method Check_target_components (PTargetID).

6. For <Rule> elements, **step 3.b.iii** calls the method Create_rule_table (PolicyID) that creates a Rule table: **Rule** (RuleID, Effect, Description, PolicyID) and loops over all the <Rule> elements, to check whether each <Rule> has a <Target>, a <Condition>, and an <ObligationExpressions> or not.

7. If the <Rule> has a non-empty <Target>, the algorithm of the method Create_rule_table (PolicyID) creates an RTarget table: **RTarget** (RTargetID, RuleID) and calls the method Check_target_components (RTargetID).

8. If the <Rule> has a <Condition>, the algorithm of the method Create_rule_table(PolicyID) calls the method Create_rule_condition_table (RuleID). The algorithm of the method Create_rule_condition_table (RuleID) creates a Condition table: **Condition** (CondID, RuleID) and a CondApply table: **CondApply** (AppID, CondID, FuncID, AppID_fk, CondID_fk), to store the function of each <Apply>
and the functions of its children and descendants. Moreover, the
algorithm of the method Create_rule_condition_table (RuleID)
creates:

- An ApplyAttVal table: **ApplyAttVal** (ID, AttDT, AttVal, AppID, CondID),
- An ApplyAttDes table: **ApplyAttDes** (ID, AttDesID, AttDesDT, AttDesCat, AppID, CondID),
- Or an ApplyAttSel table: **ApplyAttSel** (ID, Path, AttSelDT, AttSelCat, AppID, CondID), to store the arguments of each <Apply>.

9. If the <Rule> has an <ObligationExpressions>, the algorithm of the method Create_rule_table (PolicyID) calls the method Create_rule_obligation_table (RuleID). The algorithm of the method Create_rule_obligation_table (RuleID) creates a ROblExp table: **ROblExp** (OID, OblID, FulfillOn, RuleID) and loops over all the <ObligationExpression> and <AttributeAssignmentExpression> elements, to create either:

- A ROblAttDes table: **ROblAttDes** (ID, AttDesID, AttDesDT, AttDesCat, OID),
- A ROblAttSel table: **ROblAttSel** (ID, Path, AttSelDT, AttSelCat, OID),
- Or a ROblAttVal table: **ROblAttVal** (ID, AttDT, AttVal, OID), to store the attribute values that describe the action specified by the rule’s obligation.

10. If the <Policy> has an <ObligationExpressions>, **step 3.b.iv**
calls the method Create_policy_obligation_table (PolicyID).
The algorithm of the method Create_policy_obligation_table (PolicyID) creates a POblExp table: \texttt{POblExp} (\texttt{OID}, OblID, FulfillOn, PolicyID) and loops over all the \texttt{<ObligationExpression>} and \texttt{<AttributeAssignmentExpression>} elements, to create either:

- A POblAttDes table: \texttt{POblAttDes} (ID, AttDesID, AttDesDT, AttDesCat, OID),
- A POblAttSel table: \texttt{POblAttSel} (ID, Path, AttSelDT, AttSelCat, OID),
- Or a POblAttVal table: \texttt{POblAttVal} (ID, AttDT, AttVal, OID), to store the attribute values that describe the action specified by policy's obligation.

The algorithm for checking the target components, called Check_target_components (TargetID\_fk) is presented as follows:

\textbf{Check_target_components (TargetID\_fk):}

\textbf{BEGIN:}

\textbf{Step 1:} IF a \texttt{<Subjects>} exists and is non-empty, create a Subjects table:

\texttt{Subjects} (ID, SubMatchID, AttVal, AttDT, AttDesID, AttDesDT, AttDesCat, TargetID\_fk).

END IF.

\textbf{Step 2:} IF a \texttt{<Resources>} exists and is non-empty, create a Resources table: \texttt{Resources} (ID, ResMatchID, AttVal, AttDT, AttDesID, AttDesDT, AttDesCat, TargetID\_fk).

END IF.
Step 3: IF an <Actions> exists and is non-empty, create an Actions table:

Actions (ID, ActionMatchID, AttVal, AttDT, AttDesID, AttDesDT, AttDesCat, TargetID_fk).

END IF.

Step 4: IF an <Environments> exists and is non-empty, create an Environments table: Environments (ID, EnvMatchID, AttVal, AttDT, AttDesID, AttDesDT, AttDesCat, TargetID_fk).

END IF.

END.

The steps of the algorithm of the method Check_target_components (TargetID_fk) are executed as follows:

1. Step 1 checks whether a <Subjects> exists and is non-empty or not. If the <Subjects> exists and is non-empty, it creates a Subjects table: Subjects (ID, SubMatchID, AttVal, AttDT, AttDesID, AttDesDT, AttDesCat, TargetID_fk).

2. Step 2 checks whether a <Resources> exists and is non-empty or not. If the <Resources> exists and is non-empty, it creates a Resources table: Resources (ID, ResMatchID, AttVal, AttDT, AttDesID, AttDesDT, AttDesCat, TargetID_fk).

3. Step 3 checks whether an <Actions> exists and is non-empty or not. If the <Actions> exists and is non-empty, it creates an Actions table: Actions (ID, ActionMatchID, AttVal, AttDT, AttDesID, AttDesDT, AttDesCat, TargetID_fk).

4. Step 4 checks whether an <Environments> exists and is non-empty or not. If the <Environments> exists and is non-empty, it
creates an Environments table: **Environments** (ID, EnvMatchID, AttVal, AttDT, AttDesID, AttDesDT, AttDesCat, TargetID_fk).

### 5.2.1.2 Policy is the Root

When a `<Policy>` is the root, the mapping algorithm invokes the method `Policy_as_root()`. The steps of the algorithm of the method `Policy_as_root()` are as follows:

**Policy_as_root():**

**BEGIN:**

**Step 1:** Create a Policy table: **Policy** (PolicyID, RCAlgId, Description).

**Step 2:** IF the `<Target>` is non-empty, create a PolicyTarget table:

PTarget (PTargetID, PolicyID) and call

Check_target_components (PTargetID).

END IF.

**Step 3:** Call Create_rule_table (PolicyID).

**Step 4:** IF the `<Policy>` has an `<ObligationExpressions>`, call

Create_policy_obligation_table (PolicyID).

END IF.

**END.**

The steps of the algorithm of the method `Policy_as_root()` are executed as follows:

1. **Step 1** creates a Policy table: **Policy** (PolicyID, RCAlgId, Description).
2. **Step 2** checks whether the `<Policy>` has a non-empty `<Target>` or not. If the `<Target>` is non-empty, it creates a PTarget table and calls the method Check_target_components (PTargetID).

3. **Step 3** calls the method Create_rule_table (PolicyID). The algorithm of the method Create_rule_table (PolicyID) creates a Rule table and loops over all the `<Rule>` elements, to checks whether each `<Rule>` has a `<Target>`, a `<Condition>`, and an `<ObligationExpressions>` or not.

4. If the `<Policy>` has an `<ObligationExpressions>`, **step 4** calls the method Create_policy_obligation_table (PolicyID). The algorithm of the method Create_policy_obligation_table (PolicyID) creates a POblExp table and loops over all the `<ObligationExpression>` and over all the `<AttributeAssignmentExpression>` elements, to create for each `<AttributeAssignmentExpression>` either a POblAttDes, POblAttSel, or POblAttVal table, to store the attribute values that describe the action specified by the policy's obligation.

### 5.2.1.3 Rule is the Root

Finally, when a `<Rule>` is the root of the XACML document, the mapping algorithm calls the method Rule_as_root (). The steps of the algorithm of the method Rule_as_root () are as follows:

**Rule_as_root ()**: 

BEGIN: 

**Step 1**: Create a Rule table: **Rule (RuleID, Effect, Description)**.

**Step 2**: IF a `<Target>` exists and is non-empty, create a RuleTarget table:
**RTarget (RTargetID, RuleID)** and call Check_target_components (RTargetID).

END IF.

**Step 3:** IF the <Rule> has a <Condition>, call

Create_rule_condition_table (RuleID).

END IF.

**Step 4:** IF the <Rule> has an <ObligationExpressions>, call

Create_rule_obligation_table (RuleID).

END IF.

END.

The above code of the method runs as follows:

1. **Step 1** creates a Rule table: **Rule (RuleID, Effect, Description).**

2. If the <Rule> has a non-empty <Target>, **step 2** creates an RTarget table and calls the method Check_target_components (RTargetID).

3. If the <Rule> has a <Condition>, **step 3** calls the method Create_rule_condition_table (RuleID). The algorithm of the method Create_rule_condition_table (RuleID) creates a Condition table and a CondApply table, to store the function of each <Apply> and the functions of its children and descendants. Moreover, the algorithm of the method Create_rule_condition_table (RuleID) creates tables, namely, an ApplyAttVal, ApplyAttDes, or ApplyAttSel, to store the arguments of each <Apply>. 
4. If the <Rule> has an <ObligationExpressions>, step 4 calls the method Create_rule_obligation_table (RuleID). The algorithm of the method Create_rule_obligation_table (RuleID) creates a ROblExp table and loops over all the <ObligationExpression> and over all the <AttributeAssignmentExpression> elements, to create for each <AttributeAssignmentExpression> either a ROblAttDes, ROblAttSel, or ROblAttVal table, to store the attribute values that describe the action specified by the rule's obligation.

In addition to the mapping of the rule, policy, or policy set documents, the proposed mapping algorithm also maps the request documents into relations, using the same idea of the method Check_target_components (arg).

5.3 IMPLEMENTATION AND RESULTS

The implementation of the proposed mapping algorithm has been carried out in Java platform, using MS Access as a database. The MS Access tables are used for storing the XACML policies and rules in the forms of relational rules, and triggers have been created for these tables, to provide integrity and security. The performance evaluation of the mapping algorithm has been done on a Toshiba laptop running Windows 7 with Pentium (R) Dual-Core CPU 2.20 GHz and 3GB RAM. Before using the application, the XACML document has been enhanced as follows:

1. If the <PolicySet>, <Policy>, or <Rule> has a <Target>, change the <Target> to be <PSTarget>, <PTarget>, and <RTarget> respectively, to differentiate among them in parsing.
2. If the <Policy> or <Rule> has an <ObligationExpressions>, change it to <PObligationExpressions> and <RObligationExpressions> respectively, to distinguish between them in parsing.

3. Remove the empty elements.

5.3.1 Creating Authorization Relations

The following example explains the applying of the proposed mapping algorithm to the XACML Policy Set document in Figure 5.2, and the Request document in Figure 5.3:

```
<PolicySet PolicySetId = "pls-0001" PolicyCombiningAlgId = "urn:oasis:names:tc:xacml:3.0:policy-combining-algorithm:deny-overrides">
  <Description>
    It is the description of the policy set
  </Description>
  <PSTarget>
    <Subjects>
      <Subject>
        <SubjectMatch MatchId = "urn:oasis:names:tc:xacml:1.0: function:string-equal">
          <AttributeValue DataType = "http://www.w3.org/2001/XMLSchema#string">
            Omar
          </AttributeValue>
        </SubjectMatch>
        <SubjectAttributeDesignator
          AttributeId= "urn:oasis:names:tc:xacml:1.0:subject:subject-id"
          DataType = "http://www.w3.org/2001/XMLSchema#string"/>
      </Subject>
    </Subjects>
  </PSTarget>
</PolicySet>
```

Figure 5.2 (Continued)
<Policy PolicyId = "pol-0001" RuleCombiningAlgId = "urn:oasis:names:tc:xacml:3.0: rule-combining-algorithm:deny-overrides">
  <Description>
    It is the description of the policy
  </Description>
  <Rule RuleId = "rul-0001" Effect = "Permit">
    <Condition>
      <Apply FunctionId = "urn:oasis:names:tc:xacml:1.0:function:string-equal">
        <Apply FunctionId = "urn:oasis:names:tc:xacml:1.0:function:string-one-and-only">
          <AttributeDesignator
            AttributeId = "urn:oasis:names:tc:xacml:1.0:subject:subject-id"
            DataType = "http://www.w3.org/2001/XMLSchema#string"/>
          <AttributeValue
            DataType = "http://www.w3.org/2001/XMLSchema#string">
            Omar
          </AttributeValue>
        </Apply>
      </Apply>
    </Condition>
  </Rule>
  <Rule RuleId = "rul-0002" Effect = "Permit">
    <Condition>
      <Apply FunctionId = "urn:oasis:names:tc:xacml:1.0:function:string-equal">
        <Apply FunctionId = "urn:oasis:names:tc:xacml:1.0:function:string-one-and-only">
          <AttributeDesignator
            AttributeId = "urn:oasis:names:tc:xacml:1.0:subject:subject-id"
            DataType = "http://www.w3.org/2001/XMLSchema#string"/>
          <AttributeValue
            DataType = "http://www.w3.org/2001/XMLSchema#string">
            Ahmed
          </AttributeValue>
        </Apply>
      </Apply>
    </Condition>
  </Rule>
</Policy>

Figure 5.2 (Continued)
Figure 5.2 Example of an XACML Policy Set Document
Figure 5.3 (Continued)
Figure 5.3 Example of an XACML Request Document

When the application of the mapping algorithm is running, a main form appears that allows the user to select the Policy Set document or the Request document. Hence, when the selection is the Policy Set document as in Figure 5.2 or the Request document as in Figure 5.3, the application shows a form with the corresponding relations of the Policy Set document or the Request document, according to the proposed mapping algorithm. The application executes as follows:

1. When the selection is the Policy Set document in Figure 5.2, the application executes step 1 of the mapping algorithm that
parses the XACML Policy Set document and find that, the root is a <PolicySet>.

2. **Step 2** of the mapping algorithm calls the method PolicySet
    _as_root () that creates the following **authorization relations**: 

   PolicySet (PolicySetID, PCAlgId, Description).

   PSTarget (PSTargetID, PolicySetID).

   PSTargetSubjects (ID, SubMatchID, AttVal, AttDT, AttDesID, AttDesDT,
    AttDesCat, PSTargetID).

   Policy (PolicyID, RCAlgId, Description, PolicySetID).

   Rule (RuleID, Effect, Description, PolicyID).

   Condition (CondID, RuleID).

   CondApply (AppID, CondID, FuncID, AppID_fk, CondID_fk).

   ApplyAttVal (ID, AttDT, AttVal, AppID, CondID).

   ApplyAttDes (ID, AttDesID, AttDesDT, AttDesCat, AppID, CondID).

   POblExp (OID, OblID, FulfillOn, PolicyID).

   POblAttDes (ID, AttDesID, AttDesDT, AttDesCat, OID).

   POblAttSel (ID, Path, AttSelDT, AttSelCat, OID).

   POblAttVal (ID, AttDT, AttVal, OID).

3. When the selection is the Request document in Figure 5.3, the
   application creates the following relations:

   Request (ReqID).

   ReqSubject (SubID, ReqID).

   SubjectAtt (ID, Cat, AttID, AttVal, AttDT, SubID).
**ReqResource** (ResID, ReqID).

**ResourceAtt** (ID, Cat, AttID, AttVal, AttDT, ResID).

**ReqAction** (ActionID, ReqID).

**ActionAtt** (ID, Cat, AttID, AttVal, AttDT, ActionID).

**ReqEnv** (EnvID, ReqID).

**EnvAtt** (ID, Cat, AttID, AttVal, AttDT, EnvID).

Therefore, by using the authorization relations, the access control to the XML documents will be easier and more effective, than while, using the complex structure of the XACML policies.

### 5.3.2 Querying on Authorization Relations

Now, the authorization relations can be used to provide the conditions results, obligations, and response to the request as follows:

Since the application created a Condition relation, the following condition query is executed to get the results of the conditions that may be true or false:

```
SELECT RuleID, CondID, IIf ([SubjectAtt.AttVal] = [ApplyAttVal.AttVal], True, False) AS CondRes
FROM CondApply, Condition, ApplyAttDes, ApplyAttVal, SubjectAtt
```
This query retrieves the name value of the subject and compares it with the two name values stated in the two conditions. If the name value of the subject equals any name value of the two name values stated in the two conditions, the query returns True. Otherwise, it returns False. If the result of the condition query is False, the corresponding rule is NotApplicable. Otherwise, the corresponding rule is applicable. The results of the condition query are shown in Table 5.1:

**Table 5.1 Condition Query Results**

<table>
<thead>
<tr>
<th>RuleID</th>
<th>CondID</th>
<th>CondRes</th>
</tr>
</thead>
<tbody>
<tr>
<td>rul-0001</td>
<td>1</td>
<td>True</td>
</tr>
<tr>
<td>rul-0002</td>
<td>2</td>
<td>False</td>
</tr>
</tbody>
</table>

Table 5.1 states that, the result of the condition of the first rule is True; thus it is applicable, and the result of the condition of the second rule is False; thus it is NotApplicable. Since the first rule is applicable, the response query is executed to get the Effect of the first rule:

```sql
FROM Rule, Policy, PolicySet, ConditionQuery, PSTarget,
PSTargetSubjects, SubjectAtt, Request, ReqSubject,
ReqAction, ActionAtt
```


The using of the RTarget, PTarget, or PSTarget relation in the response query is based on the following:

- IF the RTarget relation is non-empty, use it in the response query.
- ELSE IF the PTarget relation is non-empty, use it in the response query.
- ELSE use the PSTarget relation in the response query.

END IF.

Therefore, the PSTarget is used in the response query and the join is done between the PSTarget and SubjectAtt relations, because the RTarget and PTarget relations are empty.

The wizard creates the above response query in an easy way, but the SQL is used, instead of using the figures of the wizard. The rules' effects are shown in Table 5.2:

<table>
<thead>
<tr>
<th>RuleID</th>
<th>FEffect</th>
<th>CondRes</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>rul-0001</td>
<td>Permit</td>
<td>True</td>
<td>read</td>
</tr>
<tr>
<td>Rul-0002</td>
<td>NotApplicable</td>
<td>False</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.2 states that, the evaluation of the first rule is Permit and the evaluation of the second rule is NotApplicable. Hence, according to the
rule-combining algorithm, named Deny overrides algorithm, the policy evaluates to Permit, and according to the policy-combining algorithm, named Deny overrides algorithm, the policy set evaluates to Permit.

Since the response permits the access request to read the resource, the obligation must be performed. To know the obligation and perform it, the following obligation query is executed:

\[
\text{SELECT} \quad \text{OblID, FulfillOn, Path, SubjectAtt.AttVal AS Subject,} \\
\text{POblAttVal.AttVal AS Value} \\
\text{FROM} \quad \text{POblExp, POblAttDes, POblAttSel, POblAttVal, SubjectAtt} \\
\text{WHERE} \quad \text{POblExp.OID = POblAttSel.OID and POblExp.OID = POblAttDes.OID and POblExp.OID = POblAttVal.OID and POblAttDes.AttDesDT = SubjectAtt.AttDT and POblAttDes.AttDesID = SubjectAtt.AttID;}
\]

From the above obligation query, the result is shown in Table 5.3:

<table>
<thead>
<tr>
<th>OblID</th>
<th>FulfillOn</th>
<th>Path</th>
<th>Subject</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>email</td>
<td>Permit</td>
<td>Zizo/Contact/email</td>
<td>Omar</td>
<td>Your record is accessed</td>
</tr>
</tbody>
</table>

In Table 5.3, the obligation is sending an email to the subject Omar. The email is sent to Omar when the response's effect is Permit. The email address of the subject Omar is an element, called email, contained in the resource, called Zizo element, and its path is Zizo/Contact/email. If the resource is stored in native XML databases, the email address of the subject is retrieved by an XPath query. If the resource is stored in relational databases,
the XPath query is translated into an SQL query on the resulting relations of the resource. The body of the email is "Your record is accessed".

5.4 SUMMARY

In this chapter, new algorithms for mapping the XACML policies and rules into relational rules, and storing them in the form of rules in temporal relations, are proposed. The resulting temporal relations control the access to the XML documents effectively. These proposed algorithms relieve the users from the complexity of the XACML structure, and allow them to efficiently control the access to the XML documents, leading to reduction in their time and effort. From the experimental results obtained from this work, it is clearly proved that, these proposed algorithms map the XACML policies and rules effectively.