Synthesising Verified Access Control Systems in XACML

Nan Zhang
School of Computer Science
University of Birmingham
Birmingham, UK, B15 2TT
n.zhang@cs.bham.ac.uk

Mark Ryan
School of Computer Science
University of Birmingham
Birmingham, UK, B15 2TT
m.d.ryan@cs.bham.ac.uk

Dimitar P. Guelev
School of Computer Science
University of Birmingham
Birmingham, UK, B15 2TT
d.p.guelev@cs.bham.ac.uk

ABSTRACT

The *eXtensible Access Control Markup Language* (XACML) was proposed by the OASIS committee to be used as a standard language in e-business [6]. However, policy files written in XACML are hard to read and analyse directly. In this paper, we present a tool which generates verified XACML scripts from access control system descriptions in simple but expressive language proposed in [3], which admits algorithmic verification of access control systems against appropriately formalised policies. This allows the generation of XACML scripts for systems that can be formally verified to be implementing the relevant policies.

Categories and Subject Descriptors

F.4.3 [**Theory of Computation**]: Mathematical Logic and Formal Languages—*formal languages*

General Terms

Security, Verification

Keywords

Access control model, XACML, Access control policy language

1. INTRODUCTION

Access control is of increasing importance in a world in which computers are ever-more interconnected through networks. In order to protect resources and information from illegal access, access control systems are built to provide the ability of protection. Access control systems usually implement access control policies. Several formalisations of such policies have been proposed. For instance, *role-based access control* (RBAC) [7], prescribes assigning access rights to agents on the grounds of their having certain roles.

This paper is about a formalism for describing access control systems introduced in [3], which allows access permissions to be defined using arbitrary conditions on the current state of the system in question. Since the basic actions which are subject of permissions in that formalism are *r*eading and over*w*riting data, including permission records themselves, we denote the formalism from [3] by *RW* . *RW* descriptions admit algorithmic verification of their adherence to access control policies written in the *RW* language. In this paper we propose a machine-readable syntax for the language of *RW* and show how systems described in it can be translated into the *eXtensible Access Control Markup Language* (XACML), which was proposed by the OASIS committee to be used as a standard language in e-business [6]. Access control systems described in XACML can be bolted onto existing applications. One of the major implementations of XACML was made by *Sun Microsystems* [8]. This implementation includes a program which runs as a simple *Policy Decision Point* (PDP), which is the entity in the XACML framework whose responsibility is to make decisions on granting access requests. This program can be used to test the tool we have implemented for translating *RW* descriptions of access control systems into XACML.

The benefit from translating *RW* into XACML is twofold: First, this allows the relatively concise descriptions of access control systems in *RW* to be automatically translated into the machine-oriented XACML. Second, since the properties of systems described in *RW* can be verified algorithmically, the translation can be guaranteed to produce systems which correctly implement the required policies (See Figure 1).

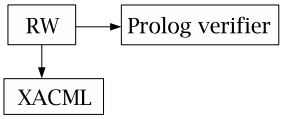

Figure 1: Two working threads

Structure of the paper

After preliminaries on the *RW* framework and XACML, we explain our translation of *RW* into XACML and draw some conclusions. We present the machine-readable syntax of *RW* in terms of examples in the main text of the paper. The complete syntax of the *RW* language and some practical instructions on the use of our implementation are in appendices.

FMSE'04, October 29, 2004, Washington, DC, USA.

2. THE *RW* ACCESS CONTROL FORMALISM

2.1 Definition

Given a set of propositional variables P, we denote the set of the propositional logic formulas built using these variables by $\mathtt{L}(P)$. An access control system S is a tuple $\langle A, P, \mathtt{r}, \mathtt{w}\rangle$, where A is a set of *agents*, P is a set of *propositional variables* and $\mathtt{r}$ and $\mathtt{w}$ are mappings of type $P \times A \to \mathtt{L}(P)$. States of S are valuations of the variables from P. Given a state s of S, agent $a \in A$ is allowed to read and overwrite variable p iff $s \models \mathtt{r}(p,a)$ and $s \models \mathtt{w}(p,a)$, respectively. Thus the formulas $\mathtt{r}(p,a)$ and $\mathtt{w}(p,a)$ define the conditions for agents to access S as functions on its state.

2.2 Example

The example access control system below is taken from [3]. We use it as a running example in this paper. It is about a conference paper reviewing system. The system includes fixed sets of agents and papers to review. Some of the agents are authors of the papers and/or participate in the conference Programme Committee (PC). The Programme Chair is a fixed agent too. Access control policies which apply in this system include[1]:

1. PC members and authors of papers are known to everybody. Authors of papers cannot be changed.
2. The PC chair appoints the PC members. A PC member can resign his membership.
3. The PC chair can assign a paper to a PC member for reviewing, except if he is one of its authors.
4. All PC members, except the author(s) of a paper can know who are the reviewers for this paper.
5. The reviewer of a paper can assign the paper to be sub-reviewed by an agent who is not an author of the paper and has not been assigned the same paper by another reviewer.
6. A reviewer of a paper p can resign, unless he has already appointed a sub-reviewer for the paper.
7. Subreviewers are known to all PC members who are not authors of the respective papers.
8. A sub-reviewer can resign, unless he has already submitted his review.

Following the RBAC approach [7], we can identify three roles in this system — *PC member*, *chair* and *anybody*. Generally speaking, rules 1, 4, and 7 deal with *reading* privileges as described in the current state of the system. The other rules are about assigning privileges, which means *overwriting* this state. The purpose of having these rules is to avoid conflicts of interest in the review process.

[1] A full list can be found in [3].

To put this example in the *RW* formalism, let `Papers` be the set of papers, `Agents` be the set of agents and P include the propositional variables for all $a, b \in \mathtt{Agents}$ and $p \in \mathtt{Papers}$:

$\mathtt{author}(p,a)$	a is an author of p
$\mathtt{pcmember}(a)$	a is a PC member
$\mathtt{chair}(a)$	a is the chair of PC
$\mathtt{reviewer}(p,a)$	p is assigned to PC member a for review
$\mathtt{subreviewer}(p,a,b)$	p is assigned by PC member a to b as a sub-reviewer
$\mathtt{submittedreview}(p,a)$	a review on p has been submitted by (sub-)reviewer a
$\mathtt{review}(p,a)$	the review on p from a itself

The permission mappings $\mathtt{r}$ and $\mathtt{w}$ can be defined as follows ("$\rightleftharpoons$" denotes "is defined as", $\mathtt{A}$ denotes the set of agents, `sub` stands for `subreviewer`, `submit` stands for `submittedreview`):

$$\mathtt{r}(\mathtt{author}(p,a),x) \rightleftharpoons \mathtt{true},\ \mathtt{r}(\mathtt{pcmember}(a),x) \rightleftharpoons \mathtt{true} \qquad \text{rule 1}$$

$$\mathtt{w}(\mathtt{pcmember}(a),x) \rightleftharpoons \mathtt{chair}(x) \vee (\mathtt{pcmember}(a) \wedge x = a) \qquad \text{rule 2}$$

$$\mathtt{r}(\mathtt{reviewer}(p,a),x) \rightleftharpoons \mathtt{pcmember}(x) \wedge \neg\mathtt{author}(p,x) \qquad \text{rule 4}$$

$$\mathtt{w}(\mathtt{reviewer}(p,a),x) \rightleftharpoons \left(\begin{array}{l} \left(\begin{array}{l} (\mathtt{chair}(x) \\ \wedge\mathtt{pcmember}(a)) \\ \wedge \\ (\neg\mathtt{author}(p,a) \\ \wedge\neg\mathtt{reviewer}(p,a)) \end{array}\right) \\ \vee \left(\begin{array}{l} \mathtt{pcmember}(a) \\ \wedge x = a \\ \wedge\mathtt{reviewer}(p,a) \\ \wedge\neg(\exists\ b \in \mathtt{A}) \\ \mathtt{sub}(p,a,b) \end{array}\right) \end{array}\right) \qquad \text{rules 3,6}$$

$$\mathtt{r}(\mathtt{sub}(p,a,b),x) \rightleftharpoons \left(\begin{array}{l} (\mathtt{pcmember}(x) \\ \wedge\neg\mathtt{author}(p,x)) \\ \vee x = b \end{array}\right) \qquad \text{rule 7}$$

$$\mathtt{w}(\mathtt{sub}(p,a,b),x) \rightleftharpoons \left(\begin{array}{l} \left(\begin{array}{l} \left(\begin{array}{l} (\mathtt{reviewer}(p,a) \\ \wedge\neg\mathtt{author}(p,b)) \\ \wedge x = a \end{array}\right) \\ \wedge\neg(\exists\ d \in \mathtt{A}) \\ \left(\begin{array}{l} \mathtt{sub}(p,a,d) \\ \vee\mathtt{sub}(p,d,b) \end{array}\right) \end{array}\right) \\ \vee \left(\begin{array}{l} (\mathtt{sub}(p,a,b) \\ \wedge x = b) \\ \wedge\neg\mathtt{submit}(p,b) \end{array}\right) \end{array}\right) \qquad \text{rules 5,8}$$

Note that the quantifier prefixes $(\exists\ d \in \mathtt{Agents})$ occurring above can be read as $\bigwedge_{a \in \mathtt{Agents}}$, because `Agents` is a fixed finite set. Every action which is not explicitly allowed is denied in this example.

2.3 A machine-readable syntax for *RW*

Figure 2 shows a description of the above example in the machine-readable syntax we propose for *RW*, in order to illustrate it. This syntax is described in full in Appendix A.

The description starts with the key word `AccessControlSystem`, followed by an identifier to name the particular system. Identifiers begin with a letter and can include letters, digits, "-" and "_". They are case-sensitive. The keyword `AccessControlSystem` and the name after form the *title* of this description.

```
AccessControlSystem Conference
        Class  Paper;

        Predicate       author(paper: Paper, agent: Agent), pcmember(agent: Agent), chair(agent: Agent),
                        reviewer(paper: Paper, agent: Agent),
                        subreviewer(paper: Paper, appointer:Agent, appointee:Agent),
                        submittedreview(paper: Paper, agent: Agent),
                        review(paper: Paper, agent: Agent);
        author(p, a){
                read : true;
        }
        pcmember(a){
                read : true;

                write : chair(user)|pcmember(a);
        }
        reviewer(p, a){
                read : pcmember(user)&~author(p, user);

                write : (chair(user)&pcmember(a)&~author(p,a)&~reviewer(p,a))
                        or ((pcmember(a)&user=a&reviewer(p,a))&~(E b: Agent [subreviewer(p,a,b)]));
        }
        subreviewer(p, a, b){
                read : (pcmember(user)&~author(p,user)) or user=b;

                write : (reviewer(p,a)&~author(p,b)&user=a&~(E d: Agent [subreviewer(p,a,d)|subreviewer(p,d,b)]))
                        | (subreviewer(p,a,b)&~submittedreview(p,b)&user=a);
        }
        ...
End
```

Figure 2: Access control policies of the *conference paper reviewing system* described in RW language

Next is the body of a description, which consists of a declaration part and a policy-definition part. The declaration part defines components of the system, including the set of agents, sets of other objects and the logical relations on agents and objects, which are represented by appropriately parameterised sets of propositional variables in the RW model. The policy definition part defines the access rights of each agent on each variable of the system state.

The declaration part consists of class declarations which define the sets of agents and objects and predicate declarations which define the logical relations. Class declaration start with the keyword `Class` followed by any number of identifiers, which are interpreted as the names of sets of objects and must start with a capital letter. The class of agents is predefined and has the standard name `Agent`. That is why the example description has only one class definition, which is of the set of papers.

The predicate declaration starts with the keyword `Predicate` followed by any number of predicates. Each predicate defines a logical relation. A predicate consists of a name of the predicate and parameters, whose types must be defined classes. Parameter names must be distinct. Parameter names must start with lowercase letters.

The policy definition part consists of policy definitions. Each definition begins with the name of a predicate, followed by a list of formal parameters, and contains a pair of logical formulas labelled by the keywords `read` or `write`. These formulas define the rights of an agent denoted by the keyword `user` to *read* and *overwrite* the truth value of the predicate for the parameters, whose names are listed in the beginning of the definition and whose types are derived from the definition of the respective predicate. Parameters defined in this way can be used freely only within the block enclosed by the curly brackets after the parameterised predicate. Thus each block defines a local name space. Variables defined in quantified formulas can only be used inside the quantified formulas. They are invisible from outside.

The logical connectives have their usual meanings. The operator $=$ can only be applied to two elements of the same set and has the usual meaning too.

3. THE EXTENSIBLE ACCESS CONTROL MARKUP LANGUAGE

The *eXtensible Access Control Markup Language* (XACML) is an access control policies description language proposed by the OASIS committee. It is intended to be used as a standard language in the field of e-business. We briefly discuss it in this section.

3.1 The mechanism of XACML framework

In [6], there is a data-flow diagram of XACML, which illustrates the mechanism of how the framework is supposed to work. We use the simplified variant of this diagram shown on Figure 3 to explain this here.

Access control policies are written in XACML, in the format of XML, and are stored in a Policy Information Point (PIP). This PIP is known to the Policy Decision Point (PDP), which is the entity that makes decisions. The Policy Enforcement Point (PEP) is the entity to enforce access

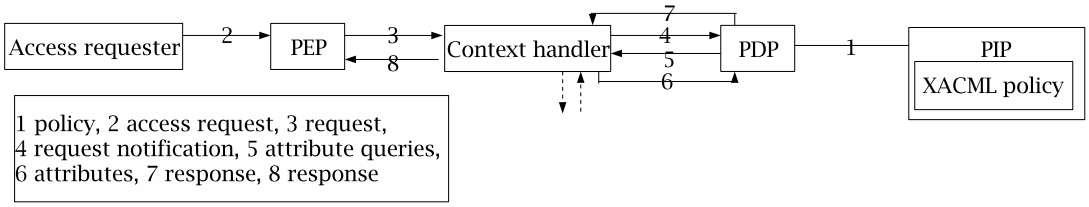

Figure 3: Data-flow diagram of XACML

control. When it receives a request, it passes it to the context handler. The request carries information about the requester and the resource to be accessed. The context handler then converts the request into XML form so that the PDP understands it. When the PDP receives the notification from the context handler, it evaluates the request on the basis of the policy provided by the PIP. It gets all the information necessary to make a decision through further communication with the context handler. The latter will get all the relevant information from different sources.

3.2 Evaluation of XACML and *RW*

There are a number of criteria for judging the expressive power of access control languages [2]. Below we briefly evaluate XACML and *RW* in terms of these criteria.

- **Conditional authorisation**. Both in XACML and *RW* , authorisation can be made to depend on arbitrary conditions, not restricted to agents' roles (see the above conference example).
- **Delegation mechanisms**. Delegation is the ability of delegators to give privileges to delegatees so that the latter can perform some actions on behalf of the former. In *RW* , delegation is achieved by establishing new relations and assigning privileges to the agents in that relation. In XACML, delegation can be achieved by simply adding new rules. Because adding new relations in *RW* corresponds to new rules being added to the XACML description.
- **Expressibility of permissions about permissions**. Permissions about permissions are also called *administrative policies* or *meta-policies*. They specify who may add, delete or modify the permissions in the system and in what manner. In *RW* , the possibility to modify policies is a consequence of the use of arbitrary conditions to determine permissions. In the XACML framework, policies are edited by administrators. However, it is still possible to convert a model in *RW* to a description in XACML without losing the ability to express permissions about permissions. This is the topic of the next section.
- **Avoidance of root bottleneck**. The chief administrator of a system is often called a *root*. The root can become a bottleneck of the system if all the administrative tasks were performed by him. Furthermore, there would be a potential danger of misusing the privileges if a root has more privileges than he needs. This is known as the *root bottleneck problem*. The including of the property of *separation of duty* in RBAC certainly reflects this consideration, while in *RW* , the integration of delegation and permissions about permissions also shows our solution to this problem. Since *RW* can be fully expressed in terms of XACML, we think that XACML also satisfies this criterion.

Note that *RW* does not have *dedicated* means for addressing each of the above issues. It rather allows the desired mechanisms to be implemented using the general means of the framework. Although *RW* and XACML have similarities, as discussed above, their motivations are very different. *RW* offers a compact, readable syntax for describing access control systems, and the possibility of model checking their properties. XACML provides the possibility to implement access control on an existing system, but its syntax is neither compact nor readable. We address these problems by providing a compiler from *RW* to XACML.

4. COMPILING *RW* TO XACML

We have written a tool in *Java* which performs the task of translating a model in *RW* to a description in XACML. In this section we explain how this is done. For instructions about how to use the compiler, one can read the *readme* file in the package [9].

4.1 The structure of the converted XACML file

The compiler reads a *RW* file and outputs the corresponding XACML file. The XACML file is a single policy unit which contains a number of rules. Each conditional formula in the *RW* file is converted into a rule in XACML plus a default rule which denies everything. The structure of the output XACML file is shown in Figure 4.

The title of the policy is composed of the content specified in the specification of XACML 1.1 [6], but one can make one's own modifications to suit one's own need. We chose the *permit-overrides algorithm*. This is one of the algorithms which are used in XACML to reconcile the decisions from the possibly several rules which apply to a given request. The algorithm produces *Permit*, provided that at least one of the applicable rules does so. If some rules produce *Deny* and all other rules produce *NotApplicable*, the algorithm returns *Deny*. In other words, *Permit* takes precedence, regardless of the result of evaluating any of the other rules in the policy.

The target of the policy is made to apply to every situation. No target applicability constraint is needed at the policy level, because each rule redefines its applicability in its own *Target* tag. Rules are defined following the *Target* tag of the policy. The effect of all rules, except the last default one, is *Permit*. Using the *permit-overrides* algorithm in this arrangement causes the system to deny whatever is not explicitly permitted.

For the example of the *conference paper reviewing system*, the generated XACML policy contains eleven rules, num-

```
<?xml version="1.0" encoding="UTF-8"?>
<Policy xmlns= ...
PolicyId="conference"
RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:ordered-permit-overrides">

<Description>add your own comment</Description>

<Target>
  <Subjects><AnySubject/></Subjects>
  <Resources><AnyResource/></Resources>
  <Actions><AnyAction/></Actions>
</Target>
...
<Rule RuleId="urn:oasis:names:tc:xacml:1.0:Rule8" Effect="Permit">
  <Target>
    <Subjects><AnySubject/></Subjects>
    <Resources>...</Resources>
    <Actions>...</Actions>
  </Target>

  <Condition ...>
  ...
  <\Condition>
<\Rule>
...
<Rule RuleId="urn:oasis:names:tc:xacml:1.0:Rule11" Effect="Deny">

  <Target>
    <Subjects><AnySubject/></Subjects>
    <Resources><AnyResource/></Resources>
    <Actions><AnyAction/></Actions>
  </Target>
</Rule>
</Policy>
```

Figure 4: The structure of an output XACML file

bered from *zero* to *ten*, plus the default rule. Each rule is generated according to a conditional formula and the predicate it applies to in the *RW* file. An example of this correspondence for the case of the predicate `pcmember`(.) is shown on Figure 5. The target of a rule is defined so that the applicable situations are defined for this rule. The requester's credentials are evaluated in the *Condition* tag, therefore no restriction is demanded on the *Subjects* tag. This is the case with all rules generated by the compiler.

The *Resources* tag has two criteria to be evaluated in this example (see Figure 5). The first one is whether the name of the requesting resource is the predicate name `pcmember`. At this point, the XACML file instructs the PDP to select the attribute value from the *resource-id* field in the request context, which is shown in Figure 6. If the name of the requesting resource is not `pcmember`, this rule is simply evaluated as *not applicable*. The second criterion is whether the name of the parameter whose type is `Agent` is *agent*, according to the definition of predicate `pcmember(agent : Agent)` in the *RW* file. This information is retrieved from the field of *agent* in the request. The value retrieved, in this case, would be *agent=a1*, which specifies that the name of the parameter equals its actual value. Here we introduce a dedicated external function which compares the string selected from the XACML, which in this case is *agent*, with the string on the left side of the equivalent formula which is expected to be *agent* too. The two criteria are enclosed in one *Resource* tag, which means the conjunction of these criteria. For the evaluation to be successful, both of these criteria must be met.

The *Actions* tag is to evaluate whether the attribute value of the *action-id* field in the request matches the applicable action. Since the condition applies to the privilege of reading in this example, the applicable action for this rule is *read*. Since the condition for reading in the *RW* file is simply *true*, the *Condition* tag is omitted in the XACML file. However, if the condition in the *RW* file is non-trivial, a *Condition* tag is added to the generated rule. We explain how such tags are generated in the next sub-section.

4.2 Generating the conditions

Logical formulas in *RW* are converted into conditions which are enclosed within *Condition* tags. For each rule the *Condition* tag is called to be evaluated if the target evaluation is passed.

The compiler converts each logical formula in *RW* into an SQL statement and puts it under the *Condition* tag. Later this SQL statement is evaluated by calling a dedicated external function, which queries a database we have set up for supporting this case study. To illustrate the idea, consider the *RW* formula and the corresponding condition shown on Figure 7.

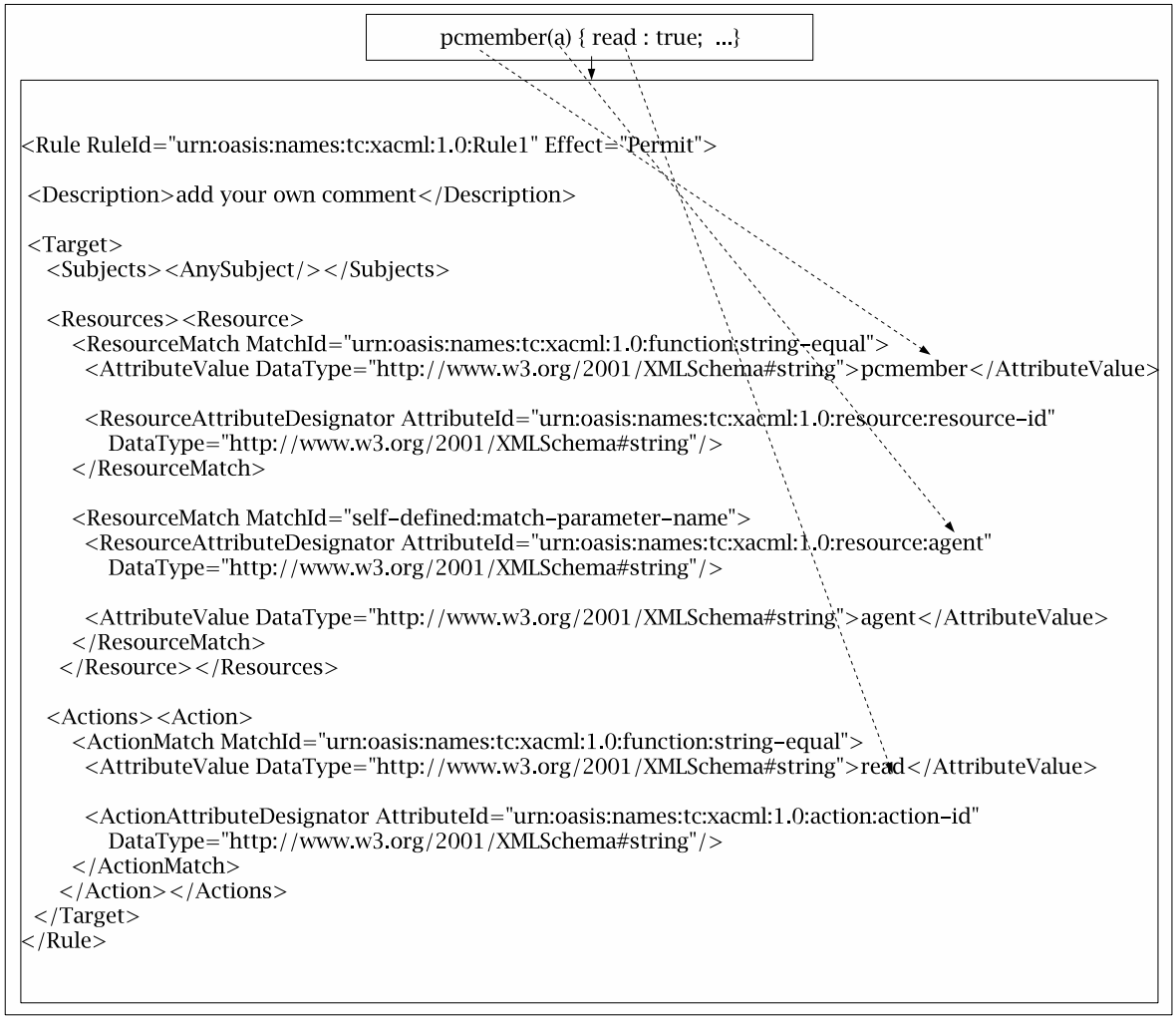

Figure 5: A condition in the *RW* file and its generated rule

The idea is to look up the truth values of the variables in P in the database. For instance, given $a \in \texttt{Agent}$, the truth value of $\texttt{pcmember}(a)$ is determined by looking up a in a table which contains all the agents who possess a role of *PC member*, which is supposed to be found in the database. Truth values of other variables are looked up similarly. Thus any boolean combination of such variables can be evaluated. To do this, each logical formula in *RW* is converted into an SQL statement. The external function *self-defined:evaluate-sql*, which appears as a conditional function in *Condition* tags, evaluates SQL statements. The mapping SQL from *RW* conditions to SQL statements described below defines the correspondence implemented by our compiler:

$\texttt{SQL}(\texttt{pcmember}(a)) \rightleftharpoons$ EXISTS (SELECT * FROM pcmember WHERE agent=arg_agent) This is an example for the simplest case, in which the condition contains just one predicate with one parameter. Then the condition is equivalent to the non-emptiness of the selection result. The column name *agent* is derived from the definition of the predicate $\texttt{pcmember}(\texttt{agent} : \texttt{Agent})$. Thus the table *pcmember* must contain a column named *agent*. The string *arg_agent* is the formal name of parameter a. The actual value of this parameter comes from the request. The external function replaces formal names by their actual values and produces an executable SQL statement.

$\texttt{SQL}(\texttt{R}(a_1, ..., a_n)) \rightleftharpoons$ EXISTS (SELECT * FROM R WHERE $\texttt{R}_{c1}{=}a'_1$ AND ... AND $\texttt{R}_{cn}{=}a'_n$) If the predicate has more than one parameter, the SQL selection condition is a conjunction. Again, the *RW* condition is equivalent to the non-emptiness of the selection result. Here $\texttt{R}_{c1}...\texttt{R}_{cn}$ stand for the column names derived from the definition of predicate R. $a'_1...,a'_n$ are the formal names for $a_1...,a_n$.

$\texttt{SQL}(a_1 = a_2) \rightleftharpoons (a'_1{=}a'_2)$ The translation of equality formulas in *RW* is straightforward. a'_1 and a'_2 are the formal names for a_1 and a_2.

$\texttt{SQL}(\neg f) \rightleftharpoons$ NOT $\texttt{SQL}(f)$, $\texttt{SQL}(f_1 \wedge f_2) \rightleftharpoons (\texttt{SQL}(f_1))$ AND $(\texttt{SQL}(f_2))$ Logical operators are expressed by their counterparts in SQL. We only give the clauses for negation and conjunction. Other connectives can be defined using these.

```
<?xml version="1.0" encoding="UTF-8"?>
<Request
    xmlns="urn:oasis:names:tc:xacml:1.0:context"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="urn:oasis:names:tc:xacml:1.0:context
    cs-xacml-schema-context-01.xsd">

 <Subject> <AnySubject/></Subject>

 <Resource>
        <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"
                DataType="http://www.w3.org/2001/XMLSchema#string">
                <AttributeValue>pcmember</AttributeValue>
        </Attribute>
        <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0:resource:agent"
                DataType="http://www.w3.org/2001/XMLSchema#string">
                <AttributeValue>agent=a1</AttributeValue>
        </Attribute>
 </Resource>

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

Figure 6: The request context for the rule of Figure 5

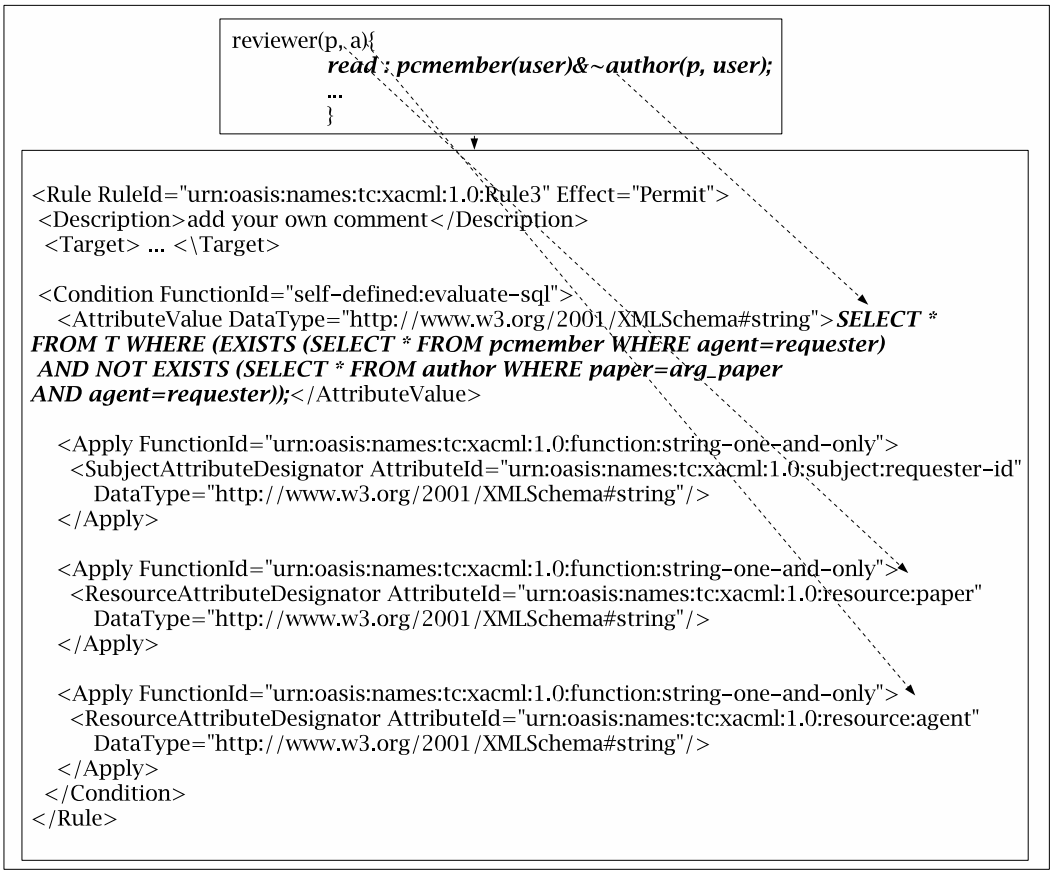

Figure 7: A logical formula in the RW file and its generated condition

SQL($\exists x \in$ D f) $\rightleftharpoons$ EXISTS (SELECT id FROM D AS D_i WHERE SQL(f)) Elements are selected from the table D, which is supposed to list all the elements from the *RW* class with the same name, and f is evaluated for each of them. The *RW* condition $\exists x \in$ D f holds iff the resulting selection is non-empty. The string *id* is the default name for a column in tables describing defined classes. The alias D_i is given to D to avoid clashes between names of bound variables. Universal formulas are expressed using existential ones by means of negation.

To obtain a complete translation of an *RW* condition, the result of SQL is prefixed **SELECT * FROM T WHERE**. Here **T** is a purpose-set table which is just supposed to contain an appropriate string to be returned by the external function which evaluates SQL statements. Thus the final form of the converted SQL statement for a given f is **SELECT * FROM T WHERE (**SQL(f)**);**.

The above clauses allow any *RW* condition to be translated into an SQL statement. Figure 7 shows an example. The string *requester* is the formal name for the access requester. It becomes replaced by its actual value by the external function.

4.3 The external function

The external function, *self-defined:evaluate-sql*, is called by a PDP program to read an SQL statement and other parameters, replace formal names in the SQL statement with their actual values, execute the query and return the result, either *true* or *false* to the PDP program. It takes at least two parameters, which are the SQL statement and the the actual value for the requester selected from the request. The compiler also puts all the parameters of the condition in question after the SQL statement and the *SubjectAttributeDesignator* on the requester, which selects the actual value for the requester from the request, as Figure 7 shows. The *ResourceAttributeDesignator* on *paper* is to select the actual value for the first parameter of predicate **reviewer**, p, and the *ResourceAttributeDesignator* on *agent* is to select the actual value for the second parameter of predicate **reviewer**, a. These two values are passed to the external function too. The external function uses the value selected by the *SubjectAttributeDesignator* for *requester-id* to replace every occurrence of the string *requester* in the SQL statement, the value selected by the *ResourceAttributeDesignator* on *paper* to replace every occurrence of the string *arg_paper* and the value selected by the *ResourceAttributeDesignator* on *agent* to replace every occurrence of the string *arg_agent*. Strings in the request are written *without* quotation marks. These are added by the external function, as required in SQL. The last formal name to be replaced is *T*. It becomes replaced by the actual name of the table, which is *test* in our example.

5. CONCLUSIONS

We have shown how the access control language *RW* of [3] can be compiled into XACML. Since *RW* descriptions may be verified[2], this allows us to produce verified XACML. We illustrated our work with the example of the *conference paper reviewing system*. Instructions on how to set up a database and how to use the compiler are given in an appendix and in the *readme* file of our the package, which is available from [9].

It should be noted that our translation from *RW* conditions, which are essentially first order formulas interpreted on a finite model, and SQL statements, was chosen for its simplicity and is by far not optimal. There is extensive literature on relational database query optimisation and, in particular, on the correspondence between relational and first order queries, see, e.g. [1].

The verification methods for *RW* from [3] apply to access control systems with fixed sets of agents and resources only. However, this limitation does not apply to the translation to XACML of such access control systems as described in this paper. *RW* permission conditions written using quantifiers can meaningfully apply to systems with varying sets of resources and agents and our implementation can handle this.

As future work, we intend to provide more formal mechanisms for verifying *RW*, the SMV [5] or Alloy [4].

[2] On this topic, please see [3]. Basically, we can verify properties like *can an agent (or a coalition of agents) reach a certain goal*, e.g. can an agent reach a state in which he/she can read who are reviewers of his/her paper.

6. REFERENCES

[1] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 77 – 90, 1977.

[2] S. D. C. di Vimercati, S. Paraboschi, and P. Samarati. Access control: principles and solutions. *Software Practice and Experience*, 33:397–421, 2003.

[3] D. P. Guelev, M. Ryan, and P. Y. Schobbens. Model-checking access control policies. To appear in the proceedings of ISC'04, Apr. 2004.

[4] D. Jackson. *Micromodels of Software: Lightweight Modelling and Analysis with Alloy*. Software Design Group, MIT Lab for Computer Science, Feb. 2002.

[5] K. L. McMillan. *The SMV language*. Cadence Berkeley Labs, 2001 Addison St. Berkeley, CA 94704 USA, Mar. 1999.

[6] OASIS committee. *eXtensible Access Control Markup Language*, 1.1 edition, Aug. 2003. Committee specification.

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

[8] Sun Microsystems. Sun's XACML implementation, Aug. 2003. For information about this implementation see web site `http://sunxacml.sourceforge.net/`.

[9] N. Zhang. Java code supporting "Synthesising verified access control systems in XACML", May. 2004. `http://www.cs.bham.ac.uk/~nxz`.

APPENDIX

A. THE SYNTAX OF THE *RW* LANGUAGE

⟨Program⟩ ::= ⟨Title⟩ ⟨Body⟩ "End"
⟨Title⟩ ::= "AccessControlSystem" ⟨ModelName⟩
⟨ModelName⟩ ::= ⟨Id⟩
⟨Body⟩ ::= [⟨ClassDefSection⟩] ⟨PredicateDefSection⟩ ⟨Rules⟩

⟨ClassDefSection⟩ ::= "Class" ⟨ClassName⟩
("," ⟨ClassName⟩)* ";"
⟨ClassName⟩ ::= ⟨UpperCaseLetter⟩
(⟨Letter⟩ | ⟨Digit⟩ | "_" | "-")*
⟨PredicateDefSection⟩ ::= "Predicate"
⟨PredicateDef⟩
("," ⟨PredicateDef⟩)* ";"
⟨PredicateDef⟩ ::= ⟨PredicateName⟩
"(" ⟨ParameterName⟩
":" ⟨ClassName⟩
("," ⟨ParameterName⟩
":" ⟨ClassName⟩)* ")"
⟨PredicateName⟩ ::= ⟨Id⟩
⟨ParameterName⟩ ::= ⟨LowerCaseLetter⟩
(⟨Letter⟩ | ⟨Digit⟩ | "_" | "-")*
⟨Rules⟩ ::= ⟨Rule⟩ (⟨Rule⟩)*
⟨Rule⟩ ::= ⟨AccessPattern⟩
"{" [⟨ReadStatement⟩]
[⟨WriteStatement⟩] "}"
⟨AccessPattern⟩ ::= ⟨PredicateName⟩
"("⟨FormalParameter⟩
("," ⟨FormalParameter⟩)* ")"
⟨FormalParameter⟩ ::= ⟨Id⟩
⟨ReadStatement⟩ ::= "read" ":" ⟨Formula⟩ ";"
⟨WriteStatement⟩ ::= "write" ":" ⟨Formula⟩ ";"
⟨Formula⟩ ::= "true" | ⟨ConditionalFormula⟩
⟨ConditionalFormula⟩ ::= ⟨ImplicationFormula⟩
⟨ImplicationFormula⟩ ::= ⟨OrFormula⟩ (⟨implies⟩
⟨OrFormula⟩)*
⟨OrFormula⟩ ::= ⟨AndFormula⟩ (⟨or⟩ ⟨AndFormula⟩)*
⟨AndFormula⟩ ::= ⟨OtherFormula⟩ (⟨and⟩
⟨OtherFormula⟩)*
⟨OtherFormula⟩ ::= ⟨AtomicFormula⟩
| "("(⟨ConditionalFormula⟩)*")"
⟨negation⟩ ⟨OtherFormula⟩
| ⟨ExistentialFormula⟩
| ⟨UniversalFormula⟩
⟨AtomicFormula⟩ ::= ⟨SinglePredicate⟩
| ⟨EquivalentFormula⟩
⟨SinglePredicate⟩ ::= ⟨PredicateName⟩
"("⟨FormalParameter⟩
("," ⟨FormalParameter⟩)* ")"
⟨EquivalentFormula⟩ ::= ⟨Term⟩ ⟨equal⟩ ⟨Term⟩
⟨Term⟩ ::= ⟨FormalParameter⟩ | ⟨QuantifiedVariable⟩
⟨ExistentialFormula⟩ ::= ⟨exist⟩ ⟨QuantifiedVariableDef⟩
("," [⟨all⟩|⟨exist⟩]
⟨QuantifiedVariableDef⟩)*
"["⟨ConditionalFormula⟩"]"
⟨QuantifiedVariableDef⟩ ::= ⟨QuantifiedVariable⟩
("," ⟨QuantifiedVariableDef⟩)*
":" ⟨ClassName⟩
⟨QuantifiedVariable⟩ ::= ⟨Id⟩
⟨UniversalFormula⟩ ::= ⟨all⟩
⟨QuantifiedVariableDef⟩
("," [⟨exist⟩|⟨all⟩]
⟨QuantifiedVariableDef⟩)*
"["⟨ConditionalFormula⟩"]"
⟨implies⟩ ::= "implies" | "→"
⟨or⟩ ::= "or" | "|"
⟨and⟩ ::= "and" | "&"
⟨negation⟩ ::= "∼"
⟨equal⟩ ::= "="
⟨exist⟩ ::= "E"
⟨all⟩ ::= "A"
⟨Id⟩ ::= ⟨Letter⟩ (⟨Letter⟩ | ⟨Digit⟩ | "_" | "-")*
⟨Letter⟩ ::= "a"-"z" | "A"-"Z"
⟨Digit⟩ ::= "0"-"9"
⟨UpperCaseLetter⟩ ::= "A"-"Z"
⟨LowerCaseLetter⟩ ::= "a"-"z"

The precedence is: "=" > "∼" > "&" > "|" > "→"

B. INSTRUCTIONS ON SETTING UP THE DATABASE

In this section, we will give instructions on how to set up a database to make the compiler work properly. The figure below illustrates the idea of how we set up the database for the conference case study. Table *test* corresponds to the virtual table T. Table *Agent* and *Paper* correspond to the defined class `Agent` and `Paper`. Table *author* is the table derived from the predicate `author`. Other tables derived from predicates are not shown in the figure. The elements in the tables are only for the purpose of illustration.

The instructions are given as follows:

- The database should contain a table for each defined class and defined predicate, including a table for class `Agent` and a table for *T*, except for the defined predicates that do not appear in any of the logical formula following the declaration part. Because if a defined predicate does not appear in any of the logical formula, it will not be evaluated. Thus we do not need to create a table for it.
- If a table is for a defined class, it must have a column named *id* to store identifiers of the elements that can uniquely identify each individual element in that class. The type of the column must be either *string* or *char*. That table must not contain duplicated records for each individual element in the class. One can store any other information about the elements in that table, however, no matter what they are, they will not be evaluated.
- If a table is for a defined predicate, it must contain a column corresponding to each its parameter, bearing the same name of that parameter. One can store other information in the table, but they will not be evaluated.
- The table for *T* needs only one column and it does not matter of the type and the name of the column. It needs also one record which does not matter of the value. However, if one is to give it a name differing from *test*, one should modify the source code of the external function which evaluates SQL statements. The modification is very simple. One only needs to change the definition of the *String* variable that stores the value for the name of the table. That variable is named *test*.

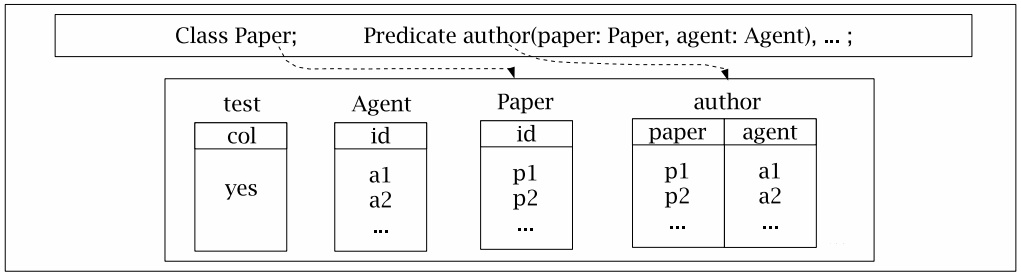

Figure 8: Tables and the structure of the database set up for the *conference paper reviewing system*

- In our example, the name of the database is *conference*. This has been hard coded into the external function to make it work. One needs to modify the variable *url* in the source code of the function, which is type of *String*, if one's database has another name. One should also modify the location of the database stored in that variable. One also needs to change the driver for the database in the code, if one uses a database system other than *Postgresql*. The driver information is stored in the *String* variable *driver*.