2010 34th Annual IEEE Computer Software and Applications Conference

Representing and Reasoning about Web Access Control Policies

Gail-Joon Ahn, Hongxin Hu, Joohyung Lee and Yunsong Meng
School of Computing, Informatics and Decision Systems Engineering
Arizona State University
Tempe, AZ 85287, USA
{*gahn,hxhu,joolee,yunsong.meng*}*@asu.edu*

***Abstract*—The advent of emerging technologies such as Web services, service-oriented architecture, and cloud computing has enabled us to perform business services more efficiently and effectively. However, we still suffer from unintended security leakages by unauthorized services while providing more convenient services to Internet users through such a cutting-edge technological growth. Furthermore, designing and managing Web access control policies are often error-prone due to the lack of logical and formal foundation. In this paper, we attempt to introduce a logic-based policy management approach for Web access control policies especially focusing on XACML (eXtensible Access Control Markup Language) policies, which have become the *de facto* standard for specifying and enforcing access control policies for various applications and services in current Web-based computing technologies. Our approach adopts Answer Set Programming (ASP) to formulate XACML that allows us to leverage the features of ASP solvers in performing various logical reasoning and analysis tasks such as policy verification, comparison and querying. In addition, we propose a policy analysis method that helps identify policy violations in XACML policies accommodating the notion of constraints in role-based access control (RBAC). We also discuss a proof-of-concept implementation of our method called XACML2ASP with the evaluation of several XACML policies from real-world software systems.**

***Keywords*-XACML; Role-based Access Control; Answer Set Programming**

I. Introduction

With the explosive growth of Web applications and Web services deployed on the Internet, the use of a policy-based approach has received considerable attention to accommodate the security requirements covering large, open, distributed and heterogeneous computing environments. Policy-based computing handles complex system properties by separating policies from system implementation and enabling dynamic adaptability of system behaviors by changing policy configurations without reprogramming the systems. In the era of distributed, heterogeneous and Web-oriented computing, the increasing complexity of policy-based computing demands strong support of automated reasoning techniques. Without analysis, most benefits of policy-based techniques and declarative policy languages may be in vain.

XACML (eXtensible Access Control Markup Language) [29], which is an XML-based language standardized by the Organization for the Advancement of Structured Information Standards (OASIS), has been widely adopted to specify access control policies for various Web applications. With expressive policy languages such as XACML, assuring the correctness of policy specifications becomes a crucial and yet challenging task. Especially, identifying inconsistencies and differences between policy specifications and their expected functions is critical since the correctness of the implementation and enforcement of policies heavily relies on the policy specification. Due to its flexibility, XACML has been extended to support specialized access control models. In particular, XACML profile for role-based access control (RBAC) [4] provides a mapping between RBAC and XACML. However, the current RBAC profile does not support *constraints* that are an important element to govern all other elements in RBAC. In RBAC, permissions of specific actions on resources are assigned to authorized users with the notion of *roles* and such assignments are constrained with specific RBAC constraints. XACML-based RBAC policies are written to specify such assignments and corresponding rules, yet security leakage may occur in specifying XACML-based RBAC policies without having appropriate constraints in place. Furthermore, designing and managing such Web access control policies are often error-prone due to the lack of logical and formal foundation.

In this paper, we propose a systematic method to represent XACML policies in answer set programming (ASP), a declarative programming paradigm oriented towards combinatorial search problems and knowledge intensive applications. Compared to a few existing approaches to formalizing XACML policies, such as as [10], [16], our formal representation is more straightforward and can cover more XACML features. Furthermore, translating XACML to ASP allows us to leverage off-the-shelf ASP solvers for a variety of analysis services such as policy verification, comparison and querying. In addition, in order to support *reasoning* about role-based authorization constraints, we introduce a general specification scheme for RBAC constraints along with a policy analysis framework, which facilitates the analysis of constraint violations in XACML-based RBAC policies. The expressivity of ASP, such as ability to handle default reasoning and represent transitive closure, helps manage XACML and RBAC constraints that cannot be handled in other logic-based approaches [16]. We also overview our tool XACML2ASP and conduct experiments with real-world

0730-3157/10 $26.00 © 2010 IEEE
DOI 10.1109/COMPSAC.2010.20

XACML policies to evaluate the effectiveness and efficiency of our solution.

The rest of this paper is organized as follows. We give an overview of XACML, RBAC and ASP in Section II. In Section III, we show how XACML can be turned into ASP and how XACML analysis can be carried out using ASP solvers. We address XACML-based RBAC policy analysis in Section IV. Section V presents the system XACML2ASP along with experiments. We overview the related work in Section VI. Section VII concludes this paper with the future work.

II. Background Technologies

A. eXtensible Access Control Markup Language

XACML has become the *de facto* standard for describing access control policies and offers a large set of built-in functions, data types, combining algorithms, and standard profiles for defining application-specific features. The root of all XACML policies is a *policy* or a *policy set*. A *policy set* is composed of a sequence of *policies* or other *policy sets* along with a *policy combining algorithm* and a *target*. A *policy* represents a single access control policy expressed through a *target*, a set of *rules* and a *rule combining algorithm*. The *target* defines a set of subjects, resources and actions the policy or policy set applies to. For applicable policy sets and policies, the corresponding targets should be true; otherwise, the policy set or policy yields no decision on the request. A *rule set* is a sequence of rules. Each *rule* in turn consists of a *target*, a *condition*, and an *effect*. The *target* of a rule has a similar structure as the target of a policy or a policy set, and decides whether the request is applicable to the rule. The *condition* is a Boolean expression to specify restrictions on the attributes in the target and refines the applicability of the rule and the *effect* is either one of "`permit`," "`deny`," or "`indeterminate`." If a request satisfies both the *target* and *condition* of a rule, the response is sent with the decision specified by the effect element in the applicable rule. Otherwise, the response yields "`notApplicable`" which is typically considered as "`deny`." Also, an XACML policy description often has conflicting rules, policies or policy sets, which are resolved by four different *combining algorithms* [29]: "Permit-overrides," "Deny-Overrides," "First-Applicable," and "Only-One-Applicable."

- Permit-Overrides: If there is any applicable rule that evaluates to `permit`, then the decision is `permit`. If there is no applicable rule that evaluates to `permit` but there is an applicable rule that evaluates to `deny`, then the decision is `deny`. Otherwise, the decision is `notApplicable`.
- Deny-Overrides: If there is any applicable rule that evaluates to `deny`, then the decision is `deny`. If there is no applicable rule that evaluates to `deny` but there is an applicable rule that evaluates to `permit`, then the decision is `permit`. Otherwise, the decision is `notApplicable`.
- First-Applicable: The decision is the effect of the first applicable rule in the listed order. If there is no applicable rule, then the decision is `notApplicable`.
- Only-One-Applicable: If more than one rule is applicable, then the decision is `indeterminate`. If there is only one applicable rule, then the decision is that of the rule. If no rule is applicable, then the decision is `notApplicable`.

```
1<PolicySet PolicySetId="ps1" PolicyCombiningAlgId="first-applicable">
2   <Target/>
3   <Policy PolicyId="p1" RuleCombiningAlgId="permit-overrides">
4     <Target/>
5     <Rule RuleId="r1" Effect="permit">
6       <Target>
7         <Subjects><Subject>     employee  </Subject></Subjects>
8         <Resources><Resource>   codes     </Resource></Resources>
9         <Actions><Action>         read      </Action>
10                  <Action>        change    </Action></Actions>
11      </Target>
12      <Condition>              8 ≤ time ≤ 17        </Condition>
13    </Rule>
14    <Rule RuleId="r2" Effect="deny">
15      <Target>
16        <Subjects><Subject>     employee  </Subject></Subjects>
17        <Resources><Resource>   codes     </Resource></Resources>
18        <Actions><Action>       change    </Action></Actions>
19      </Target>
20    </Rule>
21  </Policy>
22  <Policy PolicyId="p2" RuleCombiningAlgId="deny-overrides">
23    <Target/>
24    <Rule RuleId="r3" Effect="permit">
25      <Target>
26        <Subjects><Subject>     developer </Subject></Subjects>
27        <Resources><Resource>   codes     </Resource></Resources>
28        <Actions><Action>         read      </Action></Actions>
29      </Target>
30    </Rule>
31    <Rule RuleId="r4" Effect="deny">
32      <Target>
33        <Subjects><Subject>       tester    </Subject></Subjects>
34        <Resources><Resource>   codes     </Resource></Resources>
35        <Actions><Action>         read      </Action></Actions>
36      </Target>
37    </Rule>
38    <Rule RuleId="r5" Effect="deny">
39      <Target>
40        <Subjects><Subject>       tester    </Subject>
41                  <Subject>     developer </Subject></Subjects>
42        <Resources><Resource>   codes     </Resource></Resources>
43        <Actions><Action>       change    </Action></Actions>
44      </Target>
45    </Rule>
46  </Policy>
47</PolicySet>
```

Figure 1. An example XACML policy.

Note that "Only-One-Applicable" combining algorithm is defined only for policy sets.

Consider an example XACML policy for a software development company, which is utilized throughout this paper, shown in Figure 1. Figure 2 gives a tree structure of this example policy. The root policy set ps_1 contains two policies p_1 and p_2 which are combined using `first–applicable` combining algorithm. The policy p_1, which is the global

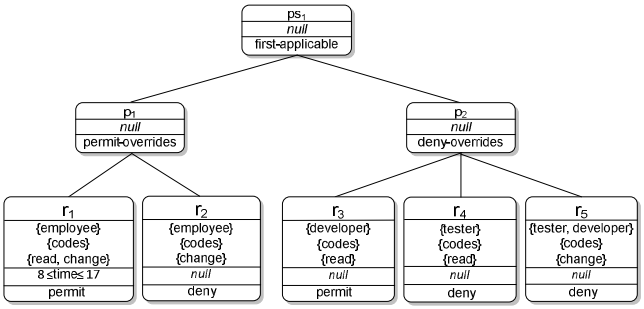

Figure 2. Tree structure of the example XACML policy.

policy of the entire company, has two rules r_1 and r_2 indicating that

- all employees can read and change codes during working hours from 8:00 to 17:00 (r_1), and
- nobody can change code during non-working hours (r_2).

On the other hand, each department is responsible for deciding whether employees can read codes during non-working hours. A local policy p_2 for a development department with three rules r_3, r_4 and r_5 is that

- developers can read codes during non-working hours (r_3),
- testers cannot read codes during non-working hours (r_4), and
- testers and developers cannot change codes during non-working hours (r_5).

Note that the rule combining algorithm for policy p_1 is `permit–overrides` and the rule combining algorithm for policy p_2 is `deny–overrides`.

B. Role-based Access Control

RBAC is a widely accepted alternative to traditional mandatory access control (MAC) and discretionary access control (DAC) [24]. As MAC is used in the classical defense arena, the access is based on the classification of objects such as security clearance [23] while the main idea of DAC is that the owner of an object has the discretion over who can access the object [15], [22]. However, RBAC is based on the role of the subjects and can specify security policy in a way that maps to an organizational structure. A general family of RBAC models called RBAC96 was proposed by Sandhu et al. [21]. Intuitively, a user is a human being or an autonomous agent, a role is a job function or job title within the organization with some associated semantics regarding the authority and responsibility conferred on the user assigned to the role, and a permission is an approval of a particular mode of access to one or more objects in the system or some privileges to carry out specified actions. Roles are organized in a partial order $\geq$, so that if $x \geq y$ then a role x inherits the permissions of a role y. Therefore, members of a role x are also implicitly members of a role y. In addition, RBAC introduces constraints that are a powerful mechanism for laying out higher-level organizational policies. Separation of duty (SoD) is a well-known principle for preventing fraud by identifying conflicting roles and has been studied in considerable depth by RBAC community [3], [7], [17]. *SoD* constraints in RBAC can be divided into *Static SoD constraints*, *Dynamic SoD constraints* and *Historical SoD constraints*. *Static SoD constraints* typically require that no user should be assigned to conflicting roles. *Dynamic SoD constraints*–with respect to activated roles in sessions– typically require that no user can activate conflicting roles simultaneously. *Historical SoD constraints* restrict the assignment and activation of conflicting roles over the course of time.

C. Answer Set Programming

ASP [20], [18] is a recent form of declarative programming that has emerged from the interaction between two lines of research—nonmonotonic semantics of negation in logic programming and applications of satisfiability solvers to search problems. The idea of ASP is to represent the search problem we are interested in as a logic program whose intended models, called "stable models (a.k.a. answer sets)," correspond to the solutions of the problem, and then find these models using an answer set solver—a system for computing stable models. Like other declarative computing paradigms, such as SAT (Satisfiability Checking) and CP (Constraint Programming), ASP provides a common basis for formalizing and solving various problems, but is distinct from others such that it focuses on knowledge representation and reasoning: its language is an expressive nonmonotonic language based on logic programs under the stable model semantics [11], [9], which allows elegant representation of several aspects of knowledge such as causality, defaults, and incomplete information, and provides compact encoding of complex problems that cannot be translated into SAT and CP [19]. As the mathematical foundation of answer set programming, the stable model semantics was originated from understanding the meaning of *negation as failure* in Prolog, which has the rules of the form

$$a_1 \leftarrow a_2, \ldots, a_m, \mathit{not}\ a_{m+1}, \ldots, \mathit{not}\ a_n \qquad (1)$$

where all a_i are atoms and *not* is a symbol for *negation as failure*, also known as *default negation*. Intuitively, under the stable model semantics, rule (1) means that if you have generated $a_2, \ldots, a_m$ and it is impossible to generate any of $a_{m+1}, \ldots, a_n$ then you may generate a_1. This explanation seems to contain a vicious cycle, but the semantics are carefully defined in terms of fixpoint.

While it is known that the transitive closure (e.g., reachability) cannot be expressed in first-order logic, it can be handled in the stable model semantics. Given the fixed extent of $edge$ relation, the extent of $reachable$ is the transitive

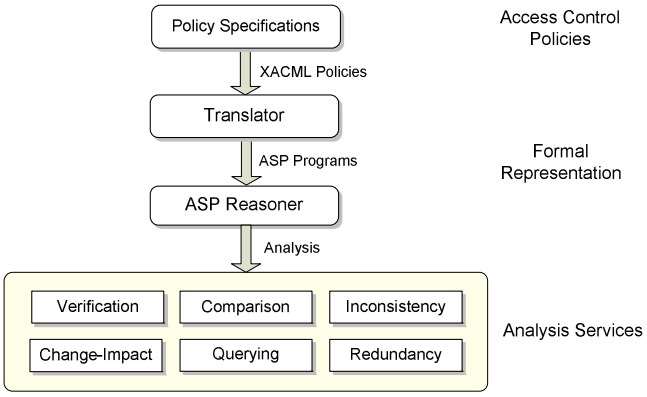

Figure 3. Logic-based policy reasoning for XACML.

closure of $edge$.

$$reachable(X,Y) \leftarrow edge(X,Y)$$
$$reachable(X,Y) \leftarrow reachable(X,Z), reachable(Z,Y)$$

Several extensions were made over the last twenty years. The addition of cardinality constraints turns out to be useful in knowledge representation. A cardinality constraint is of the form $lower\{l_1,\ldots,l_n\}upper$ where $l_1,\ldots,l_n$ are literals and $lower$ and $upper$ are numbers. A cardinality constraint is satisfied if the number of satisfied literals in $l_1,\ldots,l_n$ is in between $lower$ and $upper$. It is also allowed to contain variables in cardinality constraints. For instance,

$$more\_than\_one\_edge(X) \leftarrow 2\{edge(X,Y) : vertex(Y)\}.$$

means that $more\_than\_one\_edge(X)$ is true if there are at least two edges connect X with other vertices.

The language also has useful constructs, such as strong negations, weak constraints, and preferences. What distinguishes ASP from other nonmonotonic formalisms is the availability of several efficient implementations, answer set solvers, such as SMODELS[1], CMODELS[2], CLASP[3], which led to practical nonmonotonic reasoning that can be applied to industrial level applications.

III. General XACML Policy Analysis

We introduce a logic-based policy reasoning approach for XACML as shown in Figure 3. First, XACML policies are converted to ASP programs. Then, by means of off-the-shelf ASP solvers, several typical policy analysis services, such as policy verification, comparison, redundancy and querying are utilized. For instance, *policy verification* is to check if ASP-based representation of XACML policies entails the property as certain formulas in its specification, *policy comparison* checks the equivalence between two answer set programs, and *policy redundancy checking* can be viewed as an instance of simplification of ASP programs.

[1]`http://www.tcs.hut.fi/Software/smodels` .
[2]`http://www.cs.utexas.edu/users/tag/cmodels.html` .
[3]`http://potassco.sourceforge.net` .

A. Abstracting XACML Policy Components

We consider a subset of XACML that covers more constructs than the ones considered in [28] and [16]. We allow the most general form of *Target*, take into account *Condition*, and cover all four combining algorithms.

XACML components can be abstracted as follows: *Attributes* are the names of elements used by a policy. *Attributes* are divided into three categories: *subject attributes*, *resource attributes* and *action attributes*. In the example policy above, `developer`, `tester` and `employee` are subject attributes; `read` and `change` are action attributes; `codes` is a resource attribute. A *Target* is a triple $\langle Subjects, Resources, Actions\rangle$. A *Condition* is a conjunction of comparisons. An *Effect* is either "`permit`," "`deny`," or "`indeterminate`."

- An XACML rule can be abstracted as

  $$\langle RuleID, Effect, Target, Condition\rangle$$

  where *RuleID* is a rule identifier. For example, rule $\mathtt{r}_1$ in Figure 1 can be viewed as

  $$\langle \mathtt{r}_1, \mathtt{permit}, \langle \mathtt{employee}, \mathtt{read} \vee \mathtt{change}, \mathtt{codes}\rangle, 8 \leq \mathtt{time} \leq 17\rangle.$$

- An XACML policy can be abstracted as

  $$\langle PolicyID, Target, Combining\ Algorithm, \langle r_1,\ldots,r_n\rangle\rangle$$

  where *PolicyID* is a policy identifier, $r_1,\ldots,r_n$ are rule identifiers and *Combining Algorithm* is either `permit–overrides`, `deny–overrides`, or `first–applicable`. For example, policy $\mathtt{p}_1$ in Figure 1 is abstracted as:

  $$\langle \mathtt{p}_1, Null, \mathtt{permit–overrides}, \langle \mathtt{r}_1, \mathtt{r}_2\rangle\rangle.$$

- Similarly we can abstract an XACML policy set as

  $$\langle PolicySetID, Target, Combining\ Algorithm, \langle p_1,\ldots,p_m, ps_{m+1},\ldots,ps_n\rangle\rangle$$

  where *PolicySetID* is a policy set identifier, $p_1,\ldots,p_m$ are policy identifiers, $ps_{m+1},\ldots,ps_n$ are policy set identifiers, and *Combining Algorithm* is either `permit–overrides`, `deny–overrides`, `first–applicable`, or `only–one–applicable`. For example, policy set $\mathtt{ps}_1$ can be viewed as

  $$\langle \mathtt{ps}_1, Null, \mathtt{first–applicable}, \langle \mathtt{p}_1, \mathtt{p}_2\rangle\rangle.$$

B. Turning XACML into ASP

We provide a translation module that turns an XACML description into a program in ASP. This interprets a formal semantics of XACML language in terms of the Answer Set semantics.

The translation module coverts an XACML rule

$$\langle RuleID, Effect, Target, Condition\rangle$$

into a set of ASP rules [4]

$$\mathit{decision}(\mathit{RuleID}, \mathit{Effect}) \leftarrow \mathit{Target} \wedge \mathit{Condition}.$$

An XACML policy

$$\langle \mathit{PolicyID}, \mathit{Target}, \mathit{Combining\ Algorithm}, \langle r_1, \ldots, r_n \rangle \rangle$$

can be also translated into a set of ASP rules. In the following we assume that R and R' are variables that range over all rule ids, and V is a variable that ranges over $\{\texttt{permit}, \texttt{deny}, \texttt{indeterminate}\}$. In order to represent the effect of each rule r_i $(1 \leq i \leq n)$ on policy, we write

$$\mathit{decision\_from}(\mathit{PolicyID}, r_i, V) \leftarrow \mathit{decision}(r_i, V).$$

Each rule combining algorithms is turned into logic programming rules under the stable model semantics as follows:

- `permit`–`overrides` of policy p is represented as

$$\begin{array}{l} \mathit{decision}(p, \texttt{permit}) \leftarrow \\ \qquad \mathit{decision\_from}(p, R, \texttt{permit}) \wedge \mathit{Target}. \\ \mathit{decision}(p, \texttt{deny}) \leftarrow \mathit{decision\_from}(p, R, \texttt{deny}) \\ \qquad \wedge\ \mathit{not}\ \mathit{decision}(p, \texttt{permit}) \wedge \mathit{Target}. \end{array}$$

- `deny`–`overrides` of policy p is represented as

$$\begin{array}{l} \mathit{decision}(p, \texttt{deny}) \leftarrow \\ \qquad \mathit{decision\_from}(p, R, \texttt{deny}) \wedge \mathit{Target}. \\ \mathit{decision}(p, \texttt{permit}) \leftarrow \mathit{decision\_from}(p, R, \texttt{permit}) \\ \qquad \wedge\ \mathit{not}\ \mathit{decision}(p, \texttt{deny}) \wedge \mathit{Target}. \end{array}$$

- `first`–`applicable` of policy p is represented as

$$\begin{array}{l} \mathit{has\_decision\_from}(p, R) \leftarrow \mathit{decision\_from}(p, R, V)\ . \\ \mathit{decision}(p, V) \leftarrow \mathit{decision\_from}(p, r_i, V) \wedge \\ \qquad \bigwedge_{1 \leq k \leq i-1} \mathit{not}\ \mathit{has\_decision\_from}(p, r_k) \wedge \mathit{Target}. \end{array}$$

The translation of a policy set is similar to the translation of a policy except that the policy combining algorithm `only`–`one`–`applicable` needs to be taken into account. For instance, `only`–`one`–`applicable` of policy set ps is represented as follows:

$$\begin{array}{l} \mathit{decision}(ps, V) \leftarrow \mathit{decision\_from}(ps, P, V) \wedge \\ \qquad 1\{\mathit{has\_decision\_from}(ps, P) : \mathit{policy}(P)\}1. \\ \mathit{decision}(ps, \texttt{indeterminate}) \leftarrow \\ \qquad 2\{\mathit{has\_decision\_from}(ps, P) : \mathit{policy}(P)\}. \end{array}$$

Figure 4 shows an ASP representation of the example XACML policy in the language of GRINGO by applying our translation approach.

C. XACML Policy Analysis Using ASP

Once we represent an XACML into an ASP program Π, we can use off-the-shelf ASP solvers for several automated analysis services. In this section, we mainly illustrate how policy verification can be handled by our policy analysis approach.

[4]We identify *Target* with the conjunction of its components. Also, we identify " $\wedge$ " with ",", " $\leftarrow$ " with " :- " and a rule of the form $A \leftarrow B, C \vee D$ as a set of the two rules $A \leftarrow B, C.$ and $A \leftarrow B, D.$

```
value(permit;deny;indeterminate).
rule(r1;r2;r3;r4;r5).
policy(p1;p2).
policyset(ps1).
time(0..23).
#domain value(V;V1).
#domain rule(R;R1).
#domain policy(P).
#domain time(T).
% domain definition
subject(employee) :- subject(developer).
subject(employee) :- subject(tester).
% r1
decision(r1,permit) :- subject(employee),action(read),
            resource(codes),8<=T,T<=17, current_time(T).
decision(r1,permit) :- subject(employee),action(change),
            resource(codes),8<=T,T<=17, current_time(T).
% r2
decision(r2,deny) :- subject(employee),action(change),
                     resource(codes).
% r3
decision(r3,permit) :- subject(developer),action(read),
                       resource(codes).
% r4
decision(r4,deny) :- subject(tester),action(read),
                     resource(codes).
% r5
decision(r5,deny) :- subject(tester),action(change),
                     resource(codes).
decision(r5,deny) :- subject(developer),action(change),
                     resource(codes).
% p1
decision_from(p1,r1,V) :- decision(r1,V).
decision_from(p1,r2,V) :- decision(r2,V).
decision(p1,permit) :- decision_from(p1,R,permit).
decision(p1,deny) :- decision_from(p1,R,deny),
                     not decision(p1,permit).
% p2
decision_from(p2,r3,V) :- decision(r3,V).
decision_from(p2,r4,V) :- decision(r4,V).
decision_from(p2,r5,V) :- decision(r5,V).
decision(p2,deny) :- decision_from(p2,R,deny).
decision(p2,permit) :- decision_from(p2,R,permit),
                       not decision(p2,deny).
% ps1
decision_from(ps1,p1,V) :- decision(p1,V).
decision_from(ps1,p2,V) :- decision(p2,V).
has_decision_from(ps1,p1) :- decision_from(ps1,p1,V).
decision(ps1,V) :- decision_from(ps1,p1,V).
decision(ps1,V) :- decision_from(ps1,p2,V),
                   not has_decision_from(ps1,p1).
```

Figure 4. ASP representation of the example XACML policy

The problem of verifying a security property F against an XACML description can be cast into the problem of checking whether the program

$$\Pi \cup \Pi_{\mathit{query}} \cup \Pi_{\mathit{config}}$$

has no answer sets, where Π is the program corresponding to the XACML specification, Π_{query} is the program corresponding to the program that encodes the negation of the property to check, and Π_{config} is the following program that generates *arbitrary* configurations.

```
subject_attributes(developer;tester;employee).
action_attributes(read;change).
resource_attributes(codes).

1{subject(X) : subject_attributes(X)}.
1{action(X) : action_attributes(X)}.
```

```
1{resource(X) : resource_attributes(X)}.
1{current_time(X) : time(X)}1.
```

If no answer set is found, this implies that the property is verified. Otherwise, an answer set returned by an ASP solver serves as a counterexample that indicates why the description does not entail F. This helps the policy designer find out the design flaws in the policy specification.

For example, consider the example XACML policy shown in Figure 1. We need to ensure that a developer cannot change codes during non-working hours. The input query Π_{query} can be represented as follows:

```
working_hours :- 8<=T, T<=17,current_time(T).
check :- decision(ps1,permit),
  subject(developer),action(change),
  resource(codes),not working_hours.
:- not check.
```

Given the corresponding ASP program of $\texttt{ps}_1$, the negation of the property, and Π_{config}, GRINGO and CLASPD return no answer set from which we conclude that the property is held.

As another example, consider the query if a developer is always allowed to read codes during non-working hours. This query Π_{query} can be represented as

```
working_hours :- 8<=T, T<=17,current_time(T).
check :- decision(ps1,deny),
  subject(developer), action(read),
  resource(codes),not working_hours.
:- not check.
```

A policy designer may intend that this property would follow based on the policy specification. However, the following answer set is found, which indicates a design flaw of the policy.

```
{subject(developer) action(read)
 action(change) resource(codes)
 decision(ps1,deny) decision(p1,deny)
 decision(p2,deny) decision(r2,deny)
 decision(r3,permit) decision(r5,deny)}
```

That is, a developer's request to read the codes is denied if his request also includes changing the codes[5]. From this answer set, the policy designer finds that $\texttt{p}_2$, which is supposed to return `permit`, returns `deny`. It is because $\texttt{r}_5$ returns `deny`, and the combining algorithm of $\texttt{p}_2$ is `deny-overrides`.

In fact, the reason that $\texttt{ps}_1$ returns `deny` is because $\texttt{p}_1$ returns `deny`. Rule $\texttt{r}_1$ is not applicable since its *condition* is not satisfied and rule $\texttt{r}_2$ returns `deny`. Then, the policy designer realizes the flaw and could disallow the concurrency of two actions within a request. However, even after adding such a constraint, another answer set is found as follows:

```
{subject(developer) subject(tester)
 action(read) resource(codes)
 decision(ps1,deny) decision(p2,deny)
 decision(r3,permit) decision(r4,deny)}
```

That is, when someone is both `developer` and `tester`, he cannot read codes during non-working hours since rule $\texttt{r}_4$ disallows it. In this answer set, $\texttt{ps}_1$ returns `deny` because $\texttt{p}_1$ is not applicable and $\texttt{p}_2$ returns `deny`. In turn, it is because $\texttt{r}_4$ returns `deny`. If we add a constraint disallowing a person to be both `developer` and `tester` roles simultaneously, the program returns no answer set as intended. Disallowing two conflicting roles to be assigned to the same user is called separation of duty (SoD) in role-based access control, which is discussed in the subsequent section.

IV. XACML-BASED RBAC POLICY ANALYSIS

A. A Policy Analysis Framework

As we discussed in Section I, the current XACML profile for RBAC [4] only supports elements and relations from core and hierarchical RBAC omitting constraints in RBAC. This section focuses on how XACML-based RBAC policies can be analyzed based on the approaches that we discussed in the previous sections while considering elements, relations and constraints in RBAC. To support the reasoning for XACML-based RBAC policy, we introduce a policy analysis framework shown in Figure 5. Our framework first transforms XACML-based representation of core and hierarchical RBAC to ASP-based RBAC representation. In addition, the policy designers can specify the RBAC constraints using a general constraint specification scheme derived from the NIST/ANSI RBAC standard [8], [1]. Those general constraint specifications are translated to ASP-based constraint specifications. Therefore, representing both RBAC system configuration (core and hierarchical RBAC) and RBAC constraints in ASP enables us to support rigorous analysis of constraints that are not addressed in the current XACML profile.

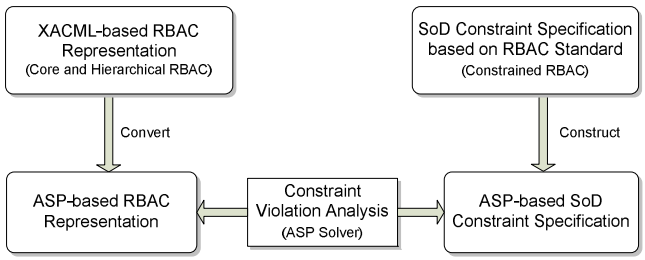

Figure 5. A policy analysis framework for XACML-based RBAC.

B. Core and Hierarchical RBAC Representation

RBAC models define sets of elements including a set of roles, a set of users, and a set of permissions, and relationships among users, roles and permissions. In XACML profile for RBAC, Role Assignment $\langle Policy \rangle$ or $\langle PolicySet \rangle$ defines which roles can be enabled or assigned to whom. Suppose that a user `john` is assigned to two roles `tester`

[5]XACML supports *multi-valued* requests, which contains multiple id-value pairs in the subject, resource, or action attribute.

and seniorDeveloper in the software development company. We can translate those user-to-role assignments (ura) to ASP as follows:

```
ura(john,tester).
ura(john,seniorDeveloper).
```

RBAC supports role hierarchy relations. For example, developer is a junior role of seniorDeveloper in the software development company. The hierarchy relation between two roles developer and seniorDeveloper represented in XACML can be converted into ASP as follows:

```
junior(developer, seniorDeveloper).
```

In addition, we assume that relation junior is reflexive.

```
junior(R,R) :- rules(R).
```

tc_junior is a transitive closure of junior relation.

```
tc_junior(R1,R2) :- junior(R1,R2).
tc_junior(R1,R3) :- tc_junior(R1,R2),
                    tc_junior(R2,R3).
```

Furthermore, the following definition is required to specify a user-to-role assignment considering the role hierarchy relations. It implies if a role $r2$ is a junior role of $r1$ and $r1$ is assigned to a user u, $r2$ is also implicitly assigned to the user u.

```
ura(U,R2) :- ura(U,R1), tc_junior(R2, R1)
```

Similarly, a session-to-role relation with respect to the role hierarchy relations is defined as follows:

```
sr(S,R2) :- sr(S,R1), tc_junior(R2, R1)
```

C. RBAC Constraint Representation

As part of RBAC constraints, we demonstrate how SoD constraints can be represented in ASP programs based on our framework. Most existing definitions of SoD constraints only consider a conflicting set as a pair of elements. For example, a constraint may declare a pair of conflicting roles r_1 and r_2, and require that no user is allowed to simultaneously assign to both r_1 and r_2. These definitions are too restrictive in the size of a conflicting set and the combination of elements in the set for which assignment operation is constrained. Thus, a more general example of SoD constraints should require that no user is allowed to be simultaneously assigned to n or more roles from a conflicting role set. In NIST/ANSI RBAC standard, SoD constraints are defined with two arguments: (a) a conflicting role set cr that includes two or more roles; and (b) a natural number n, called the cardinality, with the property that $2 \leq n \leq |cr|$ means a user can be assigned to at most n roles from conflicting role set cr. A similar definition is used in dynamic SoD constraints with respect to the activation of roles in sessions.

The NIST/ANSI RBAC standard has limitations in the constraint definitions. First, the conflicting notion is only applied to role without considering other components such as user and permission in RBAC. In the real world, we may also have notions of conflicting permissions or conflicting users based on the organizational policy. Second, historical SoD constraints are not addressed in RBAC standard. To address these issues, we provide a more general constraint specification method based on the RBAC standard.

Definition 1: (**SoD Constraint**). A SoD constraint is a tuple $SoD = \langle t, e, cs, n \rangle$, where

- $t \in \{s, d, h\}$ represents the types of SoD constraints, where s, d and h stand for $static$, $dynamic$ and $historical$, respectively;
- $e \in \{U, R, P\}$ is the RBAC element to which the constraint is applied, where U, R and P denote $User$, $Role$ and $Permission$, respectively;
- cs is the conflicting element set including conflict role set (cr), conflict user set (cu) and conflict permission set (cp); and
- n is an integer, such that $2 \leq n \leq |cs|$.

RBAC constraints defined by this general scheme can be used to construct ASP-based constraint specifications. A detailed construction algorithm is described in Algorithm 1. In this algorithm, three kinds of SoD constraints are supported depending on the value of t in a constraint specification. For static SoD constraints, if the value of e is R, the algorithm further examines the types of conflicting element, which is either user or permission indicating *user-centric* or *permission-centric* constraints, respectively. Note that us indicates a *user-to-session* relation in this algorithm.

Next, we illustrate three typical RBAC constraints specified in our general scheme and give equivalent ASP expressions generated by our construction algorithm.

Constraint 1: (SSoD-CR): The number of conflicting roles, which are from the same conflicting role set and authorized to a user, cannot exceed the cardinality of the conflicting role set.

Suppose tester and developer belong to a static conflicting role set and the cardinality of the conflicting role set is *two*. That is, these roles cannot be assigned to the same user at the same time.

Constraint Expression:

$$\langle s, U, \textit{tester}, \textit{developer}, 2 \rangle$$

Constructed ASP Expression:

```
:- 2{ura(U, tester), ura(U, developer)}.
```

Constraint 2: (User-based DSoD): The number of conflicting roles, which are from the same conflicting role set and activated directly (or indirectly via inheritance) by a user, cannot exceed the cardinality of the conflicting role set.

Assume tester and developer are contained in a dynamic conflicting role set and the cardinality of the conflicting role set is *two*. It means they are dynamic conflicting roles and cannot be activated by a user simultaneously.

Algorithm 1: Construction of ASP-based Constraint Expression

Input: A general constraint expression C.
Output: An ASP constraint expression C'.

$C \longleftarrow < t, e, cs, n >$;
/ Static Constraint*/*
if $c.t =' s'$ **then**
  if $c.e =' U'$ **then**
    foreach $r \in c.cs$ **do**
      $URA.append(ura(U, r))$;
    $C' \longleftarrow: -c.n\{URA\}.$;
  if $c.e =' R'$ **then**
    if $user(c.cs) = true$ **then**
      foreach $u \in c.cs$ **do**
        $URA.append(ura(u, R))$
      $C' \longleftarrow: -c.n\{URA\}.$;
    if $permission(c.cs) = true$ **then**
      foreach $p \in c.cs$ **do**
        $PRA.append(pra(p, R))$
      $C' \longleftarrow: -c.n\{PRA\}.$;
/ Dynamic Constraint*/*
if $c.t =' d'$ **then**
  $i \longleftarrow 0$;
  foreach $r \in c.cs$ **do**
    $i \longleftarrow i + 1$;
    $SR.append(sr(Si, r))$;
    $US.append(us(U, Si))$;
  $C' \longleftarrow: -c.n\{SR\}, US.$;
/ Historical Constraint*/*
if $c.t =' h'$ **then**
  $i \longleftarrow 0$;
  foreach $r \in c.cs$ **do**
    $i \longleftarrow i + 1$;
    $SR.append(sr(Si, r, Ti))$;
    $US.append(us(U, Si, Ti))$;
  $C' \longleftarrow: -c.n\{SR\}, US.$;
return C';

Constraint Expression:

$$\langle d, U, tester, developer, 2 \rangle$$

Constructed ASP Expression:

```
:- 2{sr(S1, tester), sr(S2, developer)},
   us(U, S1), us(U, S2).
```

Most of existing work in specifying [3], [7], [17] and analyzing [25], [27], [12] SoD constraints mainly focus on a system state at one point in time. By introducing a temporal variable *time* in ASP representation, the changing system state can be taken into account for both RBAC constraint specification and analysis in ASP representation.

Constraint 3: (Historical SoD): The number of activated roles from a conflicting role set by a user cannot exceed the cardinality of the historical conflicting role set.

Assume that two roles `tester` and `developer` are contained in a historical conflicting role set and the cardinality of the conflicting role set is *two*.

Constraint Expression:

$$\langle h, U, tester, developer, 2 \rangle$$

Constructed ASP Expression:

```
:- 2{sr(S1, tester, T1),
   sr(S2, developer, T2)},
   us(U, S1, T1), us(U, S2, T2).
```

Note that we introduce two time variables *T1* and *T2* to reflect the changing system states in this constraint representation. Thus, the constraint violations for the changing system states can be identified. For example, we can evaluate if a user ever activated two conflicting roles at different time intervals by checking the *historical SoD* constraints as a security property against the changing system states.

D. Violation Analysis of RBAC Constraints

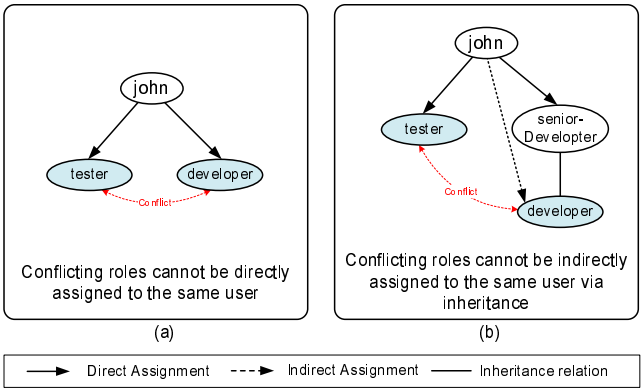

Figure 6. Violation checking for SoD constraints.

RBAC constraints can be utilized as security properties to check against access control policy configurations for identifying constraint violations. Figure 6 shows a typical example, which illustrates conflicting roles cannot be directly or indirectly (via inheritance) assigned to the same user. Figure 6 (a) shows that the user `john` is assigned to two roles `tester` and `developer` simultaneously. However, since `tester` is mutually exclusive to `developer`, the *SSoD-CR* constraint is violated. Figure 6 (b) depicts a more complex example taking role hierarchy into account. The user `john` acquires two conflicting roles `tester` and `developer` through the permission inheritance. The *SoD* property supporting the role hierarchy can be specified with ASP as follows:

```
check :- ura(U,tester), ura(U,developer).
```

If an answer set that is returned by an ASP solver contains `check`, it means that a user is assigned to two conflicting roles `tester` and `developer` in current RBAC configuration. Thus an *SoD* constraint violation is identified.

V. Implementation and Evaluation

We have implemented a tool called XACML2ASP in Java 1.6.3. XACML2ASP can automatically convert core XACML and RBAC constraint expressions into ASP. The generated ASP-based policy representations are then fed into an ASP reasoner to carry out analysis services. We evaluated the efficiency and effectiveness of our approach on several real-world XACML policies. GRINGO was employed as the ASP solver for our evaluation. Our experiments were performed on Intel Core 2 Duo CPU 3.00 GHz with 3.25 GB RAM running on Windows XP SP2.

In our evaluation, we utilized ten real-world XACML policies collected from three different sources. Six of the policies, *CodeA*, *CodeB*, *CodeC*, *CodeD*, *Continue-a* and *Continue-b* are XACML policies used by [10]; among them, *Continue-a* and *Continue-b* are designed for a real-world Web application supporting a conference management. Three of the policies *Weirdx*, *FreeCS* and *GradeSheet* are utilized by [5]. The *Pluto* policy is employed in ARCHON[6] system, which is a digital library that federates the collections of physics with multiple degrees of meta data richness.

Table I
Experimental results on real-life XACML policies

Policy	# of Rules	Converting Time(s)	Reasoning Time(s)
CodeA	2	0.000	0.000
CodeB	3	0.000	0.000
CodeC	4	0.000	0.002
CodeD	5	0.000	0.004
Weirdx	6	0.005	0.006
FreeCS	7	0.005	0.006
GradeSheet	14	0.015	0.012
Pluto	21	0.016	0.031
Continue-a	298	0.120	0.405
Continue-b	306	0.125	0.427

Table I shows the number of rules contained in each policy, the conversion time from XACML to ASP, and the reasoning time using GRINGO + CLASPD for each policy. Note that the reasoning time was measured by enabling GRINGO + CLASPD to generate answer sets representing all permitted requests for each policy. From Table I, we observe that the conversion time from XACML to ASP in XACML2ASP is fast enough to handle larger size of policies, such as *Continue-a* and *Continue-b*. It also indicates that the reasoning process for policy analysis in ASP solver is also efficient enough for a variety of policy analysis services.

VI. Related Work

In [13], a framework for automated verification of access control policies based on relational first-order logic was proposed. The authors demonstrated how XACML policies can be translated to the Alloy language [14], and checked their security properties using the Alloy Analyzer. However, using the first-order constructs of Alloy to model XACML policies is expensive and still needs to examine its feasibility for larger size of policies. In [6], the authors formalized XACML policies using a process algebra known as Communicating Sequential Processes. This utilizes a model checker to formally verify properties of policies, and to compare access control policies with each other. Fisler et al. [10] introduced an approach to represent XACML policies with Multi-Terminal Binary Decision Diagrams (MTBDDs). A policy analysis tool called Margrave was developed. Margrave can verify XACML policies against the given properties and perform change-impact analysis based on the semantic differences between the MTBDDs representing the policies. Kolovski et al. [16] presented a formalization of XACML using description logic (DL), which is a family of languages that are decidable subsets of first-order logic, and leveraged existing DL reasoners to conduct policy verification. Compared with other work in XACML, our approach provides a more straightforward formalization with ASP addressing XACML features such as all four *combining algorithms* and handling simple *conditions*.

Schaad and Moffett [25] specified the access control policies under the RBAC96 and ARBAC97 models and a set of separation of duty constraints in Alloy. They attempted to check the constraint violations caused by administrative operations. In [26], Sohr et al. demonstrated how the USE tool, a validation tool for OCL constraints, can be utilized to validate authorization constraints against RBAC configurations. The policy designers can employ the USE-based approach to detect certain conflicts between authorization constraints and to identify missing constraints. Assurance Management Framework (AMF) was proposed in [2], [12], where formal RBAC model and constraints can be analyzed. Alloy was also utilized as an underlying formal verification tool to analyze the formal specifications of an RBAC model and corresponding constraints, which are then used for access control system development. In addition, the verified specifications are used to automatically derive the test cases for conformance testing. Even though there has been a great amount of work on XACML and RBAC analysis, there is little work in providing *reasoning* in XACML-based RBAC policies.

VII. Conclusion and Future Work

In this work, we have provided a formal foundation of XACML in terms of ASP. Also, we further introduced a policy analysis framework for identifying constraint violations in XACML-based RBAC policies, explicitly demonstrating existing XACML standard does not support the constrained RBAC. In addition, we have described a tool called XACML2ASP, which can seamlessly work with existing ASP solvers for XACML policy analysis. Our experiments showed that the performance of our analysis approach could efficiently support larger access control policies.

[6] http://archon.cs.odu.edu/.

For our future work, the coverage of our mapping approach needs to be further extended with more XACML features such as handling complicated conditions, obligation and other attribute functions. Also, it is necessary to enhance our tool to provide those features and corresponding analysis services while obscuring the details of the ASP formalism.

ACKNOWLEDGMENT

The work of Gail-J. Ahn and Hongxin Hu was partially supported by the grants from National Science Foundation (NSF-IIS-0900970 and NSF-CNS-0831360). The work of Joohyung Lee and Yunsong Meng was partially supported by the grants from National Science Foundation (NSF-IIS-0916116) and by the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Projects Activity (IARPA), through US army. All statements of fact, opinion or conclusions contained herein are those of the authors and should not be construed as representing the official views or policies of IARPA, the ODNI or the U.S. Government.

REFERENCES

[1] *American National Standards Institute Inc.* Role Based Access Control, ANSI-INCITS 359–2004, 2004.

[2] G.-J. Ahn and H. Hu. Towards realizing a formal RBAC model in real systems. In *Proceedings of the 12th ACM symposium on Access control models and technologies*, page 224. ACM, 2007.

[3] G.-J. Ahn and R. Sandhu. Role-based authorization constraints specification. *ACM Transactions on Information and System Security (TISSEC)*, 3(4):207–226, 2000.

[4] A. Anderson. Core and hierarchical role based access control (RBAC) profile of XACML v2. 0. *OASIS Standard*, 2005.

[5] A. Birgisson, M. Dhawan, U. Erlingsson, V. Ganapathy, and L. Iftode. Enforcing authorization policies using transactional memory introspection. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 223–234. ACM New York, NY, USA, 2008.

[6] J. Bryans. Reasoning about XACML policies using CSP. In *Proceedings of the 2005 workshop on Secure web services*, page 35. ACM, 2005.

[7] J. Crampton. Specifying and enforcing constraints in role-based access control. In *Proceedings of the Eighth ACM symposium on Access control models and Technologies*, page 50. ACM, 2003.

[8] D. F. Ferraiolo, R. S. Sandhu, S. I. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM Trans. Inf. Syst. Secur. (TISSEC)*, 4(3):224–274, 2001.

[9] P. Ferraris, J. Lee, and V. Lifschitz. Stable models and circumscription. *Artificial Intelligence*, 2010. To appear.

[10] K. Fisler, S. Krishnamurthi, L. Meyerovich, and M. Tschantz. Verification and change-impact analysis of access-control policies. In *Proceedings of the 27th international conference on Software engineering*, pages 196–205. ACM New York, NY, USA, 2005.

[11] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. Bowen, editors, *Proceedings of International Logic Programming Conference and Symposium*, pages 1070–1080. MIT Press, 1988.

[12] H. Hu and G.-J. Ahn. Enabling verification and conformance testing for access control model. In *Proceedings of the 13th ACM Symposium on Access control Models and Technologies*, pages 195–204. ACM, 2008.

[13] G. Hughes and T. Bultan. Automated verification of access control policies. *Computer Science Department, University of California, Santa Barbara, CA*, 93106:2004–22.

[14] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

[15] S. Jajodia, P. Samarati, V. S. Subrahmanian, and E. Bertino. A unified framework for enforcing multiple access control policies. pages 474–485, 1997.

[16] V. Kolovski, J. Hendler, and B. Parsia. Analyzing web access control policies. In *Proceedings of the 16th international conference on World Wide Web*, page 686. ACM, 2007.

[17] N. Li, M. Tripunitara, and Z. Bizri. On mutually exclusive roles and separation-of-duty. *ACM Transactions on Information and System Security (TISSEC)*, 10(2):5, 2007.

[18] V. Lifschitz. What is answer set programming? In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 1594–1597. MIT Press, 2008.

[19] V. Lifschitz and A. Razborov. Why are there so many loop formulas? *ACM Transactions on Computational Logic*, 7:261–268, 2006.

[20] V. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer Verlag, 1999.

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

[22] R. Sandhu and Q. Munawer. How to do discretionary access control using roles. In *Proceedings of the third ACM workshop on Role-based access control*, pages 47–54. ACM New York, NY, USA, 1998.

[23] R. S. Sandhu. Lattice-based access control models. *IEEE Computer*, 26(11):9–19, 1993.

[24] R. S. Sandhu and P. Samarati. Access control: Principles and practice. *IEEE Communications Magazine*, 32(9):40–48, 1994.

[25] A. Schaad and J. D. Moffett. A lightweight approach to specification and analysis of role-based access control extensions. In *SACMAT '02: Proceedings of the seventh ACM symposium on Access control models and technologies*, pages 13–22, New York, NY, USA, 2002. ACM.

[26] K. Sohr, G.-J. Ahn, M. Gogolla, and L. Migge. Specification and validation of authorisation constraints using UML and OCL. *Lecture notes in computer science*, 3679:64, 2005.

[27] K. Sohr, G.-J. Ahn, and L. Migge. Articulating and enforcing authorisation policies with UML and OCL. In *Proceedings of the 2005 workshop on Software engineering for secure systems building trustworthy applications*, pages 1–7, 2005.

[28] M. C. Tschantz and S. Krishnamurthi. Towards reasonability properties for access-control policy languages. In *SACMAT '06: Proceedings of the eleventh ACM symposium on Access control models and technologies*, pages 160–169, New York, NY, USA, 2006. ACM.

[29] XACML. OASIS eXtensible Access Control Markup Language (XACML) V2.0 Specification Set. http://www.oasis-open.org/committees/xacml/, 2007.