Representing and Reasoning about Web Access Control Policies

***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 `XAnalyzer` 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 of the benefits of using policy-based techniques and declarative policy languages may be in vain.

XACML (eXtensible Access Control Markup Language) [1], 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) [2] provides a mapping between RBAC and XACML. However, the current RBAC profile cannot support *contraints* in RBAC that are an important element to govern all other elements in RBAC, while supporting core and hierarchical 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 constrains. 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. 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), which is a form of declarative programming oriented towards expressive knowledge representation and effective reasoning capability. Compared with several existing attempts [3], [4] for the formalization of XACML, 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 [4]. We also overview our tool `XAnalyzer` and conduct experiments with real-world 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 discuss our approach in representing and reasoning XACML policies. We present the implementation of our tool `XAnalyzer` along with the evaluation of our

approach in Section IV. Section V overviews the related work. Section VI 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 build-in functions, data types, combining algorithms, and standard profiles for defining application-specific features. At 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 evaluated to be *true*; otherwise, the policy set or policy is skipped when evaluating a request. A *rule set* is a sequence of rules. Each *rule* consists of a *target*, a *condition*, and an *effect*. The *target* of a rule decides whether a request is applicable to the rule and it has a similar structure as the target of a policy or a policy set; 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 `permit` or `deny`. 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 often has conflicting policies or rules, which are resolved by four different *combining algorithms* [1]: *Deny-Overrides*, *Permit-Overrides*, *First-Applicable* and *Only-One-Applicable*.

B. Role-based Access Control

RBAC is a widely accepted alternative to traditional mandatory access control (MAC) and discretionary access control (DAC) [5]. As MAC is used in the classical defense arena, the access is based on the classification of objects such as security clearance [6] while the main idea of DAC is that the owner of an object has the discretion over who can access the object [7], [8]. 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. [9]. 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 privilege to carry out specified actions. Roles are organized in a partial order $\geq$, so that if $x \geq y$ then role x inherits the permissions of role y. Therefore, members of role x are also implicitly members of role y. In addition, RBAC introduces the notion of constraints that are a powerful mechanism for laying out higher-level organizational policy. 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 [10], [11], [12]. *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 two conflicting roles. *Dynamic SoD constraints*–with respect to activated roles in sessions–typically require that no user can activate two conflicting roles simultaneously. *Historical SoD constraints* restrict the assignment and activation of conflicting roles over the course of time.

C. Answer Set Programming

ASP [13], [14] 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 in that it focuses on knowledge representation and reasoning: its language is an expressive nonmonotonic language based on logic programs under the stable model semantics [15], [16], 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 [17]. 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 looks like contain a vicious cycle, but the semantics is 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 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 strong negation ("$\neg$") turns out to be useful in knowledge representation. For instance,

$$permit(S, watch) \leftarrow \textit{not } child(S)$$

means that if S is not known to be a child then S is permitted to watch, while

$$permit(S, watch) \leftarrow \neg child(S)$$

means that if S is known to be not a child then S is permitted to watch. Such distinction is also useful to solve the frame problem— the problem of describing things that do not change by default. For instance,

$$permitted(S, read, T+1) \leftarrow permitted(S, read, T), \textit{not } \neg permitted(S, read, T+1).$$

The reading permission of S is persistent unless there is a cause for retracting that assumption.

The language also has useful constructs, such as aggregates, 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. Representing and Reasoning about XACML

A. General XACML Policy Analysis

We introduce our logic-based policy reasoning approach for XACML as shown in Figure 1. 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, inconsistency and querying are provided. For instance, *policy verification* is to check if ASP-based representation of the policy entails the property as certain formulas in its specification and *policy comparison* checks the equivalence between two answer set programs.

Figure 2 shows an example XACML policy for a bank system, which is utilized throughout this paper. The root policy set PS_1 contains two policies P_1 and P_2 which are combined using *First-Applicable* combination algorithm. The policy P_1 has two rules, r_1 and r_2, and its rule combining algorithm is *Permit-Overrides*. The policy P_2 contains

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

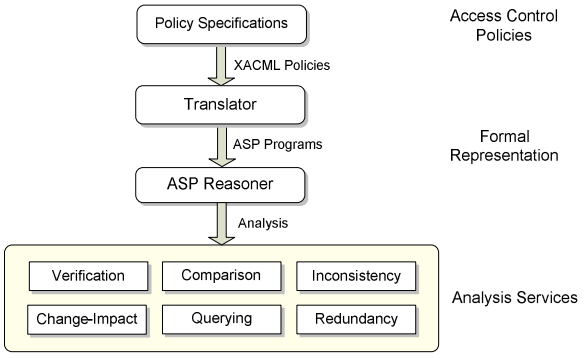

Figure 1. Logic-based policy reasoning for XACML.

```
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>      loanOfficer    </Subject></Subjects>
8         <Resources><Resource> Loan Account </Resource></Resources>
9         <Actions><Action>        Create         </Action>
10                  <Action>       Modify         </Action></Actions>
11      </Target>
12    </Rule>
13    <Rule RuleId="r2" Effect="Permit">
14      <Target>
15        <Subjects><Subject>      accountant     </Subject></Subjects>
16        <Resources><Resource> Ledger Report </Resource></Resources>
17        <Actions><Action>        Create          </Action></Actions>
18      </Target>
19    </Rule>
23  </Policy>
26  <Policy PolicyId="P2" RuleCombiningAlgId="Deny-Overrides">
27    <Target/>
28    <Rule RuleId="r3" Effect="Permit">
29      <Target>
30        <Subjects><Subject>      accountingManager </Subject></Subjects>
31        <Resources><Resource> Ledger Report       </Resource></Resources>
 9        <Actions><Action>        Create            </Action>
10                  <Action>       Modify            </Action></Actions>
33      </Target>
34    </Rule>
35  </Policy>
37</PolicySet>
```

Figure 2. Example XACML policy.

one rule r_3 with *Deny-Overrides* combination algorithm. In this example, there are three roles: *loanOfficer*, *accountant* and *accountingManager*; two resources: *Loan Account* and *Ledger Report*; and two actions: *Create* and *Modify*. r_1 is a *permit* rule which defines that a loan officer can create and modify loan accounts; r_2 is a *permit* rule which defines that an accountant can create ledger reports; and r_3 is a *permit* rule which defines that an accounting manager can create and modify ledger reports.

To formalize XACML in ASP, there exist two challenging issues that should be taken into consideration. First, XACML supports four policy combining algorithms: *Deny-Overrides*, *Permit-Overrides*, *First-Applicable*, and *Only-One-Applicable*. Our formalization process attempts to represent those combining algorithms in ASP programs. Sec-

ond, an XACML policy can contain a sequence of policies or policy sets, which may further contain policies or policy sets. Thus, we need to capture such a recursive XACML specification in ASP programs. Our approach accommodates these issues while converting XACML to ASP programs.

1) Converting XACML to ASP: Our translation method converts core XACML [18] including XACML rules, policies, policy sets and all combination algorithms to ASP programs based on Figure 2. Each translation step is described as follows:

- *Abstracting XACML policy components*

Before the formal translation, we first simplify an XACML policy to have an abstract view.

(a) An XACML rule can be abstracted as follows: $\langle RuleID, effect, subject, action, resource\rangle$
For example, three rules r_1, r_2 and r_3 in Figure 2 can be abstracted as follows respectively:
$\langle r_1, permit, loanOfficer, Create\ or\ Modify, LoanAccount\rangle$.
$\langle r_2, permit, accountant, Create, LedgerReport\rangle$.
$\langle r_3, permit, accountingManager, Create\ or\ Modify, LedgerReport\rangle$.

(b) An XACML policy can be abstracted as follows: $\langle PolicyID, combining\ algorithm, \langle r_1, \ldots, r_n\rangle\rangle$.
For example, policies P_1 and P_2 in Figure 2 can be abstracted as follows respectively:
$\langle p_1, \textit{Permit-Overrides}, \langle r_1, r_2\rangle\rangle$.
$\langle p_2, \textit{Deny-Overrides}, \langle r_3\rangle\rangle$.

(c) Similarly, an XACML policy set can be abstracted as: $\langle PolicySetID, combining\ algorithm, \langle P_1, \ldots, P_n\rangle\rangle$.
For example, policy set PS_1 can be considered as $\langle ps_1, \textit{Deny-Overrides}, \langle p_1, p_2\rangle\rangle$.

- *Mapping XACML rules to ASP*

The rule abstraction in turn can be represented in the language of ASP solver, GRINGO[4] as follows:

```
permit(r1) :- request(loanOfficer,
    create, loanAccount).
permit(r1) :- request(loanOfficer,
    modify, loanAccount).
permit(r2) :- request(accountant,
    create, ledgerReport).
permit(r3) :- request(accountingManager,
    create, ledgerReport).
permit(r3) :- request(accountingManager,
    modify, ledgerReport).
```

- *Mapping XACML policies and policy sets to ASP*

To represent that r_1 and r_2 are contained in policy P_1, for each rule R where R ranges over $\{r_1, r_2\}$, we have

```
permit_from(p1, R) :- permit(R).
deny_from(p1, R) :- deny(R).
```

To represent that r_3 is contained in policy P_2 we have

```
permit_from(p2, r3) :- permit(r3).
deny_from(p2, r3) :- deny(r3).
```

To represent that P_1 and P_2 are contained in policy set PS_1, for each policy P where P ranges over $\{P_1, P_2\}$, we have

```
permit_from(ps1, P) :- permit(P).
deny_from(ps1, P) :- deny(P).
```

Next, we need to represent the policy combining algorithms and recursive specifications in XACML. These representations are crucial components to support reasoning and analysis tasks.

- *Representing XACML combining algorithms in ASP*

The combining algorithm *Permit-Overrides* ("if any rule evaluates to *permit*, then the final decision is *permit*") of policy P_1 can be represented as

```
permit(p1) :- permit_from(p1, R).
deny(p1) :- deny_from(p1, R),
  not permit(p1).
```

The combining algorithm *Deny-Overrides* ("if any rule evaluates to *deny*, then the final decision is *deny*") of policy P_2 can be specified as

```
deny(p2) :- deny_from(p2, r3).
permit(p2) :- permit_from(p2, r3),
  not deny(p2).
```

Also, the combining algorithm *First-Applicable* ("the combined decision is the same as the effect of the first rule which matches the request") of policy PS_1 can be specified as

```
deny(ps1) :- deny_from(ps1, p1).
permit(ps1) :- permit_from(ps1, p1).
deny(ps1) :- deny_from(ps1, p2),
  not permit_from(ps1, p1).
permit(ps1) :- permit_from(ps1, p2),
  not deny_from(ps1, p1).
```

The ASP program that corresponds to the policy described in Figure 2 consists of all the above translations and some domain declarations. We name it Π_1 that is used for policy analysis.

Several related work showed how simple combining algorithms can be represented in first-order logic [18] and description logic [4]. However, their formalizations omit complex combining algorithms such as *Only-One-Applicable* ("Only one rule should be applicable and the final decision is that rule's decision"), which can be represented in ASP using default negation and cardinality constraints as follows [5]:

```
permit(p) :-  1{permit_from(p, R):
```

[4] `http://potassco.sourceforge.net/`.

[5] `R` and `Rule` are variables that range over all rule ids.

```
    rule(R)}1, not deny_from(p, Rule).
deny(p) :-  1{deny_from(p, R):
    rule(R)}1, not permit_from(p, Rule).
```

- *Converting recursive XACML specification to ASP*

To support the recursive specifications of XACML policies, we parse and model an XACML policy as a tree structure, where each terminal node represents an individual rule, each nonterminal node whose children are all terminal nodes represents a policy, and each nonterminal node whose children are all nonterminal nodes represents a policy set. At each nonterminal node, we store the target and combining algorithm. At each terminal node, the target and effect of the corresponding rule are stored. Figure 3 depicts a tree structure of the XACML example policy shown in Figure 2.

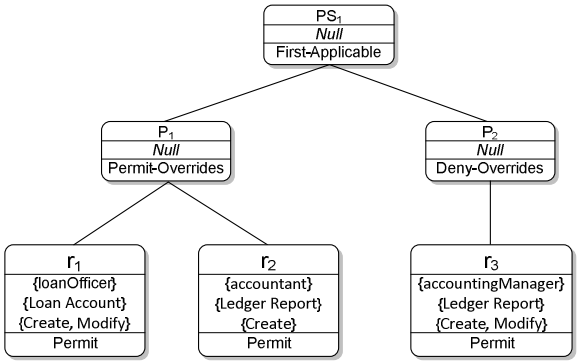

Figure 3. Tree structure of the example XACML policy.

In order to translate all nodes in the tree structure, the translator should traverse the entire tree. Algorithm 1 addresses the translation algorithm from XACML to ASP based on the tree structure. Lines 2-5 deal with the policy nodes, and lines 7-14 handle policy set nodes. Note that if the nodes of a policy set node has children nodes, the algorithm recursively calls the translation function `XACML2ASP`.

2) Analysis services for XACML: Once we represent an XACML into an ASP program Π, we can use off-the-shelf ASP solvers for a variety of comprehensive automated analysis services. This section illustrates several analysis services including policy verification, policy comparison, policy equivalence and redundancy checking, which can be provided by ASP solvers.

- **Policy Verification**. It checks if the policy P entails a certain property F. This can be cast into checking non-existence of answer sets for the program as follows:
  $\Pi \cup \{\leftarrow F'\}$
  where F' is the formula that encodes the property F.
  For example, we can say that PS_1 entails the fact that "A loan officer is permitted to create a loan account" because the program $\Pi_1 \cup \{\neg(permit(ps1) \leftarrow request(loanOfficer, create, loanAccount))\}$ has no answer set.

Algorithm 1: XACML2ASP(P)

```
   Input: A XACML policy P.
   Output: An ASP-based specification of policy A.
1  /* P is a policy*/
2  if Policy(P) = true&&P.child! = null then
3      foreach r ∈ rule(P) do
4          A.append(Translate_rule(r));
5          Remove(r);
6  /* P is a policy set*/
7  if PolicySet(P) = true then
8      if P.child! = null then
9          foreach P' ∈ P.child do
10             XACML2ASP(P');
11     if P.child = null then
12         foreach P' ∈ P.child do
13             A.append(Translate_policy(P'));
14             Remove(P');
15 return A;
```

- **Policy Comparison**. Having the ASP encoding of each policy, we can compare policies. For example, we can check if policy $p2$ is permitted whenever policy $p1$ is permitted. This is cast into checking non-existence of answer sets of program as follows:
  $\Pi \cup \{\leftarrow \textit{not } permit(p1)\} \cup \{\leftarrow permit(p2)\}$.
  We can claim that P_1 is different from P_2 in terms of permit because when $request(loanOfficer, create, loanAccount)$ holds, $permit(p1)$ is also in the answer set but not for $permit(p2)$.
- **Policy Equivalence**. *Equivalence* between policy descriptions can be defined based on *comparison* defined above. We define that two policies P and P' are equivalent if whenever P produces decision α, P' also yields the same decision, and vice versa. With the example above we can notice immediately that P_1 is not equivalent to P_2.
- **Policy Redundancy**. A policy `P` is called redundant if when removed from the policy set, the behaviors of an access control system does not change. *Redundancy checking* can be viewed as an instance of simplification of ASP programs. In other words, we can perform the redundancy check by finding a simpler ASP program that is equivalent to the target ASP program.

B. XACML-based RBAC Policy Analysis

1) A Policy Analysis Framework: As we discussed in Section II, the current XACML profile for RBAC [2] only provides a mapping from core and hierarchical RBAC to XACML but RBAC constraints are not supported in the current XACML profile. To support the reasoning for XACML-based RBAC policy, we introduce a policy analysis framework shown in Figure 4. Our framework begins with the transformation from 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 [19], [20]. Those general constraint specifications are automatically translated to ASP-based constraint specifications based on ASP formalism. Therefore, representing both RBAC system configuration (core and hierarchical RBAC) and RBAC constraints with ASP enables us to support rigorous analysis of constraints that are not addressed in the current XACML profile.

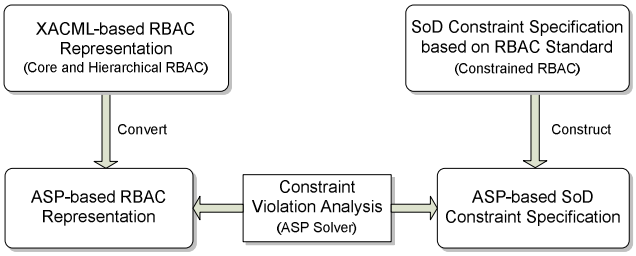

Figure 4. A policy analysis framework for XACML-based RBAC.

2) Core and Hierarchical RBAC Representation: RBAC model defines 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 $< Policy >$ or $< PolicySet >$ defines which roles can be enabled or assigned to which users. Suppose that a user *John* is assigned to two roles *loanOfficer* and *accountant* in the bank system. We can translate those user-to-role assignments (ura) to ASP as follows:

```
ura(John,loanOfficer).
ura(John,accountant).
```

RBAC supports role hierarchy relations. For example, if *accountant* is a junior role of *accountingManager* in the bank system. XACML profile for RBAC can implement role inheritance by including a $< PolicySetIdReference >$ to the Permission $< PolicySet >$ associated with one role inside the Permission $< PolicySet >$ associated with another role. The hierarchy relation between two roles *accountant* and *accountingManager* represented in XACML can be converted to ASP as follows:

```
junior(accountant, accountingManager).
```

Transitive and reflexive relations tc_junior are also needed to define for role hierarchy relations in ASP.

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

In addition, 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)
```

3) RBAC Constraint Representation: As part of RBAC constraints, we demonstrate how separation of duty (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 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 role set `cr`. The 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 roles without considering other components such as user and permission in RBAC. In the real world, we may also have notions of conflicting permission 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 $SoD =< t, e, cs, n >$, where

- $t \in \{s, d, h\}$ represents the types of SoD constraints, where s, d and h stands $static$, $dynamic$ and $historical$, respectively;
- $e \in \{U, R, P\}$ is the RBAC element to which the constraint is applied, where U, R and P denotes $User$, $Role$ 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 2. 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 elements, which

is either `user` or `permission` indicating *SSoD-CU* or *SSoD-CP* constraints, respectively.

Algorithm 2: 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**
    $i \longleftarrow 0$;
    foreach $r \in c.cs$ **do**
      $i \longleftarrow i + 1$;
      $URA.append(ura(U, Ri))$;
      $TC.append(tc\_junior(r, Ri))$;
    $C^{'} \longleftarrow: -c.n\{URA\}, TC.$;
  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**
    $i \longleftarrow i + 1$;
    $SR.append(sr(Si, Ri))$;
    $US.append(us(U, Si))$;
    $TC.append(tc\_junior(r, Ri))$;
  $C^{'} \longleftarrow: -c.n\{SR\}, US, TC.$;
/ Historical Constraint*/*
if $c.t =' h'$ **then**
  $i \longleftarrow 0$;
  foreach $r \in c.cs$ **do**
    $i \longleftarrow i + 1$;
    $SR.append(sr(Si, Ri, Ti))$;
    $US.append(us(U, Si, Ti))$;
    $TC.append(tc\_junior(r, Ri))$;
  $C^{'} \longleftarrow: -c.n\{SR\}, US, TC.$;
return $C^{'}$;

In this section, 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 *loanOfficer* and *accountant* 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:

```
<s, U, {loanOfficer,
  accountant}, 2>
```

Constructed ASP Expression:

```
:- 2{ura(U, R1), ura(U, R2)},
  tc_junior(loanOfficer, R1),
  tc_junior(accountant, R2).
```

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 exceeds the cardinality of the conflicting role set.

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

Constraint Expression:

```
<d, U, {customerServiceRep,
  loanOfficer}, 2>
```

Constructed ASP Expression:

```
:- 2{sr(S1, R1), sr(S2, R2)},
  us(U, S1), us(U, S2),
  tc_junior(customerServiceRep, R1),
  tc_junior(loanOfficer, R2).
```

Note that sr and us denote session-to-role assignment relation and user-to-session assignment relation, respectively.

Most of existing work in specifying [10], [11], [12] and analyzing [21], [22], [23] SoD constraints mainly focus on a system state at one point in time. By introducing a temporal variable *time* in ASP representation, we can overcome such a gab. Hence, 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 exceeds the cardinality of the historical conflicting role set.

Assume that two roles *customerServiceRep* and *accountant* are contained in a historical conflicting role set and the cardinality of the conflicting role set is *two*.

Constraint Expression:

```
<h, U, {customerServiceRep,
  accountant}, 2>
```

Constructed ASP Expression:

```
:- 2{sr(S1, R1, T1), sr(S2, R2, T2)},
  us(U, S1, T1), us(U, S2, T2),
  t_junior(customerServiceRep, R1),
  t_junior(accountant, R2).
```

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 considered. For example, we can evaluate if a user ever activated two conflicting roles in different

time interval by checking the *historical SoD* constraint as a security property against the changing system states.

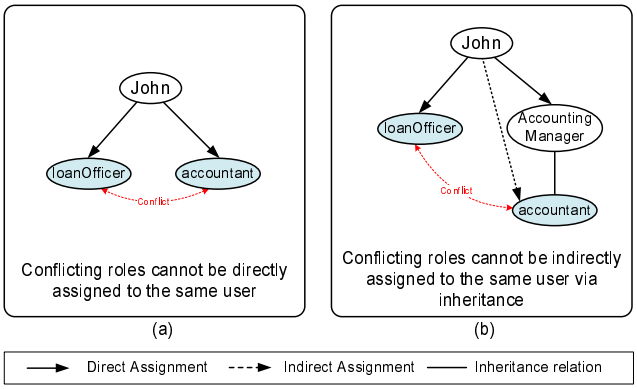

Figure 5. Violation checking for SoD constraints.

4) Violation Analysis of RBAC Constraints : RBAC constraints can be utilized as security properties to check against access control policy configurations for identifying constraint violations. Figure 5 shows a typical example, which illustrates conflicting roles cannot be directly or indirectly (via inheritance) assigned to the same user. Figure 5 (a) shows that the user *John* is assigned to two roles *loanOfficer* and *accountant* simultaneously. However, since *loanOfficer* is statically mutually exclusive to *accountant*, the *SSoD-CR* constraint is violated. Figure 5 (b) depicts a more complex example taking role hierarchy into account. The user *John* acquires two conflicting roles *loanOfficer* and *accountant* through permission inheritance. In order to identify a constraint violation, the *negated* constraint specification is used as a security property. For example, the following is a *negated* specification of the *SSoD-CR* constraint:

```
:- not 2{ura(U, R1), ura(U, R2)},
  tc_junior(loanOfficer, R1),
  tc_junior(accountant, R2).
```

We can check this *negated* specification against ASP-based RBAC configuration in an ASP solver, GRINGO. If the ASP solver finds out an answer set {`ura(John,loanOfficer)`, `ura(John,accountant)`}, that means a user *John* is assigned to two conflicting roles *loanOfficer* and *accountant* at the same time in current RBAC configuration. Thus, a *SoD* constraint violation is identified.

IV. Implementation and Evaluation

We have implemented a tool called `XAnalyzer` in Java 1.6.3. `XAnalyzer` can automatically convert the core XACML and the 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 [3]; among them, *Continue-a* and *Continue-b* are designed for a real-world web application supporting a conference management. Three of the policies, *Weirds*, *FreeCS* and *GradeSheet* are utilized by [24]. 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 size of policy, the conversion time from XACML to ASP, and the reasoning time using GRINGO for each policy. Note that the reasoning time was measured by enabling GRINGO to discover all permitted requests for each policy. From Table I, we observe that the conversion time from XACML to ASP in `XAnalyzer` is fast enough to handle a larger set of policies, such as *Continue-a* and *Continue-b*. It also indicates the reasoning process for policy analysis in ASP solver is also efficient enough for a variety of policy analysis services.

V. Related Work

In [25], a framework for automated verification of access control policies based on relational first-order logic was proposed. The authors demonstrated how to translate XACML policies to the Alloy language [26], 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 [27], the authors formalized XACML policies using a process algebra known as Communicating Sequential Processes (CSP). This utilizes a model checker to formally verify properties of policies, and to compare access control policies with each other. Fisler et al. [3] introduced an approach to represent XACML policies with Multi-Terminal Binary Decision Diagrams

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

(MTBDDs). A policy analysis tool called Margrave was developed. Margrave can verity XACML policies against the given properties and perform change-impact analysis based on the semantic differences between the MTBDDs representing the policies. In [4], a description logic-based approach for analyzing XACML was presented. The authors provided a formalization of XACML that explores the space between propositional logic analysis tools and first-order logic XACML analysis tools. As a basis for the XACML formalization they use description logic, which is a family of languages that are decidable subsets of first-order logic. Compared with other work in XACML, our approach provides a more straightforward formalization with ASP and can cover more XACML features as discussed in the previous sections. Our work with ASP solvers also shows superior performance in handling XACML combining algorithms.

Schaad and Moffett [21] 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 [28], Shor 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 [23], 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 a RBAC model and 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 supporting *reasoning* in XACML-based RBAC policies.

VI. 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 as existing XACML standard does not support constrained RBAC. In addition, we have described a tool called `XAnalyzer`, which can seamlessly work with existing ASP solvers for policy analysis. Our experimental results showed that the performance of our analysis approach could efficiently support larger access control policies.

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 hiding the details of the ASP formalism.

References

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

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

[3] 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*. ACM New York, NY, USA, 2005, pp. 196–205.

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

[5] R. S. Sandhu and P. Samarati, "Access control: Principles and practice," *IEEE Communications Magazine*, vol. 32, no. 9, pp. 40–48, 1994.

[6] R. S. Sandhu, "Lattice-based access control models," *IEEE Computer*, vol. 26, no. 11, pp. 9–19, 1993.

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

[8] 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*. ACM New York, NY, USA, 1998, pp. 47–54.

[9] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman, "Role-based access control models," *IEEE computer*, vol. 29, no. 2, pp. 38–47, 1996.

[10] G.-J. Ahn and R. Sandhu, "Role-based authorization constraints specification," *ACM Transactions on Information and System Security (TISSEC)*, vol. 3, no. 4, pp. 207–226, 2000.

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

[12] N. Li, M. Tripunitara, and Z. Bizri, "On mutually exclusive roles and separation-of-duty," *ACM Transactions on Information and System Security (TISSEC)*, vol. 10, no. 2, p. 5, 2007.

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

[14] V. Lifschitz, "What is answer set programming?" in *Proceedings of the AAAI Conference on Artificial Intelligence*. MIT Press, 2008, pp. 1594–1597.

[15] M. Gelfond and V. Lifschitz, "The stable model semantics for logic programming," in *Proceedings of International Logic Programming Conference and Symposium*, R. Kowalski and K. Bowen, Eds. MIT Press, 1988, pp. 1070–1080.

[16] P. Ferraris, J. Lee, and V. Lifschitz, "Stable models and circumscription," *Artificial Intelligence*, 2010, to appear.

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

[18] 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*. New York, NY, USA: ACM, 2006, pp. 160–169.

[19] 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)*, vol. 4, no. 3, pp. 224–274, 2001.

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

[21] 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*. New York, NY, USA: ACM, 2002, pp. 13–22.

[22] 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*, 2005, pp. 1–7.

[23] H. Hu and G. Ahn, "Enabling verification and conformance testing for access control model," in *Proceedings of the 13th ACM Symposium on Access control Models and Technologies*. ACM, 2008, pp. 195–204.

[24] 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*. ACM New York, NY, USA, 2008, pp. 223–234.

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

[26] D. Jackson, "Alloy: a lightweight object modelling notation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 11, no. 2, pp. 256–290, 2002.

[27] J. Bryans, "Reasoning about XACML policies using CSP," in *Proceedings of the 2005 workshop on Secure web services*. ACM, 2005, p. 35.

[28] K. Sohr, G. Ahn, M. Gogolla, and L. Migge, "Specification and validation of authorisation constraints using UML and OCL," *Lecture notes in computer science*, vol. 3679, p. 64, 2005.