Analyzing Web Access Control Policies

Vladimir Kolovski
Department of Computer Science
University of Maryland
College Park, MD
kolovski@cs.umd.edu

James Hendler
Department of Computer Science
University of Maryland
College Park, MD
hendler@cs.umd.edu

Bijan Parsia
Schoole of Computer Science
University of Manchester
Manchester, UK
bparsia@lcs.man.ac.uk

ABSTRACT

XACML has emerged as a popular access control language on the Web, but because of its rich expressiveness, it has proved difficult to analyze in an automated fashion. In this paper, we present a formalization of XACML using description logics (DL), which are a decidable fragment of First-Order logic. This formalization allows us to cover a more expressive subset of XACML than propositional logic-based analysis tools, and in addition we provide a new analysis service (policy redundancy). Also, mapping XACML to description logics allows us to use off-the-shelf DL reasoners for analysis tasks such as policy comparison, verification and querying. We provide empirical evaluation of a policy analysis tool that was implemented on top of open source DL reasoner Pellet.

Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming—*program verification*; I.2.4 [**Artificial Intelligence**]: Knowledge Representation Formalisms and Methods—*representation languages*; H.3.5 [**Information Storage and Retrieval**]: Online Information Services—*Web-based services*

General Terms

Security, Verification, Languages

Keywords

XACML, access control, policy analysis, policy verification, description logics

1. INTRODUCTION

With the widespread use of Web services, systems on the Web are becoming more connected and integrated. To protect the sensitive information that is often contained in these systems, there is an increased need for adequate security and privacy support. As a result, there has been a great amount of attention to access control policy languages for web services which accommodate large, open, distributed and heterogeneous environments like the Web. These languages aim to be flexible and extensible, with enough features to capture expressive and distributed access control policies. Currently, policy languages with the largest momentum include WS-Policy [17] (a W3C submission), which has been designed to specify the constraints and capabilities of web services, and the more general eXtensible Access Control Markup Language (XACML [7]).

The OASIS standard XACML is an expressive, general purpose XML-based language (with significant deployment[1]) that is used to specify policies on web resources. XACML enables the use of arbitrary attributes in policies, allows for expressing negative authorization, conflict resolution algorithms and enables the use of hierarchical Role Based Access Control, among other things.

With policy languages as expressive as XACML, a new issue has emerged: users have difficulty understanding the overall effect and consequences of their security policies. Even arguably the most important feature in access control - checking that the access control policy will not result in the leakage of permissions to an unintended or unauthorized principal, i.e., *safety* - has become difficult, if not impossible, to do manually. For example, incomplete security policies might unintentionally give access to an intruder. How can a security administrator be certain that her policy covers all corner cases? Even if the administrator does discover a bug in the policy, and fixes it accordingly, the consequences of that fix (policy change) are difficult to analyze.

To address the above issues, there has been a great amount of attention to using logic and formal reasoning techniques for analysis and verification of policies. There have been several attempts to provide a formalization of XACML [11, 4, 18, 6] – unfortunately they either support a small subset of the language, or they severely limit the analysis services offered. To the best of our knowledge, Margrave [6] is the only one analysis tool for XACML that provides both verification and change analysis – and it does so in a quite efficient manner.

In this paper, we provide a formalization of XACML that explores the ground between propositional logic analysis tools (such as Margrave) and full First-Order logic XACML analysis tools (like Alloy [12]). As a basis for the XACML formalization we use description logics (DL), which are a family of languages that are decidable subsets of First-Order logic and are the basis for the Web Ontology Language (OWL) [5]. Because of the correspondence of policy analysis services to DL reasoning services (e.g., policy inclusion can be reduced to concept subsumption, whereas change impact analysis and verification can be reduced to concept satisfiability), the framework can easily provide a variety of policy

[1]See [2] for a list of systems incorporating XACML.

WWW 2007, May 8–12, 2007, Banff, Alberta, Canada.
ACM 978-1-59593-654-7/07/0005.

analysis services and leverage the availability of off-the-shelf DL reasoners optimized for these services. In addition to the analysis services, grounding the framework in DL (and consequently, OWL), provides other benefits:

- the web nature of OWL (it uses URIs for naming and allows for links between ontologies) is suitable for representing an access control language for web resources
- with OWL we can use the framework to extend access control policies with ontology-based descriptions for objects used in the policy. Using DL reasoners at analysis time, we are able to easily integrate these expressive descriptions of the policy domain with the policy itself, without sacrificing any of the services. Also, being described in a standard machine processable language allows these domain descriptions to be easily shared and re-used on the web.

We emphasize that at the moment we intend these services to be used at design (or audit) time, not in a policy enforcement point (PEP) to *enforce* policies. First, it is not entirely clear how useful these services would be for enforcement in today's set-ups. They are not particularly designed for enforcement, and even where they could be used to optimize enforcement (e.g., by pruning redundant tests) such optimization can be done off-line. Second, these services can be computationally expensive. Given that one requirement on a PEP is that it can handle the response requirements of applications under load, we must take care not to introduce too much overhead.

We also provide a prototype implementation of our analysis services on top of open source DL reasoner Pellet [15]. We performed preliminary evaluation of our tool on Margrave's test policy set. While slower than Margrave (as expected), Pellet finished all of the tests in a reasonable amount of time (verifications took less than a second), thus exhibiting encouraging preliminary results. Our results also show that the overhead of using OWL ontologies of different sizes to describe policy objects is manageable.

2. PRELIMINARIES

2.1 Overview of XACML

In this section we provide an overview of XACML (version 2.0 [7]), with focus on the subset of the language that we support. At the end of the section we will discuss the XACML features that we do *not* support.

At the root of all XACML policies is a `Policy` or a `PolicySet`. A `PolicySet` is a container that can hold other `Policies` or `PolicySets`, as well as references to policies found in remote locations. A `Policy` represents a single access control policy, expressed through a set of `Rules`. Each XACML policy document contains exactly one `Policy` or `PolicySet` root element.

2.1.1 Combining Algorithms

Because a `Policy` or `PolicySet` may contain multiple policies or `Rules`, each of which may evaluate to different access control decisions, XACML needs some way of combining the decisions each makes. This is accomplished using a collection of combining algorithms, where each algorithm represents a different way of combining multiple access decisions into a single one. There are Policy Combining Algorithms and Rule Combining Algorithms which have similar semantics. For example, with the `Deny-overrides` Algorithm, if any of the child elements return `Deny`, then the final result is also `Deny` (no matter what the other children return). Table 2.1.1 presents the most common combining algorithms.

2.1.2 Attributes and Rules

Attributes are the most basic unit of a XACML policy. They represent characteristics of the `Subject`, `Resource`, `Action`, or `Environment` in which the access request is made. For example, a user's role, their name, the file they want to access, the current date are all attribute values. Access requests in XACML represent a list of attribute-value pairs.

We provide some datatype support for values of attributes. More specifically, we offer support for built-in and user-defined XML Schema datatypes (currently we only support datetime and integer). For example , we could state that *age* attribute can have value ≥ 18, or that it must be one of $18, 19, 20, 21$.

`Rules` are the most basic element of XACML that actually makes access decision. Essentially, a `Rule` is a function that takes an access request as input and yields an access decision (`Permit`, `Deny`, or `Not-Applicable`). To determine if a Rule is applicable to an access request, the `Target` element is used. A `Target` is a set of simplified conditions for the `Subject`, `Resource` and `Action` that must be met for a `Rule` to apply to a given request. These use boolean functions to compare values found in a request with those included in the `Target`. Example of a rule that returns `Deny` for access requests that have value *read* value for *action* attribute is given below:

```
<Rule RuleId="rule" Effect="Deny">
 <Target>
  <Subjects><AnySubject/></Subjects>
  <Resources><AnyResource/></Resources>
  <Actions>
   <ActionMatch MatchId="function:string-equal">
    <AttributeValue DataType="#string">read</AttributeValue>
    <ActionAttributeDesignator
      AttributeId="action"
      DataType="...#string"/>
   </ActionMatch>
  </Actions>
 </Target>
</Rule>
```

As shorthand notation, we will express a XACML `Rule` as a triple $r = (\texttt{Target}, AD, P)$ where AD is a `Permit` or a `Deny`, and P is the parent `Policy` or `PolicySet`.

2.1.3 Advanced XACML Features

We also support the Hierarchical Role-based Access Control Profile of XACML [3], which allows us to specify inheritance relationships between roles. For example, Role A may be defined to inherit all permissions associated with Role B. In this case, Role A is considered to be senior to Role B in the role hierarchy.

As for the part of XACML that we do *not* support, this includes multi-subject requests, complex attribute functions, rule `Conditions` and some combining algorithms (see Table 2.1.1). While some features (like complex `Conditions`) may be impossible to analyze at development time, there are others which we believe could be handled in our formalization (some types of `Conditions`, more expressive datatypes and the `Only-one-applicable` overriding algorithm) – this is part of our ongoing work.

Name	Summary	Supported
Permit-overrides	If any rule evaluates to Permit, then the final decision is also Permit.	yes
Deny-overrides	If any rule evaluates to Deny, then the final decision is also Deny.	yes
First-applicable	The effect of the first rule that applies is the decision of the policy. The rules must be evaluated in the order that they are listed.	yes
Only-one-applicable	If more than one rule is applicable, return Indeterminate. Otherwise return the access decision of the applicable rule	
Ordered-permit-overrides	Same as Permit-Overrides, except the order in which rules are evaluated is the same as the order in which they are in the policy.	
Ordered-deny-overrides	Same as Deny-Overrides, except the order in which rules are evaluated is the same as the order in which they are in the policy.	

Table 1: Rule Combining Algorithms. Third column indicates whether the particular combining algorithm is supported in our formalization.

2.2 Description Logics

In this section we provide an overview of DL and present the syntax and semantics of a commonly used logic ($\mathcal{SHOIN}$).

In DL, the domain of interest is modeled using individuals, concepts and roles[2], denoting objects of the domain, unary predicates and binary predicates respectively. Atomic concepts (C) and atomic roles (R) are elementary descriptions and complex ones can be built on top of them using constructors. The available concept constructors determine the expressive power of the description logic in question. For example, in $\mathcal{SHOIN}$the following constructors are available:

$$C \leftarrow A|\neg C|C_1 \sqcap C_2|C_1 \sqcup C_2|\exists R.C|\forall R.C|\bowtie nS|\{a\}$$

where A is an atomic concept, a is an individual, $C_{(i)}$ a $\mathcal{SHOIN}$ concept, R a role, S a *simple* role[3] and $\bowtie \in \{\leq, \geq\}$. We write $\top$ and $\bot$ to abbreviate $C \sqcup \neg C$ and $C \sqcap \neg C$ respectively.

For C, D concepts, a *concept inclusion axiom* is an expression of the form $C \sqsubseteq D$. A TBox T is a finite set of concept inclusion axioms. An ABox $\mathbf{A}$ is a finite set of concept assertions of the form $C(a)$ (where C can be an arbitrary concept expression) and role assertions of the form $R(a, b)$.

An *interpretation* $\mathcal{I}$ is a pair $\mathcal{I} = (\mathcal{W}, .^{\mathcal{I}})$, where $\mathcal{W}$ is a non-empty set, called the *domain* of the interpretation, and $.^{\mathcal{I}}$ is the *interpretation function*. The interpretation function assigns to each atomic concept A a subset of $\mathcal{W}$, to each role R a subset of of $\mathcal{W} \times \mathcal{W}$ and to each individual a an element of $\mathcal{W}$. The interpretation function is extended to complex roles and concepts as given in [10].

The satisfaction of a $\mathcal{SHOIN}$axiom α in an interpretation $\mathcal{I}$, denoted $\mathcal{I} \models \alpha$ is defined as follows: (1) $\mathcal{I} \models R_1 \sqsubseteq R_2$ iff $(R_1)^{\mathcal{I}} \subseteq (R_2)^{\mathcal{I}}$; (2) $\mathcal{I} \models Trans(R)$ iff for every $a, b, c \in \mathcal{W}$, if $(a, b) \in R^{\mathcal{I}}$ and $(b, c) \in R^{\mathcal{I}}$, then $(a, c) \in R^{\mathcal{I}}$; (3) $\mathcal{I} \models C \sqsubseteq D$ iff $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$; The interpretation $\mathcal{I}$ is a model of the RBox R (respectively of the TBox T) if it satisfies all the axioms in R (respectively T). $\mathcal{I}$ is a model of $\mathsf{K} = (\mathsf{T}, \mathsf{R})$, denoted by $\mathcal{I} \models \mathsf{K}$, iff $\mathcal{I}$ is a model of T and R.

For this work, it is important to discuss two basic reasoning services offered by DL: satisfiability and subsumption. Determining satisfiability of a concept C in a KB K amounts to a check whether K admits a model in which the interpretation of C is nonempty. Subsumption between two concepts C and D in K, amounts to a check whether $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ for every interpretation I of K, denoted as $\mathsf{K} \models C \sqsubseteq D$. Subsumption is reduced to concept satisfiability as follows: C is subsumed by D in K iff $C \sqcap \neg D$ is not satisfiable in K.

[2]Note that there is a distinction between DL roles, which are binary predicates, and roles in access control, which are usually unary predicates. We will refer to either of these as roles only when clear from context.

[3]See [10] for a precise definition of simple roles

3. RUNNING EXAMPLE

In this section we will introduce an example which will be used throughout the paper to illustrate the services that our formalization provides. In this toy example, initially there are two security roles, *Manager* and *Developer*; one resource: *Report*; and two actions: *read*, *write*. The root policy set contains two policy sets which are combined using `First-applicable` combining algorithm. The policy is presented in graphical form in figure 1.

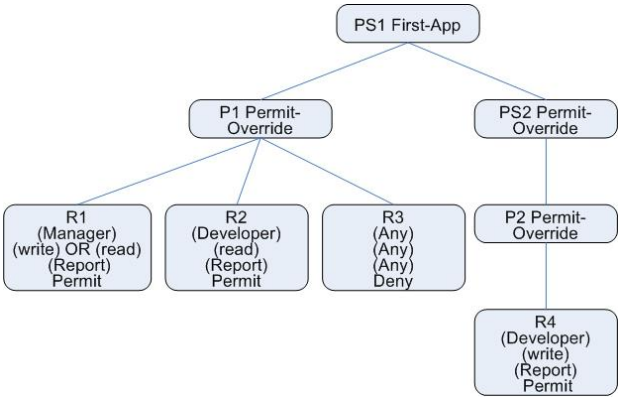


Figure 1: Example Policy

The safety property for this example is: *Developers* cannot *write* to *Reports*. Checking the example policy against this property with Pellet produces a fail, with the following counter example returned:

role=Manager, role=Developer, action=write, resource=report

Thus, if a requester comes along that is a member of both security roles (*Manager* and *Developer*), then she can gain *write* access to *Report*. To prevent a *Developer* who is masquerading as a *Manager* from *writing* to *Report*, we use a

separation of duty constraint: no user can be a member of both *Manager* and *Developer* role at the same time. However, the policy fails to satisfy the property even after adding the above constraint. This time, the counter example given is:

role=Developer, action=write, action=read, resource=report

Apparently, there is another way for a *Developer* to gain write access: if he tries to both *read* and *write* to *Report* at the same time. To prevent this from happening, we can restrict R_2 such that *only* one value (*read*) is allowed for action attribute. After adding this constraint the policy satisfies the property.

With this simple policy we can also illustrate a new analysis service that we provide: policy *redundancy*. A policy element is redundant if removing the element does not change the behavior of the access policy. To understand our motivation for this service, consider `Rule` R_4 in Figure 1. R_4 will always be overridden by R_3 , since the policy combination is `First-applicable`, and the `Target` of R_3 subsumes the `Target` of R_4. In a policy evaluation engine, R_4 can be dropped without any consequences to the security policy. This elimination of unnecessary `Rules` could potentially provide significant optimizations of the policy engine.

The above example illustrated only a subset of the services provided by our framework; a full list follows:

- **Constraints** - We already mentioned separation of duty constraints. In addition, we can also specify more general *cardinality constraints*; for example, a user cannot be a member of more than 3 security roles at a time. Property (attribute) hierarchies are allowed as well: if X is a *brother-of* Y, then he is a *relative-of* Y.
- **Policy Comparison** - For two policies (or policy sets) P_1 and P_2 check if whenever P_1 yields a decision α, P_2 will yield α, too. If not, give a counter example.
- **Policy Verification** - Check if the policy satisfies a particular policy property. If not, give a counter-example.
- **Policy Incompatibility** - If for two policies P_1 and P_2, there cannot exist an access request where both policies apply (yield a decision), then these policies are *incompatible*.
- **Policy Redundancy** - For a policy and an access decision (Permit or Deny), check whether the policy can ever satisfy that decision (or it will be always overridden by some other policy higher up the hierarchy).
- **Policy Querying** - Search for policies in the document based on attribute values.

4. FORMALIZATION OF XACML

The basic unit in XACML that yields an access decision is a `Rule`. To capture the behavior of XACML correctly, we need to formalize the prerequisite of the `Rule` (which is the `Target` element), and its head (the access decision). We also need to capture how the access decision is propagated upwards toward the root `PolicySet` - for this, the rule and policy combining algorithms have to be taken into account.

While the `Target` element of `Rules` and `Requests` can be mapped to a DL concept expression (we discuss this in more detail below), the interaction of the access decision of various policy elements is difficult, if not impossible, to do using only description logics. This is because of the semantics of the combining algorithms which requires us to use a formalism that supports preferences. To capture the behavior of the XACML combining algorithms, we use Defeasible description logics (DDL [16]), which is a formalism that allows for expressing defeasible rules on top of description logics.

Only a limited fragment of DDL is needed to formalize the combining algorithm. In Section 4.1 we provide the description of this fragment. Then, we formalize the four main policy elements in XACML: `Rules`, `Requests`, `Policies` and `PolicySets`. Considering the obvious similarities between some of them (and for better presentation), we have grouped them in two: `Rules` with `Requests` and `Policies` with `PolicySets` . Finally, in Section 4.4 we show how it is possible to reason about XACML policies in this formalization using DL reasoners.

4.1 DDL Preliminaries

The subset of DDL that we use, termed DDL^-, is expressive enough for our purposes and at the same time has the same computational complexity as the underlying description logics [16].

We represent a rule in DDL^- as

$$Pre(r) \longmapsto Con(r),$$

($Pre(r)$ and $Con(r)$ obviosly referring to the antecedent and consequent).

In DDL^-, we do not allow any predicates from the DL KB in the head of the rules. Instead, for each policy or policy set P, we create two literals, which we call *effect-literals* : $Permit\text{-}P, Deny\text{-}P \in \mathcal{L}$. If the theory derives that $Permit\text{-}P(r)$ for some request r, that means we inferred a `Permit` access decision for policy element P to request r. To distinguish these literals for different policy elements, we append the policy element id to the literal name. Instead of denoting it as $Permit\text{-}id(P)$ we abuse notation and simply write $Permit\text{-}P$. Note here that only effect-literals are allowed in heads of rules in $\mathcal{R}$.

A set of rules $\mathcal{R}$ can contain both *strict* and *defeasible* rules. Strict rules cannot be overridden: whenever the body of the rule is derived, the head is derived as well. With defeasible rules, even though the body of the rule might be derived, the rule might still be overridden by another conflicting, higher priority rule. Following we give a definition of strict and defeasible rules that takes into account XACML's combining algorithms.

Definition 1. For each rule $r \in \mathcal{R}$, let P, P_{par} correspond to the policy element and its parent in the XACML document. r is a strict rule in the following cases:

1. If $Permit\text{-}P_{par} \in Con(r)$, and the combining algorithm of P_{par} is `Permit-overrides`
2. If $Deny\text{-}P_{par} \in Con(r)$, and the combining algorithm of P_{par} is `Deny-overrides`
3. If P is the first element in the list of children of P_{par}, and the combining algorithm of P_{par} is `First-applicable`

All other rules in $\mathcal{R}$ are defeasible.

Definition 2. A DDL^- theory $\mathcal{D}$ is a tuple $(\mathsf{K}, \mathcal{R}, >, \mathcal{L})$ where K is a DL knowledge base, $\mathcal{R} = \mathcal{R}_s \cup \mathcal{R}_d$ a finite set that contains strict and defeasible rules , $>$ a superiority relation on $\mathcal{R}$, and $\mathcal{L}$ a set of effect-literals used in the rules of $\mathcal{R}$.

Following we provide procedural semantics for the defeasible theory (derived from [16]). A derivation (proof) is a finite sequence $P = (P(1), \ldots, P(n))$ of literals that belong to $\mathcal{L}$. Conclusion in a DDL^- theory is a set of tagged effect-literals. We have only two tags: for a literal l, $+l$ means l was derived, and -l means it cannot be derived. We use $\mathcal{D} \vdash \alpha$ iff there is a derivation sequence $P = (P(1), \ldots, P(n))$ s.t. $+\alpha \in P$; we say that α is *provable* in $\mathcal{D}$.

If $P(i+1) = +l$ **then** either
(1) $\exists r \in R_s$ s.t.
$l \in Con(r)$ **and**
$\mathsf{K} \models Pre(r)$ **or** $Pre(r) \in P(1..i)$
(2)or $\exists r \in R_d$ s.t.
$l \in Con(r)$ **and**
$\mathsf{K} \models Pre(r)$ **or** $Pre(r) \in P(1..i)$
$\forall q \in (R_d \cup R_s)$ s.t. $Con(q) \sqsubseteq \neg Con(r)$ either
$\mathsf{K} \not\models Pre(q)$ **and** $Pre(q) \notin P(1..i)$
or $r > q$

Figure 2: Derivation Procedure

Figure 2 gives the procedural semantics for the derivation. In (1), which is the strict rule case, we infer l if there exist a strict rule that concludes l and the prerequisite of that rule is either entailed by the DL knowledge base or was derived before in this proof. The defeasible rule case is more involved: we also need to make sure that the defeasible rule that infers l is not also overridden by a conflicting rule with a higher priority.

Deriving that a literal cannot be proven from a DDL^- theory is similar to above, but all of the conditions are negated (defined as strong negation in [16]). We omit the procedural semantics here for brevity and point the interested reader to [13].

4.2 Mapping XACML Requests and Rules

A XACML `Rule` is translated to a rule in $\mathcal{R}$. The `Target` element is translated to a DL concept expression C and becomes the antecedent of the new rule. The `Effect` is mapped to an effect-literal $L \in \mathcal{L}$ and becomes the conclusion. The effect-literal can be either *Permit-P* or *Deny-P* where P is the `Policy` that contains the `Rule`. This new rule, denoted $C \mapsto L$, is added to $\mathcal{R}$. For any policy P and rules r_1, r_2 s.t. *Permit-P* $\in Con(r_1)$ and *Deny-P* $\in Con(r_2)$, we state that r_1 and r_2 are conflicting.

The full mapping of the `Target` element to a DL concept expression is given in Table 2. The main idea is that attribute-value pairs are mapped to existential restrictions – for example (*role Developer*) would be mapped to $\exists role.Developer$. We also allow for propositional combinations of attribute-value pairs. Note here that we enforce a one-to-one mapping from attribute names and values used in the XACML policy to their corresponding DL roles and concepts in K (we create a DL role or a concept with the same name as the XACML attribute or value).

For the XACML construct *Any*, we formalize it as a disjunction where each disjunct corresponds to an attribute. For each attribute, we create another disjunction from all possible attribute values for that attribute. For example, formalizing *Any* using this mapping would create 15 disjuncts (there are 3 attributes and 5 attribute values). By assuming that the values for role, action and resource attributes are disjoint, we can prune the search space significantly (see how the `Any` occurring in the running example is mapped below).

EXAMPLE 1. *The rules in our running example are mapped to:*

R_1 : $\exists role.Manager \sqcap \exists resource.Report \sqcap$
$(\exists action\text{-}type.read \sqcup \exists action\text{-}type.write) \mapsto Permit\text{-}P_1$

R_2 : $\exists role.Developer \sqcap \exists action\text{-}type.read \sqcap$
$\exists resource.Report \mapsto Permit\text{-}P_1$

R_3 : $\exists role.Manager \sqcup \exists role.Developer \sqcup \exists action.Write \sqcup$
$\exists action.read \sqcup \exists resource.Report \mapsto Deny\text{-}P_1$

R_4 : $\exists role.Developer \sqcap \exists action\text{-}type.write \sqcap$
$\exists resource.Report \mapsto Permit\text{-}P_2$

XACML requests are mapped in the same manner as rules, since they also can be represented as a list of attribute value pairs. To check whether a request r matches a rule with target T, we only need to check whether $\mathsf{K} \models \pi(T)(r)$ (equivalent to instance checking in description logics).

4.3 Mapping XACML Policies and Policy Sets

To propagate the access decisions from `Rules` to the root `PolicySet`, we introduce the following rules in $\mathcal{R}$:

1. For each XACML `Rule` r : (`Target Deny` P), add an axiom to R,
$$R = R \cup \{\pi(Target) \mapsto Deny\text{-}P\}$$
2. For each XACML `Rule` r : (`Target Permit` P), add an axiom to R,
$$R = R \cup \{\pi(Target) \mapsto Permit\text{-}P\}$$
3. For each policy element P and parent policy element PS introduce the following axioms:
$$Permit\text{-}P \mapsto Permit\text{-}PS$$
$$Deny\text{-}P \mapsto Deny\text{-}PS$$

A `Policy` or a `PolicySet` can also have a `Target` element. However, we can propagate the constraints specified in `Target` down to `Target`s of its children. In this manner, we propagate the constraints to the XACML `Rule` elements. Thus, without loss of generality, we can assume that `Policy` and `PolicySet` elements have empty `Target` – all of the constraints are propagated down to the `Target` of their XACML `Rule`s descendants.

In Table 3, we provide the full translation of our toy example to a DDL^- theory.

4.4 Reduction to DL Concept Expressions

This section provides an important result: an algorithm to reduce derivability in DDL^- to concept satisfiability in description logics. In particular, for a effect-literal α we will

Syntax	π
R ::= (`Rule` T Effect)	$\pi(T) \mapsto \pi(\text{Effect})$
Effect ::= `Permit` | `Deny`	$Permit\text{-}P$ | $Deny\text{-}P$ (P is parent policy)
T ::= ((Sub) (Act) (Res))	$\pi(Sub) \sqcap \pi(Act) \sqcap \pi(Res)$
Sub | Act | Res ::= Fcn	$\pi(Fcn)$
Fcn ::= AV | Fcn ∩ Fcn | Fcn ∪ Fcn | ¬ Fcn	$\pi(AV)\|\pi(Fcn) \sqcap \pi(Fcn)\|\pi(Fcn) \sqcup \pi(Fcn)\|\neg\pi(Fcn)$
AV ::= (attr-id attr-val)	$\exists\pi(attr\text{-}id).\pi(attr\text{-}val)$
attr-id	DL role corresponding to *attr-id*
attr-value	DL concept corresponding to *attr-val*

Table 2: Mapping `Rule` to a DL class expression

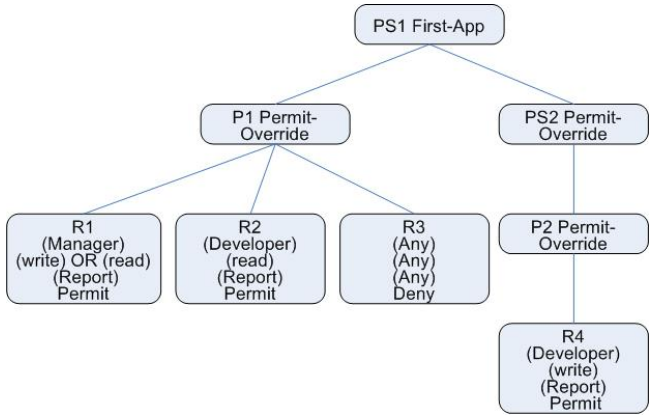
Diagram	Formalization
	$\mathcal{R} = \{$ $P_1 : Permit\text{-}P1 \mapsto Permit\text{-}PS1,$ $P_2 : Deny\text{-}P1 \mapsto Deny\text{-}PS1,$ $P_3 : Permit\text{-}PS2 \mapsto Permit\text{-}PS1,$ $P_4 : Deny\text{-}PS2 \mapsto Deny\text{-}PS1,$ $P_5 : Permit\text{-}P2 \mapsto Permit\text{-}PS2,$ $P_6 : Deny\text{-}P2 \mapsto Deny\text{-}PS2$ $R_1 : \exists role.Manager \sqcap \exists resource.Report \sqcap$ $(\exists action.read \sqcup \exists action\text{-}type.write) \mapsto Permit\text{-}P_1$ $R_2 : \exists role.Developer \sqcap \exists action\text{-}type.read \sqcap$ $\exists resource.Report \mapsto Permit\text{-}P_1$ $R_3 : \exists role.Manager \sqcup \exists role.Developer \sqcup$ $\exists action.Write \sqcup \exists action.read \sqcup$ $\exists resource.Report \mapsto Deny\text{-}P_1$ $R_4 : \exists role.Developer \sqcap \exists action\text{-}type.write \sqcap$ $\exists resource.Report \mapsto Permit\text{-}P_2\}$ $\delta = \{P_1 > P_4, P_2 > P_3, R_1 > R_3, R_2 > R_3\}$ $\mathcal{L} = \{Permit\text{-}PS1, Deny\text{-}PS1, Permit\text{-}PS2,$ $Deny\text{-}PS2, Permit\text{-}P1, Deny\text{-}P1,$ $Permit\text{-}P2, Deny\text{-}P2\}$

Table 3: Mapping the example access control policy to a DDL^- theory.

show how to generate a DL concept expression A in K s.t. α is derivable in $\mathcal{D}$ iff A is satisfiable in K.

First we will illustrate the intuition with a simple example. In the following DDL^- theory: $r_1 : A \mapsto Permit\text{-}1, r_2 : B \mapsto Deny\text{-}1, r_1 < r_2$ we have only two rules whose prerequisites are DL expressions A and B. To check if $Permit$-1 can be derived, we need to check if the concept A is satisfiable (since it is the only way to derive $Permit$-1). However, this alone is not enough, since r_1 can be overridden by r_2. Thus, we need to check the satisfiability of: $A \sqcap \neg B$. If this expression is satisfiable, then there can exist an access request such that it satisfies A and not B. If this expression was unsatisfiable, then there would be no possibility of deriving $Permit$-1 since it would always be overridden by r_2.

The transformation function that takes a DDL^- theory $\mathcal{D}$ and an effect-literal l as input, and generates a DL concept expression is given below:

Definition 3. If $a \notin \mathcal{L}$, then $map(a) = a$. When $a \in \mathcal{L}$, there are two cases - it is a `Permit` effect-literal or a `Deny` effect-literal. For a `Permit` effect-literal,

$map(a) = \bigsqcup(map(C) \sqcap \neg(\bigsqcup map(J)))$ where

- $C \mapsto Permit\text{-}P \in \mathcal{R}$
- $J \mapsto Deny\text{-}P \in \mathcal{R}$
- $J > C$

for some $Permit$-P, $Deny$-P.

$map(a)$ is defined analogously for `Deny` effect-literals.

THEOREM 1. *For a DDL^- theory $\mathcal{D} = (\mathsf{K}, \mathcal{R}, >, \mathcal{L})$ and literal $\alpha \in \mathcal{L}$, $map(\alpha)$ is satisfiable iff there exists a request i s.t. $\mathcal{D} \vdash +\alpha(i)$ (α is provable). $\neg map(\alpha)$ is satisfiable iff there is a request j s.t. $\mathcal{D} \vdash -\alpha(j)$ (α is not always provable).*

Proof of this theorem is available in the accompanying report [13].

5. POLICY IDIOMS

One of the distinguishing features of our mapping is that the subjects, actions and resources used in the access policies are mapped to DL concepts and roles. As a result, we can extend the access policy with semantic descriptions of the policy domain. In other words, we can use a domain ontology (expressed in DL) to provide a semantic description for the entities used in the access policy. If the policy is about *Managers*, *Developers* and *Reports*, we can have an ontology that describes the company, and link the policy entities with concepts in the company ontology using subclass relationships. For example, we can state that a *Manager* is an *Employee* that is a boss of at least one *Person*:

$$Manager \sqsubseteq Employee \ \sqcap \ \exists boss.Person$$

Using such ontologies, we show how common policy idioms can be expressed in description logics:

1. *Role hierarchies* are easily captured with subclass axioms. For example, stating that a *LeadDeveloper* inherits all of the access privileges of the *Developer* role can be expressed as:

$$\exists role.LeadDeveloper \sqsubseteq \exists role.Developer$$

2. Hierarchies on Attributes, can be captured using property hierarchies in DL. For example, to state that if a person is a CIO of a company, that means he is also an employee of that company, we write:

$$CIO\text{-}of \sqsubseteq employee\text{-}of$$

3. *Separation of duty* constraints can be captured with disjoint axioms. To state separation of duty for two security role A and B, we use:

$$\exists role.A \sqsubseteq \neg \exists role.B$$

4. *Cardinality constraints* can be expressed on any given attribute. To state that the *role* attribute cannot have more than k values, we can write:

$$\geq k \ role.\top \sqsubseteq \bot$$

We can even specify maximum number of users that a security role can have, with a combination of inverses and cardinality constraints. For example, the following says that a role cannot have more than k users:

$$\geq k \ role^{-}.\top \sqsubseteq \bot$$

6. ANALYSIS SERVICES

In this section be provide an overview of the analysis services provided by our formalization.

6.1 Policy Comparison

The *map* function defined above allows us to easily compare the behaviors of two policies. For example, we can check for policy *subsumption*: P_2 subsumes P_1 ($P_1 \sqsubseteq P_2$) if whenever P_1 produces access decision α, P_2 also yields the same access decision. We can restrict our attention to *Permit*, *Deny* or both.

To determine whether $P_1 \sqsubseteq P_2$, we try to build a request i s.t. $\mathcal{D} \vdash Permit\text{-}P_1(i)$ and $\mathcal{D} \nvdash Permit\text{-}P_2(i)$. If it is possible to build a request s.t. $Permit\text{-}P_1(i)$ was derived, and $Permit\text{-}P_2(i)$ was not derived, then $P_1 \not\sqsubseteq P_2$. Translating to DL expressions, this reduces to checking whether the concept $map(Permit\text{-}P_1) \sqcap \neg map(Permit\text{-}P_2)$ is satisfiable. If the concept is satisfiable, such request exists.

As an example, consider adding a new security role , *LeadDeveloper*, to our running example. The updated policy now contains an additional `Rule`:

$$\exists role.LeadDev \sqcap \exists action.write \sqcap \exists resource.Report \ \mapsto Permit\text{-}P_1$$

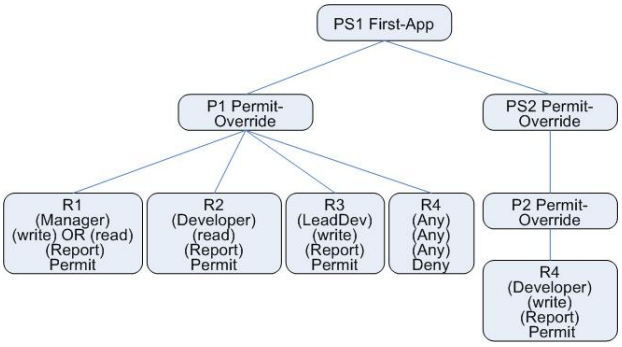


Figure 3: Updated Policy (with LeadDev role)

To check whether we have given any unintended access to other roles, we use the policy subsumption algorithm, that is, we generate the following concept expressions:

$$map(Permit\text{-}PS_{old}), map(Deny\text{-}PS_{new}),$$
$$map(Permit\text{-}PS_{old}), map(Deny\text{-}PS_{new})$$

Subsumption holds only both of the following hold:

$$map(Permit\text{-}PS_{old}) \sqsubseteq map(Permit\text{-}PS_{new})$$

$$map(Deny\text{-}PS_{old}) \sqsubseteq map(Deny\text{-}PS_{new})$$

Pellet reports the first statement is true, which is obvious from looking at the `Rules` themselves (the `Rule` that we added in PS_{new} yields a `Permit`). However, Pellet reports subsumption does not hold w.r.t. `Deny`.

In cases of non-subsumption, it is useful to know what are the counter examples, i.e., to show the user a request where PS_{new} and PS_{old} would yield different decisions. Since we use a tableau-based DL reasoner for policy analysis, to check whether $A \sqsubseteq B$, we try to build a representation of a model for $A \sqcap \neg B$. If such representation *can* be built, it means the subsumption does not hold. In that case the completion graph (representation of the model) just built can easily be extracted from the internals of the reasoner and used as a counter example. Here we get several counter-examples:

1) role=LeadDev, action=read, resource=report,
action=write,
resource=report
2) role=LeadDev, action=write, resource=report
3) role=LeadDev, role=Developer, action=write,
resource=report

The first two are expected (because of the new Permit rule), however the third counter example represents a potentially dangerous access leak to a person who is a member of role Developer. It is possible to fix this bug by separating the roles of *Developer* and *LeadDev*:

$$\exists role.LeadDev \sqsubseteq \neg\exists role.Developer$$

We can generalize the technique used for policy subsumption to policy *comparison*. For two policies P_1 and P_2, we first specify the access decisions we are interested in (say, `Deny` for the first policy and `Permit` for the second), and then check satisfiability of the *map* expressions for those decisions :

$$map(Permit\text{-}PS_{old}) \sqcap map(Deny\text{-}PS_{new})$$

If the above expression is not satisfiable, then there cannot be an access request s.t. the first policy yields a `Deny` and the second one yields a `Permit`. If it is satisfiable, there is such an access request, and we can extract the counter example from the internals of the reasoner. To get all counter examples, we need to retrieve *all* open and complete completion graphs (representations of models) that the concept expression admits; this involves saturation of the tableau, a technique for which DL reasoners are not particularly optimized.

Finally, the service of verifying changes was introduced in [6], we show here that it can be accomplished using description logics as well. The safety properties that are to be checked are simply added to the conjunction of the *map* expressions. For example, if we want to verify that there were all of the changes from `Permit` to `Deny` in the above policy involved the *LeadDev* role, we could test the following concept expression:

$$map(Permit\text{-}PS_{old}) \sqcap map(Deny\text{-}PS_{new}) \sqcap \forall role.\neg LeadDev$$

6.2 Policy Redundancy

Another service we provide is determining redundant `Rules`[4] Intuitively, a redundant rule is one that whenever fires, it is always overridden by some other rule or policy with higher priority in the hierarchy. It does not matter whether the rule is part of the policy document or not – in both cases, for any access request i, the evaluation engine will give identical results. A simple way to check redundancy of a rule r is to perform change impact analysis for a policy with and without the rule. However, there is a more optimal way of checking redundancy, by building a concept expression that ignores the policy elements that *cannot* override the rule we are checking. Algorithm 1 contains the pseudo-code.

The function starts with an input `Rule` r and works its way up to the root policy element. At the same time, it builds a disjunction, that consists of the concept expressions for every `Policy` or `PolicySet` that *can override* the access decision made by r. If the prerequisite of r is subsumed by this disjunction, then we know for certain that the access decision of r will always be overridden by some policy element. In this case, r is redundant.

Redundant rules do not have to be evaluated, and can be safely removed from a policy file. This simplifies the policy and improves runtime performance of the policy evaluator because there are less rules to match requests against.

[4]The technique can be easily generalized to `Policies` or `PolicySets` – for brevity we focus on `Rules` only.

Algorithm 1 is_redundant$(r, \mathcal{D})$

Input:
r: defeasible rule of the form $T \mapsto \alpha$
$\mathcal{D}$: DDL^- theory $\mathcal{D} = (\mathsf{K}, \mathcal{R}, >, \mathcal{L})$
Output:
b: returns true if r is redundant, false otherwise

```
1: J ← ⊥
2: r_old ← r                                   ▷ cache r
3: while true do
4:     J ← J ⊔ map(Pre(q)) where
5:       q ∈ R and
6:       parent(q) = parent(r) and
7:       q > r                                 ▷ q can override r
8:     if parent(r) ≠ null then
9:       r ← getParentRule(r)
10:    else
11:      break
12:    end if
13: end while
14: if Pre(r_old) ⊑ J then                     ▷ request is subsumed
15:    return true
16: else
17:    return false
18: end if
```

6.3 Policy Verification

We allow for specification and formal verification of properties of policies. We admit policy properties of the same form as rules, i.e, they have a DL concept expression as body and a `Deny` or `Permit` as head. To check whether a policy P satisfies a property, first we compute $map(Permit\text{-}P)$ and $map(Deny\text{-}P)$ using the techniques described in Section 4.4. Then, we try to build a (finite representation of a) model where a request can match both the policy property and the *map* concept for the effect-literal that has the *opposite* access decision than the policy property. If the expression is not satisfiable, then the policy is formally verified against the property. If the expression is satisfiable, the model that is built is returned as a counter example.

When specifying properties, we can use the full expressiveness of description logics. For example, we can state that the a user who is not a *Manager* and is less than 25 years old is not allowed to do perform more than one action at a time:

$$\exists role.\neg Manager \sqcap \exists age. \leq_{21} \sqcap > 1 action \longmapsto \texttt{Deny}$$

7. IMPLEMENTATION AND EVALUATION

We have implemented a prototype XACML analysis tool on top of open source DL reasoner Pellet. As a test case, we used the access policy for the conference paper manager Continue. The authors of Margrave [6] translated this policy to XACML, used it to test their analyzer and published the policy at [1]. This realistic access policy was used to evaluate the correctness and performance of our tool as well.

Parsing the policy file using sunxacml and loading the policy ontology took 2.1 seconds. Converting the policies to description logics took additional 1.7 seconds.

For the purpose of this paper, we tested verification of properties. There are 12 safety properties for the Continue policy. We encoded them in description logics and used

Pellet to verify them.

EXAMPLE 2. *A sample property from the set is:*

If a subject is not a pc-chair or admin, then he may not set the meeting flag.

This can be encoded in DL as:

$$P \equiv \neg(\exists role.pc\text{-}chair \sqcup \exists role.admin) \sqcap \exists action.write \sqcap \exists resource.meetingFlag$$

P can be verified by checking the concept satisfiability of $map(Permit) \sqcap P$ *where* $map(Permit)$ *is the concept expression corresponding to access requests that map to Permit (details on map function in Section 4.4). If* $map(Permit) \sqcap P$ *is not satisfiable, then property P holds for the policy.*

It is important noting that, to improve performance, tableau reasoners reduce concept expressions to a simplified normal form before checking concept satisfiability. Due to the sheer size of the concept expressions returned from our mapping function (the XACML construct `Any` dramatically increases the size of the DL representation, and it is used liberally in Continue) this function proved to be the bottleneck of the analyzer. To alleviate the issue we extracted the simplification function outside the verification algorithm - so once the concept expression is reduced to normal form when verifying property A, there is no need to do it again for other properties. This produced significant time savings overall.

Preprocessing the concept expressions in Pellet took 10.6 seconds, while verification of the properties took .420 seconds on average. For comparison, Margrave's loading and preprocessing time was 0.9 seconds, and the properties were verified in less than a millisecond. Pellet's performance is worse since it is optimized for reasoning about a more expressive logic than Margrave. Nonetheless, considering that verification time was still under half a second, we consider the results to be suitable for practical purposes.

Next, to evaluate the overhead of integrating the access control policy with a domain ontology, we tested Continue with three ontologies with various sizes. For the evaluation, we simply added subclass relationship between the concepts in the access policy and random concepts from the domain ontology. The goal was to simulate reasoning about policies when their concepts have rich domain descriptions expressed in DL. Results are in Table 4. Overhead is noticeable only in the cases of Galen, which is a quite large ontology.

Ontology	Ontology Info			Performance	
Name	C	P	Classif.	Initial	Verif.
no-ont	0	0	0	10.295	0.420
koala	25	5	0.171	10.415	0.421
lubm	43	32	0.210	10.425	0.414
tambis	395	100	1.713	10.876	0.441
galen	3097	413	49.251	10.385	1.088

Table 4: Reasoning about policies with domain ontologies. *C*, *P*, *Classif.* stand for number of DL concepts, roles and classification time respectively. We added subclass relationships connecting *every* concept in the policy with a random concept from the ontology. The first row represents the case without a domain ontology.

8. DISCUSSION AND FUTURE WORK

For future work, we intend to extend our coverage of XACML even further: adding `Only-one-applicable` as a combining algorithm and handling more attribute functions using specific datatype reasoners are some of our short term goals. We believe that `Only-one-applicable` can be handled simply by adding additional defeasible rules in our framework (we will need to extend the *map* function as well).

As for datatype reasoning, user-defined datatypes are part of OWL 1.1 [8] and are already implemented in Pellet. Currently, there is support for datetime and integer user-defined XML Schema Datatypes and common datatype facets (minInclusive, maxInclusive). We plan to add more expressive datetime datatype support to allow for policies with periodicity constraints. For example: Permit access only on every other Friday form 5PM-6PM starting from today, *unless* it is a national holiday.

9. RELATED WORK

Logic programming systems seem to be a popular choice for formalization of access control policy languages, which is not surprising considering logic programming is a mature research area, with efficient implementations, and rules being the most natural way to model access control policies. We used DL as the basis for the formalization instead because of the correspondence of policy analysis services to DL reasoning services, which allowed us to provide a variety of policy analysis services and leverage the availability of off-the-shelf DL reasoners optimized for these services. Also, we have provided a set of services that, to the best of our knowledge, has not been offered by rule-based policy systems.

Moving to DL based systems, Zhao et al [19] present a formalization of RBAC based on the description logic $\mathcal{ALCQ}$. They also show how RBAC policy constraints (separation of duty, security role hierarchies) can be captured with this logic. We generalize their approach by formalizing a more expressive access control language (XACML uses overridding algorithsm which are not covered by their approach) using a more expressive description logic ($\mathcal{SHOIN}$).

Massacci [14] formalizes RBAC using multi modal logic and presents a decision method based on analytic tableaux. Because he is using tableau-based algorithms, he is able to provide services similar to ours: logical consequence, model generation and consistency checking of policies. Again in this case, policy combining algorithms are not taken into account, so it is not applicable to XACML.

Hughes et al. [11] propose a framework for automated verification of access control policies based on relational First-Order Logic. They introduce a formal model for systematically specifying access to resources, and show that the access control policies in XACML can be translated to a simple form which partitions the input domain to four classes: permit, deny, error, and not-applicable. The authors show how to automatically verify policies using an existing automated analysis tool, Alloy [12]. Because using the first-order constructs of Alloy to model XACML policies is prohibitively expensive (in terms of performance), the authors use only the propositional constructs. However, it is unclear from their results whether it is feasible for larger policies. In addition, the results of policy analysis are an internal Alloy representation that can only be explored with Alloy's visualization tools.

In [18] the authors present a model-checking algorithm which can be used to evaluate access control policies, and a tool which implements it. The evaluation includes not only assessing whether the policies give legitimate users enough permissions to reach their goals, but also checking whether the policies prevent intruders from reaching their malicious goals. Policies of the access control system and goals of agents are described in the access control description and specification language *RW* [9].

Finally, in terms of services offerred, Margrave [6] is the tool most similar to ours. Margrave is a software suite for analyzing role-based access-control policies. It includes a verifier that analyzes policies written in XACML, translating them into a binary decision-diagram to answer queries. It also provides semantic differencing information between versions of policies. On one hand, by using description logics to analyze policies, we sacrifice some of the performance compared to Margrave. Although, results show that our approach is still practical. Also, we have not evaluated yet comprehensive change analysis, where *all* access requests that map to different access decisions are returned. On the other hand, by having DL-based formalization we are able to provide expressive descriptions of the subjects, resources and actions that are referred to in the policies, and offer support for XML Schema datatypes.

10. CONCLUSION

Understanding the effects and consequences of sets of access control policies has always been an issue for security officers, especially with expressive policy languages. In this paper, we addressed this issue for XACML by proposing a formalization based on a decidable fragment of FOL. We were able to provide a similar suite of analysis services such as propositional logic-based tools, while adding extra expressiveness by describing subjects, actions and resources using ontologies. We also showed how common policy constraints, such as role cardinality, separation of duty and role hierarchies can be easily captures by these logics. Finally, we demonstrated through empirical evaluation that off the shelf DL reasoners are practical as XACML analysis tools.

11. ACKNOWLEDGEMENTS

This work was supported in part by grants from Fujitsu, Lockheed Martin, NTT Corp., Kevric Corp., SAIC, the National Science Foundation, the National Geospatial-Intelligence Agency, DARPA, US Army Research Laboratory, and NIST.

The authors would like to thank Kathi Fisler, Christian Halaschek-Wiener, Yarden Katz and Taowei Wang and for all of their comments and feedback.

We would also like to thank the authors of [6] for translating the Continue policy example to XACML, generating the policy safety properties and making them publicly available.

12. REFERENCES

[1] Continue access control policy example., 2005. http://www.cs.brown.edu/research/plt/software/margrave/versions/01-01/examples/continue/.

[2] Xacml references, v1.65. http://docs.oasis-open.org/xacml/references/xacmlrefsv1.65.html, 2006.

[3] A. Anderson. Core and hierarchical role based access control (rbac) profile of xacml v2.0, February 2005.

[4] J. Bryans. Reasoning about xacml policies using csp. In *SWS '05: Proceedings of the 2005 workshop on Secure web services*, pages 28–35, New York, NY, USA, 2005. ACM Press.

[5] M. Dean and G. Schreiber. Owl web ontology language reference w3c recommendation., feb 2004.

[6] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz. Verification and change-impact analysis of access-control policies. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 196–205, 2005.

[7] S. Godik and T. Moses. Oasis extensible access control markup language (xacml) version 1.1. oasis committee specification, July 2003.

[8] B. C. Grau, I. Horrocks, B. Parsia, P. Patel-Schneider, and U. Sattler. Next steps for owl. In *OWL Experienced and Directions*, 2006.

[9] D. P. Guelev, M. Ryan, and P.-Y. Schobbens. Model-checking access control policies. In *ISC*, pages 219–230, 2004.

[10] I. Horrocks and U. Sattler. A tableaux decision procedure for SHOIQ. In *Proc. of the 19th Int. Joint Conf. on Artificial Intelligence (IJCAI 2005)*. Morgan Kaufman, 2005.

[11] G. Hughes and T. Bultan. Automated verification of access control policies (technical report). Technical Report 2004-22, Department of Computer Science, University of California, Santa Barbara, September 2004.

[12] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.

[13] V. Kolovski. Formalizing XACML Using Defeasible Description Logics. Technical Report TR-233-11, University of Maryland - College Park, 2006.

[14] F. Massacci. Reasoning about security: A logic and a decision method for role-based access control. In *ECSQARU-FAPR*, pages 421–435, 1997.

[15] B. Parsia and E. Sirin. Pellet: An OWL DL reasoner. In *Third International Semantic Web Conference - Poster*, 2004.

[16] K. Wang, D. Billington, J. Blee, and G. Antoniou. Combining description logic and defeasible logic for the semantic web. In *RuleML*, pages 170–181, 2004.

[17] WS-Policy. Web services policy framework (ws-policy). http://www-106.ibm.com/developerworks/library/specification/ws-polfram/.

[18] N. Zhang, M. D. Ryan, and D. Guelev. Evaluating access control policies through model checking. In *Eighth Information Security Conference (ISC05)*, 2005.

[19] C. Zhao, N. Heilili, S. Liu, and Z. Lin. Representation and reasoning on rbac: A description logic approach. In *ICTAC*, pages 381–393, 2005.