

# Formalizing XACML Using Defeasible Description Logics

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  
School 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. Previous attempts to analyze XACML policies either use propositional logic or full First-Order logic. In this paper, we present a formalization of XACML using Description Logics (DL). This formalization allows us to extend the subset of XACML supported by propositional logic-based analysis tools, and in addition we provide a new analysis service (policy redundancy). In addition, mapping XACML to description logics allows us to use off-the-shelf DL reasoners for XACML analysis tasks such as policy comparison, policy verification and querying. We provide empirical evaluation of a policy analysis tool that was implemented on top of open source reasoner Pellet.

## Categories and Subject Descriptors

I.2.2 [Artificial Intelligence]: Knowledge Representation Formalisms and Methods; I.2.4 [Artificial Intelligence]: Automatic Programming—*program verification*

## General Terms

Access Control, Verification, Knowledge Representation

## Keywords

XACML, access control policies, policy analysis, policy verification, Description Logics

## 1. INTRODUCTION

As the amount of sensitive information available on the Web grows, access control becomes a key issue. Over time, access control models have evolved from simple access control lists, through discretionary (DAC), mandatory (MAC) and role-based access control (RBAC) paradigms to declarative, rule-based languages that allows users to specify sophisticated policies.

The OASIS standard eXtensible Access Control Markup Language (XACML [10]) is a highly expressive access control language with significant deployment<sup>1</sup>. XACML enables the use of arbitrary attributes in policies, allows for expressing

<sup>1</sup>See [1] for a list of systems incorporating XACML.

'deny' policies and enables the use of hierarchical RBAC, among other things.

When using such an expressive language, managing and analyzing an access control policy becomes a concern: how can an administrator be certain that her policy covers all possible corner cases? One way to alleviate the situation is with testing, but testing is not exhaustive, and as policies grow in complexity, coming up with useful tests is more challenging.

Another approach that has emerged recently is to use logic and formal reasoning techniques for analysis and verification of policies. There have been a number of attempts to provide a formalization of XACML [13, 7, 19, 9] – unfortunately they either support a very small subset of the language, or they severely limit the analysis services offered.

Margrave [9] is an analysis tool that captures an interesting subset of XACML while providing quite useful analysis services. In addition to providing means to verifying and querying a single policy, it also allows for change impact analysis between two policies and querying and verifying changes between two policies. Margrave is restricted to a subset of XACML that is expressible using propositional logic, and implemented using binary decision diagrams (BDD), which makes the tool quite efficient. Unfortunately, restricting to propositional logic means that some useful features (such as role hierarchies and role cardinality constraints) are not covered. In addition, Margrave does not offer datatype support for XACML attribute values. In their evaluation, First-Order logic-based analysis tools were incapable of handling XACML policies of non-trivial sizes.

In this paper, we provide a formalization of XACML that explores the middle ground between propositional logic analysis tools (such as Margrave) and full First-Order logic XACML analysis tools (like Alloy [14]). With this we are able to extend the analysis services offered by Margrave in the following way:

1. We cover additional policy idioms, such as role hierarchies and role cardinality constraints, and provide some datatype support
2. We also introduce a new analysis service: policy *redundancy*, which means that the access decision of the policy is always overridden by some other policy higher-up the hierarchy.

We use Description Logics (DL) to provide a formalization of XACML. DL are a family of knowledge representation languages which are decidable subsets of First-Order logic

commonly used as the formal bases of object/class style ontology languages. Expressing policies in DL allows us both to define and effectively implement an array of policy analysis services for a fairly large subset of XACML. Our formalization also benefits from Description Logics being the basis for the Web Ontology Language (OWL [8]), which allows us a) to use off-the-shelf OWL reasoners as policy analysis tools and b) to potentially integrate ontology-based policy descriptions with XACML policies (a la [5]).

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 [16]. We performed preliminary evaluation of our tool on Margrave's test policy set. While slower than Margrave (which was expected), our tool finished all of the tests in a reasonable amount of time, thus exhibiting encouraging results (especially when compared to other First-Order logic XACML analyzers [14], which were not able to process the entire policy).

## 2. PRELIMINARIES

### 2.1 Overview of XACML

In this section we provide an overview of XACML (version 2.0), 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.

XACML [10] is an XML-based language for describing access control policies. 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 simply 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  $\geq 18$ , 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-type* 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-type"
          DataType="...#string"/>
      </ActionMatch>
    </Actions>
  </Target>
</Rule>
```

#### 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 attribute functions and Only-One-Applicable overriding algorithm) – this is part of our ongoing work.

## 2.2 Description Logics

Because our formalization is based on Description Logics, in this section we briefly present the syntax and semantics of the logic *SHOIN*, which provides all of the expressiveness that we need.

In DL, the domain of interest is modeled using individuals, concepts and roles, denoting objects of the domain, unary predicates and binary predicates respectively. Atomic concepts (C) and atomic roles (R) are elementary descriptions

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.

and complex ones can be built on top of them. Concepts are defined inductively using the following grammar:

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

where  $A$  is an atomic concept,  $a$  is an individual,  $C_{(i)}$  a *SHOIN* concept,  $R$  a role,  $S$  a *simple* role<sup>2</sup> and  $\bowtie \in \{\leq, \geq\}$ . We write  $\top$  and  $\perp$  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  $\mathbf{T}$  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  $\mathcal{I} = (\mathcal{W}, \cdot^{\mathcal{I}})$ , where  $\mathcal{W}$  is a non-empty set, called the *domain* of the interpretation, and  $\cdot^{\mathcal{I}}$  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  $\mathcal{W} \times \mathcal{W}$  and to each individual  $a$  an element of  $\mathcal{W}$ . The interpretation function is extended to complex roles and concepts as given in [12].

The satisfaction of a *SHOIN* axiom  $\alpha$  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)  $\mathcal{I} \models \text{Trans}(R)$  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  $\mathbf{R}$  (respectively of the TBox  $\mathbf{T}$ ) if it satisfies all the axioms in  $\mathbf{R}$  (respectively  $\mathbf{T}$ ).  $\mathcal{I}$  is a model of  $\mathbf{K} = (\mathbf{T}, \mathbf{R})$ , denoted by  $\mathcal{I} \models \mathbf{K}$ , iff  $\mathcal{I}$  is a model of  $\mathbf{T}$  and  $\mathbf{R}$ .

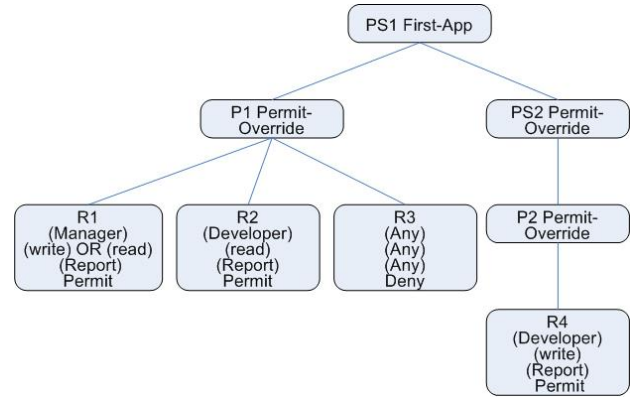
For the purposes of our work, it is important to discuss two basic reasoning services offered by DL: satisfiability and subsumption. Determining satisfiability of a concept  $C$  in a KB  $\mathbf{K}$  amounts to a check whether  $\mathbf{K}$  admits a model in which the interpretation of  $C$  is nonempty.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

### 3. RUNNING EXAMPLE

In this section we will introduce an example which will be used throughout this paper to illustrate the services that our formalization provides. In this toy example, initially there are two roles, *Manager* and *Developer*; one resource: *Report*;

<sup>2</sup>See [12] for a precise definition of simple roles

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 property we want to check is whether *Developers* can *write* to *Reports*. Running Pellet on this property 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 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. 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
  
```

Seems 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. In order 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 example we can also illustrate a new analysis service that we provide: policy *redundancy*. To understand our motivation for this service, consider a large policy system, which keeps getting requests that need to be evaluated against all of the rules in the system. Evaluation starts on the lowest level, and the access decisions are propagated towards the root policy set and combined using the rule- and policy-combining algorithms. There may exist some rules where the decision simply *does not matter*, i.e, their access decision is always overridden by a higher-priority policy.

An example of a redundant rule is  $R_4$  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, rule  $R_4$  can be dropped without any consequences to the access decisions of the policy. This elimination of unnecessary (redundant) 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 roles at a time. Combinations of restrictions are allowed as well: a Developer cannot perform more than three actions at a time.
- **Policy Comparison** - For two policies (or policy sets)  $P_1$  and  $P_2$  check if whenever  $P_1$  yields a decision  $\alpha$ ,  $P_2$  will yield  $\alpha$ , 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 be a request s.t. both policies yield the same 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

We formalize the four main policy elements in XACML: Rules, Requests, Policies and Policy Sets. Considering the obvious similarities between some of them (and for better presentation), we have grouped them in two: Rules and Requests in the first group, and Policies and Policy Sets in the second.

While the Target element of Rules and Requests can be mapped to a DL class 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. For this reason, to capture the semantics of the combining algorithms we use defeasible

description logics (DDL [17]), which is a formalism that allows for expressing defeasible rules on top of description logics.

The subset of *DDL* that we define, termed *DDL<sup>-</sup>*, is expressive enough for our purposes and at the same time has the same computational complexity as the underlying description logics. More importantly, reasoning services in *DDL<sup>-</sup>* can be reduced to description logic reasoning, which allows us to use DL reasoners to analyze XACML policies. Following we give some basic definitions of *DDL<sup>-</sup>* terms.

**DEFINITION 1.** A rule  $r$  consists of an antecedent  $Pre(r)$  and a consequent  $Con(r)$ . Given a set  $\mathcal{R}$  of rules, we denote the set of all strict rules (that cannot be overridden) in  $\mathcal{R}$  by  $\mathcal{R}_s$  and the set of defeasible rules in  $\mathcal{R}$  by  $\mathcal{R}_d$ .

A set of rules  $\mathcal{R}$  can contain both *strict* and *defeasible* rules. Following we give a definition of strict rules.

**DEFINITION 2.** For each rule  $a \in \mathcal{R}$ , let  $P, P_{par}$  correspond to the policy element and its parent in the XACML document.  $a$  is a strict rule in the following cases:

1. if  $a$  is a Permit rule, and the combining algorithm of  $P_{par}$  is Permit-Overrides
2. if  $a$  is a Deny rule, 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  $P$  is First-Applicable

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

**DEFINITION 3.** A *DDL<sup>-</sup>* theory  $\mathcal{D}$  is a tuple  $(\Sigma, \mathcal{R}, >, \mathcal{L})$  where  $\Sigma$  is a DL knowledge base,  $\mathcal{R}$  a finite set of rules,  $>$  a superiority relation on  $\mathcal{R}$ , and  $\mathcal{L}$  a set of literals used in the rules of  $\mathcal{R}$ .

For each policy or policy set  $P$ , we create two literals, which we call *effect-literals*:  $Permit-P, Deny-P \in \mathcal{L}$ . If the theory derives that  $Permit-P(r)$  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  $Permit-id(P)$  we abuse notation and simply write  $Permit-P$ . Note here that only effect-literals are allowed in heads of rules in  $\mathcal{R}$ .

### 4.1 Mapping XACML Requests and Rules

A XACML **Rule** is translated to a rule in  $\mathcal{R}$ . The **Target** element is translated to a DL class 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 conclude  $r_1$  and  $r_2$  are conflicting.

The full mapping of the **Target** element to a DL class expression is given in Table 2. In it, Subjects, Resources and Actions are treated the same. Attributes-value pairs are mapped to existential restrictions – for example (*role-type Developer*) would be mapped to  $\exists role-type.\{Developer\}$ . We also allow for free combination of attribute-value pairs using propositional operators. Note here that we enforce a

Syntax	$\pi$
$R ::= (\text{Rule } T \text{ Effect})$	$\pi(T) \mapsto \pi(\text{Effect})$
$\text{Effect} ::= \text{Permit} \mid \text{Deny}$	$\text{Permit-}P \mid \text{Deny-}P \text{ (} P \text{ is parent policy)}$
$T ::= ((\text{Sub}) (\text{Act}) (\text{Res}))$	$\pi(\text{Sub}) \sqcap \pi(\text{Act}) \sqcap \pi(\text{Res})$
$\text{Sub} \mid \text{Act} \mid \text{Res} ::= \text{Any} \mid \text{Fcn}$	$\top \mid \pi(\text{Fcn})$
$\text{Fcn} ::= \text{AV} \mid \text{Fcn} \cap \text{Fcn} \mid \text{Fcn} \cup \text{Fcn} \mid \neg \text{Fcn}$	$\pi(\text{AV}) \mid \pi(\text{Fcn}) \sqcap \pi(\text{Fcn}) \mid \pi(\text{Fcn}) \sqcup \pi(\text{Fcn}) \mid \neg \pi(\text{Fcn})$
$\text{AV} ::= (\text{attr-id attr-val})$	$\exists \pi(\text{attr-id}).\pi(\text{attr-val})$
$\text{attr-id}$	DL property corresponding to $\text{attr-id}$
$\text{attr-value}$	DL individual corresponding to $\text{attr-val}$

**Table 2: Mapping Rule to a description logics class expression**

one-to-one mapping from attribute names and values used in the XACML policy to their corresponding properties and individuals in  $\Sigma$  (we simply create individuals or properties with the same name as the attribute or value).

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

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

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

$$R_3 : \top \mapsto \text{Deny-}P_1$$

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

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  $\Sigma \models \pi(T)(r)$  (equivalent to instance checking in description logics).

## 4.2 Mapping XACML Policies and Policy Sets

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

1. For each Deny rule  $r : \text{Target Deny } P$ , add an axiom to  $R$ ,

$$R = R \cup \{\pi(\text{Target}) \mapsto \text{Deny-}P\}$$

2. For each Permit rule  $r : \text{Target Permit } P$ , add an axiom to  $R$ ,

$$R = R \cup \{\pi(\text{Target}) \mapsto \text{Permit-}P\}$$

3. For each policy element  $P$  and parent policy element  $PS$  introduce the following axioms:

$$\text{Permit-}P \mapsto \text{Permit-}PS$$

$$\text{Deny-}P \mapsto \text{Deny-}PS$$

A Policy or a PolicySet can also have a Target element. However, we can propagate the constraints specified in Target down to Targets of its children. In this manner, we propagate the constraints all the way 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 Rules descendants.

## 4.3 Semantics of $DDL^-$

Following we provide procedural semantics for the defeasible theory (derived from [17]). A derivation (proof) is a finite sequence  $P = (P(1), \dots, P(n))$  of literals that belong to  $\mathcal{L}$ . Conclusion in a  $DDL^-$  theory is defined as 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  $P = (P(1), \dots, P(n))$  s.t.  $+\alpha \in P$ ; we say that  $\alpha$  is *provable* in  $\mathcal{D}$ .

**If  $P(i+1) = +l$  then either**

(1)  $\exists r \in R_s$  s.t.  
 $l \in \text{Con}(r)$  **and**  
 $\Sigma \models \text{Pre}(r)$  **or**  $\text{Pre}(r) \in P(1..i)$

(2) **or**  $\exists r \in R_d$  s.t.  
 $l \in \text{Con}(r)$  **and**  
 $\Sigma \models \text{Pre}(r)$  **or**  $\text{Pre}(r) \in P(1..i)$

$\forall q \in (R_d \cup R_s)$  s.t.  $\text{Con}(q) \sqsubseteq \neg \text{Con}(r)$  either  
 $\Sigma \not\models \text{Pre}(q)$  **and**  $\text{Pre}(q) \notin P(1..i)$   
**or**  $r > q$

**Figure 2: Derivation Procedure**

Figure 2 gives the procedure 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.

**If  $P(i+1) = -l$  then**

(1)  $\forall r \in R_s$   
 $l \notin \text{Con}(r)$  **or**  
 $\Sigma \not\models \text{Pre}(r)$  **and**  $\text{Pre}(r) \notin P(1..i)$

(2) **and**  $\forall r \in R_d$  s.t.  
 $l \notin \text{Con}(r)$  **or**  
 $\Sigma \not\models \text{Pre}(r)$  **and**  $\text{Pre}(r) \notin P(1..i)$

$\exists q \in (R_d \cup R_s)$  s.t.  $\text{Con}(q) \sqsubseteq \neg \text{Con}(r)$   
 $\Sigma \models \text{Pre}(q)$  **or**  $\text{Pre}(q) \in \bar{P}(1..i)$   
**and**  $r < q$

**Figure 3: Derivation Procedure**

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 [17]). The procedural semantics is given shown in Figure 3.

$DDL^-$  is subsumed by DDL [17], i.e., for a literal  $l$ , a

$DDL^-$  theory  $\mathcal{D}^-$  and a DDL theory  $\mathcal{D}$ , if  $\mathcal{D}^- \vdash +l$  then  $\mathcal{D} \vdash +\delta l$  and if  $\mathcal{D}^- \vdash -l$  then  $\mathcal{D} \vdash -\Delta l$ . This result is due to  $DDL^-$  being a specialization of  $DDL$  where

- no dl-literals are allowed in heads of rules
- only one literal is allowed in heads of rules
- there is no distinction between defeasible and definite derivation of facts

Thus, we inherit the nice complexity properties of DDL ( $DDL^-$  has the same complexity as the underlying description logic).

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

#### 4.4 Reducing $DDL^-$ Literal Derivability to Description Logic Concept Satisfiability

This section provides an important result: an algorithm to reduce literal provability in  $DDL^-$  to concept satisfiability in Description Logics. In particular, for an effect-literal  $\alpha$  we will show how to generate a DL concept expression  $A$  in  $\Sigma$  s.t.  $\alpha$  is derivable in  $\mathcal{D}$  iff  $A$  is satisfiable in  $\Sigma$ .

First we will illustrate the intuition with a simple example. In the following  $DDL^-$  theory:  $r_1 : \{A \mapsto Permit-1\}, r_2 : \{B \mapsto Deny-1\}, r_1 < r_2$  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 4.** *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,*

$$map(a) = \sqcup (map(C) \sqcap \neg(\sqcup map(J))) \text{ where}$$

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

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

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

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

**Proof Sketch** Proof by structural induction, exploiting the tree-like structure of a XACML policy document (rules in different levels of tree do not conflict with each other).

**Base case** is when  $\alpha$  is a XACML Policy (hence it is composed only of Rules).

$\leftarrow$  direction first. For a Policy  $\alpha$ ,  $Permit-\alpha$  is derivable only if exists at least one Rule  $C$  s.t.  $\{C : r \mapsto Permit-\alpha\} \in \mathcal{D}$  and  $map(C)$  is satisfiable. Thus, we need to construct a disjunction of rules who have the literal  $Permit-\alpha$  as a conclusion and check whether the expression is satisfiable. Constructing this expression, we get

$$map(\alpha) = \sqcup map(C) \text{ where } \{C \mapsto Permit - P\} \in \mathcal{D}$$

However, the above expression is not complete – it is possible for it to be satisfiable, and  $Permit-\alpha$  still not to be derivable in  $\mathcal{D}$ . This can happen because there might be other rules that can fire in  $\alpha$ , producing a  $Deny$  decision that could potentially override the  $Permit$  decision. Thus, for every rule  $C$  that yields a  $Permit-\alpha$ , we need to take into account its corresponding overriding rules  $J$ . If it is true that  $map(C) \sqsubseteq \sqcup map(J)$  where each  $J$  is an overriding rule, then the literal  $Permit-\alpha$  is not derivable. This can be rewritten as:

$$map(a) = \sqcup (map(C) \sqcap \neg(\sqcup map(J))) \text{ where}$$

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

Which is exactly the definition of the map function.

$\rightarrow$  If the expression  $map(\alpha)$  is satisfiable, that means there exists an access request  $i$  s.t. a)  $i$  satisfies the prerequisite  $map(C)$  of some rule  $R : C \mapsto Permit-\alpha$  and b)  $i$  is a member of the *negated* prerequisites of all of the overriding rules. In other words, there exist a request  $i$  s.t. a **Permit** rule will fire and all of the overriding **Deny** rules won't, thus  $Permit-\alpha$  is derivable.

**General case.** In this case,  $\alpha$  is a **PolicySet** and can contain **Policies** and/or **PolicySets**. The inductive assumption is that the hypothesis holds for all permit-literals  $\beta$  s.t.  $\{Permit - \beta \mapsto Permit - \alpha\} \in \mathcal{D}$ .

$\leftarrow$   $Permit-\alpha$  is derivable only if there exists at least one rule whose prerequisite is derivable and not overridden. If a prerequisite  $C$  is derivable, we know from the inductive assumption that  $map(C)$  is satisfiable. We have to make it a disjunction because if  $Permit-\alpha$  is derivable, then at least one of the  $map(C)$  expressions has to be satisfiable. As in the base case, we have to be careful with the overriding rules, so we include the negation of all of the rules  $J$  which give a **Deny** access decision, and are of higher priority than  $C$ . To derive  $Permit-\alpha$  from a prerequisite  $C$ , there has to exist a request  $i$  s.t.  $\mathcal{D} \vdash C(i)$  and  $\mathcal{D} \not\vdash J(i)$  where  $J$  is a prerequisite of a rule that overrides  $C$ . To rephrase it in terms of  $map$ ; we know that if  $C(i)$  is derivable for some request  $i$  then  $map(C)$  is satisfiable, and if  $J(i)$  is not derivable for some request then  $\neg map(J)$  is satisfiable. Thus, we ask whether there can be a request  $i$  s.t. both  $map(C)$  and  $\neg map(J)$  are satisfiable:

$$map(C) \sqcap \neg(\sqcup map(J))$$

which corresponds exactly to one disjunct in  $map(Permit-\alpha)$ . The disjunction matches what the definition of a  $map$  function for a general effect-literal  $\alpha$ .

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

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

→ If the expression  $map(\alpha)$  is satisfiable, that means there exists an access request  $i$  s.t. a)  $i$  satisfies the prerequisite  $map(C)$  of some Policy or PolicySet  $P : \text{Permit-}\beta \mapsto \text{Permit-}\alpha$  and b)  $i$  is a member of the *negated* prerequisites of all of the overriding rules. In other words, there exist a request  $i$  s.t. we will derive a  $\text{Permit-}\alpha$  and will not derive any of the overriding  $\text{Deny-}\alpha$  literals, thus  $\text{Permit-}\alpha$  is derivable.

□

PROPOSITION 1. For a  $DDL^-$  theory  $\mathcal{D} = (\Sigma, \mathcal{R}, >, \mathcal{L})$ , and a request  $i$ ,

- if  $\Sigma \models i : map(\alpha)$ , then  $\mathcal{D} \vdash \alpha(i)$ .
- if  $\Sigma \models i : \neg map(\alpha)$ , then  $\mathcal{D} \not\vdash \alpha(i)$ .

Note that in above proposition, the opposite does not hold. Consider a counter example:

$$\mathcal{D} = (\Sigma, \mathcal{R}, \{r_1 > r_2\}, \{\text{Permit-}P, \text{Deny-}P\})$$

where

$$\mathcal{R} = \{r_1 : A \mapsto \text{Permit-}P, r_2 : B \mapsto \text{Deny-}P\}$$

In this case,  $map(\text{Deny-}P) = B \sqcap \neg A$ . If the request  $i$  is of type  $B \sqcap \neg A$ , then we will derive  $\text{Deny-}P(i)$ . However, it is not true that if we derive  $\text{Deny-}P(i)$  for a request that it has to be of type  $B \sqcap \neg A$ . As a counter example,  $i : B$  or  $i : B \sqcap C$  would still derive  $\text{Deny-}P(i)$ .

## 5. ANALYSIS SERVICES

Before discussing analysis services in more detail, first we show how it is possible to express common policy idioms using our formalization:

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:

$$\exists \text{role-type.}\{LeadDeveloper\} \sqsubseteq \exists \text{role-type.}\{Developer\}$$

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

$$\exists \text{role-type.}\{A\} \sqsubseteq \neg \exists \text{role-type.}\{B\}$$

3. *Cardinality constraints* can be expressed on any given attribute. To state that a user cannot be a member of more than  $k$  roles at a time we can write:

$$\geq k \text{ role-type.}\top \mapsto \perp$$

We can even specify maximum number of users that a 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:

$$\geq k \text{ role-type}^-. \top \mapsto \perp$$

We can use this policy idioms to either constrain individual XACML Rules (only fire Rule if user is member of at most 2 roles), or we can also use them to express global properties for all policies (under no circumstance give access if user is member of more than 2 roles).

### 5.1 Policy Comparison

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

To determine whether  $P_1 \sqsubseteq P_2$ , we try to build a request  $i$  s.t.  $\mathcal{D} \vdash \text{Permit-}P_1(i)$  and  $\mathcal{D} \not\vdash \text{Permit-}P_2(i)$ . If it is possible to build a request s.t.  $\text{Permit-}P_1(i)$  was derived, and  $\text{Permit-}P_2(i)$  was not derived, then the policies  $P_1 \not\sqsubseteq$

$P_2$ . Translating to DL expressions, this reduces to checking whether the concept  $map(Permit-P_1) \sqcap \neg map(Permit-P_2)$  is satisfiable. If the concept is satisfiable, then it is possible to have such request.

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

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

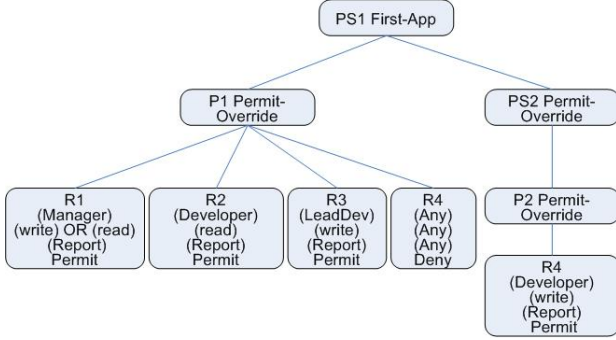


Figure 4: Updated Policy (with LeadDev role)

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

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

Subsumption holds only both of the following hold:

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

Pellet reports the first statement is true, which is obvious from looking at the rules themselves (the rule that we added in  $PS_{new}$  is a permit-rule). 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 inspect whether the expression  $A \sqcap \neg B$  is unsatisfiable (which means a model that admits that concept expression cannot be built). If the expression is satisfiable, then no subsumption holds and we can extract a counter example directly from the model built by the reasoner. In this example, we get a number of 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*:

$$\exists role-type.\{LeadDev\} \sqsubseteq \neg \exists role-type.\{Developer\}$$

We can generalize the technique used for policy subsumption to policy *comparison*. For two policies  $P_1$  and  $P_2$ , we first specify what kind of decision 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 decision. If the expression is satisfiable, then we can also get *all* consistent models that it admits – this will correspond to all possible requests where  $P_1$  returns Deny and  $P_2$  returns permit.

Generating all requests for which the policies yield different decisions involves saturation of the tableau, i.e., computing all possible models that admit the concept expression. While DL reasoners are not particularly optimized for computation of all models, since we are only saturating one concept expression at a time, we believe (and evaluation shows) that our approach suffices for moderately sized policies.

## 5.2 Policy Redundancy

Another service we provide is determining redundant rules. 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. In other words, 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. To check whether a rule  $r : T \mapsto \alpha$  is redundant, we use the algorithm in Figure 5.

```
function isRedundant(r,D)
Input:
rule r : T ↦ α
DDL- theory D = (Σ, R, >, L)
p = α
while true do
  J = ⋃ map(j) where
    {j ↦ β} ∈ R and conflicts(β, α) and
    {j ↦ β} > {p ↦ α}
  if T ⊆ J return false
  if ∃ pnew s.t. {p ↦ pnew} ∈ R
    p = pnew
  else
    break
return true
```

Figure 5: Derivation Procedure

The function starts from the Rule  $r$  and works its way to the root policy element, checking if there can exist a request s.t. the  $r$  yielded an access decision and none of the overriding, higher priority rules gave an conflicting decision. If it is always the case that whenever  $r$  fires, it will be overridden by a combination of higher priority rules and/or policies, then  $r$  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.

## 5.3 Policy Verification

We allow for specification and formal verification of properties of policies. The properties that we admit are limited by the expressiveness of description logics. We admit policy properties of the same form as rules, i.e, they have a DL class expression as the target and a Deny or Permit in the head of rule. To check whether a policy  $P$  satisfies a property, first

we compute the  $map(Permit - P), map(Deny - P)$  using the techniques described previously. Then, we try to build 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 unsatisfiable, then we have formally verified the policy 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 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:

$$\exists role\text{-}type. \neg \{Manager\} \sqcap \exists age. \leq_{21} \sqcap > 1 \text{ action}\text{-}type. \top \longmapsto Deny$$

## 6. IMPLEMENTATION AND EVALUATION

We have implemented a prototype XACML analysis tool as an extension to description logics reasoner Pellet. As a test case, we used the same access control policy that [9] used ( a conference paper submission application Continue). The authors of Margrave translated the Continue access control policy to XACML and published it at [2]. We used this policy file to evaluate the correctness and performance of our implementation.

Parsing the Continue policy file (which contains 25 policies) into Pellet took 700ms. As a first test, we checked each policy pair for subsumption (for both Permit and Deny access decisions); performing a total of  $25*24*2 = 1200$  satisfiability tests in Pellet. The total time for all tests was 220 seconds, which amounts to 180ms per individual subsumption check.

We also performed formal verification of properties described by the Margrave authors (following is an incomplete list):

- $P_1$  : If a subject is not a pc-chair or admin, then he may not set the meeting flag.
- $P_2$  : If the subject role attribute is empty and the resource-class is not conferenceInfo\_rc, do not allow permit.
- $P_3$  : If the subject is a pc-chair who called a meeting, then he can read any part of a review if the meeting is about it.

All of our verification results coincided with the results of Margrave, which was as expected. On average, it took 17 seconds to verify a property against the policy. As expected, our performance is much slower than Margrave – it is due to the fact that we use a tableau reasoner optimized for description logics whereas Margrave is restricted to propositional logic (and is implemented using binary decision diagrams). By using a DL reasoner we essentially sacrificed some of the performance in order to have additional features. For example, we also tested the access control policy against this property:

$P_{new}$ : A subject cannot be a member of more than two roles at a time.

Verifying  $P_{new}$  took 19 seconds. Considering that we plan for our tool to be used offline (during policy development

time), we believe that the runtime performance is reasonable.

## 7. 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 conjecture 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 [6] and are already implemented in Pellet. Currently, we have implemented datetime and integer user-defined XML Schema Datatypes and provide support for common datatype facets (minInclusive, maxInclusive). We plan to add much more expressive date time datatype support to allow for policies such as: Permit access only on every other Friday from 5PM-6PM starting from today, *unless* it is a national holiday.

Recently we proposed a mapping of WS-Policy [18] to OWL-DL which allowed use to provide a similar suite of analysis services for the WS-Policy framework. Unfortunately, much of the policy information in WS-Policy is specified in using domain-specific policy assertion languages, which do not always have clear semantics and are difficult to analyze. There has been a proposal [4] recently to use XACML to specify these policy assertions in a domain-independent manner. As part of our future work, we plan to develop a tool that will cover both WS-Policy and XACML, essentially providing us with analysis support for WS-Policy assertions thereby covering a practical subset of the WS-Policy framework.

In addition, even though our formalization also allows for policy querying, we have not implemented the service yet. Initially, we intend to allow two types of queries:

- Querying access requests. Similar to Margrave, we plan to allow users to search through results of policy comparison or verification. The output of policy comparison is a set of models, where each model represents a possible *access request*. For a query  $q$ , we can go through every model  $m$  and check whether  $m \models q$ .
- Querying XACML rules based on attribute values we are interested in – output is given as a list of XACML Rules that match the query expression.

## 8. 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 being a mature research area, with efficient query algorithms and rules being the most natural way to model access control policies. We chose to formalize XACML using description logics instead because we believe that the suite of analysis services that we provide with DL cannot be matched by logic programming systems. For example (as is also stated in [9]), fine-grained policy comparison cannot be provided by logic programming systems.

Moving to DL based systems, Zhao et al [20] present a formalization of RBAC based on the description logic  $\mathcal{ALCQ}$ . They also show how RBAC policy constraints (separation

of duty, role hierarchies) can be captured with this logic. We generalize their approach by formalizing a more expressive access control language (XACML uses overriding algorithms which are not covered by their approach) and a more expressive description logics (*SHOIN*).

Massacci [15] 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. [13] 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 [14]. 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, and cannot be queried or processed in more detail.

In [19] 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* [11].

The tool most similar to ours (and one which inspired our work) is Margrave [9]: a software suite for analyzing role-based access-control policies. Margrave 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. We extend the subset of XACML that Margrave covers with role hierarchies, attribute value datatypes and cardinality constraints, and in addition provide a new reasoning service (policy redundancy).

## 9. CONCLUSION

In this paper, we provided a formalization of a subset of XACML and described how we can use automated reasoning to cover a suite of policy analysis services. We were inspired by Margrave, which uses binary decision diagrams to represent and reason about a subset of XACML. In this paper, by using a decidable subset of first order logic we were able to analyze a larger subset of XACML, and also allow users to specify policy idioms such as *role hierarchies* and *cardinality constraints*. In addition, we presented a new reasoning service (policy redundancy) which could provide hints for optimizing a XACML policy engine. The preliminary evaluation, while showing that our tool performs slower than Margrave ( which was expected since we are using a more

expressive logic), still shows that our approach is practically feasible.

## 10. 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 Christian Halaschek, Taowei Wang and Yarden Katz for all of their contributions to this work.

## 11. REFERENCES

- [1] Xacml references, v1.65. <http://docs.oasis-open.org/xacml/references/xacmlrefsv1.65.html>.
- [2] Margrave continue example. available at <http://www.cs.brown.edu/research/plt/software/margrave/versi01/examples/continue/>, 2005.
- [3] A. Anderson. Core and hierarchical role based access control (rbac) profile of xacml v2.0, February 2005.
- [4] A. Anderson. Domain-independent, composable web services policy assertions. In *7th International IEEE Workshop on Policies for Distributed Systems and Networks*, 2006.
- [5] C. A. Ardagna, E. Damiani, S. D. C. di Vimercati, C. Fugazza, and P. Samarati. Offline expansion of xacml policies based on p3p metadata. In *ICWE*, pages 363–374, 2005.
- [6] B. P. P.-S. B. Cuenca Grau, I. Horrocks and U. Sattler. Next steps for owl. In *OWL Experienced and Directions*, 2006.
- [7] 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.
- [8] M. Dean, D. Connolly, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. Web Ontology Language (OWL) Reference Version 1.0. W3C Working Draft 12 November 2002 <http://www.w3.org/TR/2002/WD-owl-ref-20021112/>.
- [9] 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.
- [10] S. Godik and T. Moses. Oasis extensible access control markup language (xacml) version 1.1. oasis committee specification, July 2003.
- [11] D. P. Guelev, M. Ryan, and P.-Y. Schobbens. Model-checking access control policies. In *ISC*, pages 219–230, 2004.
- [12] 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.
- [13] 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.

- [14] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.
- [15] F. Massacci. Reasoning about security: A logic and a decision method for role-based access control. In *ECSQARU-FAPR*, pages 421–435, 1997.
- [16] Pellet. Pellet - owl dl reasoner, 2003.  
<http://www.mindswap.org/2003/pellet>.
- [17] 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.
- [18] WS-Policy. Web services policy framework (ws-policy). <http://www-106.ibm.com/developerworks/library/specification/ws-polfram/>.
- [19] N. Zhang, M. D. Ryan, and D. Guelev. Evaluating access control policies through model checking. In *Eighth Information Security Conference (ISC05)*, 2005.
- [20] C. Zhao, N. Heilili, S. Liu, and Z. Lin. Representation and reasoning on rbac: A description logic approach. In *ICTAC*, pages 381–393, 2005.