

XACML Policy Analysis Using Description Logics

VLADIMIR KOLOVSKI

University of Maryland-College Park

JAMES HENDLER

Rensselaer Polytechnic Institute

XACML is a standardized, expressive and increasingly popular language for writing access control policies about distributed resources. XACML enables the use of arbitrary attributes in policies, allows for expressing negative authorization, conflict resolution algorithms and hierarchical Role Based Access Control, among other things. Due to its rich expressiveness, the language has proved difficult to analyze in automated fashion.

In this paper, we present a logic-based approach to XACML policy engineering that provides a comprehensive set of automated analysis services (policy verification, policy comparison, redundancy checking) for XACML. Our approach maps a large fragment of XACML (including core XACML, XMLSchema datatypes and the Administrative Policy profile) to a decidable subset of First Order Logic called Description Logics (DL), and uses DL reasoners to provide analysis services. We provide these services by using software engineering metaphors (e.g., verification is presented as enhanced unit testing), effectively hiding the details of the background logic formalism and the XACML mapping.

We also show how our analysis framework can cover XACML policies extended with rich ontology-based models representing subject and resource information. Finally, we provide empirical evaluation of our policy analysis tool against propositional logic XACML analyzers.

Categories and Subject Descriptors: D.4.6 [Software]: Operating Systems—*Security And Protection*; 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

Additional Key Words and Phrases: XACML, access control, policy analysis, policy verification, description logics

1. INTRODUCTION

The eXtensible Access Control Markup Language (XACML [Rissanen 2007]) is a standardized, expressive and increasingly popular¹ language for writing access control policies about distributed resources. XACML enables the use of arbitrary attributes in policies, allows for expressing negative authorization, conflict resolution algorithms and hierarchical Role Based Access Control, among other things. The language specifies a processing model for checking policy compliance at run time. However, the standard does not include support for development time activities

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

Author's address: Department of Computer Science, University of Maryland, College Park, MD 20740, USA; email: kolovski@cs.umd.edu , hendler@cs.rpi.edu

Preliminary version of this paper appears in the *Proceedings of the 15th International World Wide Web Conference (WWW 2007)* [Kolovski et al. 2007]

such as testing, verifying or debugging large sets of complex policies.

Due to the expressiveness of XACML and its lack of "compile time" support, it is non-trivial for a policy developer to understand the overall effect and consequences of the XACML policies he/she writes. Even arguably the most important feature in access control - checking that the policy will not result in leakage of permissions to an unintended or unauthorized principal, i.e., *safety* - is 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 possible 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 concerns, there has been research on security analysis for access control [Li et al. 2003; Li and Tripunitara 2006] that uses the notions of *states* of policy systems and *transitions* (for example, adding a role, or changing a permission) that alter those states. Then, usually a set of queries is proposed that investigates the possible consequences of certain changes in the policy. Simple safety checking is an example of a query; it checks if there exists a reachable state in which a (presumably untrusted) principal has access to a resource. Although early results show that this type of safety analysis can easily lead to undecidability [Harrison et al. 1976], there has been recent work that demonstrates a class of access control models and queries for which safety is decidable and efficient algorithms exist [Li et al. 2003]. Unfortunately, there are limitations to the expressiveness of the models that are analyzed: the states are described using positive Datalog programs, so there is no support for classical negation.

Another line of research has focused on using logic-based methods for security policy analysis in general [Schaad and Moffett 2002; Fisler et al. 2005; Guelev et al. 2004; Halpern and Weissman 2003; Massacci 1997; Zhao et al. 2005; Hughes and Bultan 2007; Li et al. 2003; Zhang et al. 2005; Tschantz and Krishnamurthi 2006; Bryans 2005; Damiani et al. 2004], and XACML analysis in particular [Fisler et al. 2005; Guelev et al. 2004; Zhang et al. 2005; Bryans 2005]. The basic service provided by these approaches is formal verification of a policy against given safety properties. In addition, some tools [Fisler et al. 2005; Hughes and Bultan 2007] support change analysis, using queries of the form 'Show me all access requests that used to map to Deny but now are mapping to Permit'. Main limitation of previous work lies in the expressiveness/analyzability trade-off: either the analysis services provided are very limited (e.g., most Datalog-based approaches do not support change analysis) or the subset of XACML being analyzed is limited. The issue of reasoning about policies with rich background domains is also not addressed.

In this paper, we propose a logic-based XACML analysis framework that overcomes the previously mentioned limitations. By using a decidable subset of first-order logic, we extend previous work by both a) providing a more extensive set of analysis services, and b) covering a larger fragment of XACML - including, but not limited to core XACML [Tschantz and Krishnamurthi 2006], RBAC profile [Anderson 2005], XML Schema data types and the Administrative (Delegation) Policy Profile [Rissanen et al. 2007].

The main analysis services we provide include verification, policy comparison and

redundancy checking. Our goal is to provide all of these services while hiding the details of the logic formalism and the internals of the analysis tool. To accomplish this, we allow users to specify their security properties in XACML and we also present the verification results back in XACML (if the property fails, we return a counterexample XACML access request). We present the verification services to the end user as an adaptation of the software engineering technique of unit testing. In addition to verification, our tool also provides policy comparison (including checking for subsumption and compatibility/disjointness), and detecting redundant ("dead") policies.

As a basis for this framework we use Description Logics (DL), which are a family of formalisms that are decidable subsets of First-Order logic, and are the formal basis for the Web Ontology Language(OWL²) [Dean et al.]. Because of the correspondence of policy analysis services to DL reasoning services (e.g., policy comparison can be reduced to concept subsumption, whereas formal verification can be reduced to concept satisfiability), our framework can leverage off-the-shelf DL reasoners optimized to provide the above-mentioned analysis services.

An important benefit of using a logic compatible with OWL is that we can leverage OWL being a W3C standard for representing information on the Web. Thus, we extend previous work of coupling XACML with OWL ontologies [Damiani et al. 2004] by providing a unified reasoning framework about that covers both XACML and OWL.

1.1 Contributions and Organization

The contributions of our paper are as follows:

- We present a mapping of a large fragment of XACML to Description Logics. This mapping extends previous work by covering the XACML Administrative Profile[Rissanen et al. 2007] and XML Schema Datatypes.
- We provide an extensive suite of analysis services for XACML, with the main services offered being policy verification, comparison and redundancy checking. The services are exposed in such a way that no knowledge of the internals of our analyzer or the background logic formalism is needed for XACML end users.
- We provide a mechanism to extend XACML with OWL Ontologies. Moreover, we also show how common policy idioms can be captured using Description Logics.
- Because of lack of publicly available large XACML policy sets, we provide a synthetic XACML policy benchmark. The synthetic policies are used in conjunction with smaller real-world policies to empirically evaluate performance of our analyzer on larger, more complex policies. The evaluation demonstrates the scalability of our approach (we compared our approach against a BDD-based analyzer [Fisler et al. 2005]).

This paper is organized as follows. In the next section we present an overview of XACML and Description Logics, the formalism we use to analyze XACML. In Section 3, we present a few use cases for policy analysis that motivate our work. In

²OWL has three subsets: OWL-Lite, OWL-DL and OWL Full, the first two of which are grounded in description logics. From now on, I will use OWL to refer to the OWL-DL sublanguage.

Section 4, we discuss the formalization in detail. In Section 5, we discuss the analysis services provided by the framework. Section 6 discusses the manner in which we couple XACML with OWL ontologies. We have performed a thorough performance evaluation of our analyzer, both against real world and synthetic XACML policies. Evaluation results (along with information on the implementation) are presented in Section 7. Finally, we discuss related work and conclude in Sections 8 and 9, respectively.

2. PRELIMINARIES

In this section we first provide an overview of XACML (version 3.0 [Rissanen 2007]), with focus on the subset of the language that we support. Then we briefly discuss the features of XACML that we do not cover.

After XACML, we provide an overview of Description Logics, focusing on the logic *SHOIN*, which is formally aligned with OWL-DL and provides all of the expressiveness we need.

2.1 Overview of XACML

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 Rules, Targets and Attributes. `Rules` are the most basic policy element of XACML that actually makes an access decision. Essentially, a `Rule` is a function that takes an access request as input and yields a `Permit`, `Deny` or `Not-Applicable`. To determine if a `Rule` is applicable to an access request, the `Target` element is used.

The `Target` defines the set of requests to which the rule is intended to apply in the form of a logical expression on attributes in the request. `Target` is comprised of a conjunction of `DisjunctiveMatch` elements, where each `DisjunctiveMatch` contains a set of `ConjunctiveMatch` elements. Finally, each `ConjunctiveMatch` contains a list of attributes and values.

Attributes are the basic unit in XACML. They represent characteristics of the subjects, resources, actions or the environment where the access request was 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 are represented as a list of attribute-value pairs. Each attribute can belong to a category - the most common categories in XACML are `Subject`, `Resource`, `Action` and `Environment`³.

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>
    <DisjunctiveMatch>
```

³The full names of these categories consist of a lengthy prefix, e.g., `urn:oasis:names:tc:xacml:3.0:attribute-category:Action`. For clarity, we use shortened names in this paper.

```

<ConjunctiveMatch>
  <Match MatchId="function:string-equal">
    <AttributeValue DataType="#string">read</AttributeValue>
    <AttributeDesignator
      AttributeId="action-type"
      Category="...attribute-category:action"
      DataType="...#string"/>
    </Match>
  </ConjunctiveMatch>
</DisjunctiveMatch>
</Target>
</Rule>

```

2.1.2 *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. Following is a list of the most common combining algorithms:

- `Permit-overrides`. If any rule evaluates to `Permit`, then the combined decision is also `Permit`.
- `Deny-overrides`. If any rule evaluates to `Deny`, then the combined decision is also `Deny`.
- `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.
- `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.

Throughout this paper, we use the following notation: for a XACML policy element P , we refer to its `Target`, `Effect` (in cases of `Rules`), its ordered list of children policy elements, its parent policy element and combining algorithm using $P.target$, $P.effect$, $P.children$, $P.parent$, $P.comb$ respectively. We also use $P.pos$ to refer to the position of P w.r.t its sibling policy elements.

2.1.3 *Administrative XACML.* In XACML 3.0, there are access and administrative policies. So far, we have only been discussing access policies, i.e., policies that specify the situations under which users are granted or denied access.

An administrative policy, on the other hand, specifies who (and under what conditions) is authorized to write access policies. For example, an administrative policy might state that members of group `Administrators` are allowed to write access policies about `Files`. Following, we describe the basic element of an administrative policy.

2.1.3.1 Policy Issuers and Delegates. A policy in XACML can contain a `PolicyIssuer` element that describes the source of the policy. A special form of the `PolicyIssuer` element, called the *trusted* issuer, is used to specify that a policy is trusted by the Policy Decision Point (PDP). A missing `PolicyIssuer` element is shorthand for the trusted issuer. In previous versions of XACML with only access policies, the `PolicyIssuer` element is missing and all access policies are trusted (authorized) by default. If a policy's issuer is not trusted, then the policies has to be authorized by using available administrative policies, in a manner described below.

The `Delegate` element of the administrative policy is used for matching against other policies. Essentially, if the `Delegate` of policy A matches the `PolicyIssuer` of policy B, that means that A can authorize policy B.

When an administrative policy authorizes an access policy, it can also specify under which conditions the access policy is authorized. This is called a constrained situation in [Rissanen et al. 2007], and is analogous to the `Target` attribute in access policies.

2.1.3.2 Processing Model. When a new access request A is to be checked against a XACML policy set, it is first applied against the access policies in the set. This is done in the same manner as in XACML 2.0, where administrative policies are not supported. If some policy applies to A and yields an access decision, then the access decision needs to be authorized by a trusted policy using a process defined in [Rissanen et al. 2007] as *reduction*.

An access policy P can be authorized for an access request A only by its *sibling* administrative policies. In particular, if we want to authorize policy P for A then we generate a administrative request (let's call it N), which contains the same attributes as A but is augmented with the policy issuer attributes of A . Then, this new request is applied against any suitable administrative policy that is a sibling of P . To find suitable policies, we inspect a) the policy delegate attributes of the administrative policy and the issuer attribute of N , and b) the situation attributes with the other attributes of A . If both of these sets of attributes match, then we *reduce* the request by substituting the issuer attribute with the issuer attributes of the just-applied administrative policy. Then, this new request is applied against other (sibling) administrative policies. The authorization process ends when the request is successfully applied against a trusted administrative policy.

2.1.4 Advanced XACML Features. We support the Hierarchical Role-based Access Control Profile of XACML [Anderson 2005], which allows us to specify inheritance relationships between roles. In addition, we provide some datatype support for values of attributes. More specifically, we offer support for built-in and user-defined XML Schema datatypes. For example, we could state that *age* attribute can have value ≥ 18 , or that it must be one of 18, 19, 20, 21.

As for the part of XACML that we do *not* support, this includes multi-subject requests, higher order attribute functions in `Conditions` and some combining algorithms (`Only-one-applicable`). 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 (XPath datatypes and `Only-one-applicable`

overriding algorithm) – this is part of our ongoing work.

2.2 Description Logics

Description logics (DL) are a family of logic-based knowledge representation languages which can be used to represent and reason about the terminological knowledge of an application domain [Baader et al. 2003]. 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 and complex ones can be built on top of them. Concepts are defined inductively using the following grammar:

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

where A is an atomic concept, a is an individual, $C_{(i)}$ a *SHOIN* concept, R a role, S a *simple* role⁴ 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 \mathbb{T} is a finite set of concept inclusion axioms. An ABox \mathbb{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 [Horrocks and Sattler 2005].

The satisfaction of a *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) $\mathcal{I} \models \text{Trans}(R)$ 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 \mathbb{R} (respectively of the TBox \mathbb{T}) if it satisfies all the axioms in \mathbb{R} (respectively \mathbb{T}). \mathcal{I} is a model of $\mathbb{K} = (\mathbb{T}, \mathbb{R})$, denoted by $\mathcal{I} \models \mathbb{K}$, iff \mathcal{I} is a model of \mathbb{T} and \mathbb{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 \mathbb{K} amounts to a check whether \mathbb{K} admits a model in which the interpretation of C is nonempty. Subsumption between two concepts C and D in \mathbb{K} , amounts to a check whether $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ for every interpretation \mathcal{I} of \mathbb{K} , denoted as $\mathbb{K} \models C \sqsubseteq D$. Subsumption is reduced to concept satisfiability as follows: C is subsumed by D in \mathbb{K} iff $C \sqcap \neg D$ is not satisfiable in \mathbb{K} .

3. POLICY ANALYSIS USE CASES

In this section, we provide motivation for and description for the analysis services provided by our framework. We will give a running example to illustrate policy verification and tracing, and a distributed policy use case to illustrate policy comparison and need for policy domain models.

⁴See [Horrocks and Sattler 2005] for a precise definition of simple roles

3.1 Policy Engineering Example

Through this section, we will have a simple policy to use as a running example. 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.

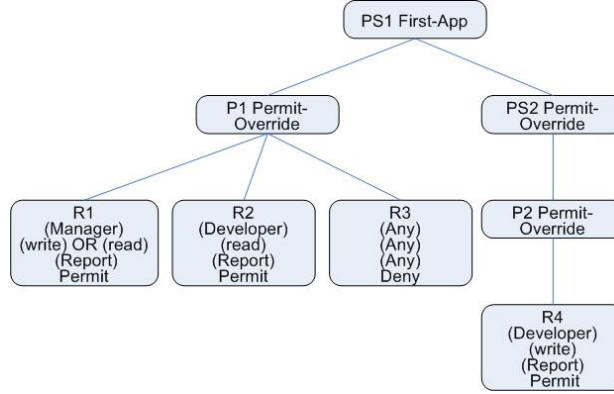


Fig. 1. Example Policy

In this example, a security administrator could specify a test condition for this policy in the form of a XACML access request, “*Developer* requests *write* access to *report*”, and the expected outcome, **Deny**. Usually, performing such policy tests without the help of a verification tool is error prone, since it is difficult to think of all possible conditions that need to be tested. In this example, it could happen that a user logs in with both a *Developer* and a *Manager* role - activating both of these roles grants him write access to the reports. Since the test condition we wrote only checks for a request containing the intern role by itself, this violation will not be caught.

Our analysis framework translates the policies and security properties into Description Logic (DL) axioms, and uses a theorem prover to verify the security (test) property holds in every model. In cases when the test fails, we can extract the access request that causes the failure. In this case, the counter example would be:

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

In addition, we also provide the corresponding *stack trace*, i.e., the XACML policy elements that were applied to this counter example (and the access decisions they yielded). In this example the corresponding stack trace is $R1(Permit) - > P1(Permit) - > PS1(Permit)$. We can also produce all counterexamples that result in test failure. For instance, the other counter example that lead to test failure is:

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

Another analysis service we provide is redundant policy detection. Analogously to dead code detection in software engineering, our redundancy checker finds policy elements (**Rules**, **Policies** or **PolicySets**) whose application does not influence the overall result. To illustrate this service through an example, 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.

3.2 Distributed and Heterogeneous Policy Example

This second example motivates the policy comparison services and the need for background domain models. We have chosen an access control system of a bank.

In this scenario, there is an access control policy assigned by the bank's headquarters, which must be followed by each branch. Each local branch policy *subsumes* the general one and extends it appropriately (customizing the general policy for its own needs). The main requirement is that for any given access request, whenever the general policy returns **Permit** (or **Deny**), the local branch policy should also return **Permit** (or **Deny**). This motivates the need for automated policy *comparison*. Policy subsumption is one type of comparison, other comparison services include: checking for *disjointness* (i.e., there is no access request s.t. both policies apply) and *equivalence* (for any access request, both policies will yield the same decision).

Now consider what happens if a new branch is added to the system. In order to compare and analyze the new policy, we might need support for information integration across different policies. This is because the policy of the new branch might use a different policy model (or at least different terminology). For example, *SystemAdministrator* in one policy might refer to the same role as *TechSupportStaff* in another. Our analysis framework has built-in support for such information integration, since it is tightly integrated with the Web Ontology Language(OWL). More specifically, our mapping reduces XACML analysis services to OWL (DL) reasoning, we can support seamless integration of XACML policy subjects with OWL entities. There have already been proposals for extending XACML with Semantic Web languages [Damiani et al. 2004], however, to the best of our knowledge, we are the first approach where a unified analysis framework is presented that can reason about XACML policies extended with domain ontologies.

4. MAPPING XACML TO DESCRIPTION LOGICS

This section will provide details of the mapping of XACML policy elements to Description Logics concepts.

At the core of the mapping is the translation function $\pi(P, K)$ that takes a policy element P and a DL KB K , and generates axioms in K to capture the semantics of P . More specifically, for each policy element P (**Rule**, **Policy** or **PolicySet**) we introduce two DL concepts, *Permit- P* and *Deny- P* . *Permit- P* is satisfiable (has a model) if and only if there exists an access request that will yield a **Permit** for P (analogously for *Deny- P*).

This section will describe the mapping function π . We will start with **Targets** elements, and then move to **Rules**, **Policies** and **PolicySets**. From now on, unless specified differently, K is assumed to refer to the DL KB that we're mapping

the XACML policies into.

4.1 Mapping Target

A XACML `Policy`, `PolicySet` or `Rule` can all have `Target` elements. `Target` represents the prerequisites that need to be satisfied by an access request for the policy element to apply. A `Target` element in XACML 3.0 is a conjunction of `DisjunctiveMatch` elements, where each `DisjunctiveMatch` is a disjunction of `ConjunctiveMatches`. A `Target` is matched if and only all `DisjunctiveMatches` are, and a `DisjunctiveMatch` is matched if at least one of its `ConjunctiveMatches` is matched. Finally, a `ConjunctiveMatch` is a (conjunctive) list of attribute-value pairs (`Match` elements).

We translate the `Target` element to a DL concept expression. The main idea is that attribute-value pairs are mapped to existential restrictions – for example (*role Developer*) would be mapped to $\exists \text{role}.\text{Developer}$. We combine the attribute-value pairs from different `ConjunctiveMatch` and `DisjunctiveMatch` using the appropriate conjunctive and disjunctive DL constructs (resp. \sqcap, \sqcup). 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 \mathcal{K} (we create a DL role or a concept with the same name as the XACML attribute or value). The full mapping of the `Target` element to a DL concept expression is given in Table I.

Syntax	Mapping π
$T ::= DMatch \mid T \sqcap T$	$\pi(DMatch) \mid \pi(T) \sqcap \pi(T)$
$DMatch ::= CMatch \mid DMatch \sqcup DMatch$	$\pi(CMatch) \mid \pi(DMatch) \sqcup \pi(DMatch)$
$CMatch ::= Match \mid CMatch \sqcap CMatch$	$\pi(Match) \mid \pi(CMatch) \sqcap \pi(CMatch)$
$Match ::= (\text{attr-id attr-val})$	$\exists \pi(\text{attr-id}).\pi(\text{attr-val})$
attr-id	<i>attr-id</i> DL role
attr-value	<i>attr-val</i> DL concept

Table I. Mapping `Target` to a DL concept expression

In Table I, we made the simplifying assumption that each attribute value in the policy is matched to an attribute value in the request using the string equality function. We are also able to capture other comparison (numeric, datetime, etc.) functions, these are described in further detail in Section 4.4.

We represent the XACML construct *Any* (i.e., empty `Target`) as a disjunction of possible attributes. To capture the possible values, we set the role filler of each DL role corresponding to a XACML attribute to be a disjunction of all found values for that XACML attribute. Note that this is done for XACML string attributes only - in cases of other (numeric and datetime) datatypes, it is not feasible to enumerate all possible values. In this case we use concrete domains in a manner explained in Section 4.4.

Assuming that the attribute values for different categories (like Subject, Action and Resource) are disjoint, we can prune the size of the *Any* mapping significantly.

Following is a mapping of *Any* for the running example:

$$\text{Any} \equiv \exists \text{role} . (\text{Manager} \sqcup \text{Developer}) \sqcup \exists \text{action} . (\text{read} \sqcup \text{write}) \sqcup \exists \text{resource} . \text{Report}$$

4.2 Mapping XACML Rules

For each XACML **Rule** we introduce a *Permit-R* and *Deny-R* concept.

Rules in XACML consist of an identifier (**ID**), **Target** representing the preconditions, and an **Effect** (we mark it with α), representing the head of the rule. The *Permit* and *Deny* concepts for a rule depend on the effect of the rule itself. For example, for a rule *R* that yields a **Permit**, the *Deny-R* concept will be equal to a contradiction, because there cannot be an access request where *R* will yield a **Deny**. The *Permit-P* concept for *R* will be equal to the mapping of the **Target** of *R*.

DEFINITION 1. *Mapping a Rule to DL* For a rule $R = (\text{ID} \ T \ \text{Permit})$, \mathbb{K} is extended as follows:

$$\mathbb{K} = \mathbb{K} \cup \{ \text{Permit-ID} \equiv \pi(\text{Target}, \mathbb{K}), \ \text{Deny-ID} \equiv \perp \}$$

Analogously, for a rule $R = (\text{ID} \ T \ \text{Deny})$, \mathbb{K} is extended as follows:

$$\mathbb{K} = \mathbb{K} \cup \{ \text{Permit-ID} \equiv \perp, \ \text{Deny-ID} \equiv \pi(\text{Target}, \mathbb{K}) \}$$

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

$$\begin{aligned} \text{Permit} - R_1 &\equiv \exists \text{role} . \text{Manager} \sqcap \exists \text{resource} . \text{Report} \sqcap \\ &\quad \exists \text{action} . (\text{read} \sqcup \text{write}) \end{aligned}$$

$$\text{Deny} - R_1 \equiv \perp$$

$$\text{Permit} - R_2 \equiv \exists \text{role} . \text{Developer} \sqcap \exists \text{action} . \text{read} \sqcap \exists \text{resource} . \text{Report}$$

$$\text{Deny} - R_2 \equiv \perp$$

$$\text{Permit} - R_3 \equiv \perp$$

$$\begin{aligned} \text{Deny} - R_3 &\equiv \exists \text{role} . \text{Manager} \sqcup \text{Developer} \sqcup \exists \text{action} . (\text{read} \sqcup \text{write}) \sqcup \\ &\quad \sqcup \exists \text{resource} . \text{Report} \end{aligned}$$

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

$$\text{Deny} - R_4 \equiv \perp$$

XACML access requests represent a conjunction of attribute-value pairs, so they are mapped similarly to the **Target** element of rules. To check whether a request r matches a rule with target T , we only need to check whether $\mathbb{K} \models \{ \pi(r) \sqsubseteq \pi(T) \}$ (equivalent to instance checking in description logics).

4.3 Mapping Policies and PolicySets

A **Policy** or a **PolicySet** can also have a **Target** element. In addition, **Policies** are a collection of **Rules**, whereas **PolicySets** can contain other **Policies** or **PolicySets**. The access decisions of the children elements are combined using designated **Policy** and **Rule** combining algorithms.

For a **Policy** P , the **Permit** and **Deny** concepts have to take into account the **Target**, and in addition the **Permit-** and **Deny-** concepts of P 's children. This is

Algorithm 1 *can_override*(p, r, α)**Input:**

p, r : XACML policy elements
 α : access decision

Output:

b : returns true if whenever both p, r apply, and r yields α , the decision of p would override r

```

1: if  $\alpha = \text{Permit}$  and  $p.\text{comb} = \text{Permit-Overrides}$  then
2:   return true
3: else if  $\alpha = \text{Deny}$  and  $p.\text{comb} = \text{Deny-Overrides}$  then
4:   return true
5: else if  $p.\text{comb} = \text{First-Applicable}$  and  $p.\text{pos} > r.\text{pos}$  then
6:   return true
7: else
8:   return false
9: end if

```

because the access decision a **Policy** yields depends on the **Target** and the results of its children's decisions. Intuitively, a **Permit-P** is matched only if P's **Target** is matched and at least one of its children's **Permit-** concepts is matched (satisfied) as well. However, for each child policy element, we also need to make sure that whenever it yields a **Permit** it will not be overridden by a **Deny** from a different child policy element. Taking these considerations into account, the definition for a **Permit-P** concept for a policy is as follows:

DEFINITION 2. *Generating a permit concept for a Policy P:*

$$\mathbf{K} = \mathbf{K} \cup \{ \text{Permit-}P \equiv \pi(\text{Target}) \sqcap \sqcup (\text{Permit-}R_i \sqcap \neg \sqcup \text{Deny-}R_j) \} \text{ where}$$

— $R_i, R_j \in P.\text{Rules}$

— $\text{can_override}(R_j, R_i, \text{Permit}) = \text{true}$

Deny-P is defined analogously.

The **Permit** and **Deny** concepts for **PolicySets** are very similar to ones for **Policy**; the only change is that **PolicySets** contain **Policy** or **PolicySet** elements as children.

DEFINITION 3. *Permit concept for a PolicySet PS:*

$$\mathbf{K} = \mathbf{K} \cup \{ \text{Permit-}PS \equiv \pi(\text{Target}) \sqcap \sqcup (\text{Permit-}P_i \sqcap \neg \sqcup \text{Deny-}P_j) \} \text{ where}$$

— $P_i, P_j \in PS.\text{Children}$

— $\text{can_override}(P_j, P_i, \text{Permit}) = \text{true}$

Deny-PS is defined analogously.

EXAMPLE 2. *The policies and policysets in our running example are mapped as*

follows:

$$\begin{aligned}
\textit{Permit-PS1} &\equiv \textit{Permit-P1} \sqcup (\textit{Permit-PS2} \sqcap \neg \textit{Deny-P1}) \\
\textit{Deny-PS1} &\equiv \textit{Deny-P1} \sqcup (\textit{Deny-PS2} \sqcap \neg \textit{Permit-P1}) \\
\textit{Permit-P1} &\equiv \textit{Permit-R1} \sqcup \textit{Permit-R2} \\
\textit{Deny-P1} &\equiv \textit{Deny-R3} \sqcap \neg (\textit{Permit-R1} \sqcup \textit{Permit-R2}) \\
\textit{Permit-PS2} &\equiv \textit{Permit-P2} \\
\textit{Deny-PS2} &\equiv \textit{Deny-P2} \\
\textit{Permit-P2} &\equiv \textit{Permit-R2} \\
\textit{Deny-P2} &\equiv \textit{Deny-R4}
\end{aligned}$$

4.4 Datatypes

So far, in our mapping we have assumed only string attribute values and simple string equality comparison to match those values. However, XACML offers extensive datatype support and various attribute matching functions. In this subsection, we will discuss how additional datatypes and matching functions are mapped in our framework.

4.4.1 Datatypes. XACML supports the XML schema datatypes and in addition it defines four datatypes of its own: `ipAddress`, `dnsName`, `rfc822Name`, `x500Name`. Description Logics provide some support for datatypes based on the notion of concrete domains. Since DL reasoners such as Pellet [Parsia and Sirin 2004] and Fact++ [Tsarkov and Horrocks 2006] already have built-in reasoning support for XML schema datatypes, mapping the encountered datatypes attribute values is trivial - for each datatype attribute value we create the same datatype value in the DL KB. Thus, we can support the following XML schema datatypes: `string`, `boolean`, `integer`, `double`, `time`, `date`, `datetime`, `anyURI`, `hexBinary`, `base64Binary`.

4.4.2 Attribute Matching Functions. As part their datatype reasoning support, DLs already provide support for the various attribute comparison functions in XACML (such as `datetime-greater-than`) by way of *user-defined* XML schema datatypes. This support is currently being standardized in OWL 1.1 [Motik et al.] and is implemented in Fact++ and Pellet. User-defined (restricted) datatypes are supported through the `datatypeRestriction` constructor, which creates a restricted range by applying a facet to a particular data range. Built-in XML schema facets include: `length`, `minLength`, `maxLength`, `pattern`, `minInclusive`, `minExclusive`, `maxInclusive`, `maxExclusive`, `totalDigits`, and `fractionDigits`. These facets cover the numeric and non-numeric comparison functions in the XACML specification (Appendix A.3 in [Rissanen 2007]).

More detailed information about the mapping is presented in Table 4.4.2. For brevity, we only presented integer datatypes in the table; comparison functions involving the other XML schema datatypes can be mapped in the same manner. Since the OWL functional syntax is already standardized for user-defined datatypes, we used it instead of a DL syntax. For a XACML attribute matching function *fen*, if its datatype is `string`, then we match the attribute *a* to an *object* role, and the

value v to a corresponding class in the KB. In all other cases, we map a to a *datatype* role, and v to a datatype value (with the appropriate type) in the KB. Please note here that we currently handle only one variable constraints, and only cover XML schema datatypes.

Attribute matching function fcn	$\pi(a, v, fcn)$
integer-equal	DataHasValue($\exists\pi(a) v$)
integer-greater-than	DataHasValue($\pi(a)$ DatatypeRestriction (xsd:integer minExclusive v))
integer-greater-than-or-equal	DataHasValue($\pi(a)$ DatatypeRestriction (xsd:integer minInclusive v))
integer-less	DataHasValue($\pi(a)$ DatatypeRestriction (xsd:integer maxExclusive v))
integer-less-than-or-equal	DataHasValue($\pi(a)$ DatatypeRestriction (xsd:integer maxInclusive v))

Table II. Mapping attribute comparison functions to OWL user-defined datatypes. As input, the π takes an attribute a , value v and a matching function fcn .

4.5 Mapping Administrative XACML

The Administrative Policy Profile[Rissanen 2007] adds support for administrative/delegation policies to XACML 3.0. In this version of XACML, after a policy element yields an access decision, that decision might need to be authorized by an administrative policy. In contrast, in previous versions of XACML all access decisions were assumed to be authorized by default.

At the core of our translation of Administrative XACML lies the *reduction graph*.

DEFINITION 4. *Reduction Graph*

A *reduction graph* is a directed graph $\mathcal{G} = \{V, E\}$ where each $v \in V$ corresponds to a XACML *Policy* and has four labels: *target*, *issuer*, *delegate* and *trusted*. For a node v , $v.target$ corresponds to the situation (*target*) to which v applies, *issuer* holds the attributes that describe the issuer of v and *delegate* contains information about the issuer to whom v can be delegated. In addition, *trusted* is a boolean field that indicates if the policy was issued by a trusted issuer.

For two nodes v, w that correspond to two sibling policy elements, a directed edge $(v, w) \in E$ exists if and only if $v.issuer \sqsubseteq w.delegate$.

Intuitively, the edges in \mathcal{G} indicate whether the issuer and delegate of two policy nodes are compatible. If $(v, w) \notin E$, that means it is not possible to reduce an administrative request from v to w . *Target* elements of each node in the reduction graph are mapped to DL concept expressions in the same manner as **Targets** of access policies (presented in Section 4.1). In cases when the node is an access policy, the *delegate* field is empty.

Algorithm 2 $get_admin_expr(node, \mathcal{G})$ **Input:**

$node$: node in graph corresponding to input policy
 \mathcal{G} : reduction graph of composed of $node$'s sibling policies

Output:

b : returns an expression that corresponds to all possible ways $node$ can be authorized

```

1:  $paths \leftarrow []$ 
2:  $searchGraph(node, \top, paths)$ 
3:  $result \leftarrow \perp$ 
4: foreach  $expr$  in  $paths$ 
5:    $result \leftarrow result \sqcup expr$ 
6: return  $result$ 

```

4.5.1 *Extending the XACML Mapping.* *Permit* and *Deny* concept definitions are augmented with the constraints coming from the administrative policies. The definition of a *Permit* concept is extended as follows:

$$Permit-P \equiv \pi(Target) \sqcap \sqcup (Permit-R_i \sqcap \neg \sqcup Deny-R_j) \sqcap get_admin_expr(P, \mathcal{G})$$

The added function (get_admin_expr) generates a concept expression that is a disjunction of all unique reduction paths from the current policy (P) to a policy with a trusted issuer. Since this is the only way P can be authorized, the result of get_admin_expr is added as a constraint to the existing *Permit* concepts. *Deny* concepts are extended in the same manner.

Details of get_admin_expr shown in Algorithm 2. The function takes the reduction graph \mathcal{G} and the node corresponding to policy P as input, and generates a disjunction of all unique possible paths from P to a trusted policy.

The algorithm that generates the paths (Algorithm 3) performs a depth first search on \mathcal{G} . It starts with the policy node (for the *Permit* or *Deny* concept), and tries to build a path along administrative policy (access policies cannot authorize administrative requests) nodes to a trusted node, accumulating constraints along the way. These constraints come from the *Target* elements of each administrative policy, since administrative policies can restrict the cases in which they are applied. When the accumulated constraints become unsatisfiable at a point in the path, the algorithm backtracks and tries a different node. All unique paths are added to the *Permit* or *Deny* concept.

5. SERVICES

This section will discuss the analysis services provided by our framework. One of our goals is to expose these services in such a way that no knowledge of the internal details of our analysis tool (or the formalism used) is necessary.

5.1 Formal Verification

Formal verification is the most commonly offered analysis service for access policies. In most previous work, the security property to be verified is specified programmatically using the particular analysis tool's API, thus requiring users to be familiar

Algorithm 3 *search_graph(node, constraint, paths)*

Input:

node: current node being explored
constraint: accumulated constraints so far
paths: empty list

Output:

paths: list of found paths to a trusted policy

```

1: if node has trusted issuer then
2:   paths.add(constraint)           ▷ found a new path, remember constraint
3: else
4:   foreach nbor in n.getnbors()
5:     if nbor.getVisited() and constraint  $\sqcap$  nbor.target  $\not\sqsubseteq \perp$  then
6:       nbor.setVisited(true)
7:       searchGraph(nbor, constraint  $\sqcap$  nbor.target, paths)
8:       nbor.setVisited(false)
9:     end if
10: end if

```

with the internals of the tool. Our approach, on the other hand, allows users to specify their security properties in XACML and also presents the verification results back in XACML. To accomplish this, we chose to expose the service as an adaptation of the well known software engineering technique of unit testing⁵.

Traditional unit testing applied to policies amounts to users creating an access request (the unit test) and specifying its expected value (*Permit*, *Deny*, *NotApplicable* or *Indeterminate*). Then, whenever changes are made to the access policy, all of the test requests are run against a XACML engine to make sure no bugs (security holes) are introduced. However, writing such unit tests in this manner is tedious and error-prone, since it is difficult to think of all possible conditions that need to be tested. Consider our running example (Section 3.1): it could happen that a user logs in with both a developer and a manager role - activating both of these grants him write access to reports. Since the unit test (security property in Section 3.1) we wrote only tests for a request containing the developer role by itself, this violation will not be caught.

Our formal verification-based approach to testing overcomes the above limitations. Instead of taking the access request literally and performing a shallow test based on the explicitly mentioned attributes in the request, we try to build a logic model where the attributes in the original request can produce a test failure. While building this model the DL reasoner explores all possible combinations of *additional* attributes in the request that could lead to a test failure. The test condition holds only when such a model cannot be built.

The input for this service consists of a XACML policy file P (for the access policy being tested), a XACML file that contains the test condition T , and a string denoting the type of test condition. Supported types of properties are *alwaysPermit*,

⁵The authors would like to thank Bijan Parsia for an insightful discussion on the similarities between software testing and formal verification of security policies.

alwaysDeny, *neverPermit* and *neverDeny* - these are similar to assertions for unit tests. Details on how these are reduced to entailment checking in Description Logics are presented in Table 5.1. If the policy fails, output is in the form of a XACML access request, which consists of the counter example that brings about the test failure.

Type of Property	Entailment Check
alwaysPermit	$\pi(T.target) \sqcap \neg Permit-P \models \perp$
alwaysDeny	$\pi(T.target) \sqcap \neg Deny-P \models \perp$
neverPermit	$\pi(T.target) \sqcap Permit-P \models \perp$
neverDeny	$\pi(T.target) \sqcap Deny-P \models \perp$

Table III. **Mappings of Common Types of Policy Tests.** T refers to the test policy (containing the security property) and P refers to the top level policy set. Property holds if and only if the entailment holds, i.e., the concept expression is *not* satisfiable.

5.1.1 *Tracing.* In cases when the test fails, we extract the counter example directly from the logic model, map it back to XACML and present it as an access request. However, with large policy sets, it can be difficult to find out exactly which policy elements were responsible for the error. For this purpose, we provide a 'stack trace' output of every policy that was fired while generating the counter example. Generating the trace is straightforward, assuming the counter example XACML request is available - the request is run against the policy using a XACML engine (we use Sun's reference implementation⁶), and the access decisions at each policy element are stored.

5.2 Policy Comparison

The *Permit-P* and *Deny-P* concepts defined previously allow us to easily compare the behaviors of two policies. For example, we can check for policy *subsumption*: P_2 subsumes P_1 iff 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. In our framework, policy subsumption is reduced to checking subsumption between the *Permit* and/or *Deny* concepts. For example, to check if P_2 subsumes P_1 w.r.t **Permit**, we ask the reasoner if $Permit-P_1 \sqsubseteq Permit-P_2$.

To illustrate the service, consider adding a new role, *LeadDeveloper*, to our running example. The updated policy now contains an additional **Rule** (R_3):

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

$$Permit-PS_{old}, Deny-PS_{new},$$

$$Permit-PS_{old}, Deny-PS_{new}$$

Subsumption holds only if both of the following hold:

⁶<http://sunxacml.sourceforge.net>

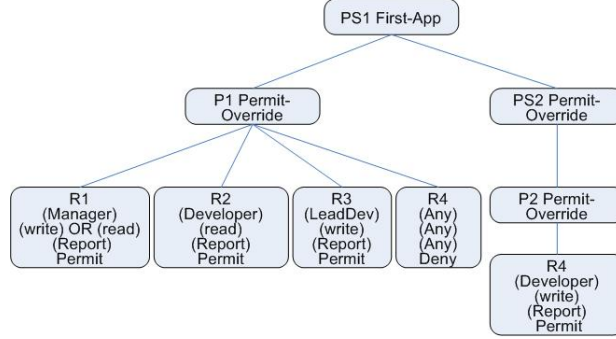


Fig. 2. Updated Policy (with LeadDev role)

$$Permit-PS_{old} \sqsubseteq Permit-PS_{new}$$

$$Deny-PS_{old} \sqsubseteq Deny-PS_{new}$$

Our analyzer reports the first subsumption holds, which is rather obvious from since the **Rule** that we added in PS_{new} yields a **Permit**). However, the analyzer 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 model for $A \sqcap \neg B$. If a model *can* be built, it means the subsumption does not hold. In that case the model that was just built can easily be extracted from the internals of the reasoner and used as a counter example. Here 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 adding a separation of duty constraint for the roles of *Developer* and *LeadDev*. The constraint is presented below, in DL syntax.

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

Note that separation of duty constraints can also be serialized in XACML. To accomplish this, a new **Policy** should be created, having the two disjoint roles in its **Target**. Since we want to prohibit a requester that has both of the roles, the effect of this new is **Deny**. Finally, this new policy needs to be set with highest priority (e.g., add it at the beginning of the root policy set and use **First-Applicable** algorithm) to ensure that the separation of duty constraint can never be overridden.

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 corresponding concept expressions for those decisions :

$$Deny-P_1 \sqcap Permit-P_2$$

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 a request, and we can extract the counter example from the reasoner. To get all counter examples, we need to retrieve *all* consistent models that the concept expression admits; this involves saturation of the tableau, a technique for which DL reasoners are not particularly optimized.

The service of verifying changes was introduced in [Fisler et al. 2005], we show here that it can be accomplished in DL as well. The safety properties to be checked are simply added to the conjunction of the *Permit* and *Deny* concepts. For example, if we want to verify that all changes from **Permit** to **Deny** in the above policy involved the *LeadDev* role, we could test the following concept expression:

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

5.3 Redundancy Checking

Another service we provide is determining redundant **Rules**⁷. A redundant rule is one that whenever fires, it is always overridden by some other rule or policy with higher priority. A simple way to check redundancy of a rule r is to perform change impact analysis for a policy with and without the rule. Here we present a more guided approach for checking redundancy, by building a concept expression that ignores the policy elements that *cannot* override the rule we are checking. Algorithm 4 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 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.

6. INTEGRATING WITH OWL ONTOLOGIES

Recently, there has been a great amount of interest in extending the expressive power of XACML with Semantic Web technologies [Damiani et al. 2004; Damiani et al. 2005; Ardagna et al. 2005]. This is because the Semantic Web provides a data sharing and re-use framework for applications across enterprise and community boundaries. One of the foundational languages of the Semantic Web is the Web

⁷The technique can be easily generalized to **Policies** or **PolicySets** – for brevity we focus on **Rules** only.

Algorithm 4 *is_redundant*(r, \mathcal{D})

Input: $r = (ID \ T \ \alpha)$: XACML Rule**Output:** b : returns true if r is redundant, false otherwise

```

1:  $J \leftarrow \perp$ 
2:  $r_{old} \leftarrow r$  ▷ cache  $r$ 
3: while true do
4:    $J \leftarrow J \sqcup \pi(q.target)$  where
5:      $q.parent = r.parent$  and
6:      $q > r$  ▷  $q$  can override  $r$ 
7:   if  $parent(r) \neq null$  then
8:      $r \leftarrow r.parent$ 
9:   else
10:    break
11:  end if
12: end while
13: if  $r_{old}.target \sqsubseteq J$  then ▷ request is subsumed
14:  return true
15: else
16:  return false
17: end if

```

Ontology Language (OWL) [Dean et al.], which is a W3C standard for representing shared information.

Previous work has focused on extending XACML with support for rich ontology-based models representing subject and resource information. These extensions were accomplished by adding new matching functions and extended conditions to refer to the background ontologies. While these extensions do increase the expressive power of XACML, they do not offer analysis support for such ontology-extended XACML policies.

We extend previous work by providing a *unified* logical framework for analyzing XACML policies extended with OWL ontologies. Given the XACML mapping to description logics (DL) provided in this paper, and the fact the semantics of OWL is grounded in DL, our analysis framework can easily handle such policies. To connect XACML entities with background domain ontologies, we have introduced a new datatype for attributes, *OWLEntity*. The attribute of an *OWLEntity* datatype is an OWL property, whereas the value is an OWL Class or OWL Individual. This simple extension allows XACML policy editors to easily refer to background domains in their policies. For example, if one wants to say that the Subject needs to be a member of class `http://policy#Administrator`, the following syntax can be used:

```

<Match MatchId="...:function:string-match">
  <AttributeValue DataType=".../XMLSchema#string">
    http://policy#Administrator
  </AttributeValue>
</Match>

```

```

</AttributeValue>
<SubjectAttributeDesignator
  AttributeId="http://www.w3.org/1999/02/22-rdf-syntax-ns#type"
  Category="...:Subject"
  DataType="OWLEntity"/>
</SubjectMatch>

```

Currently we assume that the physical URI of the ontology is same as the logical URI. In order to process this subject s , the framework would retrieve the ontology K available at `http://policy` and check if $K \models Administrator(s)$.

6.1 Representing Policy Idioms

Here we will discuss how a domain ontology can be used to provide semantic descriptions for the entities used in the access policy. For the policy in our running example, we could develop an ontology that describes the company domain, and link the policy entities with concepts in the ontology using subclass relationships. For example, we can state that a *Manager* is of type *Employee* who 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. DL syntax is used for brevity, but the following can be easily serialized in OWL:

- (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:

$$CIO-of \sqsubseteq employee-of$$

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

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

- (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:

$$\geq k \text{ role}.\top \sqsubseteq \perp$$

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:

$$\geq k \text{ role}^{\neg}.\top \sqsubseteq \perp$$

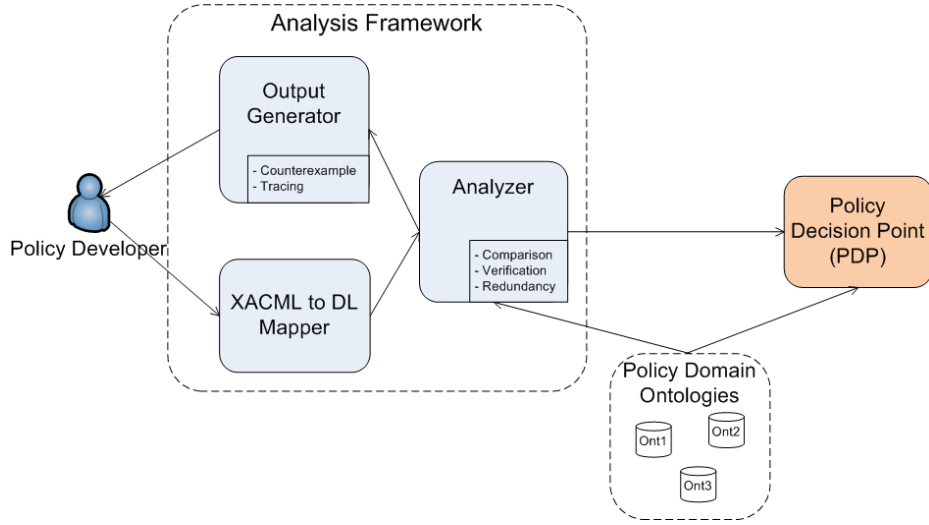
7. IMPLEMENTATION AND EVALUATION

In this section first we describe the main components as well as novel optimizations in our analysis tool. Then, we present the results of our empirical evaluation.

7.1 Implementation

The three main modules of our framework are the mapper, which converts XACML policies to DL knowledge bases, the analyzer, which reduces policy analysis services to DL reasoning tasks, and the output generator, which extracts counter examples from the internals of the reasoner and converts them to XACML access requests. We have provided an interface so different DL reasoners can be plugged in and used as analyzers. So far, we have used Pellet and Fact++.

An architectural diagram is presented in Figure 7.1.



We also implemented a few optimizations in the mapper to improve performance of analysis services:

- *Grouping by categories* - Assuming that the attributes for different categories are disjoint allows us to optimize the analyzer. This is because we can group the attributes based on the category they belong to, and then minimize unnecessary interaction between attributes in different categories. This is accomplished by introducing an additional DL role for each category (categories include **Subject**, **Resource**, **Action**) and then adding the attribute values as role fillers of the corresponding category roles. Consider the mapping of R_3 before:

$$\begin{aligned}
 \text{Deny-}R_3 \equiv & \exists \text{role}. (\text{Manager} \sqcup \text{Developer}) \sqcup \\
 & \exists \text{action-type}. (\text{read} \sqcup \text{write}) \sqcup \\
 & \exists \text{resource-type}. \text{Report}
 \end{aligned}$$

and after:

$$\begin{aligned} \text{Deny-}R_3 \equiv & \exists \text{subject} . \exists \text{role} . (\text{Manager} \sqcup \text{Developer}) \sqcup \\ & \exists \text{action} . \exists \text{action-type} . (\text{read} \sqcup \text{write}) \sqcup \\ & \exists \text{resource} . \exists \text{resource-type} . \text{Report} \end{aligned}$$

We also need to assert these category roles as *functional*, since access subjects can only have one set of attributes for each category.

- *Simplification* of concept expressions. To improve performance, tableau reasoners reduce concept expressions to a simplified normal form before checking for concept satisfiability. Due to the size of the *Permit-P* and *Deny-P* axioms returned from our mapping function (the XACML construct **Any** dramatically increases the size of the DL representation), simplifying the concepts before reasoning will improve performance. However, to gain real benefits of simplification, we needed to partially unfold some of the concepts (increase the sizes of the Permit-P and Deny-P axioms). To accomplish this, we experimented with unfolding the *Permit-P* and *Deny-P* concepts for `Rule`, `Policy` and `PolicySet`, and achieved the best results when only `Rule` and `Policy` *Permit-P* and *Deny-P* concepts are unfolded.

7.2 Evaluation

We evaluated our tool against the BDD-based policy analyzer Margrave [Fisler et al. 2005]. We used both realistic policy data set (Continue [con 2005]), as well as synthetic extensions to test for scalability with large policies. This section reports the evaluation setup and results.

7.2.1 Continue Policy. As a test case, we used the access policy for the conference paper manager Continue, which, to the best of our knowledge, is one of the most complex publicly available XACML policies. The authors of Margrave [Fisler et al. 2005] used the Continue policy to test their analyzer and made the policy publicly available [con 2005]. Fortunately, the Continue policy comes with its own set of security properties (12 of them) that can be verified. There are 26 policy files (each one representing a `PolicySet`) in the set, with a total of 13 attributes and 36 attribute values. Although it does not have many attributes and/or values, the policies in Continue are fairly interconnected and nested (up to 5 levels), which makes it non-trivial to analyze.

As part of the evaluation, we compared the performance of our analysis framework with the BDD-based analyzer Margrave (which is, to the best of our knowledge, the fastest XACML analyzer available). As can be seen in Table IV, once Margrave loads the policy and converts it to a binary decision diagram, it performs verification faster than our approach. Nevertheless, the performance of our framework overall is still very competitive.

Policy comparison and disjointness both took less than 100ms on our machine, however we do need to mention that our tool currently only reports one difference between two policies (if such difference exists). In other words, our analysis framework is not optimized for finding all differences.

Policy	Margrave			DL-based		
	Loading	Verification	Total	Loading	Verification	Total
Continue	0.924	0.010	0.934	0.641	0.594	1.235

Table IV. **Verifying *Continue* properties using Margrave and our tool.** *Loading* represents time to parse the policy and create the BDD in Margrave’s case, and the time to parse the policy and convert it to a Description Logic KB in our case. We used Pellet as our back end theorem prover. All times are in seconds.

We also checked each policy element in *Continue* for redundancy. Checking all `Rules`, `Policies` and `PolicySets` for redundancy took 6.3 seconds, 3.6 seconds and 0.9 seconds respectively. Interestingly, we did find a redundant policy: the third `Policy` in `PPS_paper-assignments.rc PolicySet`. Upon closer inspection, we realized that whenever the prerequisite of the third `Policy` is matched, the first `Policy` will fire as well, and since the `PolicySet` is using the `First-Applicable` combining algorithm, the decision of the first `Policy` will always override the third one.

7.2.2 Evaluation Benchmark. In order to test our approach against a larger policy set, we extended *Continue* in two ways: a) by adding extra attribute values, while preserving the structure of the policy, and b) by adding additional (synthetic) policies to the original policy structure. The main reason for extending *Continue* (as opposed to testing against fully synthetic policy sets) is that the policy comes with a set of properties and constraints that we can re-use in these extensions.

7.2.2.1 *Continue* extended with attribute values. The *Continue* policy uses a limited number of attribute values. For example, there are only four admissible role values: *pc-member*, *pc-chair*, *subreviewer* and *admin*. While this may be sufficient for a conference manager application, we wanted to test performance of our analyzer when there are more attribute values. For this purpose, for each attribute value present in the policy, we introduced additional ones by taking the value and augmenting it with $1 \dots limit$ where *limit* is a set in input. We modified the policy by going through each element, and whenever an attribute value *v* was encountered, it was replaced by a nondeterministically selected value from $v_1 \dots v_i$. For example, wherever *pc-member* occurs in the original policy, after extending there will be one of *pc-member*, *pc-member₂*, ..., *pc-member_{limit}*. If *limit* is set to 1, then no additional values are introduced. On the other hand, if *limit* is set to a very large number, then there will be almost no repetitions of attribute values in the policy.

Empirical results are shown in Figure 7.2.2.1. As expected, as the number of attribute values increases, both analyzers take more time to process the policy. However, as the limit of attribute values increases further, performance of the analyzers improves considerably, and reverts back to previous levels. We believe this happens because with a very large number of attribute values, there will be almost no repetition of values amongst the `Targets` of individual policy elements. Non-overlapping `Targets` in turn imply less overriding (since two policies will rarely apply at the same time), which reduces the complexity of the policy.

7.2.2.2 *Continue* extended with policies. We also wanted to evaluate our tool against a policy with a large number of policy files. For this purpose, we extended

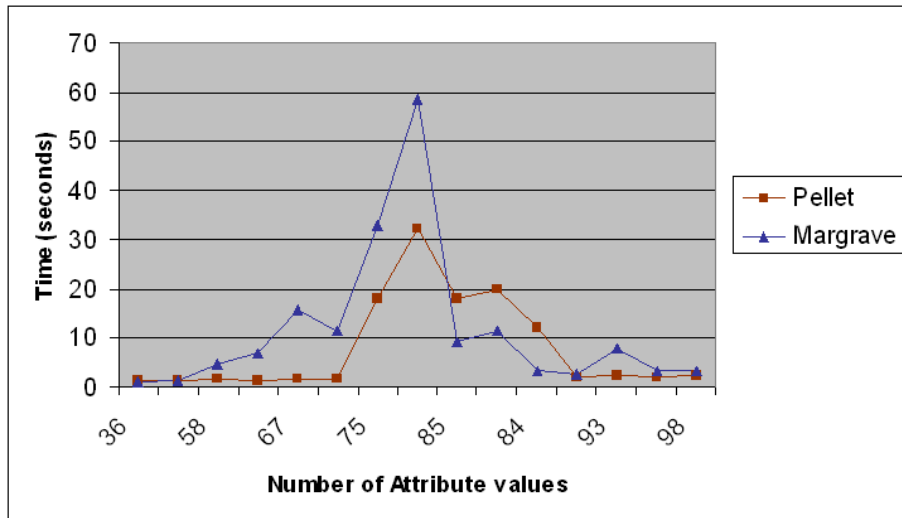


Fig. 3. Performance of Pellet and Margrave on attribute-value extended policy sets. Times include loading and verification. Both tools encounter performance decrease as the limit of possible values grows. For limit=8 (85 values), the BDD built by Margrave has 10569 nodes (compared to 1182 for the original Continue policy).

Continue with synthetic policy sets. To generate these synthetic policies we implemented a customizable XACML policy generator. The generator has the following input parameters:

- **maxDepth** - total maximum depth of policy tree
- **maxAttributesPerCategory** - for each category, a limit of possible attributes
- **maxValuesPerAttribute** - limit of possible values per attribute
- **occurrenceOfAny** - how often a policy element's `Target` is empty (`Any` construct)
- **maxChildren** - maximum number of children for `PolicySet`, `Policies` and `Rules`

To link Continue with the generated policies, we added a reference to the root generated `PolicySet` in Continue's root policy file (`RPSlist.xml`). For simplicity, we restricted the generator to use only attributes that occur in Continue itself (although we do introduce new values, like in the previous section).

Results for are shown in Figures 7.2.2.2 and 7.2.2.2. On the x-axis, there is a number $a.b$ where a stands for the number of synthetic policy sets added to the base policy, and b stands for the total number of distinct attribute values (Continue itself has 38). For lack of space, we included only these two numbers to characterize a synthetic policy. Most of the other parameters were kept constant while generating the policies (only *maxDepth* was changed between 3,4 and 5 to tweak the number of generated policy files). The values for the other generator parameters are as follows: *occurrenceOfAny* was set to .30, *maxChildren* was 8 and

maxValuesPerAttribute was 7. *MaxAttributesPerCategory* was not used, since we were reusing Continue’s attributes.

As it can be seen in Figure 7.2.2.2, the performance of Pellet remains under 10 seconds for all policy extensions. Margrave, on the other hand, runs out of memory while trying to generate the policy BDD on 4 of the test inputs. This is because the performance of its underlying BDD-based model checker decreases as the number of attribute values is increased. For example, Margrave loads the last case (78 extra policy files with only 77 attribute values) rather quickly, while it throws an out of memory error (after 5 minutes of loading) for the case of only 9 extra policies but with 101 attribute values.

In Figure 7.2.2.2, we only list results for Pellet, since we got an out of memory error when running Margrave against those inputs. While Pellet’s performance seems to grow in parallel with the number of policy sets, it is interesting to note that most of this increase comes from the mapping modules.

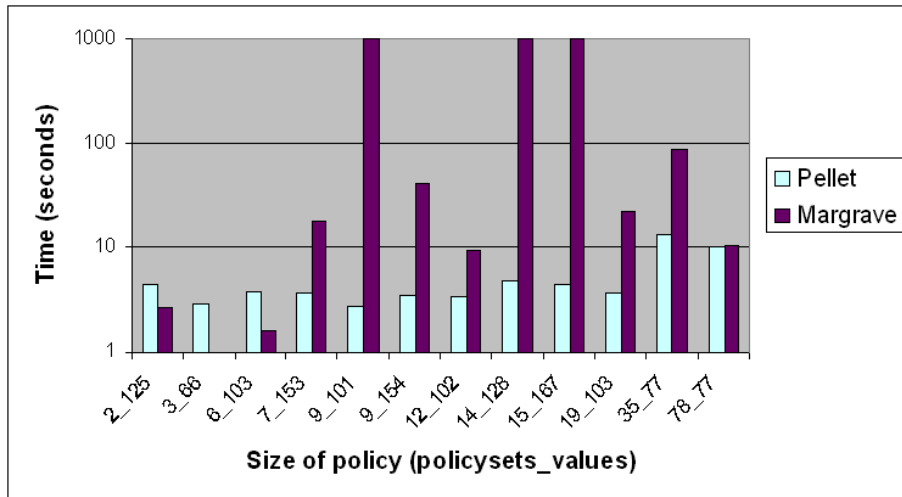


Fig. 4. **Comparing Pellet and Margrave.** Times shown are for formal *verification* of security properties (we used Continue’s 12 safety properties). It includes parsing, converting to BDD (or DL in our case) and verification of all properties. On the x-axis, the first number stands for number of policy files, whereas the second for the maximum number of attributes per category.

Even though these are purely synthetic extensions to Continue, the empirical results so far are very encouraging. In addition to being more expressive than propositional logic-based model checkers, our DL-based analysis framework seems to compare favorably in practice, especially for policy sets with large numbers of attribute values. Our better performance is due to the fact that DL reasoners are particularly optimized for finding one model (counterexample), if such model exists. Thus, seems the trade-off is that DL analyzers can perform formal verification in a manner more scalable (w.r.t. to the number of attribute values and policy sets) than BDD-based tools, as long as only one counter example needs to be returned.

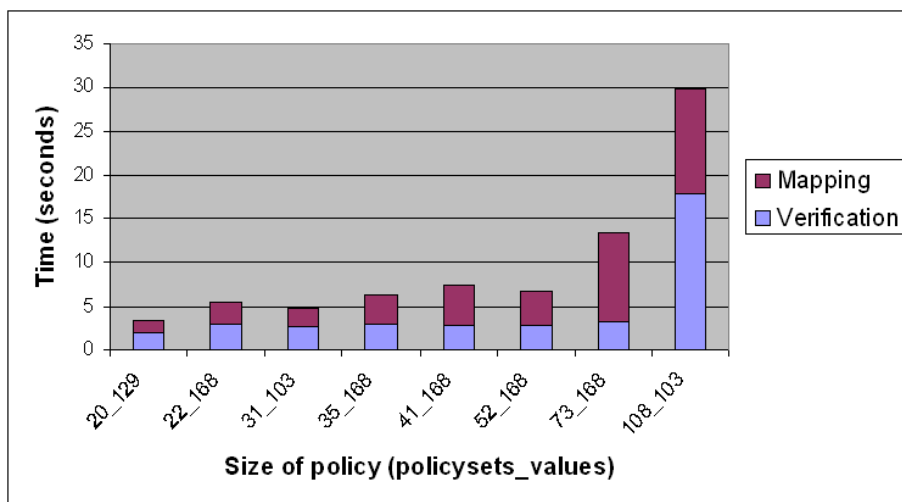


Fig. 5. Performance of Pellet on large policy sets.

Margrave, on the other hand, is more efficient - if it can load the policy in the first place - when *all* counter examples are needed.

We would like to improve this framework by adding a generator for safety properties. For this paper, we reused the properties from the original Continue data set.

8. RELATED WORK

Access policy analysis is a research area receiving a great amount of attention [Li et al. 2003; Jajodia et al. 1997; Winslett et al. 2005; Woo and Lam 1993; Li et al. 2003; Li and Tripunitara 2006; Schaad and Moffett 2002; Fislser et al. 2005; Bryans 2005]. In this section we will provide an overview of related work, starting with approaches dealing with policy analysis in general and then moving to analyzers specifically targeting XACML.

8.1 Access Control Policy Analysis

Logic programming seem to be a popular choice for formalization of access control policy languages [Li et al. 2003; Jajodia et al. 1997; Winslett et al. 2005; Woo and Lam 1993; Li et al. 2003; Li and Tripunitara 2006], 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 never been offered by rule-based policy systems.

Schaad et al. [Schaad and Moffett 2002] examine the problem of verifying a policy that is subject to change coming from another policy. Using the Alloy specification

language and its analysis facilities, they show how to specify a RBAC96-style model, ARBAC97-style extensions and a set of separation of duty properties. No tools or results were given, so it is difficult to compare with our approach - that being said, since they are using Alloy as the policy analysis engine, the same objections (scalability) apply.

In [Zhang et al. 2005] 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* [Guelev et al. 2004].

Moving to DL based systems, Zhao et al. [Zhao et al. 2005] present a formalization of RBAC based on the description logic *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 [Massacci 1997] 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.

8.2 XACML Analysis

[Bryans 2005] formalizes XACML policies using a process algebra known as Communicating Sequential Processes (CSP [Hoare 1978]). This allows them to use a model checkers such as FDR for formally verifying properties of policies, and for comparing access control policies with each other (policy subsumption and equivalence). In addition, the author shows how limited workflows can be mapped to CSP, too. The workflow is sequential in nature and in that sense their approach is more expressive than first-order logic approaches. Unfortunately the authors provide no evaluation.

In terms of services offered, Margrave [Fisler et al. 2005] 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, Margrave is very well suited for change analysis, and whenever it does manage to load and convert the policies, it can compute all differences faster than our approach. On the other hand, we have shown through empirical evaluation that a) our approach is very competitive, even faster when it comes to computing one difference, and b) As the complexity of the input policy increases, our approach scales better. In addition, with a 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. The integration of ontological domain descriptions with access control policies is one of the distinguishing features of our approach.

Hughes et al. [Hughes and Bultan 2007] analyze XACML policies by translating verification queries about XACML policies to a Boolean satisfiability problem, and using a SAT solver to solve the problem. They present a formal model for XACML policies which partitions the input domain to four classes: permit, deny, error, and not-applicable. They present several ordering relations for access control policies which can be used to specify the properties of the policies and the relationships among them. These ordering relations are then converted and a SAT solver is used to verify properties. While it covers a larger fragment of XACML than Margrave, their translation does not handle the Administrative Policy Profile. In addition, they do not address the issue of rich domain models for policies.

9. CONCLUSIONS AND OPEN ISSUES

Understanding the effects and consequences of sets access control policies has always been an issue for policy developers, especially with expressive languages such as XACML. In this paper, we addressed this problem for XACML by proposing an analysis tool based on a decidable fragment of FOL (Description Logics). We were able to provide a similar suite of analysis services to propositional logic based tools, while adding extra expressiveness by covering administrative policies, various datatypes, and ontology-based descriptions of policy entities. Finally, we demonstrated through empirical evaluation that off the shelf DL reasoners are practical as XACML analysis tools.

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 the **Only-one-applicable** overriding algorithm could be handled by extending the mapping function π and adding additional axioms in the DL KB. In addition, we plan to explore techniques for providing semi-automated policy repair.

10. ACKNOWLEDGEMENTS

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

The authors would like to thank Bijan Parsia, Christian Halaschek-Wiener and Jennifer Golbeck for their contributions to this work.

REFERENCES

- 2005. Margrave Continue Example. Available at <http://www.cs.brown.edu/research/plt/software/margrave/versions/01-01/examples/continue/>.
- ANDERSON, A. XACML References, v1.65. Available at <http://docs.oasis-open.org/xacml/references/xacmlRefsV1.65.html>.
- ANDERSON, A. 2005. Core and hierarchical role based access control (rbac) profile of xacml v2.0. Available at http://www.docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-rbac-profile1-spec-os.pdf.
- ARDAGNA, C. A., DAMIANI, E., DI VIMERCATI, S. D. C., FUGAZZA, C., AND SAMARATI, P. 2005. Offline expansion of xacml policies based on p3p metadata. In *ICWE*. 363–374.
- BAADER, F., CALVANESE, D., MCGUINNESS, D. L., NARDI, D., AND PATEL-SCHNEIDER, P. F., Eds.

2003. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press.
- BRYANS, J. 2005. Reasoning about xacml policies using csp. In *SWS '05: Proceedings of the 2005 workshop on Secure web services*. ACM Press, New York, NY, USA, 28–35.
- DAMIANI, E., DI VIMERCATI, S. D. C., FUGAZZA, C., AND SAMARATI, P. 2004. Extending policy languages to the semantic web. In *ICWE*. 330–343.
- DAMIANI, E., DI VIMERCATI, S. D. C., AND SAMARATI, P. 2005. New paradigms for access control in open environments. In *Proceedings of the Fifth IEEE International Symposium on Signal Processing and Information Technology*.
- DEAN, M., CONNOLLY, D., VAN HARMELEN, F., HENDLER, J., HORROCKS, I., MCGUINNESS, D. L., PATEL-SCHNEIDER, P. F., AND STEIN, L. A. Web Ontology Language (OWL) Reference Version 1.0. W3C Working Draft 12 November 2002 <http://www.w3.org/TR/2002/WD-owl-ref-20021112/>.
- FISLER, K., KRISHNAMURTHI, S., MEYEROVICH, L. A., AND TSCHANTZ, M. C. 2005. Verification and change-impact analysis of access-control policies. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*. 196–205.
- GUELEV, D. P., RYAN, M., AND SCHOBEBENS, P.-Y. 2004. Model-checking access control policies. In *ISC*. 219–230.
- HALPERN, J. Y. AND WEISSMAN, V. 2003. Using first-order logic to reason about policies. In *n Proceedings of the Computer Security Foundations Workshop (CSFW'03)*. IEEE Computer Society, Los Alamitos, CA, USA.
- HARRISON, M. A., RUZZO, W. L., AND ULLMAN, J. D. 1976. Protection in operating systems. *Commun. ACM* 19, 8, 461–471.
- HOARE, C. A. R. 1978. Communicating sequential processes. *Commun. ACM* 21, 8, 666–677.
- HORROCKS, I. AND SATTLER, U. 2005. A tableaux decision procedure for SHOIQ. In *Proc. of the 19th Int. Joint Conf. on Artificial Intelligence (IJCAI 2005)*. Morgan Kaufman.
- HUGHES, G. AND BULTAN, T. 2007. Automated Verification of XACML Policies Using a SAT Solver. In *Proceedings of the 7th International Conference on Web Engineering, Workshop on Web Quality, Verification and Validation (WQVV)*. 378–392.
- JAJODIA, S., SAMARATI, P., SUBRAHMANIAN, V. S., AND BERTINO, E. 1997. A unified framework for enforcing multiple access control policies. In *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data*. ACM Press, New York, NY, USA, 474–485.
- KOLOVSKI, V., HENDLER, J., AND PARSIA, B. 2007. Analyzing web access control policies. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*. ACM, New York, NY, USA, 677–686.
- LI, N., GROSOFF, B. N., AND FEIGENBAUM, J. 2003. Delegation logic: A logic-based approach to distributed authorization. *ACM Trans. Inf. Syst. Secur.* 6, 1, 128–171.
- LI, N. AND TRIPUNITARA, M. V. 2006. Security analysis in role-based access control. *ACM Transactions on Information Systems Security* 9, 4, 391–420.
- LI, N., WINSBOROUGH, W. H., AND MITCHELL, J. C. 2003. Beyond proof-of-compliance: Safety and availability analysis in trust management. In *IEEE Symposium on Security and Privacy*.
- MASSACCI, F. 1997. Reasoning about security: A logic and a decision method for role-based access control. In *First International Joint Conference on Qualitative and Quantitative Practical Reasoning ECSQARU-FAPR*. 421–435.
- MOTIK, B., PATEL-SCHNEIDER, P. F., AND HORROCKS, I. OWL 1.1 Web Ontology Language: Structural Specification and Functional-Style Syntax . Editor's Draft of 23 May 2007.
- PARSIA, B. AND SIRIN, E. 2004. Pellet: An OWL DL reasoner. In *Third International Semantic Web Conference - Poster*.
- RISSANEN, E. 2007. eXtensible Access Control Markup Language (XACML) Version 3.0 (Core Specification and Schemas). Available at <http://www.oasis-open.org/committees/download.php/23950/xacml-3.0-core-wd-02.zip>.

- RISSANEN, E., LOCKHART, H., AND MOSES, T. 2007. Xacml administrative policy version 1.0, working draft 17. Available at <http://www.oasis-open.org/committees/download.php/23951/xacml-3.0-administration-v1-wd-17.zip>.
- SCHAAD, A. AND MOFFETT, J. D. 2002. A lightweight approach to specification and analysis of role-based access control extensions. In *SACMAT '02: Proceedings of the seventh ACM symposium on Access control models and technologies*. ACM Press, New York, NY, USA, 13–22.
- TSARKOV, D. AND HORROCKS, I. 2006. Description logic reasoner: System description. In *IJCAR*. 292–297.
- TSCHANTZ, M. C. AND KRISHNAMURTHI, S. 2006. Towards reasonability properties for access-control policy languages. In *SACMAT '06: Proceedings of the eleventh ACM symposium on Access control models and technologies*. ACM Press, New York, NY, USA, 160–169.
- WINSLETT, M., ZHANG, C. C., AND BONATTI, P. A. 2005. Peeraccess: a logic for distributed authorization. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security*. ACM Press, New York, NY, USA, 168–179.
- WOO, T. Y. C. AND LAM, S. S. 1993. Authorizations in distributed systems: A new approach. *Journal of Computer Security* 2, 2-3, 107–136.
- ZHANG, N., RYAN, M. D., AND GUELEV, D. 2005. Evaluating access control policies through model checking. In *Eighth Information Security Conference (ISC05)*.
- ZHAO, C., HEILILI, N., LIU, S., AND LIN, Z. 2005. Representation and reasoning on rbac: A description logic approach. In *ICTAC*. 381–393.