Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Automated Verification of Access Control Policies Using a SAT Solver ⋆

Graham Hughes, Tevfik Bultan

Computer Science Department, University of California, Santa Barbara, CA 93106, USA e-mail: {graham,bultan}@cs.ucsb.edu

August 15, 2008

Abstract. Managing access control policies in modern computer systems can be challenging and error-prone. Combining multiple disparate access policies can introduce unintended consequences. In this paper we present a formal model for specifying access to resources, a model that encompasses the semantics of the XACML access control language. From this model we define several ordering relations on access control policies that can be used to automatically verify properties of the policies. We present a tool for automatically verifying these properties by translating these ordering relations to Boolean satisfiability problems and then applying a SAT solver. Our experimental results demonstrate that automated verification of XACML policies is feasible using this approach.

Key words: access control, automated verification

1 Introduction

A major problem in modern software systems is keeping track of which users are permitted access to shared resources. Nowadays, Web-based applications are used to access all types of sensitive information such as bank accounts, employee records and even health records. Given the ease of access provided by the Web, it is crucial to provide access control mechanisms for such applications that deal with sensitive information. Moreover, due to the increasing use of service oriented architectures, it is necessary to develop techniques for keeping the access control policies consistent across heterogeneous systems and applications spanning multiple organizations. Although effectively enforcing the access control rules within a single application is already challenging, keeping the access control policies consistent across multiple heterogeneous systems, each with their own specific access control language, is even more difficult. Several unified access policy languages attempt to guarantee consistency in such situations. In this paper we focus on one particular such language, the OASIS standard XACML [34].

OASIS (the Organization for the Advancement of Structured Information Standards) is an international standards consortium that publishes, among others, standards based on the popular markup language XML [35]. The standard that we are concerned with in this work is the "eXtensible Access Control Markup Language" (abbreviated XACML), an XML-based language for expressing access rights to arbitrary objects that are identified in XML, with a particular focus on the composition of many individual policies into a single disparate "super-policy". Having such a combined policy is useful for eliminating inconsistencies among separate policies and for achieving a uniform access control mechanism, but such a policy will inevitably become increasingly large and complex as it incorporates all the varied access rules different applications and organizations may have. It is possible, even likely, that the act of creating a unified super-policy out of several smaller policies could have unintended consequences.

In this paper we investigate static verification of access control policies, with the goal of preventing such errors. We first translate XACML policies into a simplified mathematical model, which we reduce to a normal form separating the conditions that give rise to three different classes of results: *access permitted, access denied,* and *internal error*. We define several partial orderings between access control policies, which we can use to automatically check whether a policy is over- or under-constrained with respect to another one. We show that these ordering relations can be translated to Boolean formulas which are satisfiable if and only if the corresponding relation is violated. We use a SAT solver to check the satisfiability of these Boolean logic formulas. Using our translator and a SAT solver we can check if a combination of XACML policies does or does not faithfully reproduce the properties of its sub-policies, and thus discover unintended consequences before they appear in practice.

⋆ This work is supported by NSF grants CCF-0614002 and CCF-0716095.

Although we accommodate more of the XACML language than previous efforts do, which we discuss in Sections 5.1 and 5.2, our approach also has some limitations. The first limitation is due to the fact that we perform bounded analysis. Since XACML includes several unbounded domains including strings of characters and integers, this can introduce imprecision. For example our approach may miss errors that only occur with a larger domain size than the bound used in our analysis, which can lead to false negatives.

A second cause of imprecision is due to abstraction. The XACML language includes several functions that are extremely complex or in some cases cannot be encoded as a Boolean logic formula. In these cases we use unconstrained Boolean predicates to abstract these functions. When such abstractions are used, our analysis may produce false positives, i.e. report errors that may not exist in the specification. In these cases the user may need to validate the reported errors manually. We discuss these two limitations in detail in Section 4.2. Note that for finite state specifications our approach is sound and complete as long as the user chooses a sufficiently large bound and the complex XACML functions are not used in the specification.

Finally, for ease of analysis we use slightly simplified versions of some of the XACML policy combining algorithms. The differences between the combining algorithms that we use and the XACML semantics are discussed in Section 2.2. Despite these limitations we have successfully performed analysis on several XACML policies, which we detail in Section 5.

We have organized our paper as follows: in Section 2, after giving an overview of XACML, we develop a formal model for access policies. In Section 2.3 we discuss how to transform these models into a normal form that distinguishes access permitted, access denied, and internal error conditions. In Section 3 we define several partial order relations among access policies, which we use to specify their properties. We show how to check these properties automatically in Section 4. Finally, we report the results of experiments using our tool in Section 5. We also compare our tool with two other approaches: first, Fisler et al's Margrave tool [12]; and second an approach based on translation to Alloy [17]. We discuss the related work in Section 6 and conclude the paper in Section 7.

```
<?xml version="1.0" encoding="UTF-8"?>
<Policy
  xmlns="urn:..."
  xmlns:xsi="...-instance"
  xmlns:md="http://www.medico.com/schemas/record.xsd"
  PolicySetId="urn:example:policyid:1"
  RuleCombiningAlgId="urn:...:deny-overrides">
  <Target>
    <Subjects><AnySubject/></Subjects>
    <Resources><AnyResource/></Resources>
    <Actions>
      <Action>
        <ActionMatch MatchId="urn:...:string-equal">
          <AttributeValue DataType="...#string">
            vote
          </AttributeValue>
          <ActionAttributeDesignator
            AttributeId="urn:example:action"
            DataType="...#string"/>
        </ActionMatch>
      </Action>
    </Actions>
  </Target>
  <Rule RuleId="urn:example:ruleid:1" Effect="Deny">
    <Condition FunctionId="urn:...:integer-less-than">
      <Apply FunctionId="urn:...:integer-one-and-only">
        <SubjectAttributeDesignator
          AttributeId="urn:example:age"
          DataType="...#integer"/>
      </Apply>
      <AttributeValue DataType="...#integer">
        18
      </AttributeValue>
    </Condition>
  </Rule>
  <Rule RuleId="urn:example:ruleid:2" Effect="Deny">
    <Condition FunctionId="urn:...:boolean-equal">
      <Apply FunctionId="urn:...:boolean-one-and-only">
        <SubjectAttributeDesignator
          AttributeId="urn:example:voted-yet"
          DataType="...#boolean"/>
      </Apply>
      <AttributeValue DataType="...#boolean">
        True
      </AttributeValue>
    </Condition>
  </Rule>
  <Rule RuleId="urn:example:ruleid:3" Effect="Permit"/>
</Policy>
```

Fig. 1. A simple XACML policy

2 Policy Specifications

XACML is an OASIS standard for specifying access policies. XACML policies are written in XML, and typically authored using a dedicated policy editor. The language describes three classes of objects—individual rules, collections of rules called policies, and collections of policies called policy sets. An XACML Policy Enforcement Point, the gateway that determines whether an action is permitted or not, takes *access requests*, which are specially formatted XML documents that define a set of data that we call the *environment*. Policy Enforcement Points yield one of four results: Permit, meaning that the access request is permitted; Deny, meaning that the access request will not be permitted; Not Applicable, meaning that this particular policy says nothing about the request; and Indeterminate, which means that something unexpected has occurred and the execution of the policy has failed. Which result occurs depends on what result the policy dictates, given the environment defined in the access request.

XACML rules, the most basic component of a policy, have a goal effect (either Permit or Deny), a domain of applicability, and conditions under which they can yield Indeterminate and fail. The domain of applicability is realized in a series of predicates about the environmental data that must all be satisfied for the rule to yield its goal effect. The error conditions are embedded in the domain predicates, but can be separated out into a set of predicates all their own. Policy sets combine individual policies with a domain of applicability.

XACML predicates comprise one of a number of primitive functions, with mechanisms for extension (we do not consider extensions in this work). These functions include simple equality, set inclusion, ordering within numeric types, and also more complex functions such as XPath matching and X500 name matching.

Let us consider a simple example policy. The policy states that to be able to vote a person must be at least 18 years old and a person who has voted already cannot vote. Our environment, the set of information we are interested in, consists of the age of the person in question and whether they have voted already. We can represent this as a Cartesian product of the power sets of XML Schema [36] basic types, as follows:

$$E = \mathcal{P}(\texttt{xsd:int}) \times \mathcal{P}(\texttt{xsd:boolean}) \times \mathcal{P}(\texttt{xsd:string}) \quad (1)$$

The first component of the environment E is the age of the person, the second component is whether or not they have voted already, and the third component is the action they are attempting—perhaps voting, but perhaps something else. The use of power sets is due to XACML semantics. In XACML an attribute always describes a set of values, never a scalar value. A scalar value, for example the age of a person, is represented as a singleton set.

The XACML policy for this example is shown in Figure 1. The full XACML syntax is cumbersome for discussing our techniques; accordingly we will explain the semantics of this policy using a simple mathematical notation below. Our tool accepts the policies in the XML input format defined by the XACML specification.

We illustrate our notation using our example policy. The goal for our example policy is that if a person is doing something other than voting, we do not really care what happens, and we require that there be only one age and one voting record presented. To do this we can divide E into four sets, E_a, E_v, E_p and E_d as follows, using the notation $\exists! x P$ to assert that there is a unique x that satisfies a condition P:

$$E_a = \{\langle a, v, o\rangle \in E : \exists! a_0 : a_0 \in a \wedge \exists! v_0 : v_0 \in v\}$$
$$E_v = \{\langle a, v, o\rangle \in E_a : \exists x \in o : x = \texttt{vote}\}$$
$$E_p = \{\langle \{a_0\}, \{v_0\}, o\rangle \in E_v : a_0 \geq 18 \wedge \neg v_0\}$$
$$E_d = E_v \setminus E_p = \{\langle \{a_0\}, \{v_0\}, o\rangle \in E_v : a_0 < 18 \vee v_0\}$$

Here, E_a is the set of all environments whose inputs are not erroneous, E_v is the set of all environments where voting is attempted, E_p is the set of all environments where the person can vote (their attempt to vote is *permitted*), and E_d is the set of all environments where the person cannot vote (their attempt to vote is *denied*). In the following section we will define a concise formal model for XACML policies and express our example policy in this formal model.

2.1 Formal Model

Let $R = \{\mathit{Permit}, \mathit{Deny}, \mathit{NotApp}, \mathit{Indet}\}$ be the set of valid results. Now, we can define the set of valid policies P as follows (with the semantics defined later):

$$\mathit{Permit} \in P$$
$$\mathit{Deny} \in P$$
$$\forall p \in P : \forall S \subseteq E : \mathit{Scope}(p, S) \in P$$
$$\forall p \in P : \forall S \subseteq E : \mathit{Err}(p, S) \in P$$
$$\forall p, q \in P : p \oplus q \in P$$
$$\forall p, q \in P : p \ominus q \in P$$
$$\forall p, q \in P : p \otimes q \in P$$
$$\forall p, q \in P : p \oslash q \in P$$

Informally, we regard *Permit* and *Deny* as symbols whose semantics ignore the environment, always yielding *Permit* or *Deny*, respectively. Along these same lines, *Scope* and *Err* attach conditions to policies:

- $\mathit{Scope}(p, S)$ modifies policy p to yield p's answer if the current environment is in S, or *NotApp* otherwise.
- $\mathit{Err}(p, S)$ yields *Indet* if the current environment is in S or p's answer otherwise.

The other four symbols ($\oplus, \ominus, \otimes, \oslash$) are combinators, that combine two policies in various ways:

- **Permit-overrides:** $p \oplus q$ always yields *Permit* if either p or q yield *Permit*.
- **Deny-overrides:** $p \ominus q$ always yields *Deny* if either p or q yield *Deny*.
- **Only-one-applicable:** $p \otimes q$ requires that one of p or q yield *NotApp* and then yields the other half's answer.
- **First-applicable:** $p \oslash q$ yields p's answer unless that answer is *NotApp*, in which case it yields q's answer.

To formalize the semantics of policies, we define a function $\mathrm{eff} : E \times P \to R$ that, given an environment and a policy produces a result. We will use this function to define the result indicated by a policy for any given environment, but also to define semantics-preserving transformations later. We define this function in Figure 2 so that it corresponds to the intuitive semantics we described for the policies above. To ease presentation of this function, we define two ordering relations on results $>_\oplus$ and $>_\ominus$. We define these ordering relations as follows:

$$\mathit{Permit} >_\oplus \mathit{Indet} >_\oplus \mathit{Deny} >_\oplus \mathit{NotApp}$$
$$\mathit{Deny} >_\ominus \mathit{Indet} >_\ominus \mathit{Permit} >_\ominus \mathit{NotApp}$$

Using these ordering relations we use $\sup_{>_\oplus} S$ to mean the supremum of the set S using the $>_\oplus$ ordering, and similarly for $\sup_{>_\ominus}$. For example, $\sup_{>_\oplus} \{\mathit{Deny}, \mathit{NotApp}\} = \mathit{Deny}$ and $\sup_{>_\ominus} \{\mathit{Permit}, \mathit{Deny}\} = \mathit{Deny}$.

Using this notation, we can now model our example as follows:

$$S_0 = \{\langle a, v, o\rangle \in E : \forall x \in a : x < 18\} \quad (2)$$
$$S_1 = \{\langle a, v, o\rangle \in E : \forall x \in v : x\} \quad (3)$$
$$S_2 = \{\langle a, v, o\rangle \in E : \exists x \in o : x = \texttt{vote}\} \quad (4)$$
$$S_3 = \{\langle a, v, o\rangle \in E : \neg\exists! a_0 : a_0 \in a\} \quad (5)$$
$$S_4 = \{\langle a, v, o\rangle \in E : \neg\exists! v_0 : v_0 \in v\} \quad (6)$$
$$r_1 = \mathit{Err}(\mathit{Scope}(\mathit{Deny}, S_0), S_3) \quad (7)$$
$$r_2 = \mathit{Err}(\mathit{Scope}(\mathit{Deny}, S_1), S_4) \quad (8)$$
$$p = \mathit{Scope}(r_1 \ominus r_2 \ominus \mathit{Permit}, S_2) \quad (9)$$

$$\textit{Permit} >_{\oplus} \textit{Indet} >_{\oplus} \textit{Deny} >_{\oplus} \textit{NotApp}$$
$$\textit{Deny} >_{\ominus} \textit{Indet} >_{\ominus} \textit{Permit} >_{\ominus} \textit{NotApp}$$

$$\begin{aligned}
\text{eff} &: E \times P \to R \\
\text{eff}(e, \textit{Permit}) &= \textit{Permit} \\
\text{eff}(e, \textit{Deny}) &= \textit{Deny} \\
\text{eff}(e, \textit{Scope}(p, S)) &= \begin{cases} \text{eff}(e,p) & \text{if } e \in S \\ \textit{NotApp} & \text{otherwise} \end{cases} \\
\text{eff}(e, \textit{Err}(p, S)) &= \begin{cases} \textit{Indet} & \text{if } e \in S \\ \text{eff}(e,p) & \text{otherwise} \end{cases} \\
\text{eff}(e, p \oplus q) &= \sup_{>_{\oplus}} \{\text{eff}(e,p), \text{eff}(e,q)\} \\
\text{eff}(e, p \ominus q) &= \sup_{>_{\ominus}} \{\text{eff}(e,p), \text{eff}(e,q)\} \\
\text{eff}(e, p \otimes q) &= \begin{cases} \text{eff}(e,p) & \text{if eff}(e,q) = \textit{NotApp} \\ \text{eff}(e,q) & \text{if eff}(e,p) = \textit{NotApp} \\ \textit{Indet} & \text{otherwise} \end{cases} \\
\text{eff}(e, p \oslash q) &= \begin{cases} \text{eff}(e,p) & \text{if eff}(e,p) \neq \textit{NotApp} \\ \text{eff}(e,q) & \text{otherwise} \end{cases}
\end{aligned}$$

Fig. 2. Semantics of policies

Here, S_0 is the set of environments that fail the age requirement (corresponding to lines 25–34 of Fig. 1), S_1 is the set of environments that fail the voting requirement (corresponding to lines 37–46 of Fig. 1), S_2 is the set of environments where someone's trying to vote (corresponding to lines 9–22 of Fig. 1), S_3 represents the uniqueness constraint on ages (corresponding to lines 26–30 of Fig. 1), S_4 represents the uniqueness constraint on whether or not the user has voted (corresponding to lines 38–42 of Fig. 1), r_1 represents the rule "urn:example:ruleid:1" (corresponding to lines 24–35 of Fig. 1), r_2 represents the rule "urn:example:ruleid:2" (corresponding to lines 36–47 of Fig. 1) and p represents the whole policy (with the scoping information defined on lines 8–23 of Fig. 1, and the rule combining algorithm defined on line 7 of Fig. 1).

2.2 XACML Combinators

Although our definition of $\otimes$ and $\oslash$ are identical to the corresponding policy combinators from the XACML language, our definition of $\oplus$ and $\ominus$ differ slightly from the XACML specification. In the case that there are no errors, that is there are no *Indet* results, then the operators $\oplus$ and $\ominus$ are identical to the combinator algorithms given in the XACML specification. If *Indet* results are considered, then the permit-overrides and deny-overrides algorithms in the XACML specification differ depending on whether they are applied to rules or applied to policies. As well, the policy combining algorithms for deny-overrides and permit-overrides are not symmetric. We denote the XACML versions of these combinators by using $\oplus_R$ and $\oplus_P$ for permit-overrides and $\ominus_R$ and $\ominus_P$ for deny-overrides where subscript R denotes the combinators on rules and P denotes the combinators on policies. In Figure 3 we show how these operators can be expressed using the formalism we introduced above. We will discuss this translation in more detail below; however, note that, the semantics of the operators $\oplus_R$, $\oplus_P$, $\ominus_R$ and $\ominus_P$ are considerably more complex than $\oplus$ and $\ominus$. After presenting these operators, we will discuss how they differ from $\oplus$ and $\ominus$.

To simplify our presentation of $\oplus_R$, $\oplus_P$, $\ominus_R$ and $\ominus_P$ in Figure 3, we use one auxiliary result $\textit{Indet}^{\bullet}$, two sets $P_{\oplus}, P_{\ominus} \subset P$, and four auxiliary collation functions c_S, $c_{\oplus}$, $c_{\ominus_R}$ and $c_{\ominus_P}$. $\textit{Indet}^{\bullet}$ is not normally a legal result, and can only arise through the auxiliary function $c_{\oplus}$ and $c_{\ominus_R}$ and is stripped out with c_S. We must amend the ordering relations to accommodate this result, and we do so as follows:

$$\textit{Permit} >_{\oplus} \textit{Indet} >_{\oplus} \textit{Deny} >_{\oplus} \textit{Indet}^{\bullet} >_{\oplus} \textit{NotApp}$$
$$\textit{Deny} >_{\ominus} \textit{Indet} >_{\ominus} \textit{Permit} >_{\ominus} \textit{Indet}^{\bullet} >_{\ominus} \textit{NotApp}$$

We use the subsets of P $P_{\oplus}$ and $P_{\ominus}$ to encode the XACML concept of a rule with an associated outcome. We have conflated rules with policies for simplicity but they are distinct in XACML language specification. $P_{\oplus}$ is just the set of policies that are *Permit* wrapped in *Scope* and *Err* declarations, and $P_{\ominus}$ is similarly the set of policies that are *Deny* wrapped in *Scope* and *Err* declarations.

The auxiliary collation functions $c_{\oplus}$ and $c_{\ominus_R}$ handle an unusual feature of $\oplus_R$ and $\ominus_R$. During the calculation of the result of $\oplus_R$, if the result of a rule is *Indet* but the rule itself would otherwise yield *Deny*, the result is defined to be $\textit{Indet}^{\bullet}$ instead. Otherwise processing continues normally. The auxiliary collation function $c_{\ominus_P}$ is used for the definition of $\ominus_P$, wherein an *Indet* intermediate result will yield a *Deny* for the whole policy; accordingly we map *Indet* to *Deny*.

Finally with these preliminary operators defined we may begin defining $\oplus_R$ and the like. The definition for $\oplus_R$ is in essence "if the first policy yields *Permit*, then permit; otherwise if the second policy yields *Permit*, then permit; otherwise if either policy yields *Indet* and said policy is a rule with effect *Permit* then yield indeterminate; otherwise if either policy yields *Deny* then deny; otherwise if either policy yields *Indet* and said policy is not a rule with effect *Permit* then yield indeterminate; otherwise both policies must yield *NotApp* so yield not applicable". The definition for $\ominus_R$ is symmetrical; "if the first policy yields *Deny*, then deny; otherwise if the second policy yields *Deny*, then deny; otherwise if either policy yields *Indet* and said policy is a rule with effect *Deny* then yield indeterminate; otherwise if either policy yields *Permit* then permit; otherwise if either policy yields *Indet* and said policy is not a rule with effect *Deny* then yield indeterminate; otherwise both policies must yield *NotApp* so yield not applicable".

The definition for policy combining rules is slightly simpler. The definition for $\oplus_P$ is in essence "if either policy yields *Permit*, then permit; if either policy yields *Indet*, then yield indeterminate; if either policy yields *Deny*, then deny; otherwise yield not applicable". The definition for $\ominus_P$ is in essence "if either policy yields *Deny*, then deny; if either policy yields

$$\mathit{Permit} >_{\oplus} \mathit{Indet} >_{\oplus} \mathit{Deny} >_{\oplus} \mathit{Indet}^{\bullet} >_{\oplus} \mathit{NotApp}$$
$$\mathit{Deny} >_{\ominus} \mathit{Indet} >_{\ominus} \mathit{Permit} >_{\ominus} \mathit{Indet}^{\bullet} >_{\ominus} \mathit{NotApp}$$

$$\begin{aligned}
\mathit{Permit} &\in P_{\oplus} \\
\forall p \in P_{\oplus} : \forall S \subseteq E : \mathit{Scope}(p, S) &\in P_{\oplus} \\
\forall p \in P_{\oplus} : \forall S \subseteq E : \mathit{Err}(p, S) &\in P_{\oplus} \\
\mathit{Deny} &\in P_{\ominus} \\
\forall p \in P_{\ominus} : \forall S \subseteq E : \mathit{Scope}(p, S) &\in P_{\ominus} \\
\forall p \in P_{\ominus} : \forall S \subseteq E : \mathit{Err}(p, S) &\in P_{\ominus}
\end{aligned}$$

$$c_S(r) = \begin{cases} \mathit{Indet} & \text{if } r = \mathit{Indet}^{\bullet}, \\ r & \text{otherwise.} \end{cases}$$

$$c_{\oplus}(p, r) = \begin{cases} \mathit{Indet}^{\bullet} & \text{if } r = \mathit{Indet} \text{ and } p \notin P_{\oplus}, \\ r & \text{otherwise.} \end{cases}$$

$$c_{\ominus_R}(p, r) = \begin{cases} \mathit{Indet}^{\bullet} & \text{if } r = \mathit{Indet} \text{ and } p \notin P_{\ominus}, \\ r & \text{otherwise.} \end{cases}$$

$$c_{\ominus_P}(r) = \begin{cases} \mathit{Deny} & \text{if } r = \mathit{Indet}, \\ r & \text{otherwise.} \end{cases}$$

$$\begin{aligned}
\mathrm{eff}(e, \oplus_R(p_0, \ldots, p_n)) &= c_S\left(\sup_{>_{\oplus}} \{0 \le i \le n : c_{\oplus}(p_i, \mathrm{eff}(e, p_i))\}\right) \\
\mathrm{eff}(e, \ominus_R(p_0, \ldots, p_n)) &= c_S\left(\sup_{>_{\ominus}} \{0 \le i \le n : c_{\ominus_R}(p_i, \mathrm{eff}(e, p_i))\}\right) \\
\mathrm{eff}(e, \oplus_P(p_0, \ldots, p_n)) &= \sup_{>_{\oplus}} \{0 \le i \le n : \mathrm{eff}(e, p_i)\} \\
\mathrm{eff}(e, \ominus_P(p_0, \ldots, p_n)) &= \sup_{>_{\ominus}} \{0 \le i \le n : c_{\ominus_P}(\mathrm{eff}(e, p_i))\}
\end{aligned}$$

Fig. 3. XACML semantics for permit-overrides and deny-overrides combinators

Indet, then deny; if either policy yields *Permit*, then permit; otherwise yield not applicable". These definitions are not symmetric; in particular combining policies that yield *Indet* using a deny-overrides policy combinator yields *Deny*, but combining policies that yield *Indet* using a permit-overrides policy combinator yields *Indet*. This asymmetry appears to arise from a domain requirement; if at any time the result of an XACML policy cannot be proven to be *Permit*, the enforcement machinery should reject the request.

The complex semantics of XACML policy combinators does not make analysis impossible but it does make it more cumbersome. We have defined our combinators differently to simplify the implementation of our analysis. In the case that the policies being combined do not yield *Indet*—which is hopefully an exceptional occurrence—our combinators give precisely the same result. This can be shown by noting the following: c_S, $c_{\oplus}$, $c_{\ominus_R}$ and $c_{\ominus_P}$ are all equivalent to the identity if no result is ever *Indet*. Therefore in the absence of *Indet*

$$\begin{aligned}
\mathrm{eff}(e, \oplus_R(p_0, \ldots, p_n)) &= \mathrm{eff}(e, \oplus_P(p_0, \ldots, p_n)) \\
&= \sup_{>_{\oplus}} \{0 \le i \le n : \mathrm{eff}(e, p_i)\}
\end{aligned}$$

and

$$\begin{aligned}
\mathrm{eff}(e, \ominus_R(p_0, \ldots, p_n)) &= \mathrm{eff}(e, \ominus_P(p_0, \ldots, p_n)) \\
&= \sup_{>_{\ominus}} \{0 \le i \le n : \mathrm{eff}(e, p_i)\}
\end{aligned}$$

Because we are dealing with finite sets, $\sup\{p_1, \ldots, p_n\} = \sup\{p_1, \sup\{p_2, \ldots \sup\{p_n\}\}\}$. Therefore

$$\begin{aligned}
&\mathrm{eff}(e, \oplus_R(p_0, \ldots, p_n)) = \\
&\quad \sup_{>_{\oplus}}\{\mathrm{eff}(e, p_0), \sup_{>_{\oplus}}\{\mathrm{eff}(e, p_1), \ldots \sup_{>_{\oplus}}\{\mathrm{eff}(e, p_n)\}\}\}
\end{aligned}$$

and similarly for $\ominus_R$, $\oplus_P$ and $\ominus_P$. But these precisely the form of $\oplus$ and $\ominus$: therefore

$$\mathrm{eff}(e, \oplus_R(p_0, \ldots, p_n)) = p_0 \oplus \cdots \oplus p_n$$

$$
\begin{aligned}
f &: P \to P \\
f(\mathit{Scope}(\mathit{Scope}(X,S),R)) &= f(\mathit{Scope}(X, R \cap S)) \\
f(\mathit{Scope}(\mathit{Err}(X,S),R)) &= f(\mathit{Err}(\mathit{Scope}(X, R \setminus S), S \cap R)) \\
f(\mathit{Scope}(X \oplus Y, S)) &= \mathit{Scope}(f(X),S) \oplus \mathit{Scope}(f(Y),S) \\
f(\mathit{Scope}(X \ominus Y, S)) &= \mathit{Scope}(f(X),S) \ominus \mathit{Scope}(f(Y),S) \\
f(\mathit{Scope}(X \otimes Y, S)) &= \mathit{Scope}(f(X),S) \otimes \mathit{Scope}(f(Y),S) \\
f(\mathit{Scope}(X \oslash Y, S)) &= \mathit{Scope}(f(X),S) \oslash \mathit{Scope}(f(Y),S) \\
f(\mathit{Scope}(P,S)) &= \mathit{Scope}(f(P),S) \\
&\quad \text{if no other rules apply} \\
f(\mathit{Err}(\mathit{Err}(X,S),R)) &= f(\mathit{Err}(X, R \cup S)) \\
f(\mathit{Err}(\mathit{Scope}(X,S),R)) &= f(\mathit{Err}(\mathit{Scope}(X, S \setminus R), R)) \\
&\quad \text{if } S \cap R \neq \emptyset \\
f(\mathit{Err}(X \oplus Y, S)) &= \mathit{Err}(f(X),S) \oplus \mathit{Err}(f(Y),S) \\
f(\mathit{Err}(X \ominus Y, S)) &= \mathit{Err}(f(X),S) \ominus \mathit{Err}(f(Y),S) \\
f(\mathit{Err}(X \otimes Y, S)) &= \mathit{Err}(f(X),S) \otimes \mathit{Err}(f(Y),S) \\
f(\mathit{Err}(X \oslash Y, S)) &= \mathit{Err}(f(X),S) \oslash \mathit{Err}(f(Y),S) \\
f(\mathit{Err}(P,S)) &= \mathit{Err}(f(P),S) \\
&\quad \text{if no other rules apply} \\
f(\mathit{Permit}) &= \mathit{Permit} \\
f(\mathit{Deny}) &= \mathit{Deny}
\end{aligned}
$$

Fig. 4. eff-preserving transformations for reduction to normal form

and similarly for $\ominus_R$ and the others. Therefore our policy combinators are the same as the official ones in the absence of *Indet*. If a policy does yield *Indet* our combinators will tend to propagate that to the root of a policy, whereas the official combinators will perform more complicated processing, apparently with the intent of recovering from an error. We would like to note that the verification techniques we use are not dependent on our combinators. We could use the official versions by extending our implementation. For the policies we investigated in our experiments, there is no difference between the three versions of the permit-overrides operators $\oplus_R$, $\oplus_P$ and $\oplus$, and the three versions of the deny-overrides operators $\ominus_R$, $\ominus_P$ and $\ominus$.

2.3 Policy Transformations

Now that we have defined a formal model for policies, we would like to analyze them, and it would be easier to do the analysis if we could bring the model into a normal form. To do this, first we define an equivalence relation:

$$P_1 \equiv P_2 \text{ iff } \forall e \in E : \mathrm{eff}(e, P_1) = \mathrm{eff}(e, P_2)$$

We call a function f that takes a policy and returns another policy an eff*-preserving transformation* if $\forall p \in P : f(p) \equiv p$.

For any given policy, we want to regard the subset of E that will give a *Permit* result, the subset of E that will give a *Deny* result, and the subset of E that will give an *Indet* result independently. We define a shorthand $\langle S, R, T \rangle$, where S, R and T are pairwise disjoint, as follows:

$$\langle S, R, T \rangle = \mathit{Err}(\mathit{Scope}(\mathit{Permit}, S) \otimes \mathit{Scope}(\mathit{Deny}, R), T)$$

Hence, $\langle S, R, T \rangle$ is simply a policy that yields *Permit* for any environment in S, *Deny* for any environment in R, *Indet* for any environment in T, and *NotApp* for any remaining environment. We call this *triple notation* and refer to individual nodes $\langle S, R, T \rangle$ as *triples*.

Now that we have a framework for transforming policies, we would like to transform an entire policy with *Scope*, *Err* and combinators alike into a single triple. We know that for any policy p a triple p_T that is equivalent to it exists: the triple is just

$$
\begin{aligned}
p_T = \langle &\{e \in E : \mathrm{eff}(e,p) = \mathit{Permit}\}, \\
&\{e \in E : \mathrm{eff}(e,p) = \mathit{Deny}\}, \\
&\{e \in E : \mathrm{eff}(e,p) = \mathit{Indet}\} \rangle.
\end{aligned}
$$

However, this is not a constructive definition. To transform the policies to the triple form, we define two functions $f : P \to P$ and $g : P \to \langle S, R, T \rangle$, both eff-preserving transformations, such that $g(f(p))$ is a triple representation for the policy p. The f function transforms the policy into an equivalent one that is composed of triples joined by combinators. The g function combines triples joined by combinators into a single triple. The two together generate the triple representation. We define f in Figure 4, and g in Figure 5.

As an example, applying f to the policy p defined in Equation (9) leads to the following:

$$
\begin{aligned}
p = {} & \mathit{Scope}(\mathit{Err}(\mathit{Scope}(\mathit{Deny}, S_0), S_3) \\
& \quad \ominus \mathit{Err}(\mathit{Scope}(\mathit{Deny}, S_1), S_4) \\
& \quad \ominus \mathit{Permit}, S_2) \\
f(p) = {} & \mathit{Err}(\mathit{Scope}(\mathit{Deny}, (S_2 \cap S_0) \setminus S_3), S_3 \cap S_2) \\
& \quad \ominus \mathit{Err}(\mathit{Scope}(\mathit{Deny}, (S_2 \cap S_1) \setminus S_4), S_4 \cap S_2) \\
& \quad \ominus \mathit{Scope}(\mathit{Permit}, S_2)
\end{aligned}
$$

Note that the function f pushes all *Scope* forms down to the leaves of the policy tree, and all *Err* forms down to just above the leaves.

The f function transforms a policy to a collection of expressions of the form $\mathit{Err}(\mathit{Scope}(A, B), T)$ (where $A \in \{\mathit{Permit}, \mathit{Deny}\}$, $B \subseteq E$, $T \subseteq E$, and $B \cap T = \emptyset$) combined using $\oplus, \ominus, \otimes$ and $\oslash$. Since $\forall e \in E : \mathrm{eff}(e, X \otimes \mathit{Scope}(Y, \emptyset)) = \mathrm{eff}(e, X)$, we can rewrite these expressions further into the form

$$\mathit{Err}(\mathit{Scope}(\mathit{Permit}, S) \otimes \mathit{Scope}(\mathit{Deny}, R), T)$$

combined with $\oplus, \ominus, \otimes$ and $\oslash$ where $S = B$ and $R = \emptyset$ if $A = \mathit{Permit}$ and $S = \emptyset$ and $R = B$ if $A = \mathit{Deny}$. Since S, R and T are all pairwise disjoint this is exactly the required form for our triple notation. Hence, after applying the function f we have a set of subpolicies in our triple notation combined with $\oplus, \ominus, \otimes$ and $\oslash$. We define the function g in Figure 5. The transformations for function g all preserve the disjointness

$$
\begin{aligned}
g &: P \to \langle S, R, T\rangle \\
g(\langle S_1,R_1,T_1\rangle \oplus \langle S_2,R_2,T_2\rangle) &= \langle S_1 \cup S_2, (R_1 \setminus (S_2 \cup T_2)) \cup (R_2 \setminus (S_1 \cup T_1)), (T_1 \cup T_2) \setminus (S_1 \cup S_2)\rangle \\
g(\langle S_1,R_1,T_1\rangle \ominus \langle S_2,R_2,T_2\rangle) &= \langle (S_1 \setminus (R_2 \cup T_2)) \cup (S_2 \setminus (R_1 \cup T_1)), R_1 \cup R_2, (T_1 \cup T_2) \setminus (R_1 \cup R_2)\rangle \\
g(\langle S_1,R_1,T_1\rangle \otimes \langle S_2,R_2,T_2\rangle) &= \langle (S_1 \cup S_2) \setminus ((S_1 \cap S_2) \cup T_1 \cup T_2), (R_1 \cup R_2) \setminus ((R_1 \cap R_2) \cup T_1 \cup T_2), \\
&\qquad T_1 \cup T_2 \cup (S_1 \cap S_2) \cup (R_1 \cap R_2)\rangle \\
g(\langle S_1,R_1,T_1\rangle \oslash \langle S_2,R_2,T_2\rangle) &= \langle S_1 \cup (S_2 \setminus (R_1 \cup T_1)), R_1 \cup (R_2 \setminus (S_1 \cup T_1)), T_1 \cup (T_2 \setminus (S_1 \cup R_1))\rangle \\
g(\langle S_1,R_1,T_1\rangle) &= \langle S_1,R_1,T_1\rangle \\
g(P_1 \oplus P_2) &= g(g(P_1) \oplus g(P_2)) \quad \text{if no other rules apply} \\
g(P_1 \ominus P_2) &= g(g(P_1) \ominus g(P_2)) \quad \text{if no other rules apply} \\
g(P_1 \otimes P_2) &= g(g(P_1) \otimes g(P_2)) \quad \text{if no other rules apply} \\
g(P_1 \oslash P_2) &= g(g(P_1) \oslash g(P_2)) \quad \text{if no other rules apply}
\end{aligned}
$$

Fig. 5. eff-preserving transformations for $\langle S, R, T\rangle$ reduction

property, and using the function g we can transform the policy generated by function f to a single triple $\langle S, R, T\rangle$ for some $S, R, T \subseteq E$.

When we apply the function g to our example we get the following:

$$
\begin{aligned}
f(p) = {} & \mathit{Err}(\mathit{Scope}(\mathit{Deny}, (S_2 \cap S_0) \setminus S_3), S_3 \cap S_2) \\
& \ominus \mathit{Err}(\mathit{Scope}(\mathit{Deny}, (S_2 \cap S_1) \setminus S_4), S_4 \cap S_2) \\
& \ominus \mathit{Scope}(\mathit{Permit}, S_2) \\
= {} & \langle \emptyset, (S_2 \cap S_0) \setminus S_3, S_3 \cap S_2\rangle \\
& \ominus \langle \emptyset, (S_2 \cap S_1) \setminus S_4, S_4 \cap S_2\rangle \\
& \ominus \langle S_2, \emptyset, \emptyset\rangle \\
g(f(p)) = {} & \langle S_2 \setminus (S_0 \cup S_1 \cup S_3 \cup S_4), \\
& ((S_0 \setminus S_3) \cup (S_1 \setminus S_4)) \cap S_2, \\
& ((S_3 \cup S_4) \setminus ((S_0 \setminus S_3) \cup (S_1 \setminus S_4))) \cap S_2\rangle
\end{aligned}
$$

Now that we have our policy in a form that is convenient for analysis, we would like to check it.

3 Properties of Policies

In order to check policies, we first need to figure out what sort of properties we wish to check. For this purpose, we have chosen to define several partial ordering relations that can be used to relate policies. We can specify policies using a normal XACML policy editor, and then automatically determine whether they are related in the desired manner using our analysis tool. For example, we might have a large policy composed of numerous sub-policies that we have difficulty comprehending all at once. We might want to prove that this comprehensive policy protects some resource at least as much as some simpler policy does. Similarly we might want to guarantee that the act of combining several sub-policies does not lead this new, larger policy to have a scope greater than any of its components. We can express properties like these using the ordering relations defined below.

Let $P_1 = \langle S_1, R_1, T_1\rangle$ and let $P_2 = \langle S_2, R_2, T_2\rangle$ be two policies. We define the following partial orders:

$$
\begin{aligned}
P_1 \sqsubseteq_P P_2 &\equiv S_1 \subseteq S_2 \\
P_1 \sqsubseteq_D P_2 &\equiv R_1 \subseteq R_2 \\
P_1 \sqsubseteq_E P_2 &\equiv T_1 \subseteq T_2 \\
P_1 \sqsubseteq_{P,D,E} P_2 &\equiv P_1 \sqsubseteq_P P_2 \wedge P_1 \sqsubseteq_D P_2 \wedge P_1 \sqsubseteq_E P_2
\end{aligned}
$$

Note that, we can define a partial order for any combination of P, D and E. We define $P_1 \sqsubseteq P_2 \equiv P_1 \sqsubseteq_{P,D,E} P_2$. We can regard $P_1 \sqsubseteq P_2$ as stating that for any $e \in E$ where $\mathrm{eff}(P_1, e) \neq \mathit{NotApp}$, $\mathrm{eff}(P_2, e) = \mathrm{eff}(P_1, e)$.

To demonstrate the use of these ordering relations, let us create a new policy; people are permitted to check the current results of the election, for exit polls. We encode this with the following policy

$$
\begin{aligned}
S_5 &= \{\langle a, v, o\rangle \in E : \exists x \in o : x = \texttt{getresult}\} \qquad (10) \\
r_3 &= \mathit{Scope}(\mathit{Err}(\mathit{Permit}, S_4), S_5)
\end{aligned}
$$

where S_4 is defined in Equation (6). Now, we can create a composite policy as follows $p_c = p \oplus r_3$, where p is defined in Equation (9). This policy has a bug—specifically, it permits people under 18 to vote in certain circumstances—and we will demonstrate the usefulness of our technique by showing this. First, we perform our translations on this new policy as above, getting:

$$
\begin{aligned}
g(f(r_3)) &= \langle S_5 \setminus S_4, \emptyset, S_4 \cap S_5\rangle \\
g(f(p_c)) &= \Big\langle \big((S_2 \setminus (S_0 \cup S_1 \cup S_3 \cup S_4)) \cup (S_5 \setminus S_4)\big), \\
&\qquad \Big(\big((S_0 \setminus S_3) \cup (S_1 \setminus S_4)\big) \cap S_2\Big) \setminus (S_4 \cap S_5), \\
&\qquad \Big((S_4 \cap S_5) \cup \big((S_3 \cup S_4) \setminus \\
&\qquad\qquad ((S_0 \setminus S_3) \cup (S_1 \setminus S_4))\big) \\
&\qquad\qquad \cap S_2\Big) \setminus \\
&\qquad \big((S_2 \setminus (S_0 \cup S_1 \cup S_3 \cup S_4)) \cup (S_5 \setminus S_4)\big)\Big\rangle
\end{aligned}
$$

where S_0, S_1, S_3 and S_4 are from Equations (3) to (6). Using set algebra we can simplify the expression for policy p_c to

$$g(f(p_c)) = \Big\langle \big((S_2 \setminus (S_0 \cup S_1 \cup S_3)) \cup S_5\big) \setminus S_4, \\ \big(((S_0 \setminus S_3) \cup (S_1 \setminus S_4)) \cap S_2\big) \setminus (S_4 \cap S_5), \\ \big((S_4 \cap S_5) \setminus (S_2 \setminus (S_0 \cup S_1 \cup S_3))\big) \cup \\ \big(((S_3 \cup S_4) \setminus ((S_0 \setminus S_3) \cup (S_1 \setminus S_4))) \\ \cap S_2\big)\Big\rangle$$

Now, we insist that this combined policy deny anyone trying to vote who is under 18. This is itself a policy, which we call p_v:

$$p_v = \langle \emptyset, (S_0 \cap S_2) \setminus (S_3 \cup S_4), (S_3 \cup S_4) \cap S_2 \rangle$$

The interesting thing here is whether or not $p_v \sqsubseteq_D p_c$, i.e., does the policy p_c deny every input that is denied by p_v. That would mean that everyone trying to vote who is under 18 is denied, and that our policy combination has not done any harm. However, the environmental tuple

$$e = \langle \{17\}, \{\texttt{true}\}, \{\texttt{vote}, \texttt{getresult}\} \rangle$$

demonstrates that that is not the case. Input e passes the second part of the *Permit* requirement and so is permitted by p_c (which means that it is *not* denied by p_c) but denied by p_v, i.e., e demonstrates that $p_v \not\sqsubseteq_D p_c$. The error is that this policy does not enforce that only one action be given in the third component of the input, and because of this we have the surprising result that someone who is under eighteen and has already voted, but asks for the voting results at the same time as trying to vote will be permitted, and so can cast any number of ballots. To fix this, we could insist upon a new condition, that $\exists! x : x \in o$; or we could use $\otimes$ instead of $\oplus$, which would ensure that only one of the sub-policies could be definitive on any given point (and so turn $\text{eff}(e, p_v)$ into an *Indet* result instead of a *Permit*); or we could decide that only people who have voted already can check the results.

4 Automatically Proving Properties of Policies

Given the formal model defined in Section 2.1 and properties defined in Section 3 we would like to check properties of access policies automatically. To do this we first formalize the syntax of formulas we use to specify subsets of E. Then we discuss how policies constructed using these formulas and policy combinators can be translated to Boolean logic formulas. After this translation we show that we can check properties of access policies using a SAT solver.

4.1 Characterizing Subsets of the Environment

In Section 2.1, we defined our formal model using subsets of the set of possible environments E. We showed that each policy can be expressed in triple form $P = \langle S, R, T \rangle$ where S, R, and T are subsets of E. We will assume that all subsets of E are specified in the form $\{e \in E : C\}$ where C is a constraint that evaluates to true or false for each environment. That is, the only free variables in C are the components of the environment tuple e. Note that the sets $S_0, S_1, \ldots, S_4$ in Equation (2) are expressed this way.

Given a set in the form $S = \{e \in E : C\}$ our goal is to generate a Boolean logic formula B which encodes the set S. The encoding will map each $e \in E$ to a valuation of the Boolean variables in B, and B will evaluate to true if and only if $e \in S$. Based on such an encoding we can convert questions about different policies (such as if one subsumes the other one) to SAT problems and then use a SAT solver to check them. For example, we can generate a Boolean formula which is satisfiable if and only if an access policy is not subsumed (i.e., $\not\sqsubseteq$) by another one. If the SAT solver returns a satisfying assignment to the formula, then we can conclude that the property is false, and generate a counterexample based on the satisfying assignment. If the SAT solver declares that the formula is not satisfiable then we can conclude that the property holds. We will discuss the details of such a translation below.

To present our translation we use the following notational conveniences: for elements $e \in E$, we name the components of e $e[0]$, $e[1]$, …, $e[n]$. We use s, s_0, s_1, …, s_n to denote set variables, a, a_0, a_1, …, a_n to denote scalar variables, and A, A_0, A_1, …, A_n to denote constants. We use the function $n(A)$ to define a unique non-negative integer for each constant A. Finally, *BP* is a set of basic predicates which we define as follows:

$$\begin{aligned} \textit{SCAL} &\rightarrow A \mid a \\ \textit{BSET} &\rightarrow s \mid e[i] \\ \textit{BP} &\rightarrow \texttt{true} \mid \texttt{false} \mid \textit{SCAL} = \textit{SCAL} \\ &\quad \mid \textit{SCAL} \in \textit{SET} \mid \textit{SET} \subseteq \textit{SET} \\ \textit{SET} &\rightarrow \textit{BSET} \mid \{\textit{SCAL}\} \mid \textit{SET} \cup \textit{SET} \\ &\quad \mid \textit{SET} \cap \textit{SET} \mid \textit{SET} \setminus \textit{SET} \end{aligned}$$

The above grammar is sufficient for specifying policies using only enumerated types (which obviously have finite domains) and the simple operations $\neg$, $=$, $\in$, $\subseteq$. This is sufficient for Boolean types, and also XML Schema enumerated types. We will discuss extension to other domains later in this section. This grammar is sufficient to model statements such as $x \in a$ from Equation (5), or $x = \texttt{vote}$ from Equation (4) (provided we consider this string to be an enumerated type). However we cannot yet model the bounding conditions with this grammar.

We can define them using the nonterminal C; assuming that all subsets of E are specified in the form $\{e \in E : C\}$, where there are no free variables save e in C, C is defined as follows:

$$\begin{aligned} C &\rightarrow \textit{BP} \mid C \wedge C \mid C \vee C \mid \neg C \\ &\quad \mid \forall a \in \textit{BSET} : C \mid \exists a \in \textit{BSET} : C \\ &\quad \mid \exists! a \in \textit{BSET} : C \end{aligned}$$

We use $\exists!$ to mean there exists exactly one instance that holds. We can express all set definitions on unordered and enumerated types that are permitted in XACML using the expressions

$$SCAL \rightarrow A \qquad SCAL.f := SCAL.v[n(A)] \wedge \bigwedge_{i=1, i \neq n(A)}^{k} (\neg SCAL.v[i]) \tag{11}$$

$$SCAL \rightarrow a \qquad SCAL.f := \bigwedge_{i=1}^{k} (SCAL.v[i] \leftrightarrow a[i]) \wedge \left(\bigvee_{i=1}^{k} SCAL.v[i] \right) \wedge \bigwedge_{i=1}^{k} \left(SCAL.v[i] \rightarrow \bigwedge_{j=1, j \neq i}^{k} \neg SCAL.v[j] \right) \tag{12}$$

$$BSET \rightarrow s \qquad BSET.f := \bigwedge_{i=1}^{k} (BSET.v[i] \leftrightarrow s[i]) \tag{13}$$

$$BSET \rightarrow e[i] \qquad BSET.f := \bigwedge_{j=1}^{k} (BSET.v[j] \leftrightarrow e[i][j]) \tag{14}$$

$$SET \rightarrow \{SCAL\} \qquad SET.f := SCAL.f \wedge \bigwedge_{i=1}^{k} (SET.v[i] \leftrightarrow SCAL.v[i]) \tag{15}$$

$$SET \rightarrow BSET \qquad SET.f := BSET.f \wedge \bigwedge_{i=1}^{k} (SET.v[i] \leftrightarrow BSET.v[i]) \tag{16}$$

$$SET \rightarrow SET_1 \cup SET_2 \qquad SET.f := SET_1.f \wedge SET_2.f \wedge \bigwedge_{i=1}^{k} (SET.v[i] \leftrightarrow (SET_1.v[i] \vee SET_2.v[i])) \tag{17}$$

$$SET \rightarrow SET_1 \cap SET_2 \qquad SET.f := SET_1.f \wedge SET_2.f \wedge \bigwedge_{i=1}^{k} (SET.v[i] \leftrightarrow (SET_1.v[i] \wedge SET_2.v[i])) \tag{18}$$

$$SET \rightarrow SET_1 \setminus SET_2 \qquad SET.f := SET_1.f \wedge SET_2.f \wedge \bigwedge_{i=1}^{k} (SET.v[i] \leftrightarrow (SET_1.v[i] \wedge \neg SET_2.v[i])) \tag{19}$$

$$BP \rightarrow \texttt{true} \qquad BP.f := BP.b \leftrightarrow \texttt{true} \tag{20}$$

$$BP \rightarrow \texttt{false} \qquad BP.f := BP.b \leftrightarrow \texttt{false} \tag{21}$$

$$BP \rightarrow SCAL_1 = SCAL_2 \qquad BP.f := SCAL_1.f \wedge SCAL_2.f \wedge \left(BP.b \leftrightarrow \bigwedge_{i=1}^{k} (SCAL_1.v[i] \leftrightarrow SCAL_2.v[i]) \right) \tag{22}$$

$$BP \rightarrow SCAL \in SET \qquad BP.f := SCAL.f \wedge SET.f \wedge \left(BP.b \leftrightarrow \bigwedge_{i=1}^{k} (SCAL.v[i] \rightarrow SET.v[i]) \right) \tag{23}$$

$$BP \rightarrow SET_1 \subseteq SET_2 \qquad BP.f := SET_1.f \wedge SET_2.f \wedge \left(BP.b \leftrightarrow \bigwedge_{i=1}^{k} (SET_1.v[i] \rightarrow SET_2.v[i]) \right) \tag{24}$$

Fig. 6. Translation of the basic predicates to Boolean logic formulas.

above. This is sufficient to model expressions like $\forall x \in v : x$ from Equation (3).

We will explain our translation from a constraint C defined by the above grammar to a Boolean logic formula by using attribute grammars. We will first discuss the translation of the basic predicates *BP*. In order to simplify our presentation we will assume that domains of all the scalars have the same size, call it k. We will encode a set of values from any domain using a Boolean vector of size k. Given a Boolean vector v, we will denote its components as $v[1]$, $v[2]$, ..., $v[k]$ where $v[i] \leftrightarrow$ `true` means that element i is a member of the set represented by v whereas $v[i] \leftrightarrow$ `false` means that it is not. We encode a set variable s and each component of the environment tuple e using the same encoding, i.e., as a vector of Boolean values. To simplify our presentation we also encode a scalar variable a as a set using a vector of Boolean values but restrict it to be a singleton set by making sure that at any time only one of the Boolean values in the vector can be true. In our actual implementation scalar variables are represented using $\log_2 k$ Boolean variables where k is the size of the domain.

The attribute grammar for basic predicates is shown in Figure 6. We have numbered the production rules. Each pro-

duction rule has a corresponding semantic rule next to it. Semantic rules describe how to compute the attributes of the nonterminal on the left hand side of the production rule using the attributes of the terminals and nonterminals on the right hand side of the production rule. In the attribute grammar shown in Figure 6 the nonterminals *SCAL*, *BSET* and *SET* have two attributes. One of them is a Boolean vector v denoting a set of values, and the other one is a Boolean logic formula f which accumulates the frame constraints. Again to simplify our presentation we represent scalar constants and scalar variables (i.e., the non-terminal *SCAL*) as singleton sets whereas in our actual implementation they are represented using $\log_2 k$ Boolean variables.

Equation (11) in Figure 6 states that a scalar constant A is encoded as a singleton set that contains only A. We represent this singleton set as a Boolean vector v, such that $v[n(A)]$ is set to true and all the rest of the elements of the vector are set to false. This condition is stored in the frame constraint f. Equation (12) states that a scalar variable is also encoded as a Boolean vector v. The frame constraint f makes sure that the elements of the Boolean vector v are same as the elements of the Boolean vector representing the scalar variable a and exactly one of the elements in a or v is set to true in any given time. Equations (13) and (14) show that the set variables (s) and components of the environment tuple ($e[i]$) are also encoded as Boolean vectors.

Equation (15) creates a singleton set from a scalar constant *SCAL*. However, since we encode scalar constants as singleton sets, this simply means that the Boolean vectors encoding the scalar constant *SCAL*.v and the set *SET*.v are equivalent and the frame constraint *SET*.f expresses this constraint. Note that in the attribute grammar shown in Figure 6 the frame constraint of a nonterminal on the left hand side of a production is a conjunction of the frame constraints of the nonterminals on the right hand side of the production plus some other constraints that are added based on the production rule.

Equations (17), (18), and (19) encode the set operations: union, intersection and set difference. Each set operation on two set expressions *SET*$_1$ and *SET*$_2$ results in the creation of new Boolean vector *SET*.v. The value of an element in *SET*.v is defined based on the corresponding elements in *SET*$_1$.v and *SET*$_2$.v. For example for the union operation *SET*.$v[i]$ is true if and only if *SET*$_1$.$v[i]$ is true or *SET*$_2$.$v[i]$ is true. The intersection and set difference are defined similarly.

The nonterminal *BP* corresponds to the basic predicates and it has two attributes. One of them is a Boolean variable b representing the truth value of the predicate and the other one is a Boolean logic formula f that accumulates the frame constraints.

Equations (20) and (21) create two basic predicates which have the truth value true and false, respectively.

Equation (22) is a basic predicate that corresponds to an equality expression comparing two scalars. Since scalars are expressed as Boolean vectors, the Boolean variable encoding the truth value of the predicate is true if and only if all elements of the Boolean vectors encoding the two scalar values are the same. This constraint is added to the frame constraint of the basic predicate.

Equation (23) creates a basic predicate that corresponds to a membership expression testing membership of a scalar to a set expression. Equation (24) creates a basic predicate that corresponds to a subset expression testing if a set expression is subsumed by another set expression. Since we encode scalars a singleton sets, the frame constraints generated for Equations (23) and (24) are very similar. They state that if a value is a member of the set on the left hand side, then it should also be member of the set on the right hand side.

The attribute grammar for the constraints is shown in Figure 7. The nonterminal C has two attributes. One of them is a Boolean variable b representing the truth value of the constraint, and the other one is a Boolean logic formula f that accumulates the frame constraints. Again, the frame constraint of a nonterminal on the left hand side of a production is a conjunction of the frame constraints of the nonterminals on the right hand side of the production plus some other constraints that are added based on the production rule.

Equation (25) is just a syntactic rule expressing that a constraint can be a basic predicate. Equation (26) defines the negation operation. As expected the frame constraint states that the value of the constraint on the left hand side of the production rule is the negation of the value of the constraint on the right hand side of the production rule. Equations (27) and (28) define the disjunction and conjunction operations. The frame constraints generated in Equations (27) and (28) state that the value of the constraint on the left hand side of the production rule is the disjunction or the conjunction of the values of the constraints on the right hand side of the production rule, respectively.

Equations (29), (30), and (31) deal with quantified constraints. In these equations, a denotes a scalar variable which is quantified over a basic set expression *BSET* which is either a set variable s or a component of the environment tuple $e[i]$. The quantified variable a can appear as a free variable in the constraint expression on the right hand side (C_1). The expression that follows fixes the value of a. First, we must establish as a frame condition that a is a singleton set; accordingly one value of the $a[i]$s must be true and the rest must all be false. Next, for universal quantification we want to constrain that for every value i where *BSET*.$v[i]$ is true, the condition C_1 must hold for a scalar a representing that element. Accordingly, for each *BSET*.$v[i]$ that is true, if $a[i]$ is true C_1 must hold.

Existential qualification is very similar. We use the same frame condition guaranteeing that a is a singleton set. We must find some i such that *BSET*.$v[i]$ is true, a is the scalar representing that element and C_1 holds for that a. This can be derived from Equation (29) using the identity $\exists a : b \equiv \neg\forall a : \neg b$.

Equation (31) models existentially quantified constraints which evaluate to true if and only if the constraint C_1 evaluates to true for exactly one member of the set s or $e[i]$. We use the same frame condition guaranteeing that a is a singleton set. First, we must find some i for which the condition holds; this is the same as the existential quantification case. Then,

$$C \rightarrow \mathit{BP} \qquad C.f := \mathit{BP}.f \wedge (C.b \leftrightarrow \mathit{BP}.b) \tag{25}$$

$$C \rightarrow \neg\, C_1 \qquad C.f := C_1.f \wedge (C.b \leftrightarrow \neg\, C_1.b) \tag{26}$$

$$C \rightarrow C_1 \vee C_2 \qquad C.f := C_1.f \wedge C_2.f \wedge (C.b \leftrightarrow (C_1.b \vee C_2.b)) \tag{27}$$

$$C \rightarrow C_1 \wedge C_2 \qquad C.f := C_1.f \wedge C_2.f \wedge (C.b \leftrightarrow (C_1.b \wedge C_2.b)) \tag{28}$$

$$C \rightarrow \forall a \in \mathit{BSET} : C_1 \qquad C.f := \mathit{BSET}.f \wedge C_1.f \wedge \left(\bigvee_{i=1}^{k} a[i]\right) \wedge \bigwedge_{i=1}^{k} \left(a[i] \rightarrow \bigwedge_{j=1, j\neq i}^{k} \neg\, a[j]\right) \wedge \left(C.b \leftrightarrow \bigwedge_{i=1}^{k} \mathit{BSET}.v[i] \rightarrow (a[i] \rightarrow C_1.b)\right) \tag{29}$$

$$C \rightarrow \exists a \in \mathit{BSET} : C_1 \qquad C.f := \mathit{BSET}.f \wedge C_1.f \wedge \left(\bigvee_{i=1}^{k} a[i]\right) \wedge \bigwedge_{i=1}^{k} \left(a[i] \rightarrow \bigwedge_{j=1, j\neq i}^{k} \neg\, a[j]\right) \wedge \left(C.b \leftrightarrow \bigvee_{i=1}^{k} \mathit{BSET}.v[i] \wedge a[i] \wedge C_1.b\right) \tag{30}$$

$$C \rightarrow \exists! a \in \mathit{BSET} : C_1 \qquad C.f := \mathit{BSET}.f \wedge C_1.f \wedge \left(\bigvee_{i=1}^{k} a[i]\right) \wedge \bigwedge_{i=1}^{k} \left(a[i] \rightarrow \bigwedge_{j=1, j\neq i}^{k} \neg\, a[j]\right) \wedge \left(C.b \leftrightarrow \bigvee_{i=1}^{k} \Big(\mathit{BSET}.v[i] \wedge a[i] \wedge C_1.b \wedge \bigwedge_{n=1, n\neq i}^{k} \mathit{BSET}.v[n] \rightarrow (a[n] \rightarrow \neg C_1.b)\Big)\right) \tag{31}$$

Fig. 7. Translation of the constraints to Boolean logic formulas.

we must ensure that for every other index, C_i is false; this is very similar to the universal quantification case with a negated condition. If we follow formal logical conventions and regard C in the construction $\exists! a : C$ as a predicate with argument a, then we can express $\exists! a \in \mathit{BSET} : C_1$ as $\exists a \in \mathit{BSET} : C_1(a) \wedge \forall a' \in \mathit{BSET} : a' \neq a \rightarrow \neg C_1(a')$.

4.2 Bounded Domains, Unbounded Domains, and Domain Specific Predicates

The translation we described above can handle XACML policies that only use bounded unordered and enumerated types. In fact, during our analysis we limit the size of every domain to a given fixed size and then analyze the policies as though they were specified using finite enumerated sets of that size. The problem is that if our automated analysis does not yield a counterexample to a given property, then that does not necessarily mean that no counterexample exists—perhaps if we had increased the scope just a little more we would have found one. As an example, in Equation (4) we state the condition $x = \texttt{vote}$. We can set a domain size so that x can take on the value of any string with less than, say, six characters. This is more than sufficient for our initial needs, but will not discover the flaw in Equation (10) because it will not be capable of generating the string `getresult`. The small scope hypothesis (discussed by Jackson and Damon [18], and tested and confirmed for some data structure algorithms by Marinov et al [24]) suggests that small scopes could be sufficient in practice. Note that if a counterexample is found using bounded domains, that counterexample is definite and can be translated into an error in the original policy.

Another limitation of the translation we described above is the fact that it does not handle domain specific predicates, e.g., ordering relations on domains such as integers. An example here is in Equation (2), where we state $x < 18$ in a constraint. When we translate sets described using such predicates to Boolean logic formulas we represent them as uninterpreted Boolean functions. We create a Boolean variable for encoding the value of the uninterpreted Boolean function and we generate constraints which guarantee that the value of the function is the same if its arguments are the same. Other than this restriction, the variables encoding the functions can get arbitrary values. To encode Equation (2), we would introduce a Boolean variable v_0 which represents the expression $x < 18$ and then encode the formula $\forall x \in a\ v_0$. If we had additional expressions of this sort, perhaps a constraint $x < 21$, then we would encode that with an additional variable v_1. However, the relationship between these two predicates, i.e., $v_0 \rightarrow v_1$ also needs to be added as a frame constraint to achieve pre-

cise analysis. Since our encoding does not generate all such constraints, our analysis can sometimes report spurious errors.

As discussed above, due to the scope restriction our analysis cannot guarantee absence of errors for unbounded domains. Due to our conservative approximation of domain specific predicates, it is also possible that some counterexamples may be spurious, and will need to be validated against the original policy. However, we believe that this type of automated analysis can still be useful in uncovering errors in access policies. First, our analysis is both sound and complete for bounded unordered and enumerated domains as long as the size used during analysis is not smaller than the actual size of the domain. Second, for unbounded domains, our analysis can be used to prove the absence of errors within a certain bound. Third, for bounded and ordered domains, our analysis is sound and can be used to prove the absence of errors. However, the counter-example scenarios generated for such policies need to be validated to make sure that they are not spurious.

For unbounded and ordered domains, our analysis would be neither sound nor complete unless we represent the domain specific predicates on ordered domains precisely within a certain bound. Note that it is possible to fully interpret ordering relations as long as the domain is bounded. We can encode a type with a domain of n ordered elements using n^2 Boolean variables, one for each pair of values in the domain, representing the ordering relations. However, XACML uses many complex functions such as XPath matching and X500 name matching. Attempting to fully realize these in Boolean logic is possible, but would lead to extremely complex formulas due to the need to, for example, parse the XPath expression. Hence, we believe that using uninterpreted functions for abstracting such complex functionality is a justified approach and enables us to handle a significant portion of the XACML language. This means that for unbounded and ordered domains, our analysis can be used to prove the absence of errors within a certain bound, however, the counter-example scenarios need to be validated to make sure that they are not spurious due to the conservative approximation of some of the predicates using uninterpreted functions. We would like to note that the imprecision caused by abstraction of such complex functions has not led to any spurious results in the experiments we performed so far.

4.3 Verification of Policies

As discussed in Section 3, we specify properties of policies using a set of partial ordering relations. These partial ordering relations can be used to state that a certain type of outcome for one policy subsumes the same type of outcome for another policy. In this section we will only focus on the $\sqsubseteq$ relation. Translation of properties specified using other relations are handled similarly.

Given a query like $P_1 \sqsubseteq P_2$, our goal is to generate a Boolean logic formula which is satisfiable if and only if $P_1 \not\sqsubseteq P_2$. As we discussed earlier our tool first translates policies P_1 and P_2 to triple form, such that $P_1 = \langle S_1, R_1, T_1 \rangle$ and $P_2 = \langle S_2, R_2, T_2 \rangle$ where each element of each triple is specified with a constraint expression as follows:

$$\begin{aligned} S_1 &= \{e \in E : C_{S_1}\} \\ R_1 &= \{e \in E : C_{R_1}\} \\ T_1 &= \{e \in E : C_{T_1}\} \\ S_2 &= \{e \in E : C_{S_2}\} \\ R_2 &= \{e \in E : C_{R_2}\} \\ T_2 &= \{e \in E : C_{T_2}\} \end{aligned}$$

After translating policies P_1 and P_2 in to the triple form our translator generates a Boolean logic formulas for the constraints C_{S_1}, C_{R_1}, C_{T_1}, C_{S_2}, C_{R_2} and C_{T_2} based on the attribute grammar rules described in Figures 6 and 7. For example, after this translation the truth value of the constraint C_{S_1} is represented with the Boolean variable $C_{S_1}.b$ and the frame constraint $C_{S_1}.f$ states all the constraints on the Boolean variable $C_{S_1}.b$.

Recall that for $P_1 = \langle S_1, R_1, T_1 \rangle$ and $P_2 = \langle S_2, R_2, T_2 \rangle$, $P_1 \sqsubseteq P_2$ holds if and only if

$$S_1 \subseteq S_2 \wedge R_1 \subseteq R_2 \wedge T_1 \subseteq T_2$$

Based on this, we can generate a formula F such that $F =$ `true` iff $P_1 \sqsubseteq P_2$ as follows:

$$\begin{aligned} F = (C_{S_1}.f \wedge C_{R_1}.f \wedge C_{T_1}.f \wedge C_{S_2}.f \wedge C_{R_2}.f \wedge C_{T_2}.f) \\ \rightarrow \big((C_{S_1}.b \rightarrow C_{S_2}.b) \wedge (C_{R_1}.b \rightarrow C_{R_2}.b) \\ \wedge (C_{T_1}.b \rightarrow C_{T_2}.b)\big) \end{aligned}$$

Finally, we send the property $\neg F$ to the SAT solver. If the SAT solver returns a satisfying assignment for the Boolean variables encoding the environment tuple e (which are the only free variables in the formula $\neg F$), the satisfying assignment corresponds to a counter-example environment demonstrating how the property is violated. If the SAT solver states that $\neg F$ is not satisfiable, then we conclude that the property holds, i.e., $P_1 \sqsubseteq P_2$.

Since the majority of the SAT solvers expect their input to be expressed in Conjunctive Normal Form (CNF), the last step in our translation before we send the formula $\neg F$ to the SAT solver is to convert $\neg F$ to CNF. For conversion to CNF we have implemented the structure preserving technique from [27]. The structure preserving technique, in essence, creates an auxiliary variable for each subexpression, and then combines the auxiliary variables. For example, the formula $(a \rightarrow b) \vee (b \wedge c)$ might get translated to $(v_0 \leftrightarrow (a \rightarrow b)) \wedge (v_1 \leftrightarrow (b \wedge c)) \wedge (v_0 \vee v_1)$. Following this step, the subexpressions are expanded using DeMorgan's Rule. This technique is simple to implement but introduces large numbers of auxiliary variables, which may negatively impact run time. We discuss the performance impact of this choice in more detail in Section 5.

5 Experiments

Using the algorithms given above, we are able to convert a policy property to a Boolean formula in CNF. We then apply a

SAT solver to this formula to determine if the property holds. In particular, we use the Zchaff [25] SAT solver. We conducted experiments in order to investigate if our tool runs in a reasonable amount of time on XACML policies. We also compared the performance of our tool to other XACML verification efforts. In our experiments we use the CONTINUE example [23], encoded into XACML by Fisler et al. [12]. CONTINUE is a Web-based conference management tool, aiding paper submission, review, discussion and notification. In addition, we also use the Medico example from the XACML [34] specification, which models a simple medical database meant to be accessed by physicians. Finally, we have encoded our example from Section 3 into XACML and applied our tool to the discovery of the error which we know to exist. In these examples, C1–C11 have been encoded by Fisler et al., and M1, M2 and V1 have been encoded by us.

Given that our experiments consist of two realistic XACML policies and one toy example it is difficult to generalize our experimental results. All we can say is that our approach performs efficiently on these examples, and can successfully verify nontrivial properties on these policies. Assessing the effectiveness of our verification approach in practice would require a comprehensive empirical study, which is beyond the scope of this work.

We tested 11 properties (labeled C1, C2, and so on) for subsumption on CONTINUE and two (labeled M1 and M2) on the Medico example; our voting example is property V1. Run times for verification are given in Table 1. The properties we checked can be described informally as follows:

1. Property C1 tests that the conference manager correctly denies program committee chairs the ability to review papers he/she has a conflict with.
2. Property C2 and C7 test that the conference manager permits program committee members to edit reviews they own.
3. Property C3 and C8 test that the conference manager denies access to users without a defined role.
4. Property C4 and C5 test that the conference manager will permit a program committee member who has called a meeting to read documents concerning the meeting, but not other arbitrary documents.
5. Property C6 tests whether the conference manager permits program committee members to read all parts of a review.
6. Property C9 tests whether the conference manager permits unauthorized user roles to set meetings.
7. Property C10 and C11 test that the conference manager permits program committee members who have filed their review to read the reviews of others, and denies program committee members that have not yet filed their review from reading other reviews.
8. Property M1 and M2 test whether the unified Medico policy permits a physician to edit the medical records of their patients.
9. Property V1 is just the voting property we demonstrated in Section 3.

Some of these properties are expected to be subsumed by the CONTINUE policy, and some are expected to subsume the policy. In general, if one wishes to verify that a property is upheld for any potential outcome, one should write a policy that is expected to subsume the target policy. Existence properties are most readily demonstrated with policies that are expected to be subsumed by the target policy.

We performed our analysis on a 2.8 GHz Intel Pentium 4, with 2 GB of memory, running the Linux 2.6.26 kernel. The listed values for each property are the median of five runs. The performance results shown in Table 1 indicate that analysis time is dominated by the initial parsing of the policies and by the conversion from triple form to a Boolean formula; sometimes the Boolean conversion is strongly dominant, as in the Medico examples. The resulting formulas are unexpectedly small and analysis time is so small the startup and I/O overhead of the Zchaff tool is probably dominating. This was unexpected; our tool goes to some length to simplify the Boolean formula on the assumption that run times would be dominated by the SAT solver. The results show that our assumption was wrong. However, these results are very encouraging in terms of the scalability of the proposed approach. Among the different components of our analysis, SAT solving is the one with worst case complexity. Since the examples we tested so far were easily handled by the SAT solver we believe that our approach will be feasible for analysis of large XACML policies.

The number of variables in our Boolean formulas is quite large, approximately half the number of clauses. We have made a deliberate tradeoff to get this. Our translation machinery from Section 4 introduces large numbers of tightly constrained variables, and our CNF conversion uses the structure preserving technique [27] which generates even more variables, but in exchange we get a relatively small formula, and the search space is not so large as might be presumed because of the constraints. A different CNF conversion might embody a different tradeoff between the CNF conversion and SAT solving that might be worth exploring.

Our experimental results show that the subsumption properties can be analyzed quickly for these policies. Although our experiments demonstrate the feasibility of our approach, determining its scalability would require more experiments with a larger set of policies. Inasmuch as total runtime is dominated by the Boolean formula generation and CNF transformation steps, steps which we did not initially think would be as significant a contributor to run time as the SAT computation, we believe that we could improve the performance of our tool by optimizing Boolean formula generation and the CNF transformation.

5.1 Comparison with Margrave

We have also compared our analysis with the Margrave tool written by Fisler et al. [12]. Margrave is a change impact analysis tool for the XACML language, similar in many respects to our own work. Therefore where possible we have compared the performance of our tool with Margrave's performance on

Property	I/O (ms)	Transform (ms)	Boolean (ms)	SAT (ms)	Variables	Clauses	Result
C1	429	45	617	13	56	114	No
C2	421	44	504	11	42	83	No
C3	426	45	581	11	51	108	No
C4	427	45	582	10	52	106	No
C5	425	428	854	10	79	166	No
C6	428	44	1110	15	89	190	Yes
C7	126	7	1620	12	95	218	Yes
C8	433	44	486	11	42	83	Yes
C9	430	46	578	11	51	106	Yes
C10	416	44	1464	12	108	250	Yes
C11	412	45	2462	13	129	297	Yes
M1	137	7	5144	14	109	280	No
M2	138	7	5168	13	108	279	No
V1	115	10	1765	11	52	123	Yes

Table 1. Verification performance for the properties of the CONTINUE conference manager (C1–11), Medico (M1–2) and voting (V1) examples. We divided the execution time to I/O, transformation to triple form, Boolean formula generation and CNF transformation and SAT solving. The times are in milliseconds. We also listed the size of the generated SAT problem instance (in terms of the number of Boolean variables and clauses) for verification of each property. If the result is "Yes" the generated SAT instance is satisfiable and the property is violated. If the result is "No" the generated SAT instance is not satisfiable and the property holds.

	Parse/conversion			Property verification		
Property	CPU time (ms)	Real time (ms)	GC time (ms)	CPU time (ms)	Real time (ms)	GC time (ms)
C1	1084	1102	244	4	0	0
C2	1084	1102	244	0	1	0
C3	1084	1102	244	0	0	0
C4	1084	1102	244	0	1	0
C5	1084	1102	244	4	2	0
C6	1084	1102	244	0	0	0
C7	1084	1102	244	0	0	0
C8	1084	1102	244	0	2	0
C9	1084	1102	244	0	0	0
C10	1084	1102	244	0	1	0
C11	1084	1102	244	0	0	0

Table 2. Verification performance for the properties of the CONTINUE conference manager under Margrave. Due to differences in tool architecture, only one parse/conversion step is necessary to verify any number of properties; accordingly only one such time is given.

our examples. Table 2 shows the results of running Margrave on the CONTINUE properties. As with Table 1, these figures represent a median of five runs on the same 2.8 GHz machine.

The current version of Margrave is 2–1, which has been updated for XACML 2.0. The CONTINUE example which we both use has not been updated to XACML 2.0 and only runs under Margrave 1–1. The differences between XACML 2.0 and XACML 1.0 have minimal impact upon change analysis but the syntax is very different. Margrave 1–1 only runs under PLT Scheme 209, which may not be as optimized as more recent versions.

Margrave's architecture is very different from our own, which makes a direct comparison of the time taken to analyze properties difficult. Margrave parses the XACML and converts it into a form suitable for analysis only once, and then can check as many properties as is desired. Margrave manages this by using a binary decision diagram (BDD) [6] for analysis. The conversion process can be time consuming and can also consume large amounts of memory. However, once the BDD has been created, checking properties is very quick—effectively linear in the number of variables used. This is why the figures for property verification are so small; while clearly it does not actually take 0 milliseconds to perform property verification the actual time taken is so small it is difficult to measure. Adding together the parse/conversion time and the property verification time gives a result more comparable to our own. However, this is not a fair comparison since it ignores Margrave's ability to do several checks on a given policy with only one conversion step.

Margrave itself has several important limitations which prevent us from comparing our tools using all our examples. Among the limitations, Margrave 1–1 is not capable of handling Condition elements in Rules or generally any restriction that cannot be expressed using `<Target>` elements. The

number of predicates admissible in `<Target>` elements is a fraction of those available in the language as a whole. In particular none of the predicates and functions that complicate conversion to a Boolean formula, like ordering or XPath comparison that we discuss in Section 4.2, can be used in `<Target>` elements. Margrave 2–1 adds minimal support for Condition elements, supporting only Boolean functions and string equality. Neither version of Margrave can handle the only-one-applicable policy or rule combining algorithms, nor does it understand the *Indet* result.

Because of these limitations, we cannot run the M1, M2 and V1 examples in Margrave. In particular the $x < 18$ property in S_0 which is part of V1, the XPath node matching in M1 and M2, the uniqueness declarations (for example, the `string-one-and-only` function) used extensively in M1 and M2, and the date calculations present in M1 and M2 prevent those examples from being used with Margrave.

The only examples we have been able to find that are supported both by our tool and by Margrave is the CONTINUE conference manager examples, which were in fact written for Margrave; these correspond to our C1 through C11 properties. For these examples, Margrave is able to parse the XACML, convert it to a Boolean formula and build the corresponding decision diagram in 1.1 seconds. After the decision diagram is built, the time it takes to check properties is negligible; in many cases our test harness reports 0 seconds required for completion. Our system is more sensitive to the structure of the property but for the CONTINUE examples is quite competitive. The majority of the time for our tool is spent performing I/O or generating a Boolean formula in CNF form. As well, we must regenerate the Boolean formula for each property whereas a decision diagram approach does not. However, note that, the median time for checking the SAT instances generated by our tool never exceeds 0.015 seconds; since we run the SAT solver as a separate process it is very possible that most of this time is spent in process creation overhead. Since this is the only part in our analysis that has exponential worst-case complexity, it is very encouraging to see that it is an insignificant part of the computation time of our tool for these examples. In contrast, for the decision diagram based approach used by Margrave, the time consuming step is the construction of the decision diagram and, hence, the verification time for Margrave is dominated by the time it takes to build the decision diagram.

It should be noted that while Margrave's total run time is swifter than our own, Margrave is only capable of examining policies specifically written for it, due its inability to handle any condition not expressible as a scoping restriction. It would be interesting to run our other examples against Margrave, but this is not possible due to its limitations.

5.2 Comparison with Alloy Translation

Before we built a direct XACML to SAT translator, we experimented with using the Alloy [17] analyzer as a back-end verification tool for XACML policy analysis [14]. We developed a tool that automatically translates XACML policies into the Alloy [17] declarative modeling language. Alloy is based on first order relational logic and is intended to model complex structures. It does so through extensive set manipulation, and this makes it an attractive target for translation from our mathematical model. We briefly discuss here our translation from our formal model to Alloy and the results from attempting analysis using that translation.

Alloy models consist of sets of concrete objects, called *signatures,* facts about these sets, and relations between the sets. Distinguished subsets of these signatures are possible, and these new sub-signatures are said to extend the original signature. Unlike some other modeling languages, Alloy does not require that relations be completely specified. Alloy cannot in general prove assertions about all possible models, but it can prove assertions about models with a fixed scope.

5.2.1 Translation to Alloy

The general structure we use for translation is as follows: to prove that $P_1 \sqsubseteq P_2$ we define each of P_1 and P_2 and then check that each set of P_1 is contained in the corresponding set of P_2; this is structured as follows.

```
static sig P1 extends Triple {} {
  ...
}
static sig P2 extends Triple {} {
  ...
}
assert Subset {
  P1.permit in P2.permit
  P1.deny in P2.deny
  P1.error in P2.error
}
```

We translate from our mathematical model for P_1 and P_2 to Alloy code for the same as follows. We first define a signature corresponding to our environment E; each component of E is encoded as a field. So for our example we might encode the environment E in Alloy as

```
sig E {
  age : set Integer,
  voted : set Bool,
  actions : set String
}
```

We also need to encode constants; while Alloy already defines the Boolean constants `True` and `False` we must manually define any others. So for our running example we would have a signature of constants defined as follows:

```
static sig CONSTANTS {
  eighteen : scalar Integer,
  vote : scalar String,
  ...
}
```

For each set $S \subseteq E$ we define an auxiliary set as we did the sets $S_0, S_1, \ldots, S_4$ in (2) through (6) in Section 2. For example, if we wanted to encode S_2 as an Alloy set, we might encode it as

```
sig S_2 extends E {} {
  CONSTANTS.vote in actions
}
```

We can encode relations like $<$ using an auxiliary Alloy function `LessThan` and enforcing transitivity as follows:

```
fact {
  all a,b,c:Type {
    LessThan (a, b) = True &&
    LessThan (b, c) = True =>
    LessThan (a, c) = True
  }
}
```

In fact Alloy has a library for this specific operation, but the technique is useful for converting other relations. Using this definition we can encode S_0 as

```
sig S_0 extends E {} {
  all x : age |
    LessThan(x, CONSTANTS.eighteen)
      = True
}
```

Once all the various S_is have been encoded, we can define the triples directly using Alloy's set operations.

5.2.2 Effectiveness of the Alloy Translation

Armed with the translation detailed above, we can apply our analysis of XACML policies much as we did with our direct SAT translation. Unfortunately due to limitations in the Alloy Analyzer it is not possible to analyze the same policies we did with our SAT based tool. We observed that as the policies get larger the Alloy Analyzer either runs out of memory or crashes, and that we encountered these difficulties surprisingly quickly.

We were able to analyze a simpler portion of the Medico example used by our M1 and M2 examples described in Section 5. Specifically, we took a subset of the Medico policy and checked that this subset was subsumed by the full policy (which should be true) and also checked that the full policy is not subsumed by the subset. Note that these tests are not the same as the M1 and M2 examples given above; M1 and M2 test more semantically meaningful properties. On the same hardware that we used to run the earlier examples, we found that checking that a subset is subsumed by the full policy took 21.7 seconds, and checking that the full policy is not subsumed by the subset took 42.0 seconds.

We experienced severe difficulties finding properties that the Alloy Analyzer could analyze successfully. When we attempted to check the M1 and M2 properties themselves, the Alloy Analyzer alternately crashed or ran out of memory. Sometimes restructuring the Alloy encoding of the policy ameliorated the difficulties, suggesting some inefficiency in the Boolean encoding process. Ultimately the manner in which the Analyzer failed—that is, abruptly and with no indication of what went wrong—made it extremely difficult to determine where the error lay.

To deal with these issues, we developed the direct-to-SAT translation we have detailed in Section 4.3. We use the same SAT solver (that is, Zchaff) that the Alloy Analyzer uses, but our direct-to-SAT translation handles the largest XACML policies we can find gracefully, and considerably more quickly as well.

6 Related Work

This paper is an extended version of our earlier work [15]. We extended the results reported in [15] by adding a detailed discussion of the differences between the simplified policy combinator operators we use in our tool and the corresponding policy combinator operators from the XACML language specification. We show that our operators are equivalent to their XACML counterparts under certain conditions, and, furthermore, XACML operators can be translated to our operators using a translation algorithm we describe in this paper. In this paper, we also extend the experimental results from that work [15] in two significant ways: first, we compare the performance of our tool with that of Margrave [12] using one of the policy examples from our experiments, and discuss the benefits and the limitations of Margrave compared to our tool; second, we discuss the use of Alloy [17] as the back end verification tool for XACML policy analysis (instead of a direct translation to a SAT solver), and provide experimental results in this direction.

Access control has been the subject of extensive research: Sandhu, Samarati and de Capitani de Vimercati [28, 29, 32] introduce the process; Bonatti, Bertino, et al., [3, 4, 30, 31] describe various models for access control; Damiani et al. [7–10] describe a particular fine grained access control for XML documents; Bonatti et al. [5] define an algebra for composing different parts of a model into a unified whole; and Abadi, Heckman, di Vimercati et al. [2, 11, 13] describe ways to distribute the control so that it is consistent across a distributed system.

Access policy languages, too, are not new: Abad-Piero et al. [1] describes a general purpose policy language for authorization systems, Jajodia et al. [20–22] define a model and language for access control and then present a framework for enforcing multiple access policies by expressing how to combine them in a new language. We chose XACML because it is a standardized language with tool support, and so our results are more likely to be immediately useful.

The problem with access policies becoming large and difficult to reason about has also been studied, but not in the general case: Heckman and Levitt [13] present a way of verifying a hierarchy of security servers to ensure that they are enforcing the whole access policy, and Naumovich [26] presents an algorithm for computing the flow of permissions through the Java security model, to aid static analysis. Neither of these are exactly what we want: Heckman and Levitt's work

can prove that the programs you have collectively implement the policy you specified, but their technique cannot tell you whether you have made a subtle error in creating your policy in the first place; Naumovich's work is more comprehensive but is specific to Java's security model.

Automated analysis of access control policies has also been researched; Schaad and Moffet [33] and Zao et al. [37] analyze role based access control schemas using the Alloy analyzer. Schaad and Moffet use Alloy to verify that the composition of specifications is well formed and is silent about their content, whereas we introduce a formal model of and a partial ordering on XACML specifications specifically designed for analyzing the semantics. Zao et al. model RBAC schemas in Alloy and then checks these models against predicates, also written in Alloy. We introduce a formal model for XACML with a partial ordering on policies that we then automatically check using a SAT solver as a back end; we do not insist that the user write predicates in another language and operate solely on XACML.

Jackson's Alloy Analyzer also uses a SAT solver as a back-end to solve verification queries [16, 19]. Hence, translating XACML policies to Alloy in order to verify them is in effect an indirect way of using a SAT solver for verification. We attempted to use the Alloy Analyzer for verification of XACML policies previously [14]; we detail our technique in Section 5.2. Our experiments have shown that a direct translation to SAT is much more effective then translating the verification queries to Alloy. Using a direct translation we can generate a customized encoding of the problem. The Alloy Analyzer is optimized for a more general class of models and hence, not necessarily efficient for types of verification queries we are interested in.

Recently, Fisler et al. [12] used multi-terminal decision diagrams to verify properties of XACML policies with the Margrave tool. We compare Margrave with our tool in Section 5.1. Briefly; verification queries in Margrave are expressed in the Scheme language. We use relationships between policies instead, and we believe this makes our tool easier to use. Also, a verification approach that uses decision diagrams is more likely to be successful for incremental analysis techniques; we agree that the multi-terminal decision diagrams are the appropriate representation to use the change-impact analysis Fisler et al. describe. However, for the type of verification queries we discuss in this paper we expect a verification approach based on SAT solvers to perform better on large policies than a verification approach based on decision diagrams. A final difference is that our tool handles more of XACML than Margrave does, including complex conditionals and more datatypes.

7 Conclusion

We have presented a formal model for access control policies, and shown how to verify interesting properties about such models in an automated way. In particular we translate queries about access control policies to Boolean satisfiability problems and use a SAT solver to obtain an answer. We express properties about access control policies as subsumption queries between two policies. We have implemented a tool that translates XACML policies into to our formal model and also translates subsumption queries between two XACML policies to a Boolean satisfiability problem. Our experimental results indicate that automated verification of nontrivial access policies is feasible using our approach.

Our approach is not without its limitations; we perform a bounded analysis which can lead to false negatives, and we abstract certain functions which can lead to false positives. However for finite state specifications our approach is sound and complete as long as the user chooses a sufficiently large bound and the complex XACML functions are not used in the specification. We successfully accommodate far more of the XACML specification in our analysis than previous efforts have managed.

In the future, we would like to investigate different abstraction techniques to generate more precise models for the functions that we cannot directly simulate. We would also like to experiment on more and larger policies. One issue worth investigating is the relationship between the characteristics of a policy (such as its textual size) and the time taken to perform analysis upon it. The CONTINUE examples from Section 5 are textually larger than the Medico examples we used, but the latter have more variables and clauses in the generated Boolean formula.

References

1. J. L. Abad-Peiro, H. Debar, T. Schweinberger, and P. Trommler. PLAS — Policy language for authorizations. Technical Report RZ 3126, IBM Research Division, 1999.
2. M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, Sept. 1993.
3. E. Bertino, C. Bettini, E. Ferrari, and P. Samarati. An access control model supporting periodicity constraints and temporal reasoning. *ACM Transactions on Database Systems*, 23(3):231–285, 1998.
4. P. Bonatti, E. Damiani, S. De Capitani di Vimercati, and P. Samarati. An access control model for data archives. In *Proceedings of the 16th international conference on information security: trusted information*, pages 261–276, Paris, France, 2001. Kluwer International Federation For Information Processing Series.
5. P. Bonatti, S. De Capitani di Vimercati, and P. Samarati. An algebra for composing access control policies. *ACM Transactions on Information Systems Security*, 5(1):1–35, 2002.
6. K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a bdd package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 40–45, 1990.
7. E. Damiani, S. De Capitani di Vimercati, E. Fernández-Medina, and P. Samarati. Access control of SVG documents. In *Proceedings of DBSec 2002*, pages 219–230, 2002.
8. E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati. Design and implementation of an access control processor for XML documents. *Computer Networks*, 33(1–6):59–75, 2000.

9. E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati. A fine-grained access control system for XML documents. *ACM Transactions on Information Systems Security*, 5(2):169–202, 2002.
10. E. Damiani, P. Samarati, S. De Capitani di Vimercati, and S. Paraboschi. Controlling access to XML documents. *IEEE Internet Comput.*, 5(6):18–28, 2001.
11. S. De Capitani di Vimercati and P. Samarati. Access control in federated systems. In *Proceedings of the 1996 workshop on new security paradigms*, pages 87–99, Lake Arrowhead, California, United States, 1996. ACM Press.
12. K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz. Verification and change-impact analysis of access-control policies. In *Proceedings of the 27th International Conference on Software Engineering*, pages 196–205, St. Louis, Missouri, May 2005.
13. M. Heckman and K. N. Levitt. Applying the composition principle to verify a hierarchy of security servers. In *HICSS (3)*, pages 338–347, 1998.
14. G. Hughes and T. Bultan. Automated verification of access control policies. Technical Report 2004-22, Department of Computer Science, University of California, Santa Barbara, Sept. 2004.
15. G. Hughes and T. Bultan. Automated verification of XACML policies using a SAT solver. In *Proceedings of the Workshop on Web Quality, Verification and Validation (WQVV '07)*, 2007.
16. D. Jackson. Automating first-order relational logic. In *Proc. ACM SIGSOFT Conf. Foundations of Software Engineering*, Nov. 2000.
17. D. Jackson. *Software Abstractions: Logic, Language and Analysis*. MIT Press, 2006. `http://softwareabstractions.org/`.
18. D. Jackson and C. A. Damon. Elements of style: Analyzing a software design feature with a counterexample detector. *IEEE Trans. Softw. Eng.*, 22(7):484–495, July 1996.
19. D. Jackson, I. Schechter, and I. Shlyakhter. Alcoa: the Alloy constraint analyzer. In *Proceedings of International Conference on Software Engineering*, Limerick, Ireland, June 2000. IEEE.
20. S. Jajodia, P. Samarati, M. L. Sapino, and V. S. Subrahmanian. Flexible support for multiple access control policies. *ACM Transactions on Database Systems*, 26(2):214–260, 2001.
21. S. Jajodia, P. Samarati, and V. S. Subrahmanian. A logical language for expressing authorizations. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 31–42, Oakland, CA, USA, 1997. IEEE Press.
22. S. Jajodia, P. Samarati, V. S. Subrahmanian, and E. Bertino. A unified framework for enforcing multiple access control policies. In *SIGMOD'97*, pages 474–485, Tucson, AZ, May 1997.
23. S. Krishnamurthi. The CONTINUE server. In *Symposium on the Practical Aspects of Declarative Languages*, Jan. 2003.
24. D. Marinov, A. Andoni, D. Danilinc, S. Khurshid, and M. Rinard. An evaluation of exhaustive testing for data structures. Technical Report MIT-LCS-TR-921, MIT CSAIL, 2003.
25. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *39th Design Automation Conference (DAC 2001)*, Las Vegas, June 2001.
26. G. Naumovich. A conservative algorithm for computing the flow of permissions in Java programs. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '02)*, pages 33–43, July 2002.
27. D. A. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2:293–304, 1986.
28. P. Samarati and S. De Capitani di Vimercati. *Foundations of Security Analysis and Design*, chapter 3, pages 137–196. Springer Verlag, 2001.
29. R. Sandhu and P. Samarati. Authentication, access control, and audit. *ACM Computing Surveys*, 28(1):241–243, 1996.
30. R. S. Sandhu. Lattice-based access control models. *IEEE Computer*, 26(11):9–19, Nov. 1993.
31. R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.
32. R. S. Sandhu and P. Samarati. Access control: Principles and practice. *IEEE Commun. Mag.*, 32(9):40–48, 1994.
33. A. Schaad and J. Moffet. A lightweight approach to specification and analysis of role-based access control extensions. In *7th ACM Symposium on Access Control Models and Technologies (SACMAT 2002)*, June 2002.
34. eXtensible Access Control Markup Language (XACML) version 1.0. OASIS Standard, Feb. 2003. `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml`.
35. Extensible markup language (XML) 1.0 (second edition). W3C, `http://www.w3.org/TR/REC-xml`, 2000.
36. XML Schema part 2: Datatypes. W3C Recommendation, `http://www.w3.org/TR/xmlschema-2/`, May 2001.
37. J. Zao, H. Wee, J. Chu, and D. Jackson. RBAC schema verification using lightweight formal model and constraint analysis. In *Proceedings of the eighth ACM symposium on Access Control Models and Technologies*, 2003.