Using Expert Systems to Statically Detect "Dynamic" Conflicts in XACML

Bernard Stepien and Amy Felty
School of Electrical Engineering and Computer Science
University of Ottawa
Ottawa, Canada
(bernard | afelty)@eecs.uottawa.ca

Abstract—**Policy specification languages such as XACML often provide mechanisms to resolve dynamic conflicts that occur when trying to determine if a request should be permitted or denied access by a policy. Examples include "deny-overrides" or "first-applicable." Such algorithms are primitive and potentially a risk for corporate computer security. While they can be useful for resolving dynamic conflicts, they are not justified for conflicts that can be easily detected statically. It is better to find those at compile time and remove them before run time. Many different approaches have been used for static conflict detection. However, most of them do not scale well because they rely on pair-wise comparison of the access control logic of policies and rules. We propose an extension of a Prolog-based expert system approach due to Eronen and Zitting. This approach uses constraint logic programming techniques (CLP), which are well-adapted to hierarchical XACML policy logic and avoid pair-wise comparisons altogether by taking advantage of Prolog's built-in powerful indexing system. We demonstrate that expert systems can indeed detect conflicts statically, even those that are generally believed to only be detectable at run time, by inferring the values of attributes that would cause a conflict. As a result, relying on the XACML policy combining algorithms can be avoided in most cases except in federated systems. Finally we provide performance measurements for two different architectures represented in Prolog and give some analysis.**

Keywords: XACML, access control, Prolog, constraint logic programming, hierarchical data bases.

I. INTRODUCTION

XACML [1][2] is an XML based language for specifying access control policies using a rich set of datatypes, complex logical expressions and an unlimited number of user-selected attributes. It also includes a concept of conflict resolution algorithm which is used when several policies match the values of an access control request and yield conflicting effects (permit/deny) or conflicting obligations. This facility provides the policy maker with a choice of three strategies: first-applicable, permit prevails and deny prevails. While these algorithms were thought to be satisfactory in early implementations [7] of XACML, the increasing use of XACML in industry led to the awareness that these algorithms were, in fact, not satisfactory and sometimes even led to dangerous situations. Consequently, this resulted in extensive research and eventually in new algorithm definitions in version 3.0 of XACML. Among the many proposals, we mention a few that characterize specific approaches. Reference [11] proposes a prioritization of policies to evaluate the policy combining algorithms to take into account risks. Reference [20] considers various ideas about the concept of majority in a setting where several different results may be returned when evaluating a request against a policy, and the final effect must be determined from these results. Reference [8] uses an additional policy combining language (PCL) to enhance the existing policy combining algorithms by ranking their results especially when policies or rules have returned an indeterminate or not applicable result. This leads to the use of matrices that show the final effect of combining conflicting returned effects. PCL reveals both conflicts and errors. Reference [10] introduces a method that dynamically chooses different combining algorithms based on the request context. Here the policy author status is considered using a decreasing order of precedence: legal authority, data issuer, data subject and data controller. It also considers obligation conflicts in addition to the more traditional effect conflicts. Reference [22] proposes a solution based on hybrid logic to resolve conflicts in collaborative systems.

II. BACKGROUND

The policy or rule conflict combining algorithm feature of XACML is largely based on the belief that some conflicts cannot be detected at compile time. We call such conflicts *dynamic conflicts* in this paper. There are two distinct categories of dynamic conflicts: the first results primarily from the fact that some attribute values, such as environment attributes cannot be known at compile time. For example the balance of a banking account is known to be fluctuating by definition, the second results from federated systems where each member of a federation produces its own unpredictable effect for the other members of the federation. On the other hand, static conflicts can be detected at compile time. A static conflict is the case where all attributes involved are present in both policies being compared and have identical operations that perform on identical values, but have opposite effects.

Policies in XACML are specified hierarchically. Policy *sets* are at the top level, followed by *policies*, and then *rules*. Our naming convention in examples will make it clear what level we are concerned with, though most of the discussion applies to any level in the hierarchy.

A. A Review of Conflict Detections Techniques

Much research on policy conflict detection has been summarized in [4]. It proposes a classification of 26 different approaches into six basic categories: formal methods, model checking methods, matrix based approaches, mining techniques, mutation testing techniques and others including expert systems. It also clearly identifies the type of conflict detection it can perform: static or dynamic. More important is the fact that a wide majority of approaches handle only static conflict detection. Most approaches are based on transformation of the original XML coded policies into formal models. Reference [5] uses VDM, a state-based formal modelling language. It then uses a testing approach to detect inconsistencies by defining test oracles and checking their validation. Reference [6] transforms policies using first-order temporal logic. Reference [23] describes a conflict detection algorithm for a subset of XACML that is verified to be correct using theorem proving techniques. The algorithm handles some dynamic conflicts, but a full analysis of their kinds is not given. Reference [12] uses a graph representation and applies a transformation into a decision tree. Normally conflicts are detected when a terminal node of the decision tree has both a permit and a deny effect attached to it. Reference [19] uses a combination of decision tree and data mining in order to take into account absent attributes that the authors call missing attributes.

B. The Importance of Absent Attributes

One of the characteristics of XACML is the capability to use an arbitrary number of attributes and also, importantly, to leave them out. This applies to both the target section of the different XACML components and the rule condition section. Thus, frequently, XACML conflicts result from the fact that some attributes are absent. The following example illustrates the impact of absent attributes on determining conflicts. Here we have a policy that contains two rules that use two subsets of different attributes, each that do not overlap.

rule_1 := A_1 matches v_1 /\ A_2 matches v_2, effect = permit
rule_2 := A_3 matches v_3 /\ A_4 matches v_4, effect = deny

In the above policy, rule *rule_1* uses only attributes A_1 and A_2 while rule *rule_2* only uses attributes A_3 and A_4. This creates a de facto conflict because the respective absent attributes real meaning is an implicit condition that matches *any value* implicitly. This *any value* would in turn match any specific value from the rule being compared to because any value includes the specific value of the other rule. Here we can easily discover that value v_1 for attribute A_1 in *rule_1* matches the implicit value *any value* of *rule_2* due to the absence of attribute A_1 in that rule. This can be easily verified by submitting a request to a PDP with exactly the values for each of the four attributes found in the two rules, i.e. v_1 for A_1, v_2 for A_2, v_3 for A_3 and v_4 for A_4. By setting alternatively the rule combining algorithm to either permit-prevails or deny-prevails, the PDP will return the effect corresponding to what was set in the combining algorithm, thus revealing that indeed there is a conflict.

In fact, such conflicts resulting from the wide use of subsets of attributes can be explained by the differences in concerns of various departments or sub-organizations of an enterprise. Each department focuses on the attributes that are particularly relevant to them. Such conflicts can be easily detected using expert systems, which have often been implemented in Prolog. There are two parts to an expert system as shown on **Figure 1**: the first is a knowledge base that in our case corresponds to a XACML policy set, and the second is an inference engine that consists of some logic used to verify some properties, which is achieved by querying the knowledge base. In our case, this could be queries to find who has access to a given resource or queries to determine if there are conflicts in the policy set. While the literature has shown that one of the great advantages of Prolog is the conciseness of the specification of both knowledge bases and inference engines, in this paper we determine that performance is mainly dependent on the architecture of the knowledge base. There is a quasi one-to-one mapping between XACML operators and corresponding Prolog operators either built-in or user-defined. This is in sharp contrast with other methods such as SMT [21] where several formulas are required to describe one particular XACML operation. This in turn poses the question of how to formalize a policy set in Prolog in the most effective way.

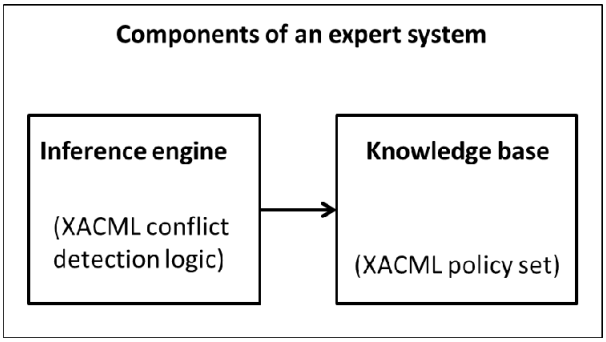

Figure 1. Expert system overview

Interestingly, it is also the case that decision trees can be ineffective for detecting conflicts in the case of absent attributes. In a decision tree, a conflict is detected when there are opposite effects attached to a leaf node. This is clearly the case of rules R_1 and R_2 in **Figure 2**, which have opposite effects. This can be seen by the fact that there are two different edges for the effect attached to the leaf node in the corresponding decision tree. However, for rule R_3 the fact that attribute A_2 is absent creates an edge for the effect that branches out before a leaf node has been reached. Thus, in theory and especially in more complex decision trees, the conflict would not be detectable using decision trees as explained in [12].

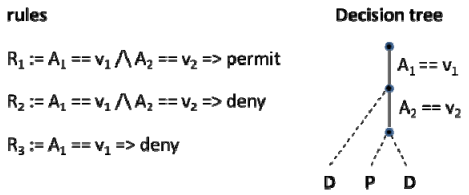

Figure 2. Absent attributes and decision trees

C. *Benefits of Absent Attributes When Specifying Golden Rules*

One particular domain where the principle of absent attributes is very useful is in the specification of golden rules usually set by upper management. Golden rules are usually high-level rules that by definition use only subsets of attributes. For example, an organization may specify that the employees of department A cannot look at information of department B. Verifying these golden rules can be easily achieved with conflict detection techniques after formalizing the golden rules and integrating them with all the fine grained policies and rules of that organization.

III. XACML POLICY REPRESENTATIONS IN PROLOG

There are several factors to consider in a Prolog representation of a XACML policy set, which represents our knowledge base. One of them is the fact that an expression on a given attribute may be distributed among different XACML components. For example attribute *A* may be used in the target of the policy for policy P_1, while it may be used in the target or condition of rule R_2 in policy P_2. This implies that conflict detection must be performed using a complete trace in the policy sets, policies and rules hierarchy.

A. *Eronen and Zitting Prolog Representation of Firewall Rules*

The basic principle of the inference engine that is used to detect conflict using an expert system, as developed by Eronen and Zitting [3], consists of calling a rule predicate twice as follows, using common Prolog variables to represent the attributes values:

```
01 find_conflict(A1, A2, A3):-
02   rule(R1, A1, A2, A3, permit),
03   rule(R2, A1, A2, A3, deny),
04   display_conflict(R1, R2, A1, A2, A3),
05   fail.
06 find_conflict(_, _, _).
```

where a typical rule would be represented by an individual predicate where the body represents the conditions on each attribute as follows:

```
rule(r1, A1, A2, A3, permit):-
  A1 #= 5,
  A2 #> 10,
  A3 #< 5.

rule(r2, A1, A2, A3, deny):-
  A1 #= 5,
  A2 #< 20,
  A3 #> 2.
```

Thus, this model uses a multi-predicate architecture approach. The first query to the knowledge base consists of finding a rule with effect `permit` in line 02. In this example it will find rule `r1` and will retrieve values that satisfy this predicate. These are found in the body of the predicate and are conveyed through its variables (5 for `A1`, lower bound 10 for `A2` and upper bound 5 for `A3`). These values, in turn, are passed via the parameters of the second rule's predicate, called in line 03 for an effect `deny`, but this time forcing Prolog to find a rule where its conditions (body of the rule) are satisfied using these values. Here both rules specify a value of 5 for attribute `A1`, a lower bound of 10 for attribute `A2` that satisfies the upper bound of 20 rule `r2` and finally an upper bound of 5 for attribute `A3` that satisfies the lower bound of 2 for rule `r2`. If the values between these two rules fully match or are within the bounds of the proposed values, then this is a conflict. All of this relies on Prolog unification. A XACML match is translated to a unification operation. Unification (A = B) works in two different ways depending which of the operands are instantiated. If both A and B are instantiated, the unification operation is equivalent to an equality operation (==) which means it will succeed if both values are equal but fail if they are different. However, a critical difference is that unification will also instantiate any open variable with the value on the other side of the operand while the equality operator will not. If both are open variables, they will remain open variables, which in our case means any value.

Note that since this expert system was designed to detect conflicts in firewalls, there is a fixed and fully predictable number of attributes that are represented as parameters of the rule predicate. In our case there are three attributes variables `A1`, `A2` and `A3`. This can only be solved using constraint logic programming [14] (CLP), which explains the special operators #=, #> and #<. These are defined in the CLP library available from SWI-Prolog [13].

Adapting the Eronen–Zitting Prolog model to XACML poses an architectural problem. The firewall rules are a list of individual self-contained rules where all the AC logic on all attributes can be found (none are absent), and importantly, where the effects are immediately given in the predicate header. XACML is a hierarchical model that is composed of components (policy sets, policies and rules) in a parent-child relationship. Moreover, AC logic is distributed among the components in an unconstrained way, i.e. a condition on a given attribute can be located in any of the basic components and/or attributes can be absent altogether. Also, effects are specified only in the leaves of the hierarchy (rule components). Thus, the implementation of XACML in Prolog can follow two different architectural patterns:

- Map each XACML component to a separate Prolog predicate which will result in a set of multiple predicates.

- Represent a XACML policy set as a single Prolog predicate with hierarchical structure of clauses in its body, thus mimicking the hierarchical nature of the XML document.

We will examine the advantages and drawbacks of these two approaches in subsequent sections using the same example policy set operating on three attributes A_1, A_2 and A_3 with identical logic in the two different approaches.

B. Multiple Predicates Approach

As mentioned, the three basic components of a XACML policy set are policy set, policies and rules. Each of them may have a target composed mainly of simple matches for attributes, while rules may also have a condition for complex expressions such as time intervals. A XACML policy set is recursive in the sense that it can have other policy sets as children. However, targets also allow representing complex expressions with the implicit disjunction of the *anyOf* construct or the implicit conjunction of the *AllOf* construct, though in a limited way. The following example shows how one can use ideas from relational databases to represent a policy set that contains two policies that in turn contain two rules each. A child component will always contain the identifiers of its parents. Each predicate contains a list of attributes as open variables. The list is identical for each predicate and shall contain all the variables whether they are used or not in the predicate's body.

```
01 policy_set('PS1', [A1, A2, A3]).

02 policy('PS1', 'P1', [A1, A2, A3]):-
03  A1 = a.

04 policy('PS1', 'P2', [A1, A2, A3]):-
05  A2 = c.

06 rule('PS1', 'P1', 'R_1',
07            [A1, A2, A3], permit):-
08  A2 = c.

09 rule('PS1', 'P1', 'R2',
            [A1, A2, A3], permit):-
10  A2 = d.

11 rule('PS1', 'P2', 'R3',
12            [A1, A2, A3], deny):-
13  A1 = a.

14 rule('PS1', 'P2', 'R4',
            [A1, A2, A3], 'permit'):-
15  A3 = b.
```

The conflict detection inference engine we define is similar to the original model of Eronen and Zitting in that it uses two queries to the Prolog knowledge base, however, in this case, we consider the natural hierarchy of XACML components, and thus the query to a rule of the firewall model is replaced by three separate queries to each XACML component that follows the hierarchy. This basically means finding the policies (line 03) that belong to a policy set (line 02) and finding the rules (line 04) that belong to its parent policy. Then once values for attributes have been obtained via the above queries they are used in another similar set of queries to determine if there is a conflict (lines 05-07). If the second set of queries are satisfied, this indicates a conflict and in this case, we can display the trace involved (policy set, policy and rule) and the values that have satisfied the conditions.

```
01 find_conflicts:-
02  policy_set(PS1, AL),
03    policy(PS1, P1, AL),
04      rule(PS1, P1, R1, AL, permit),
05  policy_set(PS2, AL),
06    policy(PS2, P2, AL),
07      rule(PS2, P2, R2, AL, deny),
08  display_conflict(PS1, P1, R1,
09                   PS2, P2,R2, AL),
10  fail.
11 find_conflicts.
```

There are some differences with the original Eronen-Zitting conflict detection inference engine. First of all, we now use lists of attributes (AL) since in XACML the number of attributes varies from application to application. However, for a particular application, the same number and nature of attributes shall be present in each predicate in order to take into account absent attributes. Then, the variables indicating the policy set and policy are present in the rule so that relational database principles can be used to extract traces in the policy set using Prolog backtracking. One drawback of this approach is that the inference engine cannot handle the recursive definition of policy sets easily, which may impact performance.

C. Single Predicate Approach

In this approach, instead, the knowledge base for the same policy set as above is represented using a single predicate where the body is a logical tree. The child components of a policy or rule are represented deeper inside the tree. The top level is a policy set (line 01), the two policy definitions start at line 06 for policy `P1` and line 18 for policy `P2`. Then the rule definitions are given at lines 09 and 12 for the rules of policy `P1` and lines 21 and 24 for the rules of policy `P2`.

```
01 policySet(PS, P, RL, [A1, A2, A3],
02                                EF):-
03  PS = 'PS1',
04   (
05    (
06     P = 'P1',
07     A1 = 'a',
08     (
09       ( RL = 'R1', A2 = 'c',
10                       EF = 'permit')
11       |
12       ( RL = 'R2', A2 = 'd',
13                       EF = 'permit')
14       )
15     )
16    |
17    (
18     P = 'P2',
19     A2 = 'c',
```

```
20     (
21       ( RL = 'R3', A1 = 'a',
22                            EF = 'deny')
23       |
24       ( RL = 'R4', A3 = 'b',
25                           EF = 'permit')
26       )
27     )
28   ).
```

With such a definition of the knowledge base, conflicts can be detected with an inference engine using two queries to the `policySet` predicate, the first (line 02) for the `permit` effect that will extract values for attributes that satisfy this predicate and the second (line 03) with the `deny` effect where the values obtained in the call of the predicate in line 02 will verify if a predicate found with the call of line 03 can be satisfied. In our case, this would find all conflicts within the unique policy set (highest level policy set that contains either other child policy sets or policies and their rule children).

```
01 find_conflicts:-
02  policySet(PS1, P1, R1, AL, permit),
03  policySet(PS2, P2, R2, AL, deny),
04  display_conflict(PS1, PS2, P1, P2, R1,
05                                R2, AL),
06  fail.
07 find_conflicts.
```

In addition, this approach does not present any problem for recursive definitions of policy sets since it is displayed naturally in the tree representation hierarchy on the knowledge base side and the inference engine calls only one predicate (policy set).

Both multiple and simple predicate approaches produce the same results executing the inference engine:

```
10 ?- find_conflicts.

found conflict for PS1 P1 R1   vs PS1 P2 R3
values:  [a,c,_G2503]
found conflict for PS1 P2 R4   vs PS1 P2 R3
values:  [a,c,b]
true.
```

There are two conflicts. The first one shows an open variable for attribute `A3` revealing an absent attribute while the second returns values for all attributes.

D. Optimizing the Use of Constraint Logic Programming

In a first step in previous work, and in order to take into account the disappearance of Eclipse Prolog used by Eronen and Zitting, we attempted to modify the SWI-Prolog CLP library by extending it to non-numeric data types by allowing non-numeric values for the unification operator '#='. This initially worked under 32 bit architectures but stopped working on a 64 bit architecture. This experience made us aware of two problems:

- It created maintenance obligations every time the SWI-Prolog CLP library was modified by their authors.
- Our own data type related modifications no longer worked since the CLP library was completely re-written and some previously used built-in predicates were replaced by new predicates that were no longer compatible.

Thus, in our new approach we decided to separate the processing of data types, using SWI-CLP only for numeric types and standard Prolog for non-numeric types. This was not initially obvious since CLP uses a separate set of logical operators (unification: '#=', conjunction: '#/\' and disjunction: '#\/'). We noticed that non-numeric attribute logic can use the standard Prolog logical operators (unification: '=', conjunction: ',' and disjunction: '|'). Note the use of the unification '=' operator instead of the '==' operator. This distinction is important when a free variable representing an absent attribute (Prolog open variable) is compared to an instantiated value.

However, the separation of processing of numeric and non-numeric constraints between regular and CLP-Prolog is not easy because some operations on non-numeric constraints involve numeric aspects. Similar problems were encountered by other methods such as [9]. The simple example of the use of string data type illustrates this problem. Effectively, there are a number of operators on strings that involve a numeric element. The built-in `sub_string` operator is a perfect illustration. Sub-strings have start and length values that are numeric. For example, in Java we have:

```
String a = b.substring(5, 15);
```

SWI-Prolog has an equivalent built-in predicate: `sub_string(AS,S,L,AL,SS)`. Thus, if we consider the following two policies, we normally should detect a conflict since any string starting with the substring `ab` would conflict with the string `abcd` that also starts with `ab`.

```
policy(1, [X], deny):-
  sub_string(X,0,2,_,ab).

policy(2, [X], permit):-
  X = abcd.
```

Here the problem is that standard Prolog would generate the error *arguments are not sufficiently instantiated* in the case we query the conflict as:

```
detect_conflict:-
  policy(P1, AL, deny),
  policy(P2, AL, permit),
  nl, write('conflict has been detected'),
  fail.
```

This is because in policy `P1` the value of the Prolog open variable `X` is not known (not instantiated for policy `P1`. When inverting the queries in the inference engine above for the policy predicate starting with `permit` followed by `deny`, the conflict is normally detected because policy `P2` gets retrieved first, returns the value `abcd` for variable `X` that is passed to policy `P1` where the built-in predicate

sub_string can operate normally with a fully instantiated value for X.

Solving such cases can be done only by writing special predicates in order to handle the built-in sub_string predicate, rather than attempting to use the Prolog built-in predicate directly.

This example had at least an instantiated value for variable X in policy P2. In the case where attribute X (variable) would have been absent, this case would be considerably more complex. Here the conclusion should be that there is a conflict, which means that any string starting with ab would result in a conflict. The modified version of this policy set now uses our own defined my_sub_string predicate in line 02:

```
01 policy(1, [X], deny):-
02   my_sub_string(X,0,2,_,ab).

03 policy(2, [X], permit):-
04    X = abcd.

05 policy(3, [X], permit).
```

In the policy set above, with a separate definition of the my_sub_string predicate, we can now consider the fact that some of the predicate bodies contain open variables. This is particularly the case of the newly added policy 3 with absent attribute X (an attribute on which there are no constraints specified in the body of the predicate such as in a typical default XACML catch-all rule). The definition of the custom my_sub_string predicate is as follows:

```
01 my_sub_string(X,S,L,_,SS):-
02   var(X),
03   make_contains(SS, CSS),
04   X = CSS.

05 my_sub_string(X,S,L,_,SS):-
06   nonvar(X),
07   sub_string(X,S,L,_,SS).
```

The first predicate considers the case where the value for attribute X is not known (var check in line 02 and returns a message with the word contains followed by the value of the proposed substring. The second predicate considers the normal case (nonvar check in line 06) where all attributes are instantiated and merely refers to the Prolog built-in predicate sub_string. Here, the above definition of my_sub_string will result in the value 'contains ab between position 0 and 1' for the conflict detected between policies 1 and 3. This example is particularly important because many XACML policies for controlling access to web applications are centered on the sub-domains of URLs, and are sometimes specified using the XPATH operator. All of these are cases of sub-string comparisons.

IV. Dynamic Conflicts – Myth or Reality

Dynamic conflicts have two well identified sources: the first lies in federated systems where policies from different commercial entities are evaluated commonly. Since each entity does not have access to the other participating entities' policy sets, they cannot compare them at compile time in order to pre-detect conflicts. This actually applies to both static and dynamic (attribute values unknown at compile time) conflict detection. We are not addressing conflicts due to federated systems. The second is within a single policy set belonging to a single commercial entity. This one is believed to be dynamic because values of some attributes, usually environment attributes, are not known at compile time. We have found two categories of particularly interesting examples in the literature, described in the next two subsections, where the authors claim that some conflicts can be detected only at run time, which they refer to as dynamic conflicts. The first one is a case of absent attributes like we have described at the beginning of this paper, while the second one is a case that involves numeric constraints that can be typically solved using CLP.

A. Example of Dynamic Conflicts Resulting from Absent Attributes

M.Hall-May and T. P. Kelly [16] provide a particularly interesting military example with two conflicting policies that they claim to be detectable only dynamically at run time as follows:

- Agent *a* is permitted to enter *no-fly zone*
- Agent *a* is forbidden to enter any zone that may harbour *undetected hostiles*

They claim that "the target of the policy statement is dynamically evaluated at the moment it is enforced and, if it is determined that the no-fly zone potentially harbours undetected hostiles, then the agent is both permitted and forbidden from entering that zone." These two policies certainly conflict but one does not need to wait for run time to detect the conflict. Instead, this can be detected by an expert system at compile time. First, we need to formalize these policies by converting them into Prolog as follows:

```
policy(p1, [Agent, Zone,
            HostilesPresence], permit):-
  Agent = a,
  Zone = no_fly_zone.

policy(p2, [Agent, Zone,
            HostilesPresence], deny):-
  Agent = a,
  Zone = _,
  HostilesPresence = true.
```

In fact, the above policies operate on three attributes: Agent, Zone and HostilesPresence. The first policy has no mention of hostile presence, thus its attribute can be considered as absent. The second has an explicit mention of hostile presence but specifies any zone for the zone attribute. Any zone really means any value for the Zone attribute which is represented by the "_" character, which in Prolog means open logic variable. The same could have been represented by merely omitting this attribute variable in the body of the predicate altogether which has the same effect as the explicit *any value* specification. Thus, when we query the expert system we obtain the following results:

```
?-detect_conflicts.

conflict has been detected between
policies: p1 and p2
for values:
   Agent: a
   Zone: no_fly_zone
   HostilesPresence: true
true.
```

Table 1 shows how each attribute value from one policy matches the corresponding value of the second policy, and in the case of absent attributes is inferred by Prolog unification. In this case, it is a full match between the two policies having `permit` and `deny` as effects. This is a dynamic conflict that has just been detected statically. The conclusion of this example is that many people get confused by the effects of absent attributes in matching values and conclude wrongfully that this is a case of dynamic conflict detectable only at run time. Here, CLP was not even necessary as the effect of absent attributes was sufficient to detect the conflict. This is true when the data types are string or Boolean, i.e. non-numeric.

Policy	Agent	Zone	HostilesPresence	effect
p1	a	No_fly_zone	Any value	Permit
P2	a	Any value	True	Deny

Table 1. Attribute value comparison

B. Dynamic Example Resolved with CLP

In [17] the authors claim "For example, policies which increment or decrement allocation of resources may conflict with policies related to setting upper and lower bounds for the resources. These conflicts result from current state of the resource allocation and bounds so can only be detected and resolved at run-time." While this is a perfect definition of a case of dynamic conflicts resulting from numeric constraints, their policies are not expressed in XACML.

Another classic dynamic conflict example in the financial domain comes from a policy for debit transactions on a chequing account that has two conflicting rules. The first rule permits a debit to be made as long as the transaction amount is less than the balance of the account. The second rule tries to protect the customer against fraudulent transactions and limits the amount of a debit to $3000 by specifying that any amount over $3000 is denied. Here, most people think that this conflict is not detectable at compile time because the balance, which is an environment attribute, varies in time and thus cannot be predicted. Instead, we can apply CLP immediately at compile time and discover that these two rules conflict for any transaction amount of $3001 to infinite and the balance of $3002 to infinite (when specified as integers to simplify the reasoning here). Using the simplified multiple predicates approach, this example is represented in Prolog as follows:

```
policy(p1, [Transaction, Amount,
                                    Balance]):-
  Transaction = 'debit'.

rule(p1, r1, [Transaction, Amount,
                      Balance], permit):-
  Amount #< Balance.

rule(p1, r2, [Transaction, Amount,
                       Balance], deny):-
  Amount #> 3000.
```

Effectively, for rule `r2` to evaluate to true, the amount must be over 3000, which means in the range of 3001 to infinite. This range is passed on to rule `r1`, where this lower bound of 3001 on the amount forces the lower bound of the balance to be 3002. These results have been effectively produced by CLP Prolog as a query to the Prolog engine shown as follows:

```
17 ?- find_conflicts.
found conflict for ps1 p1 r1  vs ps1 p1 r2
values:
debit
_G1721{3001..9223372036854775806}
_G1893{3002..9223372036854775807}
true.
```

However, this in an interesting case that goes beyond the conflict detection problem. Indeed, such policies are very common in banking and for good reasons. The fact that they conflict is one problem. Using a deny prevails rule conflict resolution algorithm, which is the most probable choice of banking institutions, is further adding to the confusion. This case raises the question of correct rule engineering altogether. Here, both rules are necessary to fulfil the goals of a bank. Thus, the natural solution of the problem would be to re-engineer these two rules by merely merging them together with slight logic modifications as follows:

```
rule(p1, r1, [Transaction, Amount,
                 Balance], permit):-
  Amount #< Balance
  #/\ Amount #<= 3000.
```

Thus, the problem of conflict resolution raises a new avenue of research in automating the process of re-engineering conflicting rules or policies. Here, a hint would have been that reversing the condition for the authorized ceiling of $3000 would transform rule `r2` into a rule with an effect `permit` which then can be analyzed to be complementary. Methods to achieve this solution have also been studied in [9].

V. PERFORMANCE AND SCALABILITY

A. Taxonomy of Conflicts

In order to determine a significant pattern for generated policies and rules for the purpose of performance measurement, we have after trial and error determined some interesting properties. The most obvious case of conflict is of course the one where two traces through policies and rules have identical attribute names and corresponding values and operators but with opposite effect. If we take the

simplest form this would be two rules in the same policy such as:

rule_1 := A_1 matches $v_1 \wedge A_2$ matches v_2, effect = permit
rule_2 := A_1 matches $v_1 \wedge A_2$ matches v_2, effect = deny

The second pattern consists of overlapping disjunctions such as:

rule_1 := (A_1 matches $v_1 \vee A_1$ matches v_3) $\wedge$
A_2 matches v_2,
effect = permit
rule_2 := (A_1 matches $v_3 \vee A_1$ matches v_4) $\wedge$
A_2 matches v_2,
effect = deny

In the above rules, the value v_3 for attribute A_1 would cause a conflict.

Finally, another factor for conflicts is the capability to distribute an attribute involved in a match anywhere in the XACML hierarchy, i.e. the targets of policy sets, policies or rules and within those in any sub-element (*anyOf* or *AllOf*). Thus, we may consider for this purpose to invert the order of the match expressions as follows:

rule_1 := A_1 matches $v_1 \wedge A_2$ matches v_2, effect = permit
rule_2 := A_2 matches $v_2 \wedge A_1$ matches v_1, effect = deny

The results of such inversions did not produce any difference in performance regardless of the size of the policy set. Thus, we determined that in fact there is a major difference between an XML text document that is a XACML policy set and its representation in Prolog. The Prolog representation is in fact a program, no longer a piece of text. The difference between searching for an element (attribute name and value and operation being performed) in a text requires going through the entire text. In our case, we need to go through the expression for attribute A_2 before reaching the targeted expression for attribute A_1. When converted to Prolog, the above example, even in the multiple predicates approach has a form of a function where the Prolog variable representing the attribute in the parameters of the predicate is then reachable directly, regardless to its position in the body of the predicate and even in the utmost complex policy or rule logic.

```
rule(p1, r1, [A1, A2], permit):-
  A1 = v1, A2 = v2.

rule(p1, r2, [A1, A2], deny):-
  A2 = v2, A1 = v1.
```

B. Performance Measurements

For the purpose of this study we have generated large predictable policy sets using six attributes from an attribute data model for medical access control. By predictable, we mean generating all combinations of attribute values for two identical sets of policies and rules, half of them with the effect *permit* and the other half with the effect *deny*. This always produces exactly as many conflicts as half of the total number of rules even with the disjunctive model that here would consist only of two rules. In a second step we have modified according to a set pattern the initial set of policies by removing some of the attribute conditions so as to simulate absent attributes. The number of conflicts resulting from absent attributes is still fully predictable. Effectively, a given trace will now conflict with all the traces that are derived from the various values of the alphabet of the absent attribute. In our case, we have produced three different policy sets with variable numbers of rules. First, we have illustrated the two different approaches to representing a policy set in Prolog (multiple predicates approach or single predicate approach). Performances for various sizes of rules are shown in Table 2 for the single predicate approach and in Table 3 for the multiple predicates approach. The third one illustrates a XACML programming style that we call *disjunctive predicate,* where the use of XACML targets with the *AnyOf/Allof* operators are flattened into a conjunction of disjunctions as discussed in [15]. To illustrate, the type of policy set that we generate can be collapsed to an equivalent single policy of the following form, one for each effect:

single_policy 1 =
(A_1 matches v_{11} | … | A_1 matches v_{1n}) $\wedge$ .. $\wedge$
(A_n matches v_{n1} | … | A_1 matches v_{nm}),
effect = permit

single_policy 2 =
(A_1 matches v_{11} | … | A_1 matches v_{1n}) $\wedge$ .. $\wedge$
(A_n matches v_{n1} | … | A_1 matches v_{nm}),
effect = deny

The performance of the disjunctive predicate is shown in Table 4.

number of rules	number of conflicts	Execution time (ms)
1920	960	413
4480	2240	1583
8960	4480	2552
12544	6272	8774

Table 2. Performance - single predicate approach

number of rules	number of conflicts	Execution time (ms)
1920	960	867
4480	2240	3896
8960	4480	14562
12544	6272	30667

Table 3. Performance - multiple predicate approach

number of rules	number of conflicts	Execution time (ms)
2	960	202
2	2240	425
2	4480	934
2	6272	1310

Table 4. Performance - disjunctive predicate

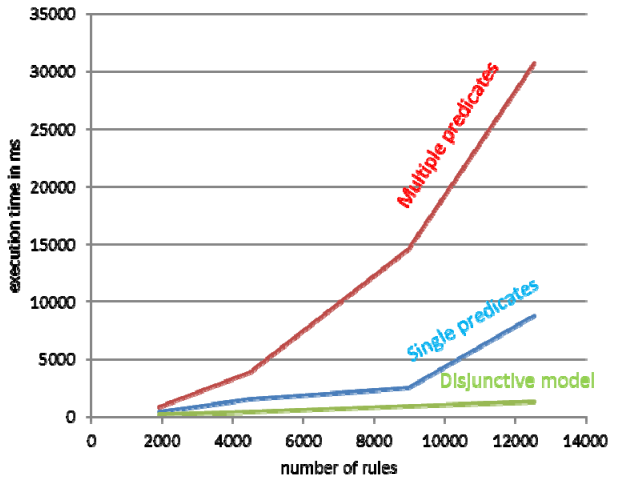

Figure 3. Performance comparison

Figure 3 shows that the multiple predicates approach has a rapidly deteriorating performance when the number of policies or rules increases while the single predicate approach follows a similar pattern but to a lesser degree. The single predicate approach performance represents a 75% reduction of execution time compared to the multiple predicates approach. More striking is the impact of programming style, which seems to be quasi-independent of the size of the problem. With this form, there are only 2 rules, one for *permit* and one for *deny* effect, but the combination of attribute values is the same and produces exactly the same number of conflicts as was already shown in [15].

However, while the single predicate approach is considerably more efficient from a performance point of view, the multiple predicates approach proves to be more efficient from an administration point of view. Effectively, the single predicate approach does not allow easy modification of the policy set in Prolog. Even a minute modification requires retracting the entire predicate and re-asserting it, while the multiple predicates approach only requires retracting and re-asserting the predicate of the targeted components, leaving all others untouched. This also applies to indexing. This is particularly useful when using the expert system for editing new policies or rules. Thus, here the time to re-load a modified policy set is dramatically different.

Also, we have observed that performance degrades rapidly when numeric constraints are present and require CLP. This is due to the fact that CLP computes the bounds of intervals in several steps compared to the simpler single unification mechanism that non-numeric values use.

C. *Tentitative Explanations of the Results*

Prolog indexing is the prime cause of the above noted performance differences. Indexing varies considerably between versions of Prolog, However, the current version of SWI-Prolog uses the just-in-time principle over multiple arguments. The SWI-Prolog documentation mentions that these indexes are not built at compile time but at call time. Thus, there is a significant amount of sometimes redundant indexing going on with the multiple predicates approach, because an indexing is performed for each separate predicate. In the single predicate approach, this indexing occurs only once. Now, the particular spectacular performance of the disjunctive model can be further attributed to the way backtracking is performed. When match expressions for the same attribute are grouped together in a single disjunction, the backtracking is easier because it can walk through the tree of solutions without having to reset some indexes and other markers, as in the case where expressions are scattered throughout the XACML elements.

D. *Architectural Strategy*

With the above performance results, it is time to make a possible choice between the two architectures, considering both run-time performance and load-time considerations. We have determined that indeed both approaches can be used simultaneously depending on the context of the use. The multiple predicates approach has a negligible reload cost since only the modified or newly added elements need to be asserted. Also, in this case, conflict detection requires comparing only these new elements to the rest of the knowledge base (a one to many comparison) and thus does not require going through each pair of policies (many to many comparisons). On the other hand, a one shot analysis for detecting conflicts of the entire knowledge base (many to many comparisons) can benefit from the single predicate approach and its 75% reduction in computing time.

VI. CONCLUSION

In this paper we have shown that dynamic conflicts resulting from unknown values of attributes can be detected at compile time using expert systems that infer the problem solely from the values of attributes present or absent in the policy logic. We have shown that the performance of expert systems applied to XACML policies can vary considerably depending on the architectural approach used to represent a policy set in Prolog as well as on the programming style

used to specify the policy set logic. We also have shown that expert system performance is high because such systems use the built-in indexing mechanism of the Prolog internal database, compared to the more natural approach of pair-wise policy comparisons. Finally, we have shown that some conflicts can be avoided by re-engineering the conflicting policies.

ACKNOWLEDGEMENTS

The authors acknowledge the support of the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

[1] OASIS, *XACML Version 2.0*, 2004, docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf.

[2] OASIS, *XACML Version 3.0*, 2013, http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html.

[3] P. Eronen and J. Zitting, "An expert system for analyzing firewall rules," in *6th Nordic Workshop on Secure IT Systems*, 2001, pp. 100–107.

[4] M. Aqib and R. A. Shaikh, "Analysis and comparison of access control policies validation mechanisms," *International Journal of Computer Network and Information Security*, vol. 7, no. 1, pp. 54–69, 2015.

[5] J. W. Bryans, "Formal analysis of access control policies," in *Proceedings of the UK e-Science All Hands Meeting*, 2006, pp. 701–708, http://www.allhands.org.uk/2006/proceedings/papers/722.pdf.

[6] D. J. Dougherty, K. Fisler, and S. Krishnamurthi, "Specifying and reasoning about dynamic access-control policies," in *3rd International Joint Conference on Automated Reasoning*, ser. Lecture Notes in Computer Science, vol. 4130. Springer, 2006, pp. 632–646.

[7] M. Lorch, S. Proctor, R. Lepro, D. Kafura, and S. Shah, "First experiences using XACML for access control in distributed systems," in *2003 ACM Workshop on XML Security*, 2003, pp. 25–37.

[8] N. Li, Q. Wang, P. Rao, D. Lin, E. Bertino, and J. Lobo, "A formal language for specifying policy combining algorithms in access control," CERIAS, Tech. Rep. 2008-9, 2008, http://core.ac.uk/download/pdf/21173941.pdf.

[9] G. Hughes and T. Bultan, "Automated verification of access control policies using a SAT solver," *International Journal on Software Tools for Technology Transfer*, vol. 10, no. 6, pp. 473–534, 2008.

[10] K. Fatema and D. Chadwick, "Resolving policy conflicts—integrating policies from multiple authors," in *Advanced Information Systems Engineering Workshops*, ser. Lecture Notes in Business Information Processing, vol. 178. Springer, 2014, pp. 310–321.

[11] I. Matteucci, P. Mori, and M. Petrocchi, "Prioritized execution of privacy policies," in *International Workshop on Data Privacy Management and Autonomous Spontaneous Security*, ser. Lecture Notes in Computer Science, vol. 7731. Springer, 2013, pp. 133–145.

[12] S. Pina Ros, M. Lischka, and F. Gómez Mármol, "Graph-based XACML evaluation," in *17th ACM Symposium on Access Control Models and Technologies*, 2012, pp. 83–92.

[13] M. Triska, Constraint Logic Programing over Finite Domains, SWI-Prolog, http://www.swi-prolog.org/man/clpfd.html.

[14] M. Triska, "The finite domain constraint solver of SWI-Prolog," in *11th International Symposium on Functional and Logic Programming*, ser. Lecture Notes in Computer Science, vol. 7294. Springer, 2012, pp. 307–316.

[15] B. Stepien, S. Matwin, and A. Felty, "Strategies for reducing risks of inconsistencies in access control policies," in *5th International Conference on Availability, Reliability, and Security*. IEEE Computer Society, 2010, pp. 140–147.

[16] M. Hall-May and T. P. Kelly, "Towards conflict detection and resolution of safety policies," in *24th International System Safety Conference*, 2006.

[17] M. Charalambides, P. Flegkas, G. Pavlou, J. Rubio-Loyola, A. K. Bandara, E. C. Lupu, A. Russo, M. Sloman, and N. Dulay, "Dynamic policy analysis and conflict resolution for DiffServ quality of service management," in *10th IEEE/IFIP Network Operations and Management Symposium*, 2006, pp. 294–304.

[18] K. Jayaraman, V. Ganesh, M. Tripunitara, M. Rinard, and S. Chapin, "Automatic error finding in access-control policies," in *18th ACM Conference on Computer and Communications Security*, 2011, pp. 163–174.

[19] R. A. Shaikh, K. Adi, and L. Logrippo, "A Data Classification Method for Inconsistencies and Imcompleteness Detection in Access Control Policy Sets," *International Journal of Information Security*, 2016.

[20] N. Li, Q. Wang, W. Qardaji, E. Bertino, P. Rao, J. Lobo, and D. Lin, "Access control policy combining: Theory meets practice," in *14th ACM Symposium on Access Control Models and Technologies*, 2009, pp. 135–144.

[21] F. Turkmen, J. den Hartog, S. Ranise, N. Zannone, "Analysis of XACML Policies with SMT," in *4th International Conference on Principles of Security and Trust,* ser. Lecture Notes in Computer Science, vol. 9036. Springer, 2015, pp. 115-134.

[22] S. Damen, J. den Hartog, and N. Zannone, "CollAC: Collaborative access control," in *International Conference on Collaboration Technologies and Systems*, 2014, pp. 142-149.

[23] M. St-Martin and A.P. Felty, "A Verified Algorithm for Detecting Conflicts in XACML Access Control Rules," in *5th ACM SIGPLAN Conference on Certified Programs and Proofs)*, 2016, pp. 166-175.