Resolving XACML Rule Conflicts using Artificial Intelligence

Bernard Stepien and Amy Felty
School of Electrical Engineering and Computer Science
University of Ottawa
Ottawa, Canada
{bstepien, afelty}@uottawa.ca

ABSTRACT

The XACML access control policy specification language provides a simple rule/policy combining algorithm that is invoked when a request is evaluated against a particular policy set, and the results of the policy decision point (PDP) include solutions with both "permit" and "deny" effects. In short, the combining algorithm allows the policy writer to specify which effect should prevail in case of such conflicts. This feature has long been considered as misleading, and a wide variety of research has been done in an attempt to extend it using supplementary language features or algorithms based on priority definitions. We propose a new algorithm that, instead of absolute priorities expressed as numbers, is based on relative priorities that do not use numerical scales. Two kinds of annotations need to be added to policies, one that says if the value of an attribute is sensitive and another that provides information that can be used to determine which attribute is most important in the case when several sensitive values are encountered during the processing of attribute values in a request. This information serves as input to our decision making mechanism, designed to respect the user-specified priorities as best as possible.

CCS Concepts

• General and reference→General conference proceedings • Theory of computation→Logic and verification •Security and privacy→Access control •Computing methodologies→Knowledge representation and reasoning •Computing methodologies→Anomaly detection •Computing methodologies→Policy iteration.

Keywords

XACML; access control; Prolog; artificial intelligence; logical reasoning.

1. INTRODUCTION

XACML [1], [2] is an XML based language for specifying access control policies. It is highly expressive and includes a rich set of datatypes, complex logical expressions and an unlimited number of user-selected attributes. However, it is very verbose and thus large specifications become rapidly unreadable by human readers. It also includes a 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. In this case, this algorithm 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 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. One of the main issues with XACML is to know whether the logic of a XACML policy set can be considered as a pure Boolean expression. Some people ascertain that theory while others deny it on the basis that a XACML policy set has rule/policy combining algorithms that they consider an integral part of the decision logic [3].

A large portion of literature on the subject of rule and policy conflict resolution is based on the belief that a conflict is an error [4] and thus must be eliminated. Thus, research on static and dynamic conflict detection at compile time has prevailed. However, when looking closely at the intention of XACML, instead we discover that policies and rules define authorization spaces for which they are specifically applicable. This is described fully in [5]. However the problem of determining with accuracy which rule prevails in case of an overlap of authorization spaces remains. Also, since policies and rules are composed by various actors who insert different rules at different times, it is difficult to constantly clean the policy sets or policies of such conflicts as discussed in [6]. Instead, it is more appropriate to define methods to determine which policy and rule is applicable in a certain context.

The following medical example is of particular interest because it provides a good illustration of the weaknesses of the XACML rule combining algorithm. Here we are trying to specify the conditions under which a nurse can access electronic records (action *read*). The first rule specifies that a nurse can read a surgery report without further restrictions. The second rule prohibits nurses from reading any document when the location is home care. And finally, the third rule has no restriction on resources or location but operates in the case of an emergency, i.e. a nurse can read anything and anywhere in an emergency. The following policy set can be viewed as a depiction of a horizontal tree. It illustrates the hierarchy of XACML elements showing the name of the XACML

ICISS 2020, March 19–22, 2020, Cambridge, United Kingdom
© 2020 Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7725-6/20/03…$15.00
https://doi.org/10.1145/3388176.3388188

element and its corresponding target logic. The corresponding full XACML specification is left as an exercise to the reader.

```
policySet ConflictingPolicySet :=
        subject-id matches nurse
 policy NurseReadPolicy :=
            action-id matches read

   rule NurseResourceRule -> permit :=
        resource-id matches surgery report

   rule NurseHomeCareRestrictionRule -> Deny
        := Location matches home care

   rule NurseEmergencyRule -> Permit :=
        Emergency matches true
```

In this example, it is clear that the home care rule conflicts with the resource rule and with the emergency rule in the case of a request of {*subject-id = nurse, action-id = read, resource-id = surgery report*, *Location = home care, Emergency = true*}. Here the use of the XACML rule combining algorithm would produce the following undesired effects:

- Deny prevails would prevent a nurse from reading any document during an emergency.
- Permit prevails would allow a nurse to read documents during home care.

Instead, these three rules provide a complex example of conflicts depending on the situations encountered. Basically, we want the *NurseHomeCareRestrictionRule* to prevail in order to deny access in the case when the location is *home care* and there is no emergency, but we would like to see the *NurseEmergencyRule* prevail to allow access regardless of the location. This example is a case of cascading conflicts that cannot be resolved by a simple XACML rule or policy combining algorithm. This conflict cannot be considered as an error and should not be corrected by removing any of its logic. The traditional recommendation of cleaning the policy of conflicts would also be undesirable because XACML rules can specify only one type of effect, *permit* or *deny*. By cleaning, we mean removing some of the rule logic that is posing a problem.

Also, some may argue that the use of the first-applicable rule combining algorithm and a proper ordering of the rules would solve the problem. It is highly recommended to avoid this. Most industry users that we have talked to have prohibited the use of the first-applicable rule combining algorithm altogether, due to bad experiences using it. In fact, while this algorithm is usable for the above small example, larger policy sets with hundreds or even thousands of rules would easily become unmanageable when trying to determine the correct order. Thus, most authors have decided to come up with new algorithms altogether.

We propose a new solution to deriving the final desirable effect. Instead of any modification such as cleaning, our approach keeps the logic of these three rules (and all rules) intact, and adds a new priority mechanism, based on simple sensitivity assessments of attributes. This mechanism is used in place of the traditional XACML rule/policy combining algorithm. However, we do not use a numerical method such as the one in[7], where priorities are scaled during the evaluation of a request against a policy set, in the process of determining the desired effect. Instead, we propose to use artificial intelligence in the form of automated logical reasoning, which relies on a two-step process of declaring relative priorities: the first step consists of determining which values of an attribute are sensitive, and the second consists of declaring which attributes are more important than other attributes. This information will be used when conflicting cases are encountered. This approach handles the concept of defining authorization spaces as in [5], however without the rule combining algorithm.

The specification of rules is based on the fact that in the absence of an appropriate target logic (i.e., when no policy rule applies), a request would return "not applicable," which is considered as an implicit *deny*. Thus, an explicit *deny* is really meant to ensure that a rule specifying a *permit* effect should exclude any cases covered by rules with an explicit *deny* specification. The problem is that the reverse may also be true.

Although it may appear that our approach supersedes the various methods for conflict detection, we note that these methods can still be very useful. Indeed, they provide material to a policy set administrator that can help to define adequate priorities among authorization spaces. This situation may arise often, mostly because users who define policies may not be aware of other users' policies as indicated in[8]. Also, there are still cases that can be considered as pure errors for which a priority algorithm proves useless. This is the case, for example, when solutions contain exactly the same attributes operating on the same values, such as in the following simple example:

Rule 1: A_1 matches $V_1 \wedge A_2$ matches V_2 => permit
Rule 2: A_1 matches $V_1 \wedge A_2$ matches V_2 => deny

2. BACKGROUND

The list below contains a sample of approaches to conflict detection resolution during the evaluation of requests against access control policies.

[9] proposes an algorithm based on deterministic formal automata, based on matrices representing the effect of a pairwise policy.

[10] proposes an ordered set of conflict resolution rules (CRR). This is in the context of multiple PDPs in collaborative systems.

[11] proposes a system of prioritization of rules and policies using numerical rankings and performing complex operations like computing Eigen values to determine which rule prevails.

[12] proposes a variety of priority concepts as follows:

- Absolute ordering where policies and rules are ordered and the highest order has priority.
- Deny by default where deny effects of rules have priority over permit cases.
- Obsolescence where more recent rules have priority over older rules.
- Specificity where a specific rule overrides a more general rule.
- Authority where a policy defined by a higher authority has priority.
- Privileges where the policy with the strongest rights has priority over weaker rights

[5]proposes a conflict resolution mechanism based on effect constraints of conflicting segments. First, conflicting segments are defined and then a reordering of conflicting segments is compulsory. Basically, no changes are made to the user specified combining algorithms.

[13] proposes a method using the concept of various degrees of majority for a given effect.

[14] proposes an ordering of attributes to determine which attributes are more important in making decisions using weights.

Among the above approaches to resolve rule conflicts at runtime, two stand out: one for the RBAC model in [5] and one for the ABAC model in[11], with the latter one being derived from[14].

3. PRIORITY-BASED CONFLICT RESOLUTION

3.1 Difficulty Determining Exceptions

One of the potential solutions we have explored involves no changes to the policy specification language. In this approach, we defined rules that express exceptions. In the presence of such rules, there are several ways to try to resolve the conflicts:

- Consider all rules as exceptions.
- Consider the fact that some rules have broader coverage than others.

In the above example, the first rule *NurseResourceRule* is restricted only to the document *surgery report,* while the second rule *NurseHomeCareRestrictionRule* has no restriction involving *surgery report,* and actually applies to any value of attribute *resource-id*. It is restricted only to location *home care*. But the reverse is also true so that there is no way to determine which rule has a broader coverage than the other. Indeed, both have broader coverage, but not on the same attribute. Consequently, the only way to determine which rule should win is to apply some priority mechanism.

3.2 Description of the Algorithm

The algorithm has been implemented using the logic programming language Prolog, used widely in artificial intelligence applications due to its suitability for implementing logical reasoning. In logic programming, there are two distinct elements. The first is the knowledge base, which is a database of facts and clauses (which express rules) about the system to be reasoned about. The second element is the logic and reasoning used to solve problems using the knowledge base as an input.

3.2.1 Structure of the Knowledge Base

In our case, the knowledge base is composed of three groups of facts:

- The description of priorities for each XACML attribute and their corresponding values;
- The description of relative priorities used to describe which attributes are more important than others;
- The actual logic of XACML rules in a given access control application.

We note here that this relative priorities approach is closer to human reasoning.

First, for the definition of priorities of attributes we consider attribute/value pairs and specify if a value of an attribute is sensitive or normal. A convincing example is the case of the *Emergency* attribute. When its Boolean value is equal to *true* we consider it as sensitive, while when it is *false* we consider it as normal. The absence of such a definition can also be used to express the fact that a given value is of no consequence in the decision process.

The above example would require the following definition of priorities to operate correctly. For the *subject-id* attribute, we consider the nurse and psychiatrist values to be *sensitive*, in this case, for two different reasons. The nurse is allowed to read medical records of a patient only under certain conditions. Thus, we consider his or her role as sensitive. On the other hand, the psychiatrist deals with highly sensitive information that only s/he can read. Also note that the sensitivity level *normal* for a surgeon is the result of the fact that a surgeon performs his/her skills only in an operating room, thus any other sensitive location is by definition irrelevant, in sharp contrast with the nurses that perform in various locations.

```
priority('subject-id', 'nurse', sensitive).
priority('subject-id', 'anesthesist',  normal).
priority('subject-id', 'generalist', normal).
priority('subject-id', 'psychiatrist', sensitive).
priority('subject-id', 'surgeon', normal).
```

The *action-id* attribute has two sensitive values, *read* and *email*. It is interesting to note that the print value is dependent on the read value. You can print only if you can read.

```
priority('action-id', 'read', sensitive).
priority('action-id', 'write', normal).
priority('action-id', 'email', sensitive).
priority('action-id', 'print', normal).
```

The *resource-id* attribute has one particular sensitive value, the *psychiatric report.*

```
priority('resource-id',  'general information', normal).
priority('resource-id', 'surgery report', normal).
priority('resource-id', 'assessment', normal).
priority('resource-id', 'psychiatric report', sensitive).
```

The *Location* attribute has sensitive values for any location outside of a hospital, which here is *ambulance* and *home care.*

```
priority('Location', 'ambulance', sensitive).
priority('Location', 'operating room', normal).
priority('Location', 'home care', sensitive).
priority('Location', 'recovery room', normal).
```

Finally, the *Emergency* attribute has a sensitive value *true*.

```
priority('Emergency', 'true', sensitive).
priority('Emergency', 'false', normal).
```

Second, we define which attributes are more important than others for the case when several sensitive values for different attributes are present in a request. Here we consider that the *Emergency* attribute prevails over any other attribute. In our case, this implies that a nurse should be able to read any medical record in any location. We specify this case using the special keyword *$all*.

```
is_more_important_than('Emergency', '$all').
```

Next, we consider the attribute *Location* as more important than *subject-id*, act*ion-id* and *resource-id*. This is, of course, in order to be able to handle appropriately the situation where the location is *home care*.

```
is_more_important_than('Location',  'subject-id').
is_more_important_than('Location',  'action-id').
is_more_important_than('Location', 'resource-id').
```

In the above definition of facts, note that we have carefully omitted a definition that would have said that *Location* is more important than *Emergency*. The absence of a specification for this case is naturally handled by Prolog since in Prolog, this would generate a fail and force the system to look at the next available solution.

Finally we consider the attribute re*source-id* more important than *subject-id* in order to handle the *psychiatric report* case.

```
is_more_important_than('resource-id',   'subject-id').
```

It is important to note that the definitions for the *is_more_important_than* fact is only partial. This is in sharp contrast with the approach of defining complete matrices used in[7]. This is inspired by the not-applicable effect of the XACML PDP system, used when a request is not matched in the policy set. However, in a Prolog implementation, if complete information were required, the use of backtracking would have the effect of forcing a search for another solution.

3.2.2 Reasoning Mechanism

When presenting a request to a policy decision point (PDP) using the specified policy set, a number of solutions are returned, possibly providing conflicting effects. A solution is defined as a path through the policy set tree and is considered in its entirety regardless of whether or not an element of logic belongs to a particular XACML structuring entity (policy set, policy or rule). Note that our reasoning mechanism is used only in case of conflicts, not redundancies, mostly because our PDP is implemented in Prolog where internal indexing is taking place, reducing considerably the search time for solutions.

In general, we work on the tree representation of a policy set as described in [5]. The tree is composed of sections of subtrees expressing the *anyOf* and *allOf* constructs in a XACML 3.0 target description, as was described in [12]. Here, the XACML *anyOf* constructs are translated into Prolog disjunctions using the “|” operator and the XACML *allOf* into Prolog conjunctions using the Prolog “,” operator. We have used the single predicate approach described in [15] both for performance and also to enable easy location of solution traces. However, there are some small but important modifications to this early model that enable collecting the names of attributes and the exact trace through the logic. Our example is represented as follows in Prolog:

```
01 policy_set(PS, P, R, T, [
02           ['subject-id', A_subject_ID],
03           ['action-id', A_action_ID],
04           ['resource-id',  A_resource_ID],
05
06           ['Location', A_Location],
07           ['Emergency', A_Emergency]],
08                                                          EF):
09
10   PS = medex,
11  (A_subject_ID = ['subject-id', nurse],
12                         TPS = tps1),
13  (
14     P = p1,
15      (A_action_ID = ['action-id', read],
16                          TP= tp1),
17          (
18     (
19        R = r1,
20             (A_resource_ID = ['resource-id',
21                      surgery_report],
22                  T = [TPS, TP, tr1]),
23                         EF = permit
24             )
25         |
26     (
27       R = r2,
28             (A_Location = ['Location',
29                         home_care],
30                  T = [TPS, TP, tr2]),
31                       EF = deny
32             )
33         |
34     (
35       R = r3,
36             (A_Emergency = ['Emergency',
37                               true],
38                T = [TPS, TP, tr3]),
39                       EF = permit
40              )
41     )
42          ).
```

Solution paths are traces composed of tree traversals through policy sets, policies and rules. They are obtained by posing a query using the Prolog built-in *findall* predicate applied to the entire tree:

```
:- findall(policy_set(PS, P, R, T, RQ,
     EF), policy_set(PS, P, R, T, RQ, EF),
                                      LS).
```

where *RQ* represents a request, which is composed of values for each attribute of the policy set, *LS* is a variable that will return a list of solution paths, and *EF* is the effect of each solution path. While the request contains values for all attributes used in the entire policy set, the returned solutions contain only subsets of attributes that are effectively used in the path. For example, the request:

```
R1 :=
  'subject-id' = 'nurse',
  'action-id' = 'read',
  'resource-id' = 'surgery_report',
  'Location' = 'home_care',
  'Emergency' = 'true'
```

will return three solution paths. The first one will traverse policy set *medex*, policy *p1* and rule *r1* with an effect of *permit*. This is achieved by the matching statements of lines 11, 12, 15, 16, 20, 21 of the Prolog representation of the XACML policy set above. The subset of attributes for this solution path that contain sensitive values is { *subject-id, action-id* }. Note that the attributes *Location* and *Emergency* are absent from this list because there are no corresponding matching expressions for them in this solution trace. The *surgery report* value for *resource-id* has been declared as non-sensitive in the priority facts above and thus does not appear in the subset of attributes. The two other solution traces are left as an exercise to the reader.

In this example, we have three results with two different effects (both *deny* and *permit*). We have tried different mechanisms to resolve such conflicts. First, we experimented with numerical values to express priorities in two different ways.

The first approach consisted of calculating the sum of each attribute's priority based on the values for the attributes that are present in a solution trace through the policy set tree. This solution was rapidly eliminated because it produces misleading results when the solution traces do not contain exactly the same number of attributes. In particular, this case arises when expressions for a given attribute are not provided, which is the way to express that any value of the attribute is applicable.

The second approach consisted of picking the solution trace for which an attribute that is present in the policy logic showed the highest priority value. This provided good results for our above example but could not be generalized.

Consequently, we began exploring an algorithm that does not rely on quantitative numerical values used to describe priorities, but instead uses qualitative relative values as expressed by the Prolog *priority* facts above.

The new algorithm has two steps:

- The first step consists of collecting the attributes for which there is a sensitive value in a particular solution path. Then, the attribute that is the most important among all of those in the subset of attributes in the solution path is chosen using the *is_more_important_than* facts. The algorithm works under the assumption that when using an attribute to specify some exception, policy writers do use sensitive values in the XACML target logic. It is clear that this approach would not work in the case of non-sensitive values. However, access control logic is mostly composed of cases where sensitive values of attributes apply. After this step, we end up with a single attribute that is the most important for a given solution trace and serves as the representative of a solution trace.
- In the second step, using the most important attributes for each solution path determined in the first step, we apply the *is_more_important_than* fact again, but this time to compare the relative priority among solution paths, which determines the most important solution path. The resulting solution path then provides the final effect desired (*permit* or *deny*).

In our case, the request *R1* produces three solutions against our policy set.

The first solution consists of the path that traverses rule *NurseResourceRule*, which is the first one returned when evaluating the request against the policy set by the Prolog inference engine:

```
Solution 1: policy_set(medex,p1,r1,[tps1,tp1,tr1],
 [[subject-id,[subject-id,nurse]],
  [action-id,[action-id,read]],
  [resource-id,[resource-id,
                   surgery_report]],
  [Location,_G1880],
  [Emergency,_G1889]],
  permit)
```

In the above first solution, we notice that Prolog open variable values *_G1880* and *_G1889* are produced when the matching logic does not contain attributes *Location* and *Emergency*. The solution trace actually considers all the attributes in the attribute list of the Prolog representation of the policy set. The solution trace traverses policy set *medex*, policy *p1* and rule *r1*.

A solution trace can be obtained using the following Prolog term to be used in a query to the knowledge base:

```
go_pdp_med_1:-
  nl, write('request 1'),
  retractall(solution(_,_)),
  assertz(solution(_,0)),
  request(request_1, RQ),

  findall(policy_set(medex, P, R, T, RQ,
                            EF),
  policy_set(medex, P, R, T, RQ, EF), LS),
  extract_solution_traces(LS, [ ], LST),
  nl, write('solution traces:'),
  select_solution(LST, SSOL),
  nl, write('overall effect: '),
  write(SSOL).
```

The second solution trace returned by the above query is as follows:

```
Solution 2: policy_set(medex,p1,r2,[tps1,tp1,tr2] ,
 [[subject-id,[subject-id,nurse]],
  [action-id,[action-id,read]],
  [resource-id,_G1961],
  [Location,[Location,home_care]],
  [Emergency,_G1979]],
  deny)
```

In the above second solution, we notice that Prolog open variable values *_G1961* and *_G1979* are produced when the matching logic is absent for attributes *resource-id* and *Emergency*. The solution trace traverses policy set *medex*, policy *p1* and rule *r2.*

And finally the third solution trace is as follows:

```
Solution 3: policy_set(medex,p1,r3,[tps1,tp1,tr3],
  [[subject-id,[subject-id,nurse]],
  [action-id,[action-id,read]],
  [resource-id,_G2051],
  [Location,_G2060],
  [Emergency,[Emergency,true]]],
  permit)
```

In the above third solution, we notice that Prolog open variable values *_G2051* and *_G2060* are produced when the matching logic is absent for attributes *resource-id* and *Location*.

The solution trace traverses policy set *medex*, policy *p1* and rule *r3*.

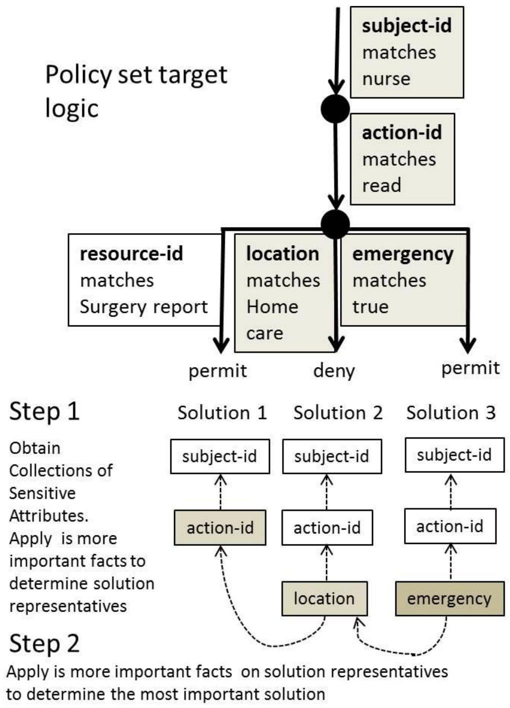
For each of these solutions we collect the attributes for which sensitive values are detected in the request and the corresponding policy set targets. For example, in the case of the third solution trace, we would have the following list.

```
[subject-id, action-id, Emergency]
```

Then we use the *is_more_important_than* fact to determine which attribute is the most important for that solution, and it will be used to represent this solution when comparing solutions to each other. In this case, it is the attribute *Emergency* because of the *is_more_important_than fact* for target attribute *$all*.

When comparing the *Emergency* attribute against other attributes with matching expressions that operate on a sensitive value, we can successfully derive that the *Emergency* attribute is the most important of all. Thus the *Emergency* attribute will represent the third solution when comparing the solutions among themselves. This is summarized in Figure 1, where solid arrows show the path of a given solution for request *R1*, grey boxes show the sensitive values for attributes and dotted arrows show the *is_more_important_than* relations.

By repeating this process for each solution, we determine that *Location* is the most important attribute for the second solution trace and the attribute *action-id* will represent the first solution, mainly because there are no *is_more_important_than* definitions for the attributes that are present in this solution path.



Policy set target logic
subject-id matches nurse
action-id matches read
resource-id matches Surgery report
location matches Home care
emergency matches true
permit
deny
permit
Step 1
Obtain Collections of Sensitive Attributes. Apply is more important facts to determine solution representatives
Solution 1
Solution 2
Solution 3
subject-id
action-id
location
emergency
Step 2
Apply is more important facts on solution representatives to determine the most important solution

Figure 1. Visual representation of algorithm applied to request 1.

Also, the results for the second request *R2*, where *Emergency* has been set to false, will produce only two solutions, with the attribute *Location* as the most important attribute. This attribute value will be used to determine the final effect, which is *deny*.

```
R2 :=
  'subject-id' = 'nurse',
  'action-id' = 'read',
  'resource-id' = 'surgery_report',
  'Location' = 'home_care',
  'Emergency' = 'false'
```

Finally, the same method applied to the request *R3* will result in only one solution produced, in which case, we don't need to try to determine priorities among attributes of this solution path. The resulting effect of this solution is *permit*.

```
R3 :=
  'subject-id' = 'nurse',
  'action-id' = 'read',
  'resource-id' = 'surgery_report',
  'Location' = 'operating room',
  'Emergency' = 'false'
```

Now, when handling request 1, the second step of our method can be applied. We compare the attribute representatives for each solution as given by the first step. Here the results of the first step produced the following most important attribute representatives for each solution path:

Solution 1: *action-id => permit*
Solution 2: *Location => deny*
Solution 3: *Emergency => permit*

Since *Emergency* has been defined as the most important attribute of all, this will make solution 3 win and the final effect will be *permit*. In other words, a nurse can read any document anywhere during an emergency.

3.2.3 Handling Concurrent Priorities

If we add one more rule that deals with psychiatric reports this system may no longer work.

```
rule NursePsychiatryRule -> Deny :=
        resource-id matches 'psychiatric report'
```

Effectively, since we have declared that the attribute *Emergency* is more important than anything else, when attribute value *Emergency* matches *true* and attribute *resource-id* matches value *psychiatric report* in a request that is presented to the PDP, it will allow a nurse to read a psychiatric report, which is what the above additional rule wants to prevent. Thus, in this context we need to improve our methodology. One easy way to handle this case is to enhance the *is_more_important_than* facts by adding a field for the highly critical value.

```
is_more_important_than('resource-id',
            'psychiatric report', '$all').
```

Then, adding a clause to the Prolog logic to handle this case (lines 01 to 06 below) solves the problem. Here these cases would be made available on the top of the list of alternative predicates and if there is a match, a Prolog cut ("!") will prevent it from considering the other cases as follows:

```
01 determine_most_important:-
02        is_more_important_than(A, V, '$all'),
03        significant(A),
04        request_value(A, V),
05        save_most_important(A),
06        !.
07
08 determine_most_important:-
09        is_more_important_than(A, '$all'),
10        significant(A),
11        request_value(A, V),
12        save_most_important(A),
13        !.
14
15 determine_most_important:-
16        significant(A),
17        (
18                most_important(nil)
19                |
20                most_important(MI),
21                is_more_important_than(A, MI)
22
23        ),
24        save_most_important(A),
25        fail.
26
```

```
27 determine_most_important.
```

The above code makes intensive use of the Prolog internal database which in a way mimics the storage of information of humans in their brains and reasoning as a retrieval of this information.

3.3 Another Example in the Military Domain

The example provided in [12] can be enhanced to create the kind of ambiguity found in the previous medical example, showing again the benefit of priorities. Here we add a policy that considers the unit being engaged.

```
  policy agent_a policy :=
     Agent matches a

 rule No_fly_zone_rule –> permit :=
        Zone matches no_fly_zone.

    rule  HostilesPresenceRule -> deny
        HostilesPresence matches true.

    rule UnitRule -> permit:=
          Zone = no_fly_zone,
          Unit matches special forces.
```

In this case, special forces are allowed to enter the no fly zone even when a hostile presence is detected. This is achieved using the following facts:

```
is_more_important(HostilePresence, Zone).
is_more_important('Unit', 'special forces',
                              '$all').
```

4. CONCLUSION

In this paper we have shown how to resolve run-time conflicts using artificial intelligence in the form of automated logical reasoning, with an algorithm that uses priorities based on sensitivity assessments defined for each policy/rule attribute and its associated values. Our approach uses a relative relationship and thus there is no need for numerical weights. This approach is closer to human reasoning, which reacts to overall sensitivity factors rather than scales of values. We also determined that compile time conflict detection algorithms are very useful for testing purposes. They can determine which requests to a PDP produce these conflicts, and thus enable the policy administrator to verify offline that the conflict resolution algorithms are performing as expected.

5. ACKNOWLEDGMENTS

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

6. 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] C.D.P.K. Ramli, H. R. Nielson, F. Nielson, The Logic of XACML, in proceedings of FACS 2011 pp 205-222.

[4] 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.

[5] H. Hu, G.-J.Ahn and K. Kulkarni, Anomaly Discovery and Resolution in Web Access Control policies in SACMAT'11 proceedings.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[12] B.Stepien, A. Felty, S.Matwin, Challenges of Composing XACML Policies in 2014 Ninth International Conference on Availability, Reliability and Security.

[13] N. Li, Q. Wang, W.Qardaji, E.bertino, P. Rao, Access Control Policy Compiling: Theory Meets Practice in SACMAT 09 proceedings pages 135-144.

[14] A.J. Rashidi, A. Rezakhani, a new method to ranking Attributes in Attribute Based Access Control using decision fusion in Natural Computing Applications Forum 2016, Springer Verlag.

[15] B. Stepien and A. Felty, Using Expert Systems to Statically Detect "Dynamic" Conflicts in XACML in ARES 2016 proceedings, pp 127-136.