A non-technical user-oriented display notation for XACML conditions

Bernard Stepien, Amy Felty, Stan Matwin
University of Ottawa – S.I.T.E., (bernard | afelty | stan)@site.uottawa.ca

Abstract. Access control rules are currently administered by highly qualified personnel. Thus, the technical barrier that specialized access control languages represent naturally prevents the prime decision maker to effectively control such access. The usability is even worse in the case of access control applications targeting an average consumer, where customers who are casual users are expected to administer their own rules, e.g. in case of financial services. XACML is one of the most powerful access control languages because it allows the definition of complex conditions. In order to make XACML usable in such applications, there is a need for fully non-technical rule editors. We propose a notation for XACML rules containing conditions that is a combination of the usual tree properties of logical expressions but with an accessible natural language like format. Our early experience indicates that such rules can be grasped by non-technical users wishing to develop and control rules for accessing their own resources.

Keywords: access control, notation, XACML

1. Motivation

The XACML (eXtensible Access Control Markup Language) [1] access control language (ACL) is naturally precise since it is based on an XML schema that represents the grammar of a given application. But this very property puts it out of reach of non-technical, and especially casual users that in some cases could even be computer illiterate. The main obstacles for a casual user in using XACML are:

- Long XML tags
- Long and complex domain references
- Prefix notation for operations
- List oriented notation for conjunction and disjunction operators

While it is practically impossible for a casual user to start coding his rules with a text editor—this would require full knowledge of XML and XACML grammars—a first step toward solving this problem could be to use an XML editor that frees the user from this knowledge up to a certain point, as the supplied XML Schema enables the selection of appropriate tags in a context-oriented way.

A number of such tools exist in different syntaxes and formats, each trying to address a specific technical problem. They can be classified into two broad categories:

- Generic XML editors.
- Specialized application oriented XML editors—where XACML belongs.

While all of these editors claim to be targeting non-technical users, their documentation indicates that they require at least a basic knowledge of XML. In fact, one of the main problems with the XACML notation is that it requires some programming skill regardless of the tools used.

Currently, there is a very limited set of XACML tools. The UMU editor [2] was the first attempt to have a general XACML editor. Others have further refined the specialization. This is the case of the visual Language hierarchy solution [4] that exclusively targets RBAC [5] applications. .

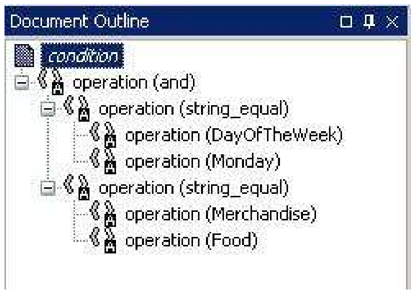
Our new approach has been guided mostly by the study of existing editors. There are a number of open source and commercial XML and XACML editors available that follow a number of basic principles.

2. Current principles in XML editors

XML editors are most often based on a tree display principle of an XML document. The tree display is most natural, mostly because an XML document is hierarchical by definition.

XAMLPad [3] is the most commonly used open source editor. It offers three different views of an XML document: the XML plain text, the grid and the table view. In addition to these views, a document outline represented as a tree is also available.

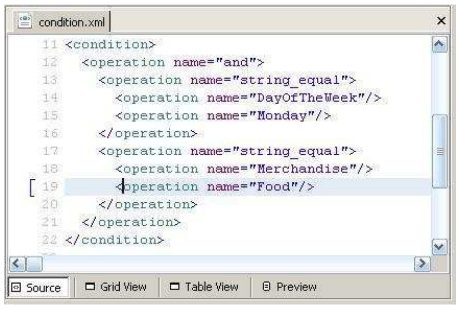
Let us imagine that we need to create a rule that authorizes a purchase action if a specific condition holds. Let us use a simple condition that says that a purchase is permitted if the day is Sunday and the merchandise purchased is food. This condition would have a document outline as shown in fig 1. Such an outline mainly shows the name of the node and the value.



```
condition
  operation (and)
    operation (string_equal)
      operation (DayOfTheWeek)
      operation (Monday)
    operation (string_equal)
      operation (Merchandise)
      operation (Food)
```

Fig. 1. Document outline of a simple condition

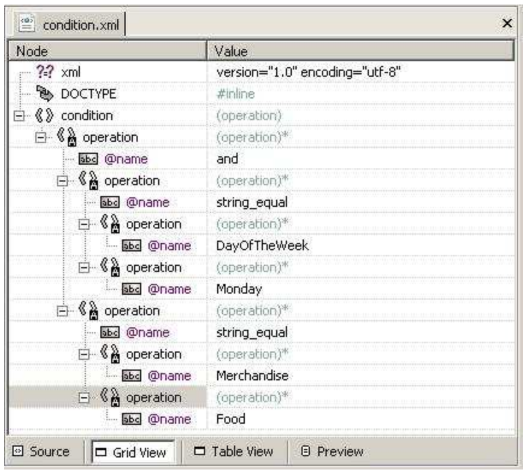
The corresponding XML source view is shown in fig 2. It can be interactively edited by positioning the cursor in a region, which triggers the appearance of a choice of actions. Examples of actions include entering the value of a new attribute if it is not already present, or appending a new tag. The editor will automatically insert the attribute or tag selected from a drop down menu. Thus here, the interesting principle is that although the user sees only plain text, the editor provides features that waive the need for in-depth knowledge of the data model (DTD or Schema) and thus reduce the risk of errors such as spelling mistakes of attribute names or forgetting an attribute altogether. The source view however allows the direct typing of tags and attributes and a parser is triggered at every save attempt and highlights errors.



```
<condition>
  <operation name="and">
    <operation name="string_equal">
      <operation name="DayOfTheWeek"/>
      <operation name="Monday"/>
    </operation>
    <operation name="string_equal">
      <operation name="Merchandise"/>
      <operation name="Food"/>
    </operation>
  </operation>
</condition>
```

Fig. 2. XML source of the condition shown in Fig. 1

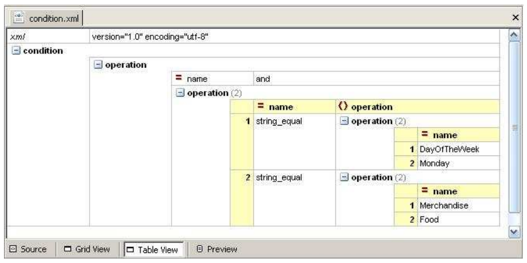
The corresponding Grid view is shown in fig 3. It corresponds to a horizontal tree where each node indicates the tag names and their corresponding attributes and also the related DTD for the current element. Again, features similar to those available in the source view are also available. Here however, the presentation of the data model could actually assist the user in planning his next move.



Node	Value
xml	version="1.0" encoding="utf-8"
DOCTYPE	#inline
condition	(operation)
operation	(operation)*
@name	and
operation	(operation)*
@name	string_equal
operation	(operation)*
@name	DayOfTheWeek
operation	(operation)*
@name	Monday
operation	(operation)*
@name	string_equal
operation	(operation)*
@name	Merchandise
operation	(operation)*
@name	Food

Fig. 3. Grid view of the condition whose XML is in Fig. 2

The table view shown in fig. 4 is just another way to represent the tree of the grid view, attempting to further reduce the programming skills required of the user. Note also the attempt to reduce the amount of information in the tree by factoring out the name of the tag when there are multiple occurrences of a tag, as in this example for the arguments of an operation.



```
xml        version="1.0" encoding="utf-8"
condition
           operation
                      name          and
                      operation (2)
                                     name           operation
                                  1  string_equal   operation (2)
                                                       name
                                                    1  DayOfTheWeek
                                                    2  Monday
                                  2  string_equal   operation (2)
                                                       name
                                                    1  Merchandise
                                                    2  Food
```

Fig. 4. Table view of the tree shown in grid view in Fig. 3

3. Current principles in XACML editors

In order to understand the implications of writing an XACML specification of the previous simple example, we need to examine the representation of the condition of this example in XACML.

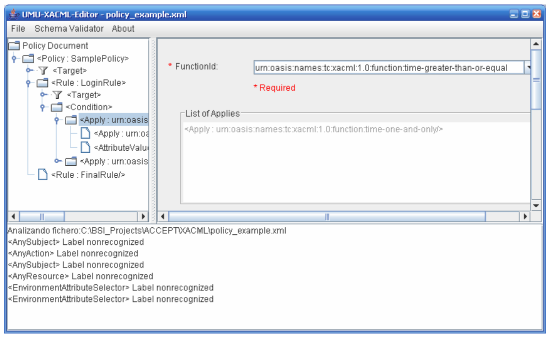
```
  <Condition  FunctionId="urn:oasis:names:tc:xacml:1.0:function:and">
     <Apply    FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-
equal">
        <Apply
FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-one-and-only">
```

```
           <SubjectAttributeDesignator        AttributeId="Merchandise"
DataType="http://www.w3.org/2001/XMLSchema#string" />
        </Apply>
        <AttributeValue
DataType="http://www.w3.org/2001/XMLSchema#string"
                                    >food</AttributeValue>
     </Apply>
     <Apply    FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-
equal">
        <Apply
FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-one-and-only">
            <SubjectAttributeDesignator AttributeId="DayOfTheWeek"
  DataType="http://www.w3.org/2001/XMLSchema#string" />
        </Apply>
        <AttributeValue
DataType="http://www.w3.org/2001/XMLSchema#string"
                                     >Monday</AttributeValue>
     </Apply>
  </Condition>
```

The first XACML editor, developed by University of Murcia [2] is shown in fig 5. It is based on two complementary views, one for the document outline and one for the attribute values and some local overviews.

The first problem this editor has addressed is omitting the need to type the domain names. Functions are merely selected from lists along with their domains.

Conditions are constructed by clicking on a node of the tree and selecting an operator from a list. Again, while this editor reduces XACML coding effort considerably, it requires a strong expertise both in XML and XACML.



urn:oasis:names:tc:xacml:1.0:function:time-greater-than-or-equal

<Apply : urn:oasis:names:tc:xacml:1.0:function:time-one-and-only/>

```
Analizando fichero:C:\BSI_Projects\ACCEPT\XACML\policy_example.xml
<AnySubject> Label nonrecognized
<AnyAction> Label nonrecognized
<AnySubject> Label nonrecognized
<AnyResource> Label nonrecognized
<EnvironmentAttributeSelector> Label nonrecognized
<EnvironmentAttributeSelector> Label nonrecognized
```

Fig. 5. UMU editor representation of the condition

This editor is not easily usable by a non-technical user, mostly because this kind of user will not know the XACML condition grammar. Also the resulting tree is again reorganizing the terms of a condition in a way that is not mapped directly on to the corresponding natural language statement of the condition. For example the *and* operator is located at the top of the tree hierarchy instead of being in the middle.

More recently the XACML Studio editor tried to alleviate some of the difficulties of use mentioned about the UMU editor [7] but with most of the same functionalities.

One principle is important in both general purpose and specialized editors presented so far. All editors provide the capability to hide or expand portions of the tree in their various views except the source view. This feature allows the user to focus on a portion of the tree and thus avoids the cluttering that naturally results from the presentation of large amounts of information. This feature has, however, an important side effect. It prevents the user from having an overview of the entire condition he is trying to assemble. This makes the reasoning about the logic of the expression being built very difficult and could lead to errors.

4. Our proposed notation

Our proposed notation is only a display notation. It is neither a new language nor a replacement for XACML. However, it bears some formal qualities that we have chosen mostly to facilitate its use in interactive editors that allow a non-technical user either to create a new policy or to modify an existing one. Effectively, if we had followed only the consideration to make the policies and rules understandable by a non-technical user we could have merely translated them into plain English but we quickly realized that plain English would have been a challenge to manipulate in an editor. Thus, we came up with the idea to use trees to represent logical expressions but again after realizing that a casual user may not grasp abstract mathematical concepts we decided to create a hybrid between formalism and plain English. This concept has already been mentioned as a goal for the XACML community by Vullings [8] but no formal paper seems to have been published yet showing evidence of results in that research area..

Furthermore, we came to the conclusion that a full non-technical representation of XACML is not really possible, mostly because XACML is a strongly typed language. Typing is not a concept that the casual user can grasp beyond the basic types, like numeric or alpha-numeric. Effectively, the nuances of data storage considerations that further divide numeric types into various levels of precision such as integer, float, double, etc. can only be knowingly manipulated by technical users. However, the actual display of a XACML condition has no real barrier of this kind, and can be considered user-friendly.

Consequently, we propose a separation of concerns between the data typing definitions that should remain in the hands of knowledgeable IT technicians, and the policy editing including its logical expression construction that can be delegated to the non-technical user.

This approach is appropriate mostly because an access control application is available within a context where there is an infrastructure organized by the provider of the service. This infrastructure naturally includes the definition of variables along with their types and potential allowed values. For example, an eStore will define what products it will sell along with the necessary parameters such as product identifiers or codes, units types to express their quantities ordered, etc.

Data typing is thus relegated to another document that we also decided to structure using XML, where variables used in a given application are defined along with their data types and potential lists of allowed values.

Our notation is based on the following basic principles:

- Stay as close as possible to the user's natural language by avoiding any technical terminology for operators and maintaining the overall structure of a natural language.
- Offer an implicit structuring by organizing the natural language into a tree.
- Organize the tree so as to make it consistent with the natural language statement of the condition by using an infix representation for conjunction and disjunction operators.
- Maintain XACML's natural non-binary nature of conjunction and disjunction operators but eliminate its original list representation.
- Use a different, yet still casual terminology for conjunction and disjunction operators depending on their position in the tree hierarchy.
- Ensure a full graphical overview of the expression being built at all times regardless of its complexity. This implies no capability to collapse portions of the tree.

Thus, our notation is very close to a natural language statement of the condition. It is actually an improvement over it, as it shows the logical structure of the condition. This will prove very important when building complex expressions requiring the concept of operator precedence. A casual user should not have to be concerned with representing operator precedence.

Our previous example augmented with an additional conjunction would be represented in our notation as follows:

```
   DayOfTheWeek is Monday
and
   Merchandise is Food
and
   BalanceOfAccount over 500
```

The simple example above has a very shallow depth. Two additional techniques can be used to express more complex conditions:

- Allowing multiple values for a given variable.
- Allowing sub-constraints on a value for a variable

The first principle is illustrated in the next example where the condition is extended to two different days of the week and to two different kinds of merchandises:

```
   DayOfTheWeek is one of Monday, Friday
```

```
and
   Merchandise is one of Food, Travel
and
   BalanceOfAccount over 500
```

The above example also illustrates that our notation is not relying on a one-to-one mapping to XACML. For example, in our notation we show only one occurance of the variable name DayOfTheWeek. In XACML this would be represented instead by a disjunction operation on two sub-expressions of the kind – DayOftheWeek is Monday or DayOftheWeek is Tuesday, both using the XACML string_equal operator. However, when the user saves this expression, it is fully translated in a XACML syntax and grammar oriented style where the variable is repeated for each sub-expression.

The second principle is illustrated by introducing sub-constraints on values by saying that travel is allowed only on Friday and food purchases only on Monday or Tuesday. Here, the conjunction operator *and* has been represented by the *provided that* terminology that is more natural since it is in the context of a disjunction.

```
Merchandise is one of
  Food
    Provided that DayOfTheWeek is one of Monday, Tuesday
  Travel
    provided that DayOfTheWeek is Friday
and
   BalanceOfAccount over 500
```

The above expression corresponds to the following plain natural language representation:

“It is permitted to purchase food on a Monday or a Tuesday or travel on a Friday provided that the balance of the account is over 500”.

As we can see from this example, the order of the sub-constraint in the pure natural language version is strictly the same as in our notation. The only difference is the graphical structuring of the tree appearance. It helps clarify the rule in its natural language form, where putting various sub-constraints in their appropriate context
requires mental effort from the user.

Another advantage of the tree notation we are proposing is that it avoids the ambiguity of the scope of the disjunction operators. In the natural language representation above it is hard to understand the exact scope of the *or* operator that applies to food or travel because of the presence of the other disjunction about Monday or Tuesday. In our tree like notation this ambiguity disappears entirely. It is a well known fact that this kind of scoping problem is the prime source of ambiguities in interpreting statements in natural language. In fact, with traditional non XACML notation for logical expression, the only way to resolve these ambiguities would be to use parentheses as follows:

```
  ((Merchandise == Food) and ((DayOfTheWeek == Monday) or (DayOfTheWeek
== Tuesday))) or ((Merchandise == Travel) and (DayOfTheWeek == Friday))
  and (BalanceOfAccount > 500)
```

As already mentioned, our notation is not intended to be formal even if it looks formal. One of the main features of this notation is that we do not represent conjunction and disjunction operators with a single equivalent. This is to follow the principles where for example a conjunction is represented either with the word *and* or a paraphrase such as "provided that" that implies the conjunction but is more conceptually precise in the context of a disjunction, thus in a way naturally resolving the ambiguities that a mix of conjunction and disjunction operators would unavoidably yield. We further resolve the ambiguities using indentation.

Another consideration is that we do not intend to cover the entire capabilities provided by the XACML grammar. This is mostly due to the fact that as pointed out already in the XACML standard, logical expressions should remain simple. to be understandable not to mention the fact that complex expressions in XACML are extremely hard to read in the first place. The only problem is that our notation allows to compose complex logical expressions without getting lost and thus may call for a full support of the XACML grammar.

We also support the XACML negation operator not by merely integrating it in the natural language representation as follows:

```
Merchandise is not Food
```

We also support the concept of XACML variables that consist in factoring out a portion of logical expression such for example as creating a variable for week days which would be either a disjunction between equalities for each day or a member of construct also provided by XACML.

5. Our notation in the context of an editor

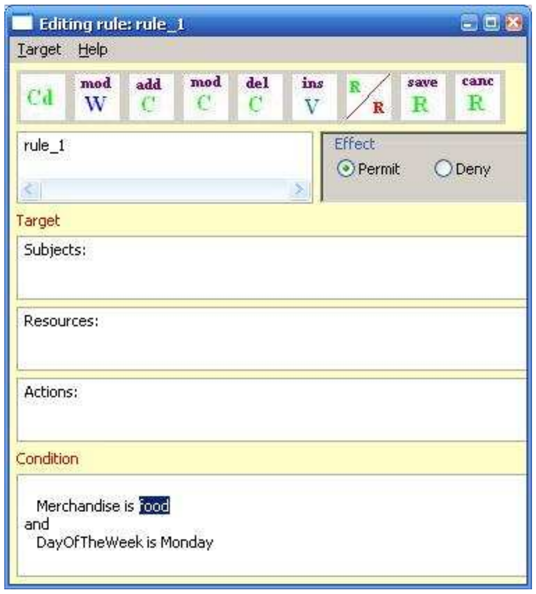
We have developed a XACML editor as a series of interfaces in which our notation is used in all cases where an expression is required such as in target subjects, resources and action specifications, and in the conditions of rules.

Our XACML editor reads a configuration file which specifies the names, data type and potentially allowed values from an XML file as in the following example:

```
<Variable name="DayOfTheWeek" type="String">
   <Values>
     <Value name="Monday"/>
     <Value name="Tuesday"/>
     <Value name="Wednesday"/>
     <Value name="Thursday"/>
     <Value name="Friday"/>
```

```
      <Value name="Saturday"/>
      <Value name="Sunday"/>
    </Values>
 </Variable>
```

The XACML policy interface allows the user to create or modify a policy. The rule interface allows creating or modifying a rule and especially its condition as shown on fig 6.



rule_1

Effect: [✓ Permit] Deny

Merchandise is food
and
DayOfTheWeek is Monday

Fig. 6. Our XACML policy interface

A modification is achieved by first double clicking a word in a condition and then invoking the requested modification by clicking one of the tool bar buttons, which allow operations such as modifying a value, adding, modifying or deleting a constraint or inserting an additional value. The insertion or modification of a value is achieved via a value selection interface show in figure 7. In fig. 6, clicking the value food is sufficient to obtain all the possible values of the Merchandise variable. The internal representation, which is a tree that is mapped exactly onto the XACML structure, enables the editor to determine which variable a clicked value corresponds to, and thus provide the appropriate value selection interface. A value node is a leaf of an operation node such as *string_equal*. Walking the tree to the parent of the value and then descending from the parent to the leaf that contains the variable makes this process possible. Once the appropriate selection is done in the value selection interface of fig 7, the resulting tree is redrawn along with all the internal references to type definitions.

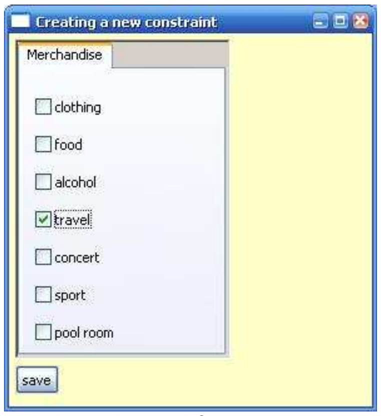


Fig. 7. Policy value modification using our editor

6. **Our notation beyond XACML**

While our efforts have concentrated on XACML, we have applied the same principles to other access control languages such as Cisco IOS [6]. This has been made particularly easy by the architecture of our editor, where the internal representation of a policy is independent of the XACML language itself. Our internal representation, however, provides the structure of XACML, but without reference to its tag names or types. Thus the XACML language structure is used as a common denominator for handling all other Access Control languages. Our editor has a policy connector component that can handle an unlimited number of languages provided that parsers for these languages are built. Another benefit of this language independent internal representation is that the editor can be used to translate one language into another language. This requires adding the appropriate code generators that all operate on the language agnostic internal representation.

The following example shows a Cisco IOS rule and its corresponding representation in our notation. The variable names are defined by the translator as they are not part of the original syntax of Cisco IOS.

```
access-list 101 deny tcp host 148.22.33.44 host 192.168.0.0 eq 3500
```

is displayed in our notation as follows:

```
   protocol is tcp
and
   srcIP is 148.22.33.44
and
   dstIP is 192.168.0.0
and
   dstPort is 3500
```

7. Conclusion

XACML editors can be an effective and highly desirable tool, assisting non-technical users in specifying complex XACML rules, e.g. for access and resource control. We have proposed here a simple yet powerful, implemented notation that allows users to perform this task by providing him a representation that is very close to natural language. Also, due to its high compactness, it provides a rare overview quality that is an important factor in reducing errors, thus helping to ensure the commercial success of the application.

Our early experience with several non-technical users confirms that our goal of empowering non-technical users with a tool giving them control of their resources can be met with the proposed notation. We need to perform a more thorough evaluation of how well this goal is realized, and collect more experience in representing a variety of resource access specifications using the approach and the editor described in this paper.

Our editor based on our notation is not intended to be a replacement for any XACML editor when the user is fully technically qualified. However, while our initial goal was to address the needs of casual, non-technical users, an additional benefit of this approach is that even technical users can easily specify very complex conditions, something that was stated as important to avoid in the past in the XACML user community. This has an important consequence of avoiding the splitting of complex rules into numerous rules with narrower targets, which produces large rule bases that become rapidly unmanageable.

References

[1] XACML, OASIS standard,
http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml

[2] University of Murcia XACML Policy Editor, http://xacml.dif.um.es/

[3] XMLPad, open source,
http://www.wmhelp.com/xmlpad3.htm

[4] M. Giordano, G. Polese, G. Scanniello, and G. Tortora, Visual Modelling of Role-Based Security Policies in Distributed Multimedia Applications, Proceedings of the IEEE Sixth International Symposium on Multimedia Software Engineering, IEEE Press, 2004.

[5] XACML Profile for Role Based Access Control (RBAC), 2004, http://docs.oasis-open.org/xacml/cd-xacml-rbac-profile-01.pdf

[6] J. Boney, Cisco IOS in a nutshell, O'Reilly, 1st edition, 2001.

[7] XACML studio, http://xacml-studio.sourceforge.net/

[8] E. Vullings, Implementing Authorized Access,
http://www.apsr.edu.au/Open_Repositories_2006/erik_vullings.ppt#256,1,Implementing Authorised Access.