XML Access Control: from XACML to Annotated Schemas

Ryma Abassi
Higher School of Communication, Sup'Com
University of the 7th November at Carthage
Tunis, Tunisia
ryma.abassi@supcom.rnu.tn

Florent Jacquemard
INRIA Saclay IdF & LSV, UMR
Cachan, France
florent.jacquemard@lsv.ens-cachan.fr

Michael Rusinowitch
INRIA Nancy-Grand Est & LORIA UMR
Nancy, France
Michael.Rusinowitch@loria.fr

Sihem Guemara El Fatmi
Higher School of Communication, Sup'Com
University of the 7th November at Carthage
Tunis, Tunisia
sihem.guemara@supcom.rnu.tn

Abstract— **XML became the *de facto* standard for the data representation and exchange on the internet. Regarding XML documents access control policy definition, OASIS ratified the XACML standard. It is a declarative language allowing the specification of authorizations as rules. Furthermore, it is common to formally represent XML documents as labeled trees and to handle secure requests through 'user views'. A user view is the part of the document accessible to a given user according to the existing policy. Moreover, control access polices can be depicted as annotated rules where annotations define for each document node whether it is accessible. Hence, an annotated schema is a formal representation of 'user views'.**

Our main contribution in this paper is then three folds. First, we compare XACML policies and annotated schemas. Second, we identify a significant fragment of XACML since this latter is very expressive and consequently complex. Third, we define adequate translation algorithms from XACML policies to annotated schemas.

Keywords; XML; XACML; XPath; tree automata; annotation

I. INTRODUCTION

XML [1] is a set of rules for encoding documents electronically. XML is the *de facto* standard for the data web exchange. It is being increasingly adopted in communication networks.

Access control for XML documents should ideally provide expressiveness, modularity, interoperability and efficiency. The expressiveness assures that a wide range of security policy specification may be written. Modularity concerns policies composition while interoperability concerns the ability of policies to interact. Finally, efficiency assures the ability to determine whether an access to an element is granted or denied by the security policy. XACML [12], the current OASIS standard, is being increasingly adopted for specifying access control policies. It is a declarative language for the specification of authorizations as rules.

Although XACML is recognized as a precise and complete policy description method, the structure of an XACML policy is complex and users need to understand XACML well and writes down the verbose policy all by hand, which make XACML difficult to master and use. Hence, our key idea in this paper is to create a 'user view' from an XACML policy. This view is intended to support and to facilitate validation task since it is based on tree formalism. Moreover, since XACML is very expressive, we have chosen to base our work on a fragment of its syntax that can be expanded later.

Several works considered access control for XML documents [2, 3, 4, 5]. In [2], Damiani and al. proposed to express access control policies using XPath queries. However, the proposed enforcement of the proposition can be complex and computationally expensive. In [3], Murata and al. introduce a static analysis for XML access control. Given an access control policy, a query expression and an optional schema, their analysis determines whether the query is safe i.e. whether it returns only accessible elements. However, this proposition seems to incur costly runtime security checks for queries. Fan and al. introduced in [4] the '*security views*' concept. Their proposition annotates the security restrictions on the schema structure and rewrites XPath queries issued. However, their annotations are limited to hiding node values. Mohan and al. [5] remedied this limit by considering constraints based on structural relationships between elements.

The rest of this paper is organized as follows. In Section 2, we introduce some basic notions such as XML, DTD, XPath and XACML. Section 3 presents the tree automata concept. In Section 4, we introduce the concept of annotation, present the adopted annotation and their beneficiary formalisms as well as the main algorithms used in order to translate an XACML policy to an annotated schema. Finally, Section 5 concludes this paper.

978-1-4244-8840-7/10/$26.00 ©2010 IEEE

II. PRELIMINARIES

In this section, we introduce the basics of XML, schema language, XACML and tree automata.

A. XML

XML (eXtensible Markup Language) [1] describes a class of data objects called XML documents and partially describes the behavior of computer programs which process them.

```
<network>
    <server>
        <name> print-server </name>
        <type> print-server </type>
    </server>
    <server>
        <name> smtp-server </name>
        <type> mail-server </type>
    </server>
    <PC>
        <name> PC1</name>
        <department> electronic </department>
    </PC>
    <PC>
        <name> PC2</name>
        <department> computer science </department>
    </PC>
</network>
```

Figure 1: XML document

An XML document is made of *elements*, *attributes* and *nodes*. Figure 1 depicts an example of an XML document representing a network topology. This document describes a university network containing two servers' type as well as two PC's type.

Moreover, the XML specification defines an XML document [1] as a text which is *well-formed*, i.e. it satisfies a list of syntax rules provided in the specification and *valid*, i.e. it contains a reference to a given schema.

XML document are defined as trees. Formally, a Σ-tree *t* can be represented by the following structure

$$t=(N_t, root_t, child_t, next_t, \lambda_t),$$

where N_t is the node set, $root_t \in N_t$ is the tree root, $child_t \subseteq N_t \times N_t$ is the relation parent-child, $next_t \subseteq N_t \times N_t$ is the relation giving the rest of child of the same parent and λ_t is the function assigning to each node his label.

B. Schema

A schema is a description of permissible XML documents. A schema language is a computer language for writing schemas. The most known are namely DTD [6], W3C XML Schema [7] and RELAX NG [8] from OASIS.

A DTD describes the structure of a class of documents via element and attribute-list declarations. Element declarations name the allowable set of elements within the document, and specify whether and how declared elements and runs of character data may be contained within each element. Attribute-list declarations name the allowable set of attributes for each declared element, including the type of each attribute value, if not an explicit set of valid value(s). Moreover, DTD describes cardinalities through a BNF grammar: '*' marks 0 or more occurrence, '+' marks one or more occurrence and '?' marks 0 or 1 occurrence.

Figure 2 depicts the DTD corresponding to XML document of Figure 1. According to this DTD, a network may be composed by zero or more occurrence of server and PC. Each server is represented by a name and a type while each PC is represented by a name and a department.

network	→	*server*, PC**
server	→	*name, type*
PC	→	*name, department*
type	→	*data*
name	→	*data*
department	→	*data*

Figure 2: An example of a DTD

Formally, a DTD is a pair D (δ^c sd) where δ is a function that maps an alphabet Σ to regular expressions over Σ and $sd \in \Sigma$. A Σ-tree *t* is valid with respect to the DTD *D* if for each node *n* having *k* child $n_1, \ldots, n_k$, we have $\lambda_t(n_1) \ldots \lambda_t(n_k) \in L(D(\lambda_t(n))$ where $L(D))$ is the set of valid trees with respect to the DTD *D* and λ_t is a the function assigning to each node his label.

The main restriction for DTDs is that they can only define local languages. In [9, 10], authors used an extension of DTD (EDTD for Extended DTD) in order to overcome this drawback and thus by allowing the definition of types instead of languages [10].

Formally, an EDTD over Σ [10] is defined as:

$$D = (\Sigma \; {}^{c}\delta \; {}^{c} sd),$$

where $(\delta^c sd)$ is a DTD over $\Sigma \cup \Sigma^N$ and $\Sigma^N := \{\sigma^n \mid \sigma \in \Sigma, n \in N\}$ a typed alphabet.

We modify the previous network example by replacing the PC tag with two new tags, one for PC located in departments for student use and the other for PC belonging to network administrators. The new requirement cannot be expressed by a DTD anymore because this would require distinguishing two rules (*PC → name department*) and (*PC → name user*). To resolve this problem one would also have to introduce two new tags for distinguishing *PC1* and *PC2*.

To capture the modification that *PC2* only belongs to a given department and that *PC1* does not belong to a department, we can define an EDTD using two types for the PC as depicted by Figure 3.

network	→	*server*, PC1*,PC2**
server	→	*name, type*
PC1	→	*name, department*
PC2	→	*name, user*

978-1-4244-8840-7/10/$26.00 ©2010 IEEE

type	→	*data*
name	→	*data*
user	→	*data*
department	→	*data*

Figure 3: An example of an EDTD

Given a valid and well formed XML document, we often want to locate some elements by specifying conditions on them. This can be done by the use of XPath.

C. XPATH

XPath (XML Path Language) [11] is a language for addressing parts of an XML document. It provides a restricted variation of regular path expressions. More precisely, a path is constructed from steps, which are:

- an axis, which specifies the tree relationship between the nodes.
- a node test, which specifies the name of the selected nodes.
- zero or more predicates, which refine the set of nodes selected by the location step.

The most useful path expressions are listed below:

- *nodename* selects all child nodes of the named node
- a child axis ('/') selects from the root node
- a descendent-or-self axis ('//') selects nodes in the document from the current node that match the selection no matter where they are
- a self axis ('.') selects the current node
- a parent axis ('..') selects the parent of the current node
- an attribute axis ('@') selects attributes.

For example,
//chapter: selects all chapters whatever they are in the document.
../chapter: selects the element chapter which is the parent of the current node.

Concerning the security of XML documents and more precisely their access control management, adequate languages must be used. One of the most used languages for securing XML documents access control is XACML.

D. XACML

XACML (eXtensible Access Control Markup Language) [12] is the OASIS standard language for access policies. Figure 4 depicts an XACML policy structure that may contain several rules. A rule has a *target,* an *effect* and a *condition*. Target is defined through a *subject*, a *resource* and an *action*. A subject is an active entity that can perform an action over a given resource. If the *target* of the rule is true then the rule's *condition* is evaluated where the rule's *effect* can be either permit or deny. Given that multiple rules may yield several decisions for the same request, an XACML policy must specify the precedence between rules in the form of policy combining algorithms e.g. "deny overrides", "permit overrides", "first applicable", "only-one applicable", etc.

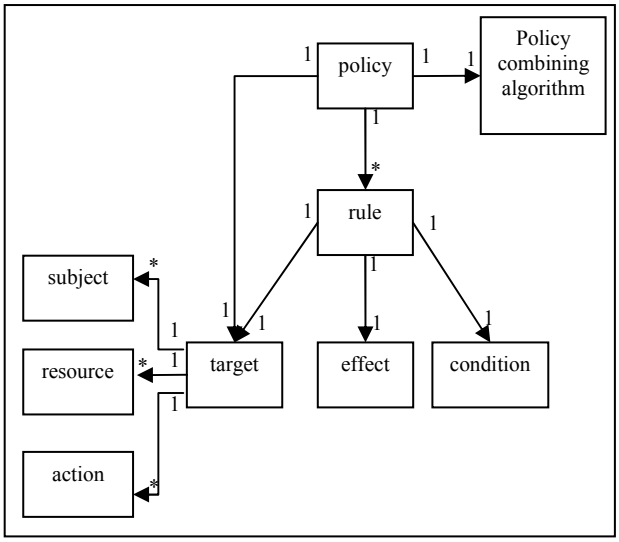

Figure 4: The XACML policy structure

An example of an XACML rule is depicted in the following. This rule stipulates that the request of a professor to read the student database will be granted.

```
<rule ruleid=1 effect = permit>
        <target>
          <subjects>
              <subject> professor</subject>
        </subjects>
        <resources>
              <resource> student-database </resource>
        </resources>
        <actions>
              <action> read </action>
        </actions>
  </target>
</rule>
```

Let's note that rule's *subject* can be specified using user ID, a role or a group name but is not limited to these. Rule's *resource* is defined trough an XPath expression where the rule's *action* value can be either read, update, create or delete, but in this paper we deal only with the read and write action.

III. TREE AUTOMATA

Let's recall that each XML document can be described as a tree where nodes correspond to elements while attributes and leafs correspond to the content of these elements [1]. Figure 6 depicts the tree corresponding to the XML document type introduced in Sub-Section II.B.

Hence, the tree TA of Figure 6 can be formalized as follows:

N_{t1}= {n_0 ,n_1 ,n_2 ,n_3 ,n_4 ,n_5 ,n_6 ,n_7 ,n_8 ,n_9 ,n_{10} ,n_{11} ,n_{12} ,n_{13} ,n_{14} ,n_{15} ,n_{16} ,n_{17} ,n_{18} ,n_{19} ,n_{20}},

$root_t = n_0$,

$child_t$ = {(n_0, n_1),(n_0, n_2),(n_0, n_3),(n_0, n_4), (n_1, n_5), (n_1, n_6), (n_2, n_7), (n_2, n_8), (n_3, n_9), (n_3, n_{10}), (n_4, n_{11}), (n_4, n_{12}), (n_5, n_{13}), (n_6, n_{14}), (n_7, n_{15}), (n_8, n_{16}), (n_9, n_{17}), (n_{10}, n_{18}), (n_{11}, n_{19}), (n_{12}, n_{20})}

$next_t$ = {(n_1, n_2), (n_2, n_3), (n_3, n_4), (n_5, n_6), (n_7, n_8), (n_9, n_{10}), (n_{11}, n_{12}), (n_{13}, n_{14}), (n_{15}, n_{16}), (n_{17}, n_{18}), (n_{19}, n_{20})}

λ_t = {(n_0, university), (n_1, server), (n_2, server),(n_3, PC), (n_4, PC), (n_5 , name),(n_6 , type), (n_7 , name),(n_8 , type),(n_9 , name,), (n_{10} , type) ,(n_{11} , name) ,(n_{12} , type) ,(n_{13} , server-imp), (n_{14} , server-imp),(n_{15}, server-imp),(n_{16} , server-imp),(n_{17} , PC1),(n_{18} , electronic), (n_{19} , PC2) ,(n_{20}, computer-science)}.

In order to define regular tree languages having a clear semantics and allowing validation tasks, automata are needed. Automata are used in order to define tree languages: they examine a given tree by assigning states to nodes in the tree. It accepts the tree if it terminates at one of the final state [13].

Formally, a TA is defined as follows [10]:

$$B = (Q, \Sigma, \delta, F),$$

where Q is the states set; Σ is an alphabet ; F is accepting states and δ is a function of the domain $Q \times \Sigma$ such as $\delta(q,a)$ is a regular expression over Q for each $a \in \Sigma$ and $q \in Q$

A *run* of B over t is a labeling $N_t : Dom(t) \rightarrow Q$ such that for each ν with n child,

$$\lambda(v1)\ldots\lambda(vn) \in \delta(\lambda(v), \lambda_t(v)).$$

A *run* is accepting iff the root is labeled with a final state, $\lambda(\varepsilon) \in F$.

The bottom up computation of a TA can be performed according to Figure 5: if the automaton reads the symbol f and that below there is a finite number of states $q_1, \ldots, q_n$ then the automaton can move to the state q. For leafs, each label corresponds directly to a given state.

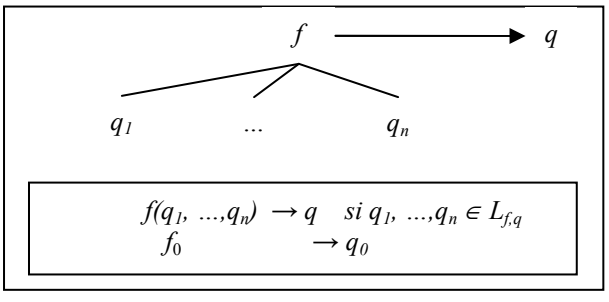

Figure 5: construction of a TA

IV. From XACML to annotated Schema

In this section, we introduce our proposition concerning the creation of 'user view' and that by translating XACML policies into adequate annotated schema.

A. Used Annotations

In this paper, we consider the five annotations defined in [14] and specifying the action to perform when constructing a given view.

$$Act = \{del, del_{rec}, id, id_{rec}\} \cup \{r_a | a \in \Sigma\}$$

where

- *del* is the annotation used in order to delete a node while preserving his children.
- del_{rec} is the annotation used in order to delete a whole arborescence, i.e. a node and all his children (the child annotations are then ignored).
- *id* is the annotation used in order to preserve a node.
- id_{rec} is the annotation used in order to preserve a whole arborescence.
- r_a is the annotation used in order to rename a given node to $a \in \Sigma$.

An annotation allows also setting the action taken at a given node by satisfaction of XPath expression. Formally the set of annotations *Ann* is defined as follows:

$$\Phi ::= \sigma \mid \text{if } f \text{ then } \Phi \text{ else } \Phi$$

where $\sigma \in Act$, f is an XPath expression and σ are actions. The semantics $[\![\Phi]\!]_t^n$ of an annotation Φ evaluated in a node n of a tree t is an action defined as follows:

$$[\![\sigma]\!]_t^n = \sigma \text{ where } \sigma \in Act,$$

$$[\![\text{if } f \text{ then } \Phi_1 \text{ else } \Phi_2]\!]_t^n = \begin{cases} [\![\Phi_1]\!]_t^n \text{ if } (n,n) \in [\![f]\!]_t, \\ [\![\Phi_2]\!]_t^n \text{ else.} \end{cases}$$

B. Proposition

Although XACML is considered as one of the best languages used for describing web documents access control policies, it has a complicated structure and is not simple to manipulate for non-expert users. Hence, we propose to hide this complexity by the use of annotated schema.

For purpose of clarity, we choose an XACML significant fragment and not all XACML syntax for the translation. Let's note that this fragment can be enriched later. Thus, we assume that an XACML policy is a collection of rules each consisting of a *resource*, a *subject*, an *action* and an *effect*. Moreover, we

consider only XPath expression using two axes: '/'and '//' and *read* and *rename* actions.

Figure 7 depicts the 4-steps proposition. Having an XML document and its EDTD, we convert it into a TA (1). In parallel, and for each subject, we consider all the rules managing his potential requests. Since a rule is defined through an annotation *a* as well as potential positions *p*, we construct an annotated EDTD corresponding to the XPath expression given the rule application position (2). The third step is then a special product of the two obtained TA (3). Let's note that this step is repeated for each rule and whenever, the initial automaton is replaced by the last obtained one (4).

Algorithm 1 presents these steps. First, a TA Tad is created from the EDTD. Then, a TA *Tar* is constructed for each rule of the XACML policy: an annotation *a* is affected according to the rule effect and the position to annotate is defined through the rule resource. Third, for each obtained TA Tar_i, Tar_i and *Tad* are superposed which allows to detect a conflict if a given node is already annotated.

Algorithm 1: security view creation

Create-view (EDTD, PS)

```
Tad:= create-edtd (EDTD)
For each subject s
do
    For each rule where s is involved
    do
        res:= resource
        eff:= effect
        act:=action
        comb:= combination
        ann :=  annotate (act, eff)
        Tar := create-xpath-TA (res, ann)
        Tas :=  product (Tad, Tar)
        Tad:=  Tas
```

In the remaining part of this section, we give more details for each procedure composing this algorithm.

C. From an EDTD to a Tree Automata

Let's recall that we formalized a TA by $B = (Q, \Sigma, \delta, F)$ where Q is the states set, Σ is an alphabet, $\delta: Q \times \Sigma \rightarrow 2^Q$ such that $\delta(q,a)$ is a regular expression over Q for each $a \in \Sigma$ and $q \in Q$ and F is the set of final states.

Hence, the creation of a TA from the EDTD of the XML document can be achieved as detailed in Algorithm 2. Given an EDTD defined by $(\Sigma, \delta, root)$, this algorithm returns a TA Ta_d defined by $(Q_{TA}, \Sigma_{TA}, \delta_{TA}, F_{TA})$ as follows:

- The final state of the TA corresponds to the tree root.
- TA states correspond to the nodes labels.
- Transitions correspond to the regular expressions of each label or directly to the label itself if the node is a leaf.

Algorithm 2: create-edtd (EDTD)

Create-edtd (EDTD)

```
F_TA := root
For each label a ∈ Σ
do
    Q_TA := Q_TA + a
    If a is a leaf
    then
        δ_TA (q_a, a):= q_a
    else
        δ_TA (q_a, a) :=  D(a)
```

Let's consider the following example. Given the EDTD *E* describing a scientific paper structure and constituted by the root *body*, the alphabet *body, paper, title, author* and *journal*, as well as two regular expressions. The corresponding TA is hence, the following:

EDTD *E*: $(\Sigma, \delta, root)$
δ(body) = paper*
δ(paper) = title, author*, journal
Σ = {body, paper, title, author, journal}
root = body

TA: $(Q_{TA}, \Sigma_{TA}, \delta)$
F_{TA}= {body}
Q_{TA}= {body, paper, title, author, journal}
Σ_{TA} = {body, papers, title, author, journal}
δ_{TA}(body, body)= paper*
δ_{TA} (paper, paper)= title, author*, journal
δ_{TA} (title, title)= title
δ_{TA} (author, author)= author
δ_{TA} (journal, journal)= journal

D. Annotation affectation

Annotations are affected according to the policy rules by the use of procedure *annotate (action, effect)* depicted in Table 1. Hence, a read action is associated to the *id* annotation if it is granted or it is associated to the *del* annotation. A rename action however, is associated to the *ren* annotation when granted.

Action	Effect	Annotation
read	permit	id
read	deny	del
rename	permit	ren

Table 1: annotation affectation

978-1-4244-8840-7/10/$26.00 ©2010 IEEE

E. From an XPATH expression to a Tree Automata

Given an XPath expression *p* and an annotation *ann*, we create an annotated TA defined as $(Q_{Tr}, S_{tr}, d_{Tr}, F_{Tr})$ where Q_{Tr} is the states set; S_{tr} is an alphabet; d_{Tr} is a function of the domain $Q \times \Sigma$ and F_{Tr} is accepting states and .

Moreover, we consider that an XPath expression can be seen as a function over alphabet $H: \quad N \cup \{//, /\}$ where N represents labels. Hence, a TA $A = (Q, \Sigma, \delta, F)$ is created from an XPath expression *p* by matching Q with prefixes of *p*; F with *p* ; Σ with H and $\delta \subseteq Q \times \Sigma \times ann \times Q$. This is depicted by Algorithm 3. Each sub-expression is added to the TA states set. Only the position corresponding to the evaluation of the XPath expression is annotated while other positions are affected to an empty annotation. Let's note that a nice side-effect is associated to this procedure. In fact, if a position is already annotated whereas we try to annotate it again, then a conflict is detected.

Algorithm 3: from XPATH expression to TA

Create-xpath-TA (p, ann)

```
do
    F_Tr := p
    For each sub- expression p_i
        Q_Tr := Q_Tr + p_i
        if i = p_pos
            then
                if node already annotated
                then ann_i := resolve-conflict (comb)
                else ann_i := ann
        elseif i <> p_pos then ann_i := 0
    δ_Tr(q_i, ann_i, h_i) := q_i+1
```

F. Conflicts resolution according to the combination algorithm

The conflict resolution procedure is depicted by Algorithm 4. This algorithm gives the priority to a rule according to the initially adopted combination algorithm of the XACML policy.

Let's recall that each XACML policy has a combination algorithm such that *deny overrides* or *permit overrides* used in order to give precedence to the positive rule (respectively negative rule). Let's recall also that we proposed three annotations types: *del*, *id* and *ren*. Hence, when the *deny overrides* strategy is adopted, the *del* annotation overrides while the *id* annotation overrides if the *permit overrides* strategy is used.

Algorithm 4: conflict resolution

resolve-conflict (alg-comb)

```
Switch alg-comb
    do
        case « deny-overrides »: ann := del;
        case « permit-overrides »: ann := id;
return (ann)
```

G. Automata product

The last step of this process is the superposition of two obtained TA: given *Tar* $(Q_{Tar}, \Sigma_{Tar}, \delta_{Tar}, F_{Tar})$ associated to the XML document EDTD and *Tad* $(Q_{Tad}, \Sigma_{Tad}, \delta_{Tad}, F_{Tad})$ associated to the security rules of a given subject, *Tas* $(Q_{Tas}, \Sigma_{Tas}, d_{Tas}, F_{Tas})$ is constructed as a product of two earlier TA.

Algorithm 5: *Tad* and *Tar product*

Product (Tad, Tar)

$F_{tas} := F_{tad} \times F_{tar}$

$Q_{tas} := Q_{tar} \times Q_{tad}$

$\delta_{tas} := \{(q^1_{tad}, q^1_{tar}), \ldots, (q^n_{tad}, q^n_{tar}) \rightarrow^a (q_{tad}, q_{tar}) \mid$
$q^1_{tad}, \ldots, q^n_{tad} \rightarrow^a q_{tad} \in \delta_{tad}$
$q^1_{tar}, \ldots, q^n_{tar} \rightarrow^a q_{tar} \in \delta_{tar} \}$

For each $s \in \Sigma_{tad}$ and $s' \in \Sigma_{tar}$

do

- **If** $s = s'$ **or** $s' = 0$
  - $s \times s' := s$
  - $\Sigma_{tas} := s$
- **If** $s = s'^{ann}$
  - $s \times s'^{ann} := s^{ann}$
  - $\Sigma_{tas} := s^{ann}$
- **else** $s \times s'^{ann} := \emptyset$
- **If** ann' = ann
  - $\Sigma_{tas} :=$ $s^{ann'} \times s'^{ann} = s^{ann}$; s^{ann}
- **else** $s^{ann'} \times s'^{ann} = \emptyset$

return $(F_{tas}, Q_{tas}, \delta_{tas}, \Sigma_{tas})$

More precisely, this superposition is made through a special product superposing nodes according to their annotations. Hence, when a label *s* is superposed with the same label or if the label doesn't exist in the second TA, then *s* is preserved. However, the superposition of two identical labels having different annotations is unfeasible. Furthermore, the superposition of states (respectively final sates) is simply a Cartesian product or the two TA states (respectively final sates). This is depicted by Algorithm 5.

V. CONCLUSION

XML is emerging as the new standard for document representing and exchange on the internet. That's why it is important to define XML document access control policies. One of the most popular specification languages is XACML. This latter has a rich and very complete syntax. Paradoxically, this completeness can constitute a weakness: it may be hard to achieve some primordial tasks such as validation or test due to this syntax complexity.

In this paper, we proposed to associate XACML policies with TA formalism which is simpler to use and to manipulate

and constitutes a well established formalism for validation, test or proving task. Hence, given an XACML policy and an XML document, we associate a 'user view' to each subject hiding all the denied parts of the document (for the subject) while preserving all granted parts according to the policy. This is achieved through the use of annotations reflecting security rules effects as well as the construction of annotated TA. So, we proposed adequate algorithms associating each XACML policy with an annotated TA basing on an EDTD (of the XML document) and access control rules.

As a future work, we plan first to implement and test the proposed algorithms and in a second time to extend this work to handle a more complete XACML and XPath fragments.

REFERENCES

[1] W3C. "Extensible Markup Language". W3C Recommandation, Novembre 2008.

[2] E. Damiani, S.D.C di Vimercati, S. Paraboschi, and P.Samarati. Securing XML documents. In Extending Databas Technology, 2000.

[3] M. Murata, A. Tozawa, M. Kudo, and S. Hada. XML access controm using static analysis. In ACM conference on Computer and communications security, 2003.

[4] W. Fan, C.Y Chan, and M. Garofalakis. Secure XML quering with security views. In Proceedings of SIGMOD 2004.

[5] S. Mohan, A. Sengupta, Y. Wu, and J. Klinginsmith. Access Control for XML-A Dynamic Query Rewrinting Approach. In Proceeding of the 31 st VLDB conference, 2005.

[6] T. Bray, J. Paoli, and C.M. Sperberg-McQueen, "Extensible Markup Language (XML) 1.0 W3C Recommandation. http://www.w3.org/TR/REC-xml, February 1998

[7] W3C. "XML Schema". W3C Recommandation, Mai 2001.

[8] J.Clark et M.Murata. "RELAX NG Specification". OASIS Commitee Specification, December 2001

[9] Y. Papakonstantinou and V. Vianu. "DTD inference for views of XML data". In 19th ACM SIGMOD-SIGACT-SIGART symposium on Principles of Database Systems, pages 35–46. ACMPress, 2000.

[10] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison et M. Tommasi, Tree Automata Techniques and Applications, 2007

[11] W3C. "XML Path Language". W3C Recommandation, Janvier 2007

[12] OASIS. eXtensible Access Control Markup Language (XACML). http://www.oasis-open.org/committees/xacml/. 2006

[13] Frank Neven, "Automata theory for XML researchers", ACM SIGMOD Record, v.31 n.3, September 2002_.

[14] Y. Andre, I. Boneva, A.-C. Caron, B. Groz, Y. Roos, S. Staworko, S. Tison, « Secure Querying of XML Documents with Annotated Schema », working draft, INRIA Lille-Nord

978-1-4244-8840-7/10/$26.00 ©2010 IEEE

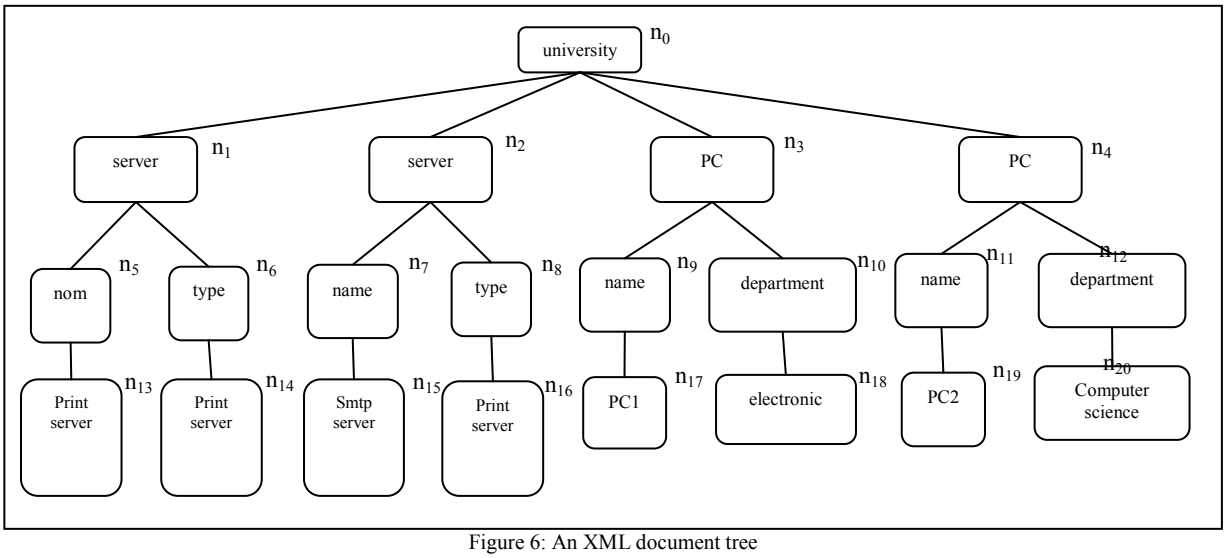

Figure 6: An XML document tree

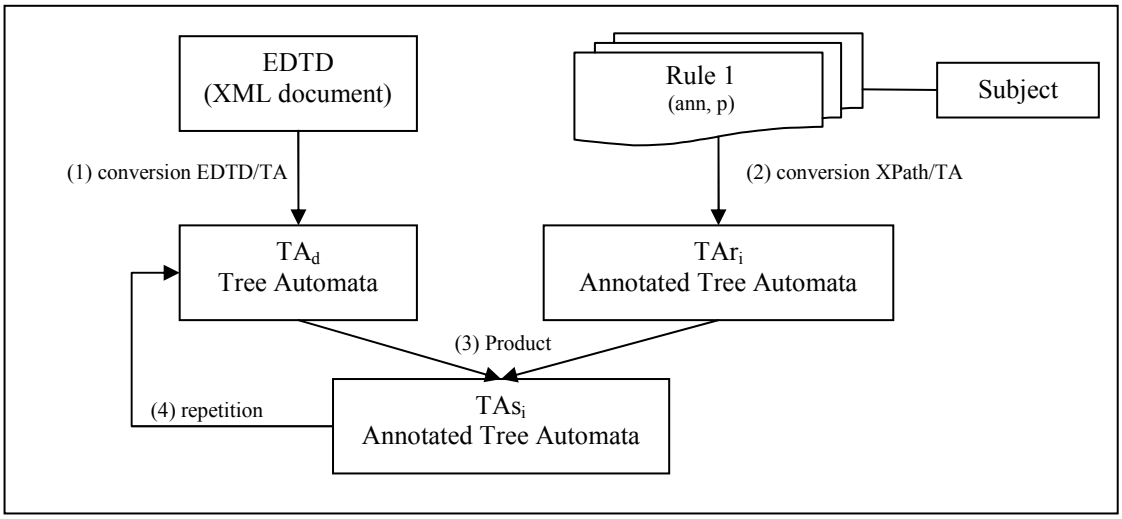

Figure 7: Security view creation process

978-1-4244-8840-7/10/$26.00 ©2010 IEEE