Specifying Access Control Policies for XML Documents with XPath

Irini Fundulaki
Network Data and Services Department
Bell Labs Research
USA
fundulaki@research.bell-labs.com

Maarten Marx[*]
Language and Inference Technology Group
University of Amsterdam
The Netherlands
marx@science.uva.nl

ABSTRACT

Access control for XML documents is a non-trivial topic, as can be witnessed from the number of approaches presented in the literature. Trying to compare these, we discovered the need for a simple, clear and unambiguous language to state the declarative semantics of an access control policy. All current approaches state the semantics in natural language, which has none of the above properties. This makes it hard to assess whether the proposed algorithms are correct (i.e., really implement the described semantics). It is also hard to assess the proposed policy on its merits, and to compare it to others (for file systems for instance).

This paper shows how XPath can be used to specify the semantics of an access control policy for XML documents. Using XPath has great advantages: it is standard technology, widely used and it has clear and easy syntax and semantics. We use the developed framework to give a formal specification of the five most prominent approaches of access control for XML documents from the literature.

Categories and Subject Descriptors

H.1 [**Information Systems Models and Principles**]: Miscalleneous

General Terms

Languages, Security

Keywords

XML, XML access control, XPath

1. INTRODUCTION

Security has always been a key issue for information systems in general. The UNIX file system [24] has a built-in access control mechanism with users and groups, and more recent file systems (AFS [3]) are using similar techniques. Relational database systems also have built-in security in the form of views or table privileges granted to users. Ditto for directory servers [25].

With the increasing number of applications that either use XML as their data model, or export relational data as XML data, it becomes critical to investigate the problem of access control for XML (either physical/native XML or views).

A number of standards such as XACML [18] (OASIS standard) and XACL [27] have already emerged to address the problem of access control for XML documents.

That XML access control is not a trivial subject is clear from the large number of different approaches suggested in the literature [9, 14, 17, 23] and recently [16].

Access control for XML is different from already existing approaches in relational databases, or file systems for a number of reasons:

- the *semi-structured* nature of XML documents [2]: unlike file hierarchies or relational tables where the structure is known ahead of time, XML documents do not necessarily have a schema[1];
- the *dependence of a node to its ancestors*: unlike relational tables that usually exist as stand-alone entities, an XML node lives with respect to its ancestors, and its children are dependent on the node itself. For instance, a meaningful scenario in the XML but not in the relational context, is the requirement that when a node is granted access, then access is also granted to its descendants;
- the *hierarchical nature of XML* (also shared by file systems): XML access control policies all have the UNIX policy as a special case, but they go much further. For one thing, in the XML context it is useful to specify access control rules that apply to a node if some condition is true (e.g. an attribute of the node has a specific value). As mentioned previously, the big difference with file systems is the absence of a schema in XML. In the UNIX policy the exact hierarchy is known and the policy is described for every node. With XML documents one might only have a vague idea of how they look. Still it can be meaningful to grant or deny access to certain parts of the document.

[*]Research supported by NWO grant 612.000.106.

[1]By schema we mean here an XML Schema [26] or an XML DTD that specify the structure of the XML document.

Our Contribution

The existing approaches deal with a number of different dimensions of the problem and offer different solutions.

Our aim was to compare these different approaches among each other and to the existing technologies for file systems. This turned out virtually impossible because none of the state of the art approaches gives a clear formal semantics of their access control policy.

Indeed, the *intended* semantics is given in natural language which is then implemented by a given algorithm. Often the natural language description was ambiguous or not precise enough for a comparison, so we had to study the given algorithms in order to understand what was meant.

We discovered that often the algorithms did something different than promised in the natural language description. Our original research goal seemed hopeless. From our comparison attempts it became clear that the topic is just too difficult to be done in natural language. So we discovered the need for a way of formally specifying the semantics of access control policies for XML documents.

Having a formal specification language for XML access control can serve a number of important goals. Then different approaches become comparable and the most prominent one could evolve as a standard. Then the community can work on implementing the standard and the implementations can be checked on their correctness and the different algorithms can be compared. So at this stage of the development of XML access control it seems absolutely vital to have such a formal specification language.

The main contributions of this paper are the following:

1. we describe how XPath 1.0 [12] can be used as a formal specification language for XML access control policies and
2. we survey and formalize the existing approaches in the literature using XPath.

Motivating Example

We end this introduction with a small example drawn from the domain of telecommunications where more and more applications require secure access to *user profile information*.

Fig. 1 shows the XML DTD that describes user profile information and Fig. 2 illustrates an XML document, a valid instance of the DTD. It is inspired from efforts of standard bodies in the telecommunications domain (3GPP GUP [1]) and Liberty Alliance [21]. A user's profile (element Profile) is associated with zero or one address book and calendar entry (elements AddressBook, Calendar resp.). The user's address book contains zero or more contact entries (element Contact) where each entry is associated with a type (e.g. attribute type), with a first and last name (elements FN, LN resp.) and a phone number (element Phone). Element Calendar is associated with zero or more events (element Event) that has a description, a time/date, and finally a location (elements Desc, Time/Date and Location resp.).

The rest of this paper is organized as follows: In Section 2 we give our formal specification language for access control for XML documents. In Section 3 we formalize the most important approaches in the literature using our specification language. Finally we give our conclusions in Section 4.

```
<!ELEMENT Profile (AddressBook?, Calendar?)>
<!ELEMENT AddressBook (Contact*)>
<!ELEMENT Contact (FN, LN, Phone)>
<!ATTLIST Contact type #CDATA>
<!ELEMENT FN|LN|Phone #PCDATA>
<!ELEMENT Calendar (Event*)>
<!ELEMENT Event (Desc, Time/Date, Location)>
<!ELEMENT Desc|Time/Date|Location #PCDATA>
```

Figure 1: XML DTD for User Profile Information

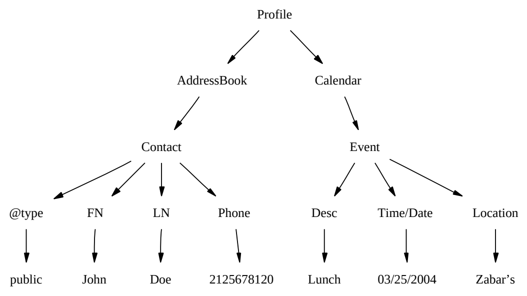

Figure 2: User Profile Document

2. SPECIFICATION LANGUAGE FOR XML ACCESS CONTROL POLICIES

We first discuss the basic notions of *access control policy* and *access control rule*. A policy is a set of rules. We undertake here the more-or-less agreed definition of an access control rule which is a tuple (*requester*, *resource*, *action*, *effect*, *propagation*) where:

- *requester* refers to the user or a set of users concerned by the authorization;
- *resource* refers to the data that the requester is (or not) authorized to access;
- *action* refers to the action (read, write, modify, delete etc.) that the requestor is (or not) allowed to perform on the resource;
- *effect* specifies whether the rule grants ('+' sign) or denies ('-' sign) access to the resource and finally
- *propagation* defines the scope of the rule.

In this paper we assume that the *requester* and *action* parameters are fixed and concentrate on the *resource*, *effect* and *propagation* components. We assume that an XML document is represented by a node labeled tree, as in the standard XPath Data Model [12]. We will refer to the access control rules that grant access to a node as *positive* and those that deny access as *negative*.

2.1 Granularity and Language for Access Control

In most of the proposed approaches, the smallest unit of protection is the XML node [9, 14, 17, 23, 18, 27] and the XPath 1.0 [12] language is used to specify the XML nodes concerned by an access control rule.

XPath 1.0 has been designed as a navigation language that returns a subset of the nodes of a document and used by a large number of XML-related technologies (XQuery [10], XSLT [15] etc.).

The different approaches use restricted versions of the XPath fragment whose syntax is given in Table 1. This fragment is used in this paper to formalize the XML access control problem.

locpath	::=	axis '::' ntst '['expr']' | '/' locpath
		| locpath '/' locpath
expr	::=	locpath | not expr | expr and expr
		| expr or expr | locpath *op* v,

Table 1: XPath Fragment

In Table 1, locpath is the start production, axis denotes the XPath axis relations[2] and ntst is a node test that can be a node label, $\star$ (that matches all labels), or function text() that tests whether a node is a text node. *op* is one of the XPath comparison operators ($<$, $>$, $\leq$, $\geq$, $\neq$, $=$) and v is a value. A formal definition of the XPath semantics is given in [19, 4].

2.2 Semantics of Access Control

As mentioned earlier, we use the XPath [12] language to specify formally the semantics of an access control policy (acp for short).

Given an absolute XPath expression q and a document D, $[\![q]\!]_D$ denotes the set of nodes obtained by evaluating q at the root node of D. We call this set of nodes the *answer set* of q on D.

The semantics of an acp should describe what is $[\![q]\!]_D$ *given the* acp. We denote the latter by $[\![q]\!]_D^{\mathsf{acp}}$. An acp restricts access to a document, so $[\![q]\!]_D^{\mathsf{acp}}$ is always a subset of $[\![q]\!]_D$.

XPath itself contains a handy mechanism to obtain subsets of an answer set. This is done by *filter expressions*, the XPath expressions specified between square brackets. This works as follows: for $n \in [\![q]\!]_D$, $n \in [\![q[\mathsf{expr}]]\!]_D$ if and only if expression expr evaluates to true when applied to n. In the latter case, we say that node n is *accessible*.

Now we have a precise way of giving meaning to an access control policy. Given such a policy P, we must describe the XPath expression expr (obtained from P) by which the answer set of a query q must be filtered to obtain *only* the accessible nodes.

Before we can do that we need to have a closer look at the access control policies themselves. Given an access control policy, there are two questions that need to be answered:

- *"when is an XML node accessible?"* and
- *"what happens to the node if there exists no access control rule that neither grants nor denies access to it?"*. In other words, what are the *default semantics* of the access control policy;

[2]XPath has 13 axis relations: self, parent, child, descendant, descendant-or-self, ancestor, ancestor-or-self, following-sibling, preceding-sibling, following, preceding, attribute and namespace. The last one will not be used in this paper.

The latter is easily answered: Most of the approaches [9, 17, 23, 27] consider that if there exists no access control rule that neither grants nor denies access to a node, then the node is considered as non-accessible. This seems most reasonable and we also adopt it.

The first question is harder. Although it seems more or less straightforward to determine whether a node is accessible or not, this is not always the case. For example, a node may be accessible if its parent is accessible (even if there does not exist a rule which either directly or indirectly grants or denies access to it).

Due to the hierarchical nature of XML, the notion of *scope* of an access control rule is introduced in most of the current approaches. The scope can be:

1. the *node* only [9];
2. the *node* and its *attributes* [14];
3. the *node* and its *text node children* [23] and
4. the *node*, its *attributes*, *all its descendants* and *their attributes* [9, 14, 17, 23, 27]. In [9] the scope of a rule can be restricted to the descendants of the node *found at a certain depth* in the tree.

If the scope of a rule is (1), (2) or (3) then the rule is called *local* [9, 14, 23]. If its scope is (4) it is called *recursive* [9, 14, 17, 23, 27].

All the above views are useful depending on the data. (4) is the most natural, but there are examples for which (2) or (3) might be better. Imagine an XML document in which all data is stored using attributes or text nodes. Then to grant permission to a node's attribute or text sub-nodes would just mean to grant access to the node.

Conflict Resolution: Recall that in an access control policy we can have both *negative* and *positive* access control rules. It might be the case that a node is granted access (by a positive rule) and denied access (by a negative rule) at the same time.

In this case, conflicts arise and there are different approaches to perform *conflict resolution:*

1. by using *priorities*: each access control rule is assigned a priority and the rule with the highest priority is considered [14, 17];
2. *deny overwrites* (negative rule takes precedence over positive rule) [14, 23, 27];
3. *grant overwrites* (positive rule takes precedence over negative rule) [27];

Note that in the last case, negative rules become obsolete, so it can be seen as a special case of the second where there are no negative rules. The last two are special cases of the first.

In the following we specify the semantics of an access control policy where *deny overwrites* is the strategy used. Together with our chosen default semantics, *deny overwrites* states that an XML node is accessible if there exist a positive rule that grants access to it, and no negative rule that denies access to it.

An access control policy is defined by four sets of XPath filter expressions $\mathbf{P}_l$, $\mathbf{P}_r$, $\mathbf{N}_l$ and $\mathbf{N}_r$. A filter expression is of the form ntst'['fexpr']' where ntst is a node label, or $\star$

that matches all labels and fexpr is a predicate as defined in Section 2.1.

Expressions in $\mathbf{P}_l/\mathbf{N}_l$ are the *positive/negative local rules* (whose scope is the node and its text or attribute sub-nodes), and $\mathbf{P}_r/\mathbf{N}_r$ are the *positive/negative recursive rules* (whose scope is the node and all its descendants and attributes).

[9, 14, 23] use absolute XPath expressions to express access control policies. As demonstrated in [22], there is an effective transformation from absolute XPath expressions into equivalent filter expressions. Thus our restriction to filter expressions is no limitation (in fact, the rules often become more natural after the transformation).

We list a few typical examples from our user profile scenario, and their formulation in the above given format:

1. grant access to all nodes;
$$\mathbf{P}_r = \{*\}$$
2. only FN nodes are accessible;
$$\mathbf{P}_l = \{\mathsf{FN}\}$$
3. all nodes in the document are accessible except Calendar nodes;
$$\mathbf{P}_r = \{*\} \;\; \mathbf{N}_l = \{\mathsf{Calendar}\}$$
4. grant access to all Calendar nodes and their descendant nodes and deny access to all other nodes;
$$\begin{array}{lcl}\mathbf{P}_r & = & \{\mathsf{Calendar}\}\\ \mathbf{N}_r & = & \{*[\mathsf{not\ ancestor\text{-}or\text{-}self :: Calendar}]\}\end{array}$$
5. grant access to Contact nodes (including their descendant nodes) only if the value of their type attribute is *"public"*;
$$\mathbf{P}_r = \{\mathsf{Contact}[\mathsf{attribute :: type} =' public']\}$$
6. grant access to the AddressBook node, and all its descendant nodes, except if they are below a Contact node whose type attribute has the value *"private"*.
$$\begin{array}{lcl}\mathbf{P}_r & = & \{\mathsf{AddressBook}\}\\ \mathbf{N}_r & = & \{\mathsf{Contact}[\mathsf{attribute :: type} =' private']\}\end{array}$$

We explain below the semantics of access control policies which consider (i) local rules only, (ii) recursive rules only and (iii) both local and recursive rules. As mentioned previously we consider here only the *deny overwrites* as the conflict resolution policy.

2.2.1 Local Rules Only Policy

In this case, a node is accessible, if there exists *at least one* positive rule that grants access to it, and *no negative rule* that denies access to it. To write the XPath filter expression which expresses exactly that, we have to distinguish between (i) element, (ii) attribute and (iii) text nodes. In the case of elements, the XPath filter expression is:

$$[\bigvee_{p\in\mathbf{P}_l} \mathsf{self} :: p \bigwedge_{f\in\mathbf{N}_l} \mathsf{not\ self} :: f]$$

$\bigvee$, $\bigwedge$ stand for disjunction and conjunction, respectively. As far as attribute and element nodes are concerned, we have to examine two cases:

1. the scope of a rule is *only* the node;
2. the scope of a rule is the *node* and also its attribute (or text) sub-nodes.

In the first case, the XPath filter expression that we must write for an attribute (resp. text) node, is exactly the filter given above. In the second case, we need to examine the parent of the element. An attribute/text node is accessible if there is a positive rule that grants access to and no negative rule that denies access to its parent. In this case, the filter expression for attributes (resp. text) nodes is:

$$[\bigvee_{p\in\mathbf{P}_l} \mathsf{parent} :: p \bigwedge_{f\in\mathbf{N}_l} \mathsf{not\ parent} :: f]$$

2.2.2 Recursive Rules Only Policy

Recall that the scope of a recursive rule is the node itself and *all its descendants*. In this context, a node is accessible if:

1. there exists a positive rule that grants access to one of its ancestors, or to the node itself *and*
2. there does not exist a negative rule that denies access to one of its ancestors or to the node itself.

Based on the above observations, the XPath filter expression with which we need to filter the answer set of a query q is:

$$\begin{array}{ll}(1): & [\bigvee_{p\in\mathbf{P}_r} \mathsf{ancestor\text{-}or\text{-}self} :: p\\ (2): & \bigwedge_{f\in\mathbf{N}_r} \mathsf{not\ ancestor\text{-}or\text{-}self} :: f]\end{array}$$

2.2.3 Local and Recursive Rules Policy

We have expressed for the different access control policies the XPath filters that should be applied to the result of a query to obtain only the accessible nodes. In this section we demonstrate how we can use the previous filters in the case where the policy contains *both* recursive and local rules.

Given an access control policy $ACP = (\mathbf{P}_l, \mathbf{P}_r, \mathbf{N}_l, \mathbf{N}_r)$, a node n is accessible if:

1. (there exists at least one positive recursive rule that grants access to it *or*,
2. there exist at least one positive local rule that grants access to it), *and*
3. (there does not exist a negative recursive rule , *and*
4. there does not exist a negative local rule that denies access to it).

The XPath filter expression which expresses exactly this is:

$$\begin{array}{ll}(1) & [(\bigvee_{p\in\mathbf{P}_r} \mathsf{ancestor\text{-}or\text{-}self} :: p \mathsf{\ or}\\ (2) & \bigvee_{p\in\mathbf{P}_l} \mathsf{self} :: p) \mathsf{\ and}\\ (3) & \bigwedge_{f\in\mathbf{N}_r} \mathsf{not\ ancestor\text{-}or\text{-}self} :: f \mathsf{\ and}\\ (4) & \bigwedge_{f\in\mathbf{N}_l} \mathsf{not\ self} :: f]\end{array}$$

One can observe how naturally the semantics of an access control policy which contains both recursive and local rules can be expressed using XPath filters. (1) and (3) above are the two expressions of the filter in Section 2.2.2 and (2) and (4) are the two expressions of the filter in Section 2.2.1.

2.3 Correctness of query evaluation

We have shown how to use XPath filter expressions to describe $[\![q]\!]_D^{\texttt{acp}}$, the answer set of q, given access control policy acp. This is an efficient and simple way for describing the semantics. To implement an access control policy in this way can be very inefficient, and none of the state of the art approaches does it in this way.

Instead they construct the *authorized document* which contains only the nodes which are deemed accessible by the access control policy, and then evaluate the query against this document. In the papers we examined, these algorithms tend to be quite complex. In all of them the access control policy itself is described in natural language which is not always unambiguous. It is fair to say that it is not possible to check whether the algorithm correctly implements the access control policy, for the sole reason that the semantics of the access control policy are not specified formally! Being able to specify the semantics of an access control policy in a simple formal language as we just did, makes it possible to assess the correctness of the algorithm.

3. FORMALIZING CURRENT APPROACHES

In this section we examine the most important approaches in the field of XML access control. As already mentioned, in these approaches, the semantics of an access control policy is expressed in natural language. Our objective in this section is to show how we can use the specification language presented previously to express their semantics.
Added in proof: At the time of writing we did not have access to the recent [16] so it is not included in our survey.

3.1 Static Analysis for XML Documents: Murata et. al.

Authors in [23] consider access control rules for the *read* action only. In their case, a rule is a tuple of the form: (*requester*, *resource*, *effect*) where these fields are as described in Section 2. Their policies contain only *recursive positive* and *negative* rules.

Granularity and Language for Access Control: Authors in [23] consider the XML node as the protection unit for which authorizations are specified. They use only the child, descendant and attribute axes of the XPath fragment specified in Section 2.1. The XPath predicates are restricted to testing value equality between constants and XML attributes. The XPath expressions in the rules are *absolute*.

Conflict Resolution Policy is *deny overwrites*: if a node is granted access by a positive rule and denied access by a negative one, then the latter overwrites the former.

Default Semantics is *deny*: if the node is neither granted nor denied access by a rule, then the node is not accessible.

Semantics: As mentioned above, their policies contain only recursive access control rules. In Section 2.2.2 we gave the XPath filter expression (in the case of a policy that contains only recursive rules) that must be applied to the answer set of a query so that only accessible nodes are obtained.

Moreover, in their policies they enforce the *denial downwards consistency* requirement according to which:

> "whenever a policy denies access to an element then it also denies access to its subordinates elements and attributes. In other words, an element is accessible only if all its ancestors are accessible."

The above statement is itself ambigious: Consider for example the XML document

```
<A><B><C/><D/></B></A>
```

and the following policy: $ACP = (\mathbf{P}_l, \mathbf{P}_r, \mathbf{N}_l, \mathbf{N}_r)$, where $\mathbf{P}_r = \{\texttt{B}\}$ and all other sets are empty. There is no rule which grants or denies access to node A. According to their default semantics, A will be marked as not accessible (the policy denies access to A). But it grants access to B and its descendant nodes. Hence this policy does not enforce the *denial downwards consistency requirement*.

In order to enforce this requirement, there must always exist a rule that grants or denies access to a node. This restriction can be formulated using the following XPath filter expression:

$$[\bigvee_{p\in\mathbf{P}_r} \mathsf{ancestor\text{-}or\text{-}self} :: p \bigvee_{f\in\mathbf{N}_r} \mathsf{ancestor\text{-}or\text{-}self} :: f]$$

If there exist a node for which the application of the above filter expression returns the empty set and not the node itself, then the policy does not satisfy the downwards consistency requirement and can be then discarded.
We can see here that in this case we can specify constraints that the policy requires using XPath filters. Hence, not only can we express the semantics of the policy in XPath but also constraints such as the above. As a consequence, we can verify at compile time whether the access control policy satisfies or not these constraints.

Query Evaluation: The authors in [23] are concerned with checking whether a given query requests only accessible nodes (in other words whether the query is safe). To do that, they translate queries and access control rules into automata and then perform their intersection. In the cases where the safety of a query cannot be statically decided, they actually evaluate the query at run-time.

3.2 Regulating Access for XML Documents: Gabillon et. al.

Authors in [17] consider access control rules for the *read* action only. A rule is a tuple of the form: (*requester*, *resource*, *effect*, *priority*) where the first three fields are as described in Section 2. Priority defines the importance of the rule. They use in their model both positive and negative recursive access control rules.

Granularity and Language for Access Control: As in most approaches, the XML node is the smallest protection unit for which rules are specified. A restricted fragment of XPath is used which considers only the child, descendant and attribute axis to specify the nodes concerned by an access control rule. The XPath predicates consider all the comparison operators between XML attribute values (or results of function calls) and constants. In addition, *parameterized* XPath expressions are allowed. For instance, in our user profile management example, an access control rule might state that a requestor has access to his contact entry in the address book. This condition is formulated as follows:

$$\mathsf{Contact}[\mathsf{child} :: \mathsf{FN} = \$requestor]$$

This can be very naturally represented in our model if we consider variables (e.g., \$*requestor*) as symbols of the language.

Conflict Resolution Policy is based on the use of *priorities*: each rule is assigned a priority (integer). Given two conflicting rules (i.e. one granting access to a node, and another denying access to it), the rule with the highest priority is selected. If there exist still conflicting rules, then the last one (in order of declaration in the access control policy) is chosen. The assignment of priorities to rules is random.

Default Semantics: To specify the default semantics of an access control policy, they use the notions of *closed* and *open policy* [20]. If the node is neither granted nor denied access by some rule, then it is considered accessible in the case of an open policy and inaccessible in the case of a closed policy.

Semantics: Their access control policy considers only recursive rules. Due to the use of priorities in their model we cannot represent the semantics of their access control using our specification language. If no numerical priorities are considered, then the semantics of their access control policy can be specified by the XPath filter expression given in Section 2.2.3.

Query Evaluation: To perform query evaluation they compute the authorized view for the document and then queries are evaluated against it. Their algorithm traverses the document in pre-order and for each visited node it (a) finds the access control rules which apply to it and (b) performs conflict resolution. A node is kept (along with its children) if it is accessible. Non accessible nodes are rejected (and hence their children).
It is evident from the above that the semantics of access control and conflict resolution policy are enforced by the algorithm for computing the authorized document.

3.3 Securing XML Documents: Damiani et. al.

In [13, 14] a rule is a tuple of the form: (*requester*, *resource*, *effect*, *type*) where the first three fields are as described in Section 2, and *type* can be the combination of any of the following:

1. *local* or *recursive*;
2. *DTD-level* defined for a DTD or *document-level* defined for a specific XML document. A rule in the former case grants or denies access to the nodes in *all* the documents, valid instances of the DTD and in the latter case to the nodes of the specific document.

DTD-level rules allow one to define organization level authorizations (e.g. employees of a company can access the profile document of their co-workers). Document-level rules allow one to define more specific authorizations (e.g. high level management cannot access the public entries of my profile document).

In general, a document-level rule takes precedence over a DTD-level one. But there are cases where we do not want that. To be able to represent such cases, they introduce the notion of *weak document-level* rules (which can be overwritten by DTD-level ones), and *hard DTD-level* rules (which cannot be overwritten by document-level ones). By combining these two types with the previous four ones, the total number of rule types is raised to eight.
Similar to the above approaches rules can be positive or negative.

In this paper we consider the access control model presented in [14] which subsumes the one presented in [13].

Granularity and Language for Access Control: As above, authors here consider the XML node as the unit of protection for which rules are defined. The language used to express the XML nodes concerned by a rule, is the XPath fragment given in Section 2.1. In addition they support XPath 1.0 [12] functions such as `last`() and `position`(). The XPath expressions used in their rules are *absolute*.

Conflict Resolution: Their conflict resolution policy is rather complex. It is a combination of the *deny overwrites* policy with a precedence policy defined for the eight different rule types:

1. *local* rules take precedence over *recursive* ones;
2. *document-level* rules take precedence over *DTD-level* ones, and finally
3. *hard DTD-level* rules take precedence over all rules.

Default Semantics: In contrast to the other approaches where the default semantics is *deny*, here a node for which there does not exist a rule that grants or denies access to it, is considered *indeterminate*.

Semantics: In our formal model we have studied the cases where the conflict resolution policy is *deny overwrites*. To capture the semantics of Damiani et. al, we will introduce here the *"local takes precedence over recursive/deny overwrites"* conflict resolution policy in which:

- a local rule takes precedence over a recursive one;
- a negative rule takes precedence over a positive rule *of the same type*;

Given an access control policy $ACP = (\mathbf{P}_l, \mathbf{P}_r, \mathbf{N}_l, \mathbf{N}_r)$, a node n is accessible if:

1. (there exists at least one positive local rule that grant access *and*
2. there does not exist a negative local rule that denies access to the node);

or

3. (there does not exist a positive local rule that grants access *and*
4. there does not exist a negative local rule that denies access *and*
5. there exists a positive recursive rule that grants access *and*
6. there does not exist a negative recursive rule that denies access to the node).

If there exist a negative local rule, because of the *deny overwrites* principle, the node is not accessible.
In `(1)-(2)` we do not need to examine recursive rules. If there exist a positive local rule that grants access and no negative local rule that denies access to the node, then it is

accessible (*local takes precedence over recursive* principle). (3)-(6) state that in the case where the node is not accessible by local rules (3 and 4), then there must exist a positive and no negative recursive rule for it (5 and 6).

According to these observations, the XPath filter expression that we must apply to the result of the evaluation of a query is defined by the disjunction of C1 and C2 below:

$$\begin{array}{lll} \mathsf{C1}: & (1) & [\bigvee_{p\in\mathbf{P}_l} \mathsf{self} :: p \\ & (2) & \bigwedge_{f\in\mathbf{N}_l} \mathsf{not\ self} :: f] \\ & & \\ \mathsf{C2}: & (3) & [\bigvee_{p\in\mathbf{P}_l} \mathsf{not\ self} :: p \\ & (4) & \bigwedge_{f\in\mathbf{N}_l} \mathsf{not\ self} :: f \\ & (5) & \bigvee_{p\in\mathbf{P}_r} \mathsf{ancestor\text{-}or\text{-}self} :: p \\ & (6) & \bigwedge_{f\in\mathbf{N}_r} \mathsf{not\ ancestor\text{-}or\text{-}self} :: f] \end{array}$$

Given the above two XPath filter expressions, we can decide when a node is accessible or not. Recall that in their semantics a node is indeterminate if there exist no rule which grants or denies access to it. The above two equations do not capture this case. To capture that, we must check if there does not exist a positive and a negative access control rule for the node.
This can be again expressed by the following XPath filter expression:

$$\begin{array}{ll} \mathsf{C3}: & [\bigwedge_{p\in\mathbf{P}_r\cup\mathbf{P}_l} \mathsf{not\ ancestor\text{-}or\text{-}self} :: p \\ & \bigwedge_{f\in\mathbf{N}_r\cup\mathbf{N}_l} \mathsf{not\ ancestor\text{-}or\text{-}self} :: f] \end{array}$$

If C3 holds for a node n, then n is indeterminate.
Table 2 shows a summary for the semantics in Damiani et. al.

Decision	Semantics
Accessible	if not $C3$ and ($C1$ or $C2$)
Not Accessible	if not $C3$ and (not $C1$ and not $C2$)
Indeterminate	if $C3$

Table 2: Summary of the semantics in Damiani et. al.

We have discussed the semantics of their approach only in the case of the *local takes precedence over recursive/-deny overwrites* conflict resolution policy. If the DTD and document-level dimension is introduced, then we define the semantics in exactly the same way as previously by combining the two policies.

Query Evaluation: Similar to [17] in order to perform query evaluation, the authors construct the authorized document. They use a two-phase algorithm where in the first phase they label each node in the XML tree with a tuple indicating whether a specific type of authorization holds or not for the node.

In the second phase, the conflict resolution policy is applied to decide whether the node is accessible or not, and based on this decision the XML document is pruned to produce the authorized document.

3.4 Secure and Selective Dissemination of XML Documents: Bertino et. al.

Bertino et. al. have published a considerable number of papers in the field of XML access control [5, 6, 7, 8, 9]. In this paper we discuss the model presented in [9].

In [9] a rule is a tuple of the form: (*requester*, *resource*, *privilege*, *propagation*) where the first two and the last field are as described in Section 2 and *privilege* defines the type of access the requester has on the resource. In their policies they consider *only positive* rules.

Granularity and Language: Similar to the above approaches, authors in [9] consider the XML node as the smallest unit of protection for which authorizations are defined.

The language used to specify the XML nodes concerned by an authorization is a variant of XPath and is based on the composition of *element names* and *attribute names*. Their expressions can be translated into XPath expressions in the XPath fragment given in Section 2.1 using the attribute and child XPath axes. Their name tests are restricted to node labels or $\star$ that matches all labels. Predicates consider equalities of path expressions with constants. Finally, they consider only absolute expressions in their rules.

In addition, the use of *element identifiers* is supported that allow one to specify access control rules for a *specific* XML node. In XML we can model that by associating each element in the document with the XML attribute identifier whose value is the element identifier. To refer to a node n whose value of the identifier attribute is m, we can simply write:

$$n[\mathsf{attribute} :: \mathsf{identifier} = m]$$

Default Semantics: The default semantics is *deny*.

Semantics: In their model, they use both *local* and *recursive* rules. The scope of a local rule is the node only and not its attributes (as in [14]) or text sub-nodes (as in [23]). The scope of a recursive rule can be either all the sub-elements of an element or those that are found at a certain depth of the XML tree. Recursive rules cannot be specified for attribute nodes.

Another important issue in their model is that they treat differently the XML attributes of type IDREF. To do that, they define two *privileges* (or actions in our terminology): the *view* privilege which allows one to read the attributes (and their values) (except the IDREF attributes) of an element. The *navigate* privilege allows one to read the IDREF attributes (and hence be aware of the horizontal relationships between elements in an XML document). They also define the privilege *browse_all* which is the combination of the *view* and *navigate* privileges.

These privileges can be defined on both elements and attributes. The restriction that they impose is that a *navigate* privilege can be defined only on attributes of type IDREF, whereas the *view* privilege can be defined on non-IDREF attributes.

We represent here formally the semantics of their model using our specification language. An access control policy is defined as $ACP = (R_v, R_n, L_v, L_n)$, where R_v (resp. L_v) is the set of recursive (resp. local) rules with *view* privilege and R_n (resp. L_n) is the set of recursive (resp. local) rules with *navigate* privilege[3]. One can observe that instead of defining different privileges as in their model for the XML attributes, we define for each privilege a set of rules.

An element node is accessible, if there is some recursive rule or a local rule that applies to it. This condition is

[3]The *browse_all* privilege on an element node can be encoded by assigning to it a rule from R_v/L_v and a rule from R_n/L_n.

formulated as the following XPath filter C1:

$$\mathsf{C1} : [\bigvee_{p \in L_v \cup L_n} \mathsf{self} :: p \bigvee_{p \in R_v \cup R_n} \mathsf{ancestor\text{-}or\text{-}self} :: p]$$

The XPath filter that we must apply to the result set of a query is straightforward from the filters in Section 2.2.1 and Section 2.2.2 when no negative rules are considered.

In the case of attributes: a non-IDREF attribute is accessible if there exist a local rule with view privilege that grants access to it. Similarly, an IDREF attribute is accessible if there exists a local rule with a navigate privilege that grants access to it. Recall that recursive rules cannot be defined on attributes, and also recursive rules defined on elements do not have impact on their attribute nodes. Conditions C2 and C3 express when a non-IDREF and IDREF attribute is accessible:

$$\mathsf{C2} : [\bigvee_{p \in L_v} \mathsf{self} :: p] \qquad \mathsf{C3} : [\bigvee_{p \in L_n} \mathsf{self} :: p]$$

3.5 XML Access Control Language

Authors in [27] propose the XACL (XML Access Control Language) which allows one to express in an XML syntax authorizations for XML documents.

A policy may contain *only* local or recursive rules but never both. Another type of recursive rules is introduced where the scope of a rule, defined for a node n, are the *ancestors* of n: if there is a rule which grants or denies access to a node then it grants or denies respectively access to the node's ancestors.

A rule is a tuple of the form (*requester, resource, action, effect, condition*) where the last specifies additional constraints that must hold for a rule to be used to decide whether a node is accessible or not. In the following we consider that *condition* is fixed and again we concentrate on the resource part of the rule.

Granularity and Language: Similar to most of the above approaches, the smallest unit of protection for which an authorization is defined is the XML node. XPath is used to define the XML nodes concerned by a policy and since they do not explicitly describe the fragment that is used, we consider that it is all of XPath 1.0 [12].

Default Semantics: If there is no rule which neither denies nor grants access to a node then the node can be either accessible or inaccessible (*grant* or *deny* as default semantics resp.).

Conflict Resolution: A policy can have one of the following conflict resolution policies: *deny*, *grant* overwrites or *default*. In the last case, the *default* semantics of the policy is considered as the conflict resolution policy.

Semantics: Based on the above observations, a policy can be associated with one of the combinations of conflict resolution policy and default semantics:

	Conflict Resolution	*Default Semantics*
Case 1	deny overwrites	deny
Case 2	deny overwrites	grant
Case 3	grant overwrites	deny
Case 4	grant overwrites	grant

We have mentioned earlier that XACL supports another type of rules whose scope are the ancestors of a node: if a rule grants (denies) access to a node, then it grants (denies) access respectively to its ancestors. In other words, a node n is accessible if:

1. there exist a rule which grants access to one of the descendants of n or to itself *and*
2. there does not exist a rule which denies access to one of the descendants of n or to itself.

In the case of deny overwrites as conflict resolution policy and deny as the default semantics, the XPath filter expression that we must write is:

$$[\bigvee_{p \in \mathbf{P}_r} \mathsf{descendant\text{-}or\text{-}self} :: p \bigwedge_{f \in \mathbf{N}_r} \mathsf{not\ descendant\text{-}or\text{-}self} :: f]$$

If the conflict resolution policy is grant, then we just remove from the previous filter the test on the negative rules.

To conclude, up until this point we have shown how we can express the semantics of XACL policies (i) in the case of local or recursive rules (with scope the descendants or ancestors of a node) (ii) *deny* or *grant* as conflict resolution policy and (iii) *deny* as default semantics. We have thus covered cases 1 and 3 above for all types of rules.

Let us see now how we express the semantics of access control policies in the case where the default semantics is *grant*. We will give the filter expressions only for local rules (for recursive rules we just need to replace appropriately the XPath self axis with the ancestor-or-self and descendant-or-self axes).

In the case where the conflict resolution policy is *deny overwrites* (Case 2) then a node is accessible only if there are no negative rules that deny access to it.

In this case, the filter expression that we must write is:

$$[\bigwedge_{f \in \mathbf{N}_l} \mathsf{not\ self} :: f]$$

In the case where the conflict resolution policy is *grant overwrites* (Case 4) above, then the only case a node *is not accessible* is when:

1. there exist a negative rule that denies access to it, *and*
2. there does not exist a positive rule that grants access to it;

If a positive rule exists for the node, because of the grant overwrites policy the node is accessible. If no rule exists for the node, then again it is accessible because of the grant as default semantics.

If we formulate that positively, a node is accessible if either there exist a positive rule for it, or does not exist a negative rule for it. Formally:

$$[\bigvee_{p \in \mathbf{P}_l} \mathsf{self} :: p \bigvee_{f \in \mathbf{N}_l} \mathsf{not\ self} :: f]$$

4. CONCLUSIONS

In this paper we proposed a formal way to represent the semantics of XML access control policies using XPath.

We have demonstrated how naturally we can express the different conflict resolution policies in the XPath dialect. The only policies that we cannot express are those that consider the assignment of "numeric" priorities to rules as in [17].

We defined the semantics of an access control policy by applying to the result of a query against a document, an XPath filter expression which finally will keep the accessible nodes. As reported in [11] this approach leads to the *non-secure* evaluation of queries, since the user may *deduce* that she is not allowed to see certain information. In this paper we were not concerned with this issue, but it seems possible to extend our approach to handle it.

We are currently examining how to use our approach for access control policies where action other than *read* is considered. Another interesting research direction is to examine the interaction between the semantics of access control policies for resources and for requestors for which a hierarchy exists.

Acknowledgments: We would like to thank Arnaud Sahuguet for discussions and comments on this work.

5. REFERENCES

[1] Third Generation Partnership Project. http://3gpp.org.

[2] S. Abiteboul, P. Buneman, and D. Suciu. *Data On the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann Publishers, October 1999.

[3] The Andrew File System (AFS). http://www.psc.edu/general/filesys/afs/afs.html.

[4] M. Benedikt, W. Fan, and G. Kuper. Structural Properties of XPath Fragments. In *ICDT*, Siena, Italy, January 2003.

[5] E. Bertino, S. Castano, and E. Ferrari. On Specifying Security Policies for Web Documents with an XML-based language. In *SACMAT*, Chantilly, Virginia, USA, May 2001.

[6] E. Bertino, S. Castano, and E. Ferrari. Securing XML Documents: The Author-X Project Demonstration. In *SIGMOD*, Santa Barbara, California, USA, May 2001.

[7] E. Bertino, S. Castano, E. Ferrari, and M. Mesiti. Specifying and Enforcing Access Control Policies for XML Document Sources. *WWW*, 3(3), 2000.

[8] E. Bertino, S. Castano, E. Ferrari, and M. Mesiti. Protection and administration of XML data sources. *DKE*, 43(3):237–260, 2002.

[9] E. Bertino and E. Ferrari. Secure and Selective Dissemination of XML Documents. *TISSEC*, 5(3):290–331, 2002.

[10] S. Boag, D. Chamberlin, M. Fernandez, D. Florescu, J. Robie, and J. Simeon. XQuery: A Query Language for XML. http://www.w3.org/TR/xquery, November 2003.

[11] S. Cho, S. Amer-Yahia, L. V. S. Lakshmanan, and D. Srivastava. Optimizing the Secure Evaluation of Twig Queries. In *VLDB*, Hong Kong, China, September 2002.

[12] J. Clark and S. DeRose (eds.). XML Path Language (XPath) Version 1.0. W3C Recommendation, November 1999. http://www.w3c.org/TR/xpath.

[13] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati. Securing XML Documents. In *EDBT*, Kostanz, Germany, March 2000.

[14] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati. A Fine-Grained Access Control System for XML Documents. *TISSEC*, 5(2):169–202, May 2002.

[15] J. Clark (ed.). XSL Transformation (XSLT) Version 1.0. W3C Recommendation, November 1999. http://www.w3c.org/TR/xslt.

[16] W. Fan, C-Y. Chan, and M. Garofalakis. Secure XML Querying with Security Views. In *SIGMOD*, 2004. To appear.

[17] A. Gabillon and E. Bruno. Regulating Access to XML Documents. In *Working Conference on Database and Application Security*, July 2001.

[18] S. Godik and T. Moses (eds). eXtensible Access Control Markup Language (XACML) Version 1.0. OASIS Standard, 2003 February.

[19] G. Gottlob, C. Koch, and R. Pichler. The complexity of XPath query evaluation. In *PODS*, pages 179–190, San Diego, California, USA, June 2003.

[20] S. Jajodia, P. Samarati, V. Subrahmanian, and E. Bertino. A Unified Framework for Enforcing Multiple Access Control Policies. In *SIGMOD*, Tuscon, Arizona, USA, May 1997.

[21] Liberty Alliance Project. http://www.projectliberty.org.

[22] M. Marx. XPath with conditional axis relations. In *EDBT*, pages 477–494, Crete, Greece, March 2004.

[23] M. Murata, A. Tozawa, and M. Kudo. XML Access Control using Static Analysis. In *CCS*, Washington, DC, USA, October 2003.

[24] A. Silberschatz, G. Gaqne, and P. Galvin. *Operating System Concepts*. John Wiley and Sons, 2002.

[25] D. Srivastava. Directories: Managing Data for Networked Applications. In *ICDE*, Los Alamos, California, USA, March 2000. Tutorial.

[26] H. Thompson, D. Beech, M. Maloney, and N. Mendelsohn (eds.). XML Schema Part 1 Structures. http://www.w3.org/TR/xmlschema-1, May 2001.

[27] XML Access Control. http://www.trl.ibm.com/projects/xml/xacl/.