Formalizing XML Access Control for Update Operations

Irini Fundulaki
University of Edinburgh
UK
Irini.Fundulaki@ed.ac.uk

Sebastian Maneth
NICTA Ltd. and UNSW
Australia
Sebastian.Maneth@nicta.com.au

ABSTRACT

Several languages have been proposed over the past years which support the specification of access control on XML data. Most of these languages consider read-access restrictions only and do not deal with access rights for updates (such as add, delete, or modify operations). Fine-grain XML update operations are subject to current research. This paper proposes XACU, a language for specifying access control on XML data in the presence of update operations. The update operations used in XACU are based on the W3C XQuery Update Facility working draft. A formal access control model is defined which allows to study properties of XACU access policies. One essential property is consistency: the policy should not allow the execution of a sequence of updates which has the same total effect as an update forbidden by the policy. Since XACU is a rich language with inherent ambiguities, checking consistency of a set of XACU rules is difficult, and undecidable in general.

Categories and Subject Descriptors: H.2.3: I.7.2

General Terms: Security

Keywords: XML Access Control, XML Updates

1. INTRODUCTION

XML is the standard for data representation and exchange on the Internet. An important issue is to secure XML content and to ensure the selective exposure of data to different classes of users. There has been a substantial amount of work on enforcing access control policies to user queries, known as *secure XML querying*. There are various aspects to this problem: languages that allow one to specify access policies [11, 20, 17, 12, 7], algorithms for enforcing policies at query time [6, 1, 16, 9, 10, 5, 14, 2], and special data structures to optimize query processing [4, 22]. Most of these proposals focus on secure XML querying for read-only queries. New challenges emerge when access control rules are used to specify update rights. One immediate problem is in the choice of update language used to specify XML update operations: no agreed upon standard language exists today, and many proposed languages lack a formal semantics. In this paper we use the operations introduced in the XQuery Update Facility Working Draft [3]. After reviewing the relevant update operations, our contributions are: (1) the access control specification language XACU, (2) a formal model that determines update rights for a given document and XACU specification, (3) the alternative language $\text{XACU}^{\text{annot}}$ that supports access control annotations at the level of the XML DTD (similar to [6]), and (4) the formalization of the notion of inconsistency in XACU specifications and the proof of its undecidability.

```
1.  <!ELEMENT conference (track+)>
2.  <!ELEMENT track (papers,reviewers)>
3.  <!ELEMENT papers (paper+) >
4.  <!ELEMENT reviewers (reviewer+)>
5.  <!ELEMENT paper (title,abstract,type,authors,reviews?)>
6.  <!ELEMENT authors (author+)>
7.  <!ELEMENT reviews (review+)>
8.  <!ELEMENT review (public?,private?,disc?,reviewer+)>
9.  <!ELEMENT author (name,school?,email?)>
10. <!ELEMENT reviewer (name,school?,email?,conflictInfo)>
11. <!ELEMENT conflictInfo (author*)>
12. <!ELEMENT type (short|long)>
```

Figure 1: Conference DTD

We now discuss a motivating example for access control with updates. Consider the DTD in Fig. 1 which describes information related to a conference. For instance, line 1 says that a `conference` has one or more tracks, line 8 says that a `review` is associated with optional `public`, `private`, and `disc` (discussion) elements and one or more `reviewers`. Line 11 says that `conflictInfo` contains zero or more authors. Each of `long` and `short` elements has an empty content. The remaining elements (such as `title`, `abstract`, etc.) are text elements. The XML document shown in Fig. 2 conforms to the DTD of Fig. 1.

```
<conference>
  <track><papers><paper>
    <title>The Essence of XML</title>
    <abstract/>
    <type><short/></type>
    <authors><author>Phil Wadler</author></authors>
    <reviews/>
  </paper></papers></track>
</conference>
```

Figure 2: XML document

Table 1 shows examples of access control rules for update operations. These rules are written in natural language and

SACMAT'07, June 20-22, 2007, Sophia Antipolis, France.

are concerned with XML documents conforming to the conference DTD of Fig. 1. They are defined for two different user groups: *author*s and *reviewer*s.
Consider that an author wishes to change the title of a paper which she co-authored. According to the rules, such a request is denied because only the rule in line 7 applies and forbids the modification of a title. It remains to be determined how to deal with (1) a request to which no rule applies, such as a reviewer wanting to modify her email address, or, (2) a request to which two conflicting rules apply. As common, we address these issues by fixing (1) a default semantics and (2) an override policy. The first one says that everything not explicitly mentioned by a rule is either by default allowed, or by default forbidden. The second says that either a positive rule overrides a conflicting negative one, or the other way around.

A third issue is the type of the data that is added or modified through an update. More precisely, we would like to specify additional constraints on the data, not covered by the DTD. E.g., an *author* may only insert a paper if she is one of the authors of that paper. This is done by annotating our access rules with additional type constraints. As in [6], the constraints may use constant parameters (such as, e.g., `$my_name`, which would hold the user's name).

```
1. an author can insert, delete and modify her information
   (school and email);
2. an author can add a paper;
3. an author can delete a paper that she has co-authored;
4. an author cannot delete a paper that she has not co-authored;
5. an author can modify the abstract of a paper;
6. an author cannot modify the abstract of a paper that she has
   not co-authored;
7. an author cannot modify the title of a paper;
8. a reviewer  can add, delete, modify private/public comments
   to a paper that she has been assigned to;
9. a reviewer can add, delete, modify the disc comments to a
   paper that she is not in conflict with;
```

Table 1: Examples of access control rules

A fourth, more challenging issue not covered by the override policy, is an update that is explicitly forbidden by a rule, but which can be achieved by an alternative sequence of allowed updates. For instance, an author may not modify the title of a paper (according to the rule in line 7). However, the author may delete the paper altogether, and then insert it again with a changed title. Should this update sequence of delete and insert be allowed, in a policy where negative rules override positive ones? – probably not. How should the system determine which update is forbidden? Should the delete already be forbidden? Or only the insert? Or, only the insert of a paper which appeared before with a different title? From our point of view, such a conflict of a positive sequence of updates with a negative rule is not a matter of override policy; rather, it as an *inconsistency* of the access rules. In other words, a set of rules is *consistent*, if no update disallowed by a negative rule can be achieved by a sequence of updates allowed by positive rules. Note that these inconsistencies arise from the ambiguity inherent in the update language used in the access control rules. E.g., a replace update is semantically equivalent to a delete followed by an insert.

Clearly, it is desirable to be able to automatically detect inconsistencies. Unfortunately, checking rules for consistency is very hard. In fact, even for a simple update language, consistency checking already becomes undecidable. We conjecture that consistency checking *can* be done automatically, if the nesting depth of the XML document is bounded by a constant.

Related Work

The problem of access control for XML updates is relatively new and has been studied by a small number of projects. As in the case of access control for read operations, the smallest unit of protection is the XML node. In general, access control rules are of the form (*subject*, *object*, *effect*, *action*) where *subject* is the user or role to whom access is granted or denied, *object* is a path expression that denotes the set of XML nodes concerned by the rule, *effect* specifies whether the operation is allowed or denied, and *action* specifies the update operation under concern. A policy is a set of access control rules associated with a conflict resolution policy and default semantics. These rules are specified for XML documents that are valid w.r.t. a given DTD. The authors of [1] consider *append*, *write* and *auth_all* actions in their positive access rules. The first allows one to modify the content of an XML node. The second, supports the modification of the content of a node by deleting the node. The last subsumes the previous two. To specify the *object* of a rule they use a language similar to XPath. They define formally the semantics of a policy by means of the *view of a document* and show how this is computed for the different flavors of their *read* operation only. In addition, *propagation options* are specified which state whether a rule applies to a single node or also to its descendants. In [5] the *object* component of a rule is an absolute XPath expression. Access control rules are defined at both the document and the DTD level and can be recursive (their scope is the node and its descendants) or local (their scope is just the node). The *write* actions supported are: insert, update and delete. The semantics of a policy is specified by means of a labelling algorithm, that annotates the nodes of an XML document with accessibility information for a given write action. Gabillon in [8] uses the update operations of XUpdate [21]. The object concerned by an authorization is explicitly specified by means of its position and value in the XML tree. The authors of [13] consider the update operations of [19] in their access control model (insert, insert before/after, delete, replace and rename operations). They support update access rights not only at the level of the XML document but also at the level of the DTD. Both positive and negative access control rules are supported and XPath 1.0 is used to specify the nodes that are concerned by a rule. The semantics of policies are specified by means of a two-step enforcement algorithm.

Rabitti et. al. [18] discuss consistency for access control rules on object oriented databases. The use *implications* between the rules' *subjects*, *objects* and *actions*. They define consistency as follows: for any positive/negative authorization A, if there exist some authorization A' that implies A, then there must not exist some other authorization A'' that implies the negative/positive authorization A. The operations supported are: *read* (R), *write* (W), *generate* (G) and *read definition* (RD). In their model, W implies R and G, G implies RD, and R implies RD. Using such implication rules, the set of all possible authorizations can be computed and inconsistencies can be detected. Since object hierarchies are trees of bounded depth, their consistency check is related to the one in Section 5, for the case of non-recursive DTDs.

2. XML UPDATE OPERATIONS

We review some update operations of the W3C XQuery Update Facility working draft [3] which are used in XACU. An update operation is applied to a set of nodes specified by an XPath *target* expression. We assume the reader to be familiar with the syntax and semantics of the XPath 1.0 language. In a *delete* operation, *target* specifies the XML nodes to be deleted. For *insert*, *replace*, and *rename*, *target* must specify a single node; moreover, the latter operations take a second argument: an XML fragment, called the *source*. Note that we always tacitly identify with an XML document a parsed tree representation consisting of element and text nodes. In the following, names in parentheses are abbreviations which we will use later.

Insert For inserts, *source* must be a sequence of nodes and *target* must evaluate to a single node; otherwise a dynamic error is raised. Let *target-node* be the node returned by the *target*.

• insert *source* as first/last into *target* (insertFirst/insertLast)
Inserts nodes in *source* as first/last children of *target-node*, respectively. As an example, consider inserting a new `paper` element under the `papers` element node (reachable by the XPath expression *//papers*) using the insertFirst operation. The result of this update is a document containing the new paper followed by the one with author `Phil Wadler`.

• insert *source* before/after *target* (insertBefore/insertAfter)
Inserts *source* as *preceding/following* sibling nodes of *target-node* respectively. Here, *target-node* must have a parent node, otherwise a dynamic error is raised. As an example, consider inserting a `paper` node using insertBefore with *target* expression *//papers/paper*[*title* = "*The Essence of XML*"] to the document in Fig. 2. The result is the same document as in the previous example.

• insert *source* into *target* (insertInto)
Inserts the nodes in *source* as children of *target-node*. Note that the position within the children of *target-node* is undetermined (i.e., may be implementation dependent). Thus, the effect of executing an insertInto command can be that of insertLast, insertFirst, insertBefore, or insertAfter applied to any child of *target*.

Delete For deletes, *target* may evaluate to an arbitrary sequence of nodes, denoted *target-nodes*. The operation delete *target* (delete) deletes the nodes in *target-nodes* along with their descendant nodes. For instance, applying delete with *target* expression *//papers/paper*[*title* = "*The Essence of XML*"] to the XML document of Fig. 2 results in a document with an empty `papers` node.

Replace and Rename For a replace operation, *target* must evaluate to a single node; For a rename operation, *target* must evaluate to a single element node. Otherwise, a dynamic error is raised. Let *target-node* be the node returned by *target*.

• replace *target* with *source* (replace)
Replaces *target-node* with *source*. If *target-node* is an element or text node, then *source* must be a sequence of element or text nodes, respectively. The *target-node* and its descendants are deleted and replaced by *source* together with its descendants. Applying the following update to the document of Fig. 2 results in a document where the title of the paper with author `Phil Wadler` has been changed to "`The Essence of XACU`".

```
replace //papers/paper/title[. ="The Essence of XML"]
with <title>The Essence of XACU</title>
```

• rename *target* as *source* (rename)
Replaces the name of *target-node* with *source*. In this case, *source* must be a string value (`PCDATA`).

```
rename //papers/paper/title[.="Note on XACU"]/type/short
as "long"
```

3. XML UPDATE ACCESS CONTROL

This section introduces the access control specification language XACU and defines a formal model for giving semantics to XACU specifications.

Syntax of XACU We support fine-grained access control, i.e., the smallest unit of protection is an XML node; this is in accordance with the majority of existing approaches for XML access control [1, 5, 9, 16, 11, 20]. While earlier approaches merely consider read-only access privileges, our concern is to propose a model that also includes *update access privileges*. When formalizing update privileges we follow the operations introduced in the previous section. That is, we distinguish between the right to insert, delete, replace, or rename an XML element or text node. We are only concerned here with update rights and assume that all nodes of the document are readable by everyone.

An *access control policy* consists of a finite set of *access control rules*. As common, an access control rule is of the form: (*subject*, *object*, *action*, *effect*) where *subject* is the user or role concerned by the rule. For this work, it is sufficient to consider that the *subject* is fixed; hence we will simply talk about rules of the form (*object*, *action*, *effect*). *object* is an XPath 1.0 expression. Evaluating this expression on an XML document returns the nodes to which the rule applies, i.e., the nodes to which the user may apply *action*. *effect* specifies whether the rule grants ('+' sign) or denies ('−' sign) *action* access to *object*. We refer to the access control rules that grant access to a node as *positive* and those that deny access as *negative*. Note that here, independent of the *action*, the *object* component of a rule is not required to return only a single node (as is the case for most of the update operations); hence, any rule may target a sequence of nodes returned by *object*. Finally, *action* is one of insertBefore[X], insertAfter[X], delete, insertFirst[X], insertLast[X], insertInto[X], replace[X], and rename[X]. In our update actions we optionally allow one to specify a name X of an element type (denoted by "[X]"). In this way it is possible to specify that a user may, for instance, insert elements only of a certain element type X.

We explain here informally the semantics of an access control rule (*object*, *action*, *effect*) for each action that we support in our model. For simplification, we assume positive rules. Let T be the XML document of concern, and let V be the set of nodes obtained from evaluating *object* on T.

- (*object*, delete, +): *subject* can *delete* the nodes in V;
- (*object*, insertInto[X], +) *subject* can *insert* nodes (of type X), as children of the nodes in V;
- (*object*, insertFirst[X]/insertLast[X], +) *subject* can *insert* nodes (of type X), as first/last resp. children nodes of the nodes in V;
- (*object*, insertBefore[X]/insertAfter[X], +) *subject* can *insert* nodes (of type X), as preceding/following resp. sibling nodes of the XML nodes in V;
- (*object*, replace[X], +) *subject* can *replace* the nodes in V (with nodes of type X);

- (*object*, rename$[X]$, +): *subject* can *rename* the XML nodes in V with name X.

Let us now take a look at some concrete examples. Consider the rules in Table 1. Rule #1 states that an author can insert, delete and replace her email information. This rule can be written in XACU as follows:

(//author[name = $my_name], insertInto[email], +)
(//author[name = $my_name]/email, delete, +)
(//author[name = $my_name]/email, replace[email], +)

An author can insert a paper as stated by rule #2. This can be written as: (//papers, insertInto[paper], +)

Rule #9 states that a reviewer can delete the discussion comments to a paper that she is not in conflict with. This rule can be written in XACU as:

(//review[not(./../../authors/author/name
≠ //reviewer[name = $my_name]/conflictInfo
/author/name)]/disc, delete, +)

Note that we do not need to distinguish between local and recursive rules, as is done in many of the approaches [1, 5, 16, 6, 9, 10] because this distinction is already supported by our use of XPath; if a node is specified by the XPath expression p, then the sequence of all its descendants is specified by the expression p/descendant-or-self::∗.

Formal Semantics of XACU Given a XACU policy P (i.e., of a set of XACU rules) and an XML document T, the semantics of P determines the nodes of T to which a user may apply a certain update operation (or a read). We assume the reader to be familiar with XML DTDs and make the following assumptions: (1) there is a given fixed DTD D, (2) T conforms to D (3) T after an update operation conforms to D. If (3) is not satisfied, then the update operation is rejected, independently of its accessibility to the user. In fact, we assume that in any access control rule (*object*, *action*, *effect*) in P, *action* (when applicable) appears *with* an optional type parameter X. Note that this is not a restriction because we can easily fill in missing type parameters with the corresponding element type of the DTD. By $\mathcal{X}$ we denote the set of all optional type parameters X which appear in the rules of P.

We denote by P_{action}/N_{action} the set of the *object* components of the positive/negative access control rules resp. where *action* is one of insertInto$[X]$, insertBefore$[X]$, insertAfter $[X]$, insertFirst$[X]$, insertLast$[X]$, delete, replace$[X]$ and rename$[X]$.

P , N	not P , not N	not P , N	P , not N
$NA[NA]$	$NA[A]$	$NA[NA]$	$A[A]$

Table 2: deny [allow] as default semantics

We now want to specify when a node is accessible for a specific action u. For this we need to answer the following questions: (1) *Default Semantics:* what happens to a node if it is not in the scope of some rule? Is it accessible or inaccessible? (2) *Conflict Resolution Policy:* what happens to a node when it is in the scope of both a positive and a negative access control rule? Several ways have been proposed to resolve conflicts, e.g., *i)* by using *priorities*: each access control rule is assigned a priority and the rule with the highest priority is used [5, 9]; *ii) deny overrides* (negative rule takes precedence over positive rule) [5, 16, 20] and *iii) grant overrides* (positive rule takes precedence over negative rule) [20]. We now discuss the semantics of an access control policy by focusing on the *deny overrides* conflict resolution policy.

In general, independently of the operation of concern, the truth table in Table 2 holds for the combination of default semantics and deny overrides as the conflict resolution policy. Here P stands for the truth value of *"has positive right to carry out the operation"*, N stands for *"has negative right to carry out the operation"*, A stands for *"accessible"* and NA for *"inaccessible"*. For instance, Table 2 says that for *deny as default semantics*, a node is accessible if and only if "P, not N". Thus, an algorithm for determining accessibility must check that the concerned node is in the scope of a positive rule, but not in the scope of a negative rule.

Next, we give absolute XPath filter expressions that determine when a node is accessible for a given operation. In [7] filters are given to determine the semantics of the read operation. Recall that here we are not concerned with read access rights, and assume that every node is readable by all users. We denote by $[\![target]\!]_T$ the set of nodes returned by evaluating *target* on the XML document T.

Insert Into Intuitively, one may execute the update "insert *source* into *target*" if one has the right to insert at any child position of the node n, $n \in [\![target]\!]_T$. For this, we check that the insert operation is allowed and that no semantically equivalent update operation is forbidden. We formally define the accessibility of a node for the different cases of default semantics.

In the following, we assume that an insertInto rule implies insertFirst and insertLast rights for a node n (in other words, if one can(not) insert children nodes of some node n as specified by some insertInto rule, then one can(not) insert first and last children of node n – unless of course there exist an insertFirst, insertLast rule that explicitly specifies otherwise). This does not hold for insertFirst, insertLast rules: an insertFirst/insertLast rule does not imply that one can add children of some node at any position. Similar for insertBefore, insertAfter rules and an insertInto rule. In the filter expressions below, $\bigwedge, \bigvee$ stand for and and or, respectively. For $X \in \mathcal{X}$, let super(X) denote the set of all $Y \in \mathcal{X}$ such that X is a subtype of Y (determined by the DTD).

1. *deny overrides as conflict resolution policy, deny as default semantics.* The update "insert *source* into *target*" is granted to node n if *i)* $n \in [\![target]\!]_T$, and *ii) source* conforms to $X \in \mathcal{X}$, and *iii)* all of (a) – (f) are satisfied;
   (a) n is in the scope of a positive insertInto$[Y]$, $Y \in$ super(X)
   (b) n is not in the scope of a negative insertInto$[Y]$, $Y \in$ super(X)
   (c) n is not in the scope of a negative insertFirst$[Y]$, $Y \in$ super(X)
   (d) n is not in the scope of a negative insertLast$[Y]$, $Y \in$ super(X)
   (e) there is no child of n in the scope of a negative insertBefore$[Y]$, $Y \in$ super(X)
   (f) there is no child of n in the scope of a negative insertAfter$[Y]$ rule, $Y \in$ super(X).

   More formally, the above conditions can be expressed by the following filter:

   (1): $[\bigvee_{p \in P_{\text{insertInto}}[Y], Y \in \text{super}(X)}$ self::p
   (2): $\bigwedge_{f \in N_{\text{insertInto}}[Y], Y \in \text{super}(X)}$ not self::f
   (3): $\bigwedge_{f \in N_{\text{insertFirst}}[Y], Y \in \text{super}(X)}$ not self::f
   (4): $\bigwedge_{f \in N_{\text{insertLast}}[Y], Y \in \text{super}(X)}$ not self::f
   (5): $\bigwedge_{f \in N_{\text{insertAfter}}[Y], Y \in \text{super}(X)}$ not child::f
   (6): $\bigwedge_{f \in N_{\text{insertBefore}}[Y], Y \in \text{super}(X)}$ not child::$f]$

2. *deny overrides as conflict resolution policy, allow as default semantics.* The update "insert *source* into *target*" is granted if, *i)* $n \in [\![target]\!]_T$, *ii) source* conforms to $X \in \mathcal{X}$, and *iii)* all of (a) – (e) are satisfied; again as above, in each line Y is arbitrary in super(X).
   (a) n is not in the scope of a negative insertInto[Y]
   (b) n is not in the scope of a negative insertFirst[Y]
   (c) n is not in the scope of a negative insertLast[Y]
   (d) there is no child of n in the scope of a negative insertBefore[Y]
   (e) there is no child of n in the scope of a negative insertAfter[Y] rule.

   More formally, the above conditions can be expressed with the following filter:

$$[\bigwedge_{f \in N_{\mathsf{insertInto}}[Y], Y \in \mathrm{super}(X)} \mathsf{not\ self}::f$$
$$\bigwedge_{f \in N_{\mathsf{insertFirst}}[Y], Y \in \mathrm{super}(X)} \mathsf{not\ self}::f$$
$$\bigwedge_{f \in N_{\mathsf{insertLast}}[Y], Y \in \mathrm{super}(X)} \mathsf{not\ self}::f$$
$$\bigwedge_{f \in N_{\mathsf{insertAfter}}[Y], Y \in \mathrm{super}(X)} \mathsf{not\ child}::f$$
$$\bigwedge_{f \in N_{\mathsf{insertBefore}}[Y], Y \in \mathrm{super}(X)} \mathsf{not\ child}::f]$$

Insert First/Last The update "insert *source* as first/last into *target*" is granted to node n if *i)* $n \in [\![target]\!]_T$, and *ii) source* conforms to $X \in \mathcal{X}$, and in the case of

1. *deny overrides as conflict resolution policy, deny as default semantics*: (a) and (b) are satisfied (and similar for insertLast):
   (a): $[\bigvee_{p \in P_{\mathsf{insertFirst}}[Y], Y \in \mathrm{super}(X)} \mathsf{self}::p$
   (b): $\bigwedge_{f \in N_{\mathsf{insertFirst}}[Y], Y \in \mathrm{super}(X)} \mathsf{not\ self}::f]$
2. *deny overrides as conflict resolution policy, allow as default semantics*: (a) is satisfied (and similar for insertLast):
   (a): $[\bigwedge_{f \in N_{\mathsf{insertFirst}}[Y], Y \in \mathrm{super}(X)} \mathsf{not\ self}::f]$

The filters for the remaining update operations are defined analogous to the ones for insertFirst.

4. ANNOTATION- BASED MODEL

In this section we introduce the language XACU$^{\text{annot}}$ for the specification of access control rules. We follow the idea of *security annotations* introduced in [6] to specify the access authorizations for XML documents in the presence of a DTD. Formally, an access specification S is a tuple (D, ann) where D is a DTD and ann is a partial mapping (the *annotation*) of the form: $\mathsf{ann}(gpath, op) ::= Y \mid N \mid [q]$ where *op* is one of insertFirst[X], insertLast[X], insertInto[X], replace[X], rename[X], insertBefore[X], insertAfter[X], or delete. Further, *gpath* is a path in the DTD graph of the form $B_1.B_2.\cdots B_k$ where the B_i are element types of D, B_{i+1} appears in the right hand side of the production rule for B_i and B_1 is the root element type of D.

As for XACU we allow the operations to use the name of an element type in the DTD to denote that one can, for instance, insert only nodes of a given type as children of some node. Notice that we use in our annotations a path in the graph of the DTD instead of a pair of element types as in [6]. The reason is that in the DTDs we consider, we allow an element type to have more than one element types as parents (i.e., element type author in the DTD of Fig. 1) which was not the case in [6]. Hence, an element type can be reached through different paths from the root element type of the DTD.

As in [6], a value of Y, N and $[q]$ denotes that in an instantiation of D, the elements that are instances of the element type designated by *gpath*, are accessible, inaccessible, or conditionally accessible, respectively, for operation *op*. If ann() is not explicitly defined, then the accessibility of the node is determined using the *default semantics*. Table 3 presents the semantics of the security annotations supported in XACU$^{\text{annot}}$ for the insert and delete operations. The semantics for the replace and rename operations are defined in a similar way.

Annotation
ann($gpath$, insertFirst[X]/insertLast[X]/insertInto[X]) = $Y\|N\|[q]$
Semantics
For node u, instance of the element type reachable by $gpath$, one can (Y)/cannot (N)/can if $[q]$ is true at u, insert nodes of type X as *first/last/at any position children* of u.
Annotation
ann($gpath$, insertBefore[X]/insertAfter[X]) = $Y\|N\|[q]$
Semantics
For node u, instance of the element type reachable by $gpath$, one can (Y)/cannot (N)/can if $[q]$ is true at u, insert nodes of type X as *preceding/following siblings* of u.
Annotation
$ann(gpath,$ delete$) = Y\|N\|[q]$
Semantics
For node u, instance of the element type reachable by $gpath$, one can (Y)/cannot (N)/can if $[q]$ is true at u, delete u and its descendant nodes.

Table 3: Semantics of security annotations

We now show how to translate access control annotations in XACU$^{\text{annot}}$ into XACU rules and vice versa. As discussed in Section 3, an access control rule is of the form (*object*, *action*, *effect*). Since XACU uses XPath expressions to denote the nodes that are concerned by a rule, the task here is to find the XPath expression p for a given *gpath* and qualifier $[q]$ that when evaluated on some XML document T returns exactly the same set of nodes designated by *gpath* and $[q]$. This XPath expression will be the *object* component of the access control rule. For an annotation $\mathsf{ann}(gpath, op) = Y|N|[q]$, we create an access control rule (*object*, *action*, *effect*) where: *i) action = op ii) effect* is '+' if ann() = $Y \mid [q]$ otherwise *effect* is '−' and finally *object* is defined as follows: if $\mathsf{ann}(gpath, op) = Y \mid N$ and $gpath = B_1.B_2 \ldots B_k$, then $object = /B_1/B_2/\ldots/B_k$, otherwise if $\mathsf{ann}(gpath, op) = [q]$ and $gpath = B_1.B_2 \ldots B_k$, then $object = /B_1/B_2/\ldots/B_k[q]$.

When going from XACU to XACU$^{\text{annot}}$ we need to construct the *gpath* expression from the *object* component of a XACU rule. For this we assume that the DTD D is *non-recursive*, i.e., that the graph of D is non-circular. This implies that we can translate any XPath expression containing descendant/ancestor axes into a union of expressions that only use the child/parent axes. Then, for a XACU rule (*object*, *action*, *effect*), we create an annotation ann(*gpath*, *op*) = $Y|N|[q]$ as follows: *i) op = action ii) gpath* is the path obtained from *object* from which the filters are removed and "/" is substituted with "." and *iii)* ann() is defined as follows: if *object* contains no filter expressions (i.e., is of the form $/B_1/B_2/\ldots B_k$), then $\mathsf{ann}(gpath, op) = Y$ if *effect* is '+' and $\mathsf{ann}(gpath, op) = N$ if *effect* is '−'. Otherwise if *object* contains filter expressions (i.e., is of the form $/B_1[F_1]/B_2[F_2]/\ldots B_k[F_l]$), then $\mathsf{ann}(gpath, op) = [q]$ where q is the filter expression obtained by transforming *object* to a filter expression as suggested in [15].

Proposition. Every XACU policy specified in terms of a non-recursive DTD D can be translated into an equivalent XACU$^{\text{annot}}$ policy and, vice versa.

5. CONSISTENCY OF XACU RULES

A set of XACU rules is *consistent* if there is no forbidden operation and sequence of allowed operations, so that both have the same effect on any input document. By "same effect" we mean that the corresponding XML documents after the updates are identical. Recall from the Introduction the example in the discussion on consistency: an author may not modify the title of a paper, but, the author may delete the paper and then insert it again with a changed title. Clearly, under "document equivalence" this modify operation has the same effect as the delete followed by the insert. Hence, a XACU policy containing corresponding rules is *inconsistent*. Note that updates are often considered in a storage model, not on a document; this typically implies that each node of the document has its own unique identifier. In such a setting of nodes with identity, the modification of a title is *not* equivalent to deleting and reinserting the paper, because the latter changes node identities of the paper node and all its descendants, while the former does not. As another example, consider that a user X is forbidden to delete a node u, but is allowed to replace u by the empty list. Clearly, these rules are *inconsistent*.

Inconsistencies are unwanted because they allow a user to achieve a forbidden operation. Unfortunately, inconsistencies can be much more complicated; imagine in the previous example that the replace rule was only applicable in the presence of some special data item. If the user X can insert the special item, then do the replace operation, and afterwards delete the data item again, then X has achieved the forbidden delete. There can be much longer chains of inserts and deletes. For instance, consider that the special item mentioned before may only be inserted if another, more special item was present, etc.

This raises the question whether it is possible to construct an algorithm which automatically tests whether or not a given set of XACU rules is consistent. In the most general case, this question has a negative answer: there is no such algorithm. The reason for this is that on the path of an XML document, it is possible to simulate the computation of a Turing machine by means of update operations. Finding a simulation sequence of update operations is reduced to the halting problem.

Theorem. It is undecidable whether or not a set of XACU rules is consistent.

Proof. Let M be a Turing machine with tape alphabet $\{0,1\}$. Our XML documents will be binary trees, the leaves of which are labeled by 0 or 1 or symbols of the form (i,q) or (i,q,m) where q is a state of M and $m \in \{\text{Up}, \text{Stay}, \text{Down}\}$. All internal binary nodes are labeled by the symbol c. We define a DTD in such a way that at most one leaf is of the form (i,q) and at most one leaf is of the form (i,q,m). A rule of the Turing machine that says, in state q with reading head on a symbol i, replace i by i', move into state q', and move the reading head $m \in \{\text{L}, \text{S}, \text{R}\}$ (denoting left, stay, and right, respectively) is simulated by XACU rules as follows: we have a right to replace (i,q) by (i',q',m') where m' is Up, Stay, and Down for $m =$ L, S, and R, respectively. Moreover, there is a right to replace the next symbol $j \in \{0,1\}$ below or above (i,q',m') by (j,q'), or to replace (i,q',m') itself by (i',q'). For instance, if $T(0,q')$ denotes the element type of a single node labeled $(0,q')$, then for a right-move of M to a symbol labeled 0 we have

$$(\texttt{//0[parent::c[parent::c[./(i',q',Down)]]]},\ replace[T(0,q')],+)$$

Finally, there is a right to remove (i,q',m'). This concludes the simulation of a rule of M. In order to reduce consistency checking to the halting problem, we have a rule that allows to replace the empty document by any document containing one occurrence of (i,f), where $i \in \{0,1\}$ and f is the unique final state of M. In this way, the rules are consistent if and only if M has a computation that ends with state f. Since the latter is undecidable, the statement of the theorem follows. □

Note that it is essential that we simulate the Turing machine on the paths of an XML document. Using XACU rules it is *not* possible to simulate a Turing machine on the children sequence of a node. In fact, it can be shown, using XML type inference rules, that for XML documents of bounded depth (which are guaranteed by a non-recursive DTD), and for XPath expressions that may only check equality of a data value with a constant, we can indeed decide consistency.

6. REFERENCES

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

[2] L. Bouganim, F.-D. Ngoc, et al. Client-Based Access Control Management for XML documents. In *VLDB*. 2004.

[3] D. Chamberlin, D. Florescu, et al. XQuery Update Facility. http://www.w3.org/TR/xqupdate/, July 2006.

[4] S. Cho, S. Amer-Yahia, et al. Optimizing the Secure Evaluation of Twig Queries. In *VLDB*. 2002.

[5] E. Damiani, S. D. C. di Vimercati, et al. A Fine-Grained Access Control System for XML Documents. *TISSEC*, 5(2):169–202, May 2002.

[6] W. Fan, C.-Y. Chan, et al. Secure XML Querying with Security Views. In *SIGMOD*. 2004.

[7] I. Fundulaki and M. Marx. Specifying Access Control Policies for XML documents with XPath. In *SACMAT*. 2004.

[8] A. Gabillon. An authorization Model for XML Databases. In *ACM Workshop on Secure Web Services*. 2004.

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

[10] A. Gabillon, M. Munier, et al. An Access Control Model for Tree Data Structures. In *ISC*. 2002.

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

[12] V. Gowadia and C. Farkas. RDF Metadata for XML access control. In *ACM Workshop on XML Security*. 2003.

[13] C.-H. Lim, S. Park, et al. Access control of XML documents considering update operations. In *ACM Workshop on XML Security*. 2003.

[14] B. Luo, D. Lee, et al. QFilter:Fine-Grained Run-Time XML Access Control via NFA-based Query Rewriting. In *CIKM*. 2004.

[15] M. Marx. XPath with conditional axis relations. In *EDBT*. 2004.

[16] M. Murata, A. Tozawa, et al. XML Access Control using Static Analysis. In *CCS*. 2003.

[17] M. Oshry, B. Porter, et al. Authorizing Read Access to XML Content Using the access-control Processing Instruction 1.0. http://www.w3.org/TR/access-control, 2006 May. gobble.

[18] F. Rabitti, E. Bertino, et al. A Model of Authorization for Next-Generation Database Systems. *TODS*, 16(1), 1991.

[19] I. Tatarinov, Z. Ives, et al. Updating XML. In *SIGMOD*. 2001.

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

[21] XUpdate: XML Update Language. http://www.xmldb.org/xupdate, 2000.

[22] T. Yu, D. Srivastava, et al. Compressed Accessibility Map : Efficient Access Control for XML Documents. In *VLDB*. 2002.