A General Approach for Securely Updating XML Data*

Houari Mahfoud
University of Nancy 2 & INRIA-LORIA Grand Est
Nancy, France
houari.mahfoud@loria.fr

Abdessamad Imine
University of Nancy 2 & INRIA-LORIA Grand Est
Nancy, France
abdessamad.imine@loria.fr

ABSTRACT

Over the past years several works have proposed access control models for XML data where only read-access rights over non-recursive DTDs are considered. A small number of works have studied the access rights for updates. In this paper, we present a general model for specifying access control on XML data in the presence of the update operations of W3C XQuery Update Facility. Our approach for enforcing such update specification is based on the notion of *query rewriting*. A major issue is that query rewriting for recursive DTDs is still an open problem. We show that this limitation can be avoided using only the expressive power of the standard XPath, and we propose a linear algorithm to rewrite each update operation defined over an arbitrary DTD (recursive or not) into a safe one in order to be evaluated only over the XML data which can be updated by the user. This paper represents the first effort for securely XML updating in the presence of arbitrary DTDs (recursive or not) and a rich fragment of XPath.

Categories and Subject Descriptors

H.2.7 [**Database Administration**]: Security, integrity and protection — *Access control*

General Terms

Algorithms, Security

Keywords

XML access control, XML views, XPath, XQuery

1. MOTIVATION

The XQuery Update Facility language [13] is a recommendation of W3C that provides a facility to modify some parts of an XML document and leave the rest unchanged, and this through different update operations. This includes rename, insert, replace and delete operations at the node level. The security requirement is the main problem when manipulating XML documents. An XML document may be queried and/or updated simultaneously by different users. For each class of users some rules can be defined to specify parts of the document which are accessible to the users and/or updatable by them. A bulk of work has been published in the last decade to secure the XML content, but only read-access rights has been considered over non-recursive DTDs [3, 4, 9]. Moreover, a few works have considered update rights.

In this paper, we investigate a general approach for securing XML update operations of the XQuery Update Facility language. Abstractly, for any update operation posed over an XML document, we ensure that the operation is performed only on XML nodes updatable by the user. Addressing such concerns requires first a specification model to define update constraints and a flexible mechanism to enforce these constraints at update time.

We now present our motivating example for controlling update access. Consider the recursive DTD[1] of a hospital depicted as a graph in Fig. 1(b) (we refer to this DTD throughout the paper to illustrate our examples). An XML document conforming to this DTD consists of different departments (*dept*) defined by a name *dname* and each department includes patients of the hospital and other patients coming from some clinics (patients under *clinical* element). For each patient (with name *pname* and category *categ*), the hospital maintains a medical history of its parents (*parent*) and a medical folder (*medicalFolder*) which includes all treatments done for this patient (*treatment* can be *analysis* or *diagnosis*); *descp* and *result* represent the description and the result of the treatment respectively. The treatment data is organized into two groups depending on whether the treatment has been done in some laboratories (*analysis* treatments) or not (the *diagnosis* treatments). Each *dname*, *pname*, *categ*, *descp*, and *result* has a single text node (`PCDATA`) as its child. An instance of the hospital DTD is given in Fig. 2. Due to space limitations, this instance is split into Figures 2 (a) and (b), where Fig. 2(b) represents the medical folder of $patient_3$.

Suppose that the hospital wants to impose an update policy that allows the doctors to update all treatment data (e.g., add some treatment results) except those of analysis. According to this policy, only the nodes $treatment_1$ and $treatment_4$ of Fig. 2(b) can be updated. As the nodes $treatment_2$ and $treatment_3$ are analysis treatments they cannot be updated.

Problem 1. The existing access control approaches are unable to specify the above policy. The model given in [3] consists in annotating the schema of the document by different update constraints, like putting attribute `@insert=`Y

*Extended version of this paper can be found in [10]

[1] A DTD is recursive iff at least one of its elements is defined (directly or indirectly) in terms of itself.

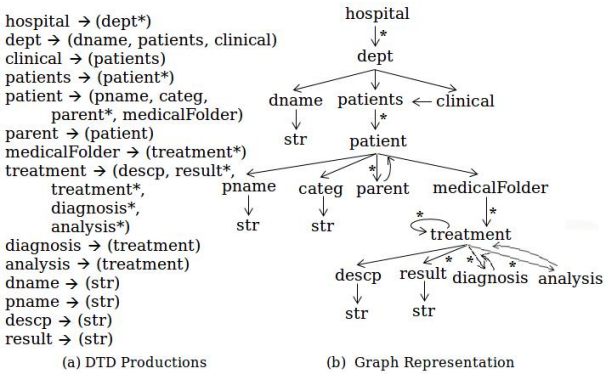

Figure 1: Hospital DTD.

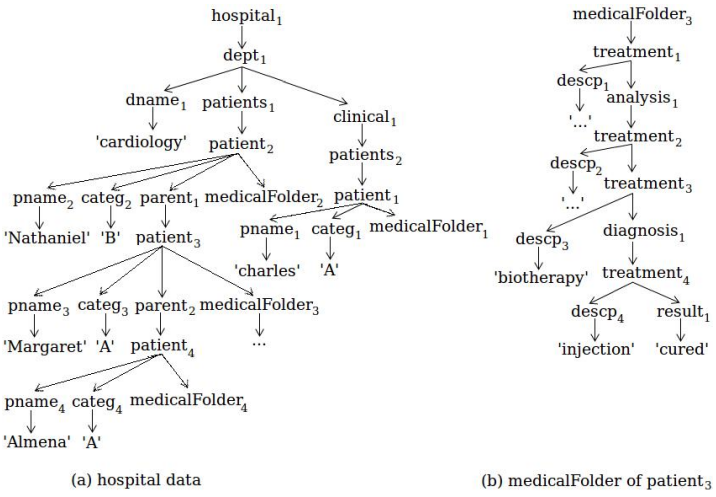

Figure 2: Example of XML Document.

in element type *treatment* of the hospital DTD to specify that some data can be inserted into nodes of type *treatment*. However, only local annotations (the update concerns only the node and not its descendants) are used which is not sufficient to define some update policies. For instance, to enforce the hospital update policy imposed, the analysis treatment data (i.e., nodes $treatment_2$ and $treatment_3$) cannot be discarded from doctors's updates by the model introduced in [3] even by using XPath upward-axes. Specifically, the annotation `@insert=[not(ancestor::`*analysis*`)]` over element type *treatment* is not the adequate constraint since it makes node $treatment_4$ not updatable.

In the XACU^{annot} language presented in [7], an update annotation over an element type of the DTD is defined with a full path from the DTD root to this element. E.g., the annotation $ann(hospital/patients/patient, \texttt{insert})=Y$ specifies that some nodes can be inserted under hospital patients. However, the XACU^{annot} language cannot be applied in the presence of recursive DTDs. For instance, due to recursion, the hospital update policy given above cannot be defined since the paths denoting updatable *treatment* nodes (not done during *analysis*) stand for an infinite set of paths. As we will see in the next, this set of paths can be expressed using the *Kleene* star operator (*) which cannot be expressed in XPath as outlined in [14]. To our knowledge, no model exists for specifying update policies over recursive DTDs.

Problem 2. For each update operation, an XPath expression is defined to specify the XML data at which the update is applied. To enforce rights restriction imposed by an update policy, the *query rewriting* principle can be applied where each update operation (i.e., its XPath expression) is rewritten according to the update rights into a safe one in order to be evaluated only over parts of the XML data updatable by the user. However, this rewriting step is already challenging for a small class of XPath. Consider the downward fragment of XPath which supports *child* and *descendant-or-self* axes, union and complex predicates. We show that, in case of recursive DTDs, an update operation defined in this fragment cannot be rewritten safely. More specifically, a safe rewriting of the XPath expression of an update operation can stand for an infinite set of paths which cannot be expressed in the downward fragment of XPath. To overcome this rewriting limitation, some solutions have been proposed [6, 8] based on the Regular XPath to express safe recursive paths. However, these solutions remain a theoretical achievement since no tool exists to evaluate Regular XPath expressions. Thus, no practical solution exists for enforcing update policies in the presence of recursive DTDs.

Our Contributions. Our first contribution is an expressive model for specifying XML update policies, based on the primitives of the XQuery Update Facility, and over arbitrary DTDs (recursive or not). Given a DTD D, we annotate element types of D with different update rights to specify restrictions on updating XML documents that conform to D through some update operations (e.g., deny insertion of new nodes of type *analysis* under *treatment* nodes). We propose a new model that supports inheritance and overriding of update privileges and overcomes expressivity limitations of existing models. Our approach for enforcing such update policies is based on the notion of *query rewriting*. However, to overcome the rewriting limitation presented above, we investigate the extension of the downward fragment of XPath using upward-axes and position predicate. Based on this extension, our second contribution is an algorithm that rewrites any update operation defined in the downward fragment of XPath into another one defined in the extended fragment to be safely evaluated over the XML data. To our knowledge, this yields the first model for specifying and enforcing update policies using the XQuery update operations and in the presence of arbitrary DTDs.

Related Work. During the last years, several works have proposed access control models to secure XML content, but only read-access has been considered over non-recursive DTDs [3, 4, 9]. There has been a few amount of work on securing XML data by considering the update rights. Damiani et al. [3] propose an XML access control model for update operations of the XUpdate language. They annotate the XML schema with the read and update privileges, and then the annotated schema is translated into two automatons defining read and update policies respectively, which are used to rewrite any access query (resp. update operation) over the XML document to be safe. However, the update policy is expressed only with local annotations which is not sufficient to specify some update rights (see *Problem 1*). Additionally, the automaton processing cannot be successful when rewriting access queries (resp. update operations)

defined over recursive schema (i.e., recursive DTD).

Authors of [7] propose an XML update access control model based on the XQuery update operations. A set of XPath-based rules is used to specify, for each update operation, the XML nodes that can be updated by the user using this operation. These rules are translated into annotations over element types of the DTD (if it exists) to present an annotation-based model called XACU^{annot}. However, this is possible only in the case of non-recursive DTDs.

The view-based access control for XML data has received an increased attention [4, 9, 11]. However, a major issue arises in the case of recursive security views when XPath query rewriting becomes not possible. To overcome this problem, some authors [6, 8] propose rewriting approaches based on the non-standard language, "Regular XPath", which is more expressive than XPath and makes rewriting possible under recursion. However, no system exists for evaluating regular XPath queries in order to demonstrate the practicality of the proposed approaches[2]. Thus, the need of a rewriting system of update operations over recursion remains an open issue.

Outline. The remainder of the paper is organized as follows. Section 2 presents basic notions on DTD, XPath, and XML update operations considered in this paper. We describe in Section 3 our specification model of update. Our approach for securing update operations is detailed in Section 4. Finally, we conclude this paper in Section 5.

2. PRELIMINARIES

This section briefly reviews some basic notions tackled throughout the paper.

DTDs. Without loss of generality, we represent a DTD D by (*Ele*, *Rg*, *root*), where *Ele* is a finite set of *element types*; *root* is a distinguished type in *Ele* called the *root type*; *Rg* is a function defining element types such that for any A in *Ele*, $Rg(A)$ is a regular expression α defined as follows:

$$\alpha \texttt{ := str} \mid \epsilon \mid B \mid \alpha\text{,}\alpha \mid \alpha\text{|}\alpha \mid \alpha\texttt{*}$$

where `str` denotes the text type `PCDATA`, ϵ is the empty word, B is an element type in *Ele*, and finally α,α, α|α, and α* denote concatenation, disjunction, and the Kleene closure respectively. We refer to $A \rightarrow Rg(A)$ as the *production* of A. For each element type B occurring in $Rg(A)$, we refer to B as a *subelement type* (or *child type*) of A and to A as a *superelement type* (or *parent type*) of B. A DTD D is *recursive* if some element type A is defined in terms of itself directly or indirectly.

For example, Fig. 1 represents (a) the productions of the hospital DTD; and (b) its graph representation.

XML Documents. We model an XML document with an unranked ordered finite node-labeled tree. Let Σ be a finite set of node labels, an XML document T over Σ is a structure defined as [14]: T=$(N, R_{\downarrow}, R_{\rightarrow}, L)$ where N is the set of nodes, $R_{\downarrow} \subseteq N \times N$ is a child relation, $R_{\rightarrow} \subseteq N \times N$ is a successor relation on (ordered) siblings, and $L : N \rightarrow \Sigma$ is a function assigning to every node its label. We use the term *XML Tree* for this type of structures.

[2]According to [12] the SMOQE system [5] has been removed because of experiment conduction for additional research.

An XML document T conforms to a DTD D if the following conditions hold: (i) the root of T is the unique node labeled with *root*; (ii) each node in T is labeled either with an *Ele* type A, called an A *element*, or with `str`, called a *text node*; (iii) for each A element with k ordered children $n_1, ..., n_k$, the word $L(n_1), ..., L(n_k)$ belongs to the regular language defined by $Rg(A)$; (iv) each text node carries a string value (`PCDATA`) and is the leaf of the tree. We call T an instance of D if T conforms to D.

XPath Queries. We consider a small class of XPath [1] queries, referred to as $\mathcal{X}$ and defined as follows:

$$\begin{aligned} p &\texttt{ := } \alpha\texttt{::}lab \mid p\texttt{[}q\texttt{]} \mid p\texttt{/}p \mid p \cup p \\ q &\texttt{ := } p \mid p\texttt{/text()='}c\texttt{'} \mid q \texttt{ and } q \mid q \texttt{ or } q \mid \texttt{not (}q\texttt{)} \\ \alpha &\texttt{ := } \varepsilon \mid \downarrow \mid \downarrow^{+} \mid \downarrow^{*} \end{aligned}$$

where p denotes an XPath query and it is the start of the production, *lab* refers to element type or $*$ (that matches all types), $\cup$ stands for union, c is a string constant, α is the XPath axis relations, and ε, $\downarrow$, $\downarrow^{+}$, and $\downarrow^{*}$ denote *self*, *child*, *descendant*, and *descendant-or-self* axis respectively. Finally the expression q is called a *qualifier* or *predicate*.

For a node n in an XML tree T, the evaluation of an XPath query p at n (called *context node* n) results in a set of nodes which are reachable via p from n, denoted by $n[\![p]\!]$.

Practically, this XPath fragment (called *downward* fragment) is commonly used and is essential to XQuery, XSLT and XML Schema [6]. Authors of [6] have shown that in the case of recursive security views, the fragment $\mathcal{X}$ is not closed under query rewriting (i.e., some update operations defined in $\mathcal{X}$ cannot be rewritten to be safe). Our solution to deal with this problem is based on the following extension:

$$\begin{aligned} p &\texttt{ := } \alpha\texttt{::}lab \mid p\texttt{[}q\texttt{]} \mid p\texttt{/}p \mid p \cup p \mid p\texttt{[}n\texttt{]} \\ q &\texttt{ := } p \mid p\texttt{/text()='}c\texttt{'} \mid q \texttt{ and } q \mid q \texttt{ or } q \mid \texttt{not (}q\texttt{)} \\ \alpha &\texttt{ := } \varepsilon \mid \downarrow \mid \downarrow^{+} \mid \downarrow^{*} \mid \uparrow \mid \uparrow^{+} \mid \uparrow^{*} \end{aligned}$$

we enrich $\mathcal{X}$ by the upward-axes *parent* ($\uparrow$), *ancestor* ($\uparrow^{+}$), and *ancestor-or-self* ($\uparrow^{*}$), and the *position* predicate. The position predicate, defined with `[`n`]`($n \in$ N), is used to return the n^{th} node from an ordered set of nodes. For instance, since we model an XML document with an ordered tree, the query $\downarrow$::$*$[1] over a node n returns its first child node. We denote this extended fragment with $\mathcal{X}^{\Uparrow}_{[n]}$.

In our case, fragment $\mathcal{X}$ is used only to formulate update operations and to define our update policies, while we will explain later how the fragment $\mathcal{X}^{\Uparrow}_{[n]}$ defined above can be used to avoid the XPath query rewriting limitation.

XML Update Operations. We review some update operations of the W3C XQuery Update Facility recommendation [13] (abbreviated as XUF). We study the use of the following operations: *insert*, *delete*, and *replace*. In each update operation an XPath *target* expression is used to specify the set of XML node(s) in which the update is applied. Moreover, a second argument *source* is required for *insert* and *replace* operations which represents a sequence of XML nodes. Note that *target* may evaluate to an arbitrary sequence of nodes, denoted *target-nodes*, in the case of *delete* operation. As for other operations, however, *target* must evaluate to a single node, denoted *target-node*; otherwise a dynamic error is raised. The XML update operations considered in this paper are detailed as follows:

• **insert** *source* **into / as first into / as last into / before / after** *target*: Inserts each node in *source* as child, as first child, as last child, as preceding sibling node, or as following sibling node of *target-node* respectively. The order defined between nodes of *source* must be preserved. We abbreviate these kinds of *insert* operations by `insertInto`, `insertAsFirst`, `insertAsLast`, `insertBefore`, and `insertAfter` respectively. In the case of `insertBefore` and `insertAfter` operations, *target-node* must have a parent node; otherwise a dynamic error is raised. For `insertInto` operation, the position of insertion is undetermined and may depend on the XUF implementation. Thus, the effect of executing an `insertInto` operation on *target* can be that of `insertAsFirst`/`insertAsLast` executed on *target* or `insertBefore`/`insertAfter` executed at any child node of *target*.

• **delete** *target*: This operation is used to delete all nodes in *target-nodes* along with their descendant nodes.

• **replace** *target* **with** *source*: Used to replace *target-node* and its descendants with the sequence of nodes specified in *source* by preserving their order. Note that *target-node* must have a parent node; otherwise a dynamic error is raised.

3. UPDATE ACCESS CONTROL MODEL

This section describes our access control model for XML update.

3.1 Update Specifications

We focus on the security annotation principle presented in [4] and on the *update access type* notion introduced in [2] to define our update specifications.

Definition 1. *Given a DTD D, an* update type *defined over D is of the form* `insertInto`$[B_i]$, `insertAsFirst`$[B_i]$, `insertAsLast`$[B_i]$, `insertBefore`$[B_i]$, `insertAfter`$[B_i]$, `d-elete`$[B_i]$ *or* `replace`$[B_i,B_j]$, *where B_i and B_j are element types of D.* □

Intuitively, an update type *ut* represents a set of update operations which are defined for specific element types. For example, the update type `replace`$[B_i,B_j]$ represents the update operations "**replace** *target* **with** *source*" where *target-node* is of type B_i and nodes in *source* are of type B_j.

Definition 2. *We define an* update specification S_{up} *as a pair (D, ann_{up}) where D is a DTD and ann_{up} is a partial mapping such that, for each element type A in D and each update type ut, $ann_{up}(A, ut)$, if defined, is an annotation of the form:*

$$ann_{up}(A,ut) := Y \mid N \mid [Q] \mid N_h \mid [Q]_h$$

where Q is a qualifier in our XPath fragment $\mathcal{X}$. □

An update specification S_{up} is an extension of a document DTD D associating update rights with element types of D.

Let n be a node of type A in an instantiation of D. Intuitively, the authorization values Y, N, and $[Q]$ indicate that, the user is *authorized*, *unauthorized*, or *conditionally authorized* respectively to perform update operations of type *ut* at n (case of *insert* operations) or over children nodes of n (case of *delete* and *replace* operations). For instance, the annotation $ann_{up}(A,$`insertInto`$[B])=Y$ specifies that the user can insert nodes of type B as children nodes of n. However, the annotation $ann_{up}(A,$`replace`$[B_i,B_j])=[Q]$ indicates that B_i children of n can be replaced by new nodes of type B_j iff: $n \vDash Q$. An annotation $ann_{up}(A,ut)=value$ is *valid* at node n iff: (*i*) $value=Y$; or, (*ii*) $value=[Q]||[Q]_h$ and $n \vDash Q$.

Our model supports *inheritance* and *overriding* of update privileges. If $ann_{up}(A,ut)$ is not explicitly defined, then an A element *inherits* the authorization of its parent node that concerns the same update type *ut*. On the other hand, if $ann_{up}(A,ut)$ is explicitly defined it may *override* the inherited authorization of A that concerns the same update type *ut*. All update operations are not permitted by default.

Finally, the semantic of the specification values N_h and $[Q]_h$ is given as follows: The annotation $ann_{up}(A,ut)=N_h$ indicates that, for a node n of type A, update operations of type *ut* cannot be performed at n and no overriding of this authorization value is permitted for descendant nodes of n. For instance, if n' is a descendant node of n whose type is A', then an update operation of type *ut* cannot be performed at n' even though $ann_{up}(A',ut)=Y$ is explicitly defined. While, with the annotation $ann_{up}(A,ut)=[Q]_h$, descendant nodes of an A element can override this authorization value only if Q is valid at this element. For instance, let n and n' be two nodes of type A and A' respectively, and consider the annotation $ann_{up}(A',ut)=[Q']$, then an update operation of type *ut* can be performed at node n' iff: $n' \vDash Q'$. Moreover, if the annotation $ann_{up}(A,ut)=[Q]_h$ is defined and n' is a descendant of n, then the annotation $ann_{up}(A',ut)=[Q']$ takes effect and an update operation of type *ut* can be performed at node n' iff: $n \vDash Q$ and $n' \vDash Q'$. We call annotation with value N_h or $[Q]_h$ as *downward-closed* annotation.

Example 1. Suppose that the hospital wants to impose an update policy that authorizes the doctors to update (insertion, deletion,...) only data of patients having category '*A*', which are under department '*cardiology*' and are not involved in any clinical trial. We define formally this policy over an update type *ut* as follows:

R_1: $ann_{up}(dept,ut)=[\downarrow::dname/text()='cardiology']_h$
R_2: $ann_{up}(clinical,ut)=N_h$
R_3: $ann_{up}(patient,ut)=[\downarrow::categ/text()='A']$

Consider the case where $ut=$`insertInto`$[treatment]$. For a node p of type *patient*, the annotation R_3 takes effect over data of p only if p is under cardiology department and outside of clinics (p has no ancestor node of type *clinical*); otherwise no insertion of *treatment* nodes is permitted under node p regardless its category. For the XML document presented in Fig. 2(a), insertions under nodes $patient_3$ and $patient_4$ are permitted (e.g., insert some *treatment* nodes into $medicalFolder_3$). □

3.2 Rewriting Problem

In the case of recursive DTD, an update operation with *target* defined in fragment $\mathcal{X}$ cannot be rewritten into an equivalent one defined in $\mathcal{X}$ in order to update only authorized data. This problem is known as the XPath closure problem [6]. Consider the following update annotations:

R_1: $ann_{up}(medicalFolder,ut)=Y$
R_2: $ann_{up}(diagnosis,ut)=Y$
R_3: $ann_{up}(analysis,ut)=N$

In the case of $ut=$`delete`$[treatment]$, all *treatment* nodes can be deleted except those of analysis data. The update operation `delete` $\downarrow^+::treatment$ cannot be rewritten into a safe update expressed in $\mathcal{X}$. Indeed, the

paths denoting updatable treatment nodes (not done during analysis) stand for an infinite set. This set of paths can be captured with: $\texttt{delete}$ $(\downarrow^+::medicalFolder \cup \downarrow^+::diagnosis)/(\downarrow::treatment)^*/\downarrow::treatment$. However, the Kleene star ($*$) cannot be expressed in XPath [14].

In the next section we explain how the extended fragment $\mathcal{X}^{\Uparrow}_{[n]}$, defined in Section 2, can be used to overcome this rewriting limitation of update operations.

4. SECURELY UPDATING XML

In this paper we focus only on update rights and we assume that every node is read-accessible by all users. Given an update specification S_{up}=(D, ann_{up}), we discuss the enforcement of such update constraints where each update operation posed over an instance T of D must be evaluated only over nodes of T that can be updated by the user w.r.t. S_{up}. We assume that the XML document T remains valid after the update operation is performed, otherwise the update is rejected. In the following, we denote by S_{ut} the set of annotations defined in S_{up} over the update type ut and by $|S_{ut}|$ the size of this set. Moreover, we denote by $\{ann\}$ the set of all annotations defined with ann, and by $|ann|$ the size of this set.

4.1 Updatability

We say that a node n is *updatable* w.r.t. update type ut if the user is granted to perform update operations of type ut either at node n (case of *insert* operations) or over children nodes of n (case of *delete* and *replace* operations). For instance, B_i children of n can be replaced with nodes of type B_j iff n is updatable w.r.t. $\texttt{replace}[B_i,B_j]$.

Definition 3. *Let S_{up}=(D, ann_{up}) be an update specification and ut be an update type. A node n in an instantiation of D is* updatable *w.r.t. ut if the following conditions hold:*

i) The node n is concerned by a valid annotation with type ut; or, no annotation of type ut is defined over element type of n and there is an ancestor node n' of n such that: n' is the first ancestor node of n concerned by an annotation of type ut, and this annotation is valid at n' (the inherited annotation).

ii) There is no ancestor node of n concerned by an invalid downward-closed annotation of type ut. □

Given an update specification S_{up}=(D, ann_{up}), we define two predicates $\mathcal{U}^1_{ut}$ and $\mathcal{U}^2_{ut}$ (expressed in fragment $\mathcal{X}^{\Uparrow}_{[n]}$) to satisfy the conditions (i) and (ii) of Definition 3 with respect to an update type ut:

$$\begin{aligned}\mathcal{U}^1_{ut} := {} & \uparrow^*::*[\bigvee_{(ann_{up}(A,ut)=Y|N|[Q]|N_h|[Q]_h)\in S_{ut}} \varepsilon::A][1]\\ & [\bigvee_{(ann_{up}(A,ut)=Y)\in S_{ut}} \varepsilon::A\\ & \bigvee_{(ann_{up}(A,ut)=[Q]|[Q]_h)\in S_{ut}} \varepsilon::A[Q]]\\ \mathcal{U}^2_{ut} := {} & \bigwedge_{(ann_{up}(A,ut)=N_h)\in S_{ut}} \text{not } (\uparrow^+::A)\\ & \bigwedge_{(ann_{up}(A,ut)=[Q]_h)\in S_{ut}} \text{not } (\uparrow^+::A[not(Q)])\end{aligned}$$

where $\bigwedge$ and $\bigvee$ denote *conjunction* and *disjunction* respectively. The predicate $\mathcal{U}^1_{ut}$ has the form $\uparrow^*::*[qual_1][1][qual_2]$. Applying $\uparrow^*::*[qual_1]$ on a node n returns an ordered set $\mathcal{S}$ of nodes (node n and/or some of its ancestor nodes) such that for each one an annotation of type ut is defined over its element type. The predicate $\mathcal{S}[1]$ returns either node n, if an annotation of type ut is defined over its element type; or the first ancestor node of n concerned by an annotation of type ut. Thus, to satisfy condition (i) of Definition 3, it amounts to check that the node returned by $\mathcal{S}[1]$ is concerned by a valid annotation of type ut, done by $\mathcal{S}[1][qual_2]$ (i.e., $n \vDash \mathcal{U}^1_{ut}$). The second predicate is used to check that all downward-closed annotations of type ut defined over ancestor nodes of n are valid (i.e., $n \vDash \mathcal{U}^2_{ut}$).

Definition 4. *Let S_{up}=(D, ann_{up}), ut, and T be an update specification, an update type and an instance of DTD D respectively. We define the* updatability predicate $\mathcal{U}_{ut}$ *which refers to an $\mathcal{X}^{\Uparrow}_{[n]}$ qualifier such that, a node n on T is* updatable *w.r.t. ut iff $n \vDash \mathcal{U}_{ut}$, where $\mathcal{U}_{ut} := \mathcal{U}^1_{ut} \bigwedge \mathcal{U}^2_{ut}$.* □

For example, the XPath expression $\downarrow^+::*[\mathcal{U}_{ut}]$ stands for all nodes which are updatable w.r.t. ut. As a special case, if $S_{ut} = \phi$ then $\mathcal{U}_{ut} = false$.

Example 2. According to the update policy of Example 1, the predicate $\mathcal{U}_{ut} := \mathcal{U}^1_{ut} \bigwedge \mathcal{U}^2_{ut}$ is defined with:

$$\begin{aligned}\mathcal{U}^1_{ut} := {} & \uparrow^*::*[\varepsilon::dept \bigvee \varepsilon::clinical \bigvee \varepsilon::patient][1]\\ & [\varepsilon::dept[\downarrow::dname/text()='cardiology']\\ & \bigvee \varepsilon::patient[\downarrow::categ/text()='A']]\end{aligned}$$

$$\begin{aligned}\mathcal{U}^2_{ut} := {} & \text{not } (\uparrow^+::clinical) \bigwedge\\ & \text{not } (\uparrow^+::dept[\text{not } (\downarrow::dname/text()='cardiology')])\end{aligned}$$

Applying $\uparrow^*::*[\varepsilon::dept \bigvee \varepsilon::clinical \bigvee \varepsilon::patient]$ over the node $medicalFolder_3$ of Fig. 2(a) returns the ordered set $\mathcal{S}$=$\{patient_3,\ patient_2,\ dept_1\}$ of nodes (each one is concerned by an annotation of type ut); $\mathcal{S}[1]$ returns $patient_3$; and the predicate $[\varepsilon::dept[\downarrow::dname/text()='cardiology'] \bigvee \varepsilon::patient[\downarrow::categ/text()='A']]$ is valid at $patient_3$. Thus $\mathcal{U}^1_{ut}$ is valid at node $medicalFolder_3$. Also, we can see that $medicalFolder_3 \vDash \mathcal{U}^2_{ut}$. Consequently, the node $medicalFolder_3$ is updatable w.r.t. ut (i.e., $medicalFolder_3 \vDash \mathcal{U}_{ut}$). This means that, in the case of ut=$\texttt{insertInto}[treatment]$, the user is granted to insert nodes of type $treatment$ under node $medicalFolder_3$. However, if ut=$\texttt{delete}[treatment]$, then $treatment$ children of node $medicalFolder_3$ can be deleted (case of node $treatment_1$ of the instance of Fig. 2). □

4.2 Rewriting of Update Operations

Finally, we detail here our approach for enforcing update policies based on the notion of *query rewriting*. Given an update specification S_{up}=(D, ann_{up}). For any update operation with $target$ defined in the XPath fragment $\mathcal{X}$, we translate this operation into a safe one by rewriting its $target$ expression into another one $target'$ defined in the XPath fragment $\mathcal{X}^{\Uparrow}_{[n]}$, such that evaluating $target'$ over any instance of D returns only nodes that can be updated by the user w.r.t. S_{up}. We describe in the following the rewriting of each kind of update operation considered in this paper. We refer to DTD D as a pair $(Ele, Rg, root)$, and to $source$ as a sequence of nodes of type B.

Delete/Replace. Consider the update operation "**delete** *target*". For any node n of type A_i referred to by $target$, parent node n' of n must be updatable w.r.t. $\texttt{delete}[A_i]$ (i.e., $n' \vDash \mathcal{U}_{delete[A_i]}$). To this end, we rewrite the $target$ expression into: $target[\bigvee_{A_i \in Ele} \varepsilon::A_i[\uparrow::*[\mathcal{U}_{delete[A_i]}]]]$.
Consider now the update operation "**replace** *target* **with** *source*". A node n of type A_i referred to by $target$ can

be replaced with nodes in *source* if its parent node n' is updatable w.r.t. $\texttt{replace}[A_i,B]$ (i.e., $n' \vDash \mathcal{U}_{replace[A_i,B]}$). Thus, the *target* expression of the `replace` operations can be rewritten into: $target[\bigvee_{A_i \in Ele} \varepsilon::A_i[\uparrow::*[\mathcal{U}_{replace[A_i,B]}]]]$.

Insert as first into/as last into/before/after. Consider the update operation "**insert** *target* **as first into** *source*". For any node n referred to by *target*, the user can insert nodes in *source* at the first child position of n, regardless the type of n, provided that he holds the $\texttt{insertAsFirst}[B]$ right on this node (i.e., $n \vDash \mathcal{U}_{insertAsFirst[B]}$). To check this, the *target* expression of the above update operation can be simply rewritten into: $target[\mathcal{U}_{insertAsFirst[B]}]$. The same principle is applied for `insertAsLast`, `insertBefore`, and `insertAfter` operations.

Insert into. In the following we assume that: if a node n is concerned by an annotation of type $\texttt{insertInto}[B]$, then this annotation implies $\texttt{insertAsFirst}[B]$ (resp. $\texttt{insertAsLast}[B]$) rights for n, and also $\texttt{insertBefore}[B]$ (resp. $\texttt{insertAfter}[B]$) rights for children nodes of n (inspired from [7]). In other words, if one can(not) insert children nodes of types B at any child position of some node n as specified by some annotations of type $\texttt{insertInto}[B]$, then one can(not) insert nodes of type B in the first and last child position of n and in preceding and following sibling of children nodes of n (unless if there is some annotations of type $\texttt{insertAsFirst}[B]$, $\texttt{insertAsLast}[B]$, $\texttt{insertBefore}[B]$, or $\texttt{insertAfter}[B]$ respectively that specify otherwise). Thus, one can execute the update operation "**insert** *source* **into** *target*" over an XML tree T iff: (i) one has the right to execute update operations of type $\texttt{insertInto}[B]$ on the node n ($n \in T[\![target]\!]$); and (ii) no annotation *explicitly prohibits* update operations of type $\texttt{insertAsFirst}[B]/\texttt{insertAsLast}[B]$ on node n (resp. $\texttt{insertBefore}[B]/\texttt{insertAfter}[B]$ on children nodes of n). When condition (ii) does not hold (e.g. update operations of type `insertAsFirst` is explicitly denied), this leads to situation where there is a *conflict* between `insertInto` and other insert operations.

The first condition is checked using the updatability predicate $\mathcal{U}_{insertInto[B]}$ (whether or not $n \vDash \mathcal{U}_{insertInto[B]}$). For the second condition, however, we define the predicate $\mathcal{U}_{ut}^{-1}$ over an update type ut such that: for a node n, if $n \vDash \mathcal{U}_{ut}^{-1}$ then update operations of type ut are *explicitly forbidden* on node n. An update operation of type ut is *explicitly forbidden* at node n iff at least one of the following conditions holds: a) the node n is concerned by an invalid annotation of type ut; b) no annotation of type ut is defined over element type of n and there is an ancestor node n' of n such that: n' is the first ancestor node of n concerned by an annotation of type ut, and this annotation is invalid at n'; c) there is an ancestor node of n concerned by an invalid downward-closed annotation of type ut.

More formally, for an update specification $S_{up}=(D, ann_{up})$, we define the predicate $\mathcal{U}_{ut}^{-1} := Cnd_{a \vee b} \bigvee Cnd_c$ over an update type ut with:[3]

$$Cnd_{a \vee b} := \uparrow^*::*[\bigvee_{(ann_{up}(A,ut)=Y|N|[Q]|N_h|[Q]_h) \in S_{ut}} \varepsilon::A]\ [1][\bigvee_{(ann_{up}(A,ut)=N|N_h) \in S_{ut}} \varepsilon::A \ \bigvee_{(ann_{up}(A,ut)=[Q]|[Q]_h) \in S_{ut}} \varepsilon::A[not(Q)]]$$

[3]As a special case, if $S_{ut} = \phi$ then $\mathcal{U}_{ut}^{-1} = false$.

$$Cnd_c := \bigvee_{(ann_{up}(A,ut)=N_h) \in S_{ut}} \uparrow^+::A \ \bigvee_{(ann_{up}(A,ut)=[Q]_h) \in S_{ut}} \uparrow^+::A[not(Q)]$$

To resolve the conflict between `insertInto` operation and other insert types, we define the predicate CRP_B ("*Conflict Resolution Predicate*") over an element type B as:

$$CRP_B := \mathcal{U}_{insertAsFirst[B]}^{-1} \bigvee \mathcal{U}_{insertAsLast[B]}^{-1} \bigvee \downarrow::*[\mathcal{U}_{insertBefore[B]}^{-1}] \bigvee \downarrow::*[\mathcal{U}_{insertAfter[B]}^{-1}]$$

Given a node n, if $n \vDash CRP_B$ then at least the update operation $\texttt{insertAsFirst}[B]$ (resp. $\texttt{insertAsLast}[B]$) is forbidden for node n or $\texttt{insertBefore}[B]$ (resp. $\texttt{insertAfter}[B]$) is forbidden for some children nodes of n. Finally, given the update operation "**insert** *source* **into** *target*" over an XML tree T, one can insert nodes of type B in *source* to the node n ($n \in T[\![target]\!]$) iff: $n \vDash \mathcal{U}_{insertInto[B]} \bigwedge not(CRP_B)$. Thus, the *target* of the `insertInto` operation can be rewritten into: $target[\mathcal{U}_{insertInto[B]} \bigwedge not(CRP_B)]$.

The overall complexity time of our rewriting approach of update operations can be stated as follows:

Theorem 1. *For any update specification $S_{up}=(D, ann_{up})$ and any update operation op (defined in $\mathcal{X}$), there exists an algorithm "`Updates Rewrite`" that translates op into a safe one op' (defined in $\mathcal{X}_{[n]}^{\Uparrow}$) in at most $O(|ann_{up}|)$ time.* □

5. CONCLUSION

We have proposed a general model for specifying XML update policies based on the primitives of the XQuery Update Facility. To enforce such policies, we have introduced a rewriting approach to securely updating XML over arbitrary DTDs and for a significant fragment of XPath. To our knowledge, this paper presents the first work for securely updating XML data over general DTDs.

6. REFERENCES

[1] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernández, M. Kay, J. Robie, and J. Siméon. Xml path language (xpath) 2.0 (second edition). *W3C Recommendation*, December 2010.

[2] L. Bravo, J. Cheney, and I. Fundulaki. Accon: Checking consistency of xml write-access control policies. *In EDBT*, 2008.

[3] E. Damiani, M. Fansi, A. Gabillon, and S. Marrara. A general approach to securely querying xml. *In Computer Standards and Interfaces*, 2008.

[4] W. Fan, C.-Y. Chan, and M. Garofalakis. Secure xml querying with security views. *In SIGMOD*, 2004.

[5] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Smoqe: A system for providing secure access to xml. *In VLDB*, 2006.

[6] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Rewriting regular xpath queries on xml views. *In ICDE*, 2007.

[7] I. Fundulaki and S. Maneth. Formalizing xml access control for update operations. *In SACMAT*, 2007.

[8] B. Groz, S. Staworko, A.-C. Caron, Y. Roos, and S. Tison. Xml security views revisited. *In DBPL*, 2009.

[9] G. Kuper, F. Massacci, and N. Rassadko. Generalized xml security views. *In SACMAT*, 2005.

[10] H. Mahfoud and A. Imine. A general approach for securely querying and updating xml data. *INRIA report*, January 2012. `http://hal.inria.fr/hal-00664975/en`.

[11] N. Rassadko. Policy classes and query rewriting algorithm for xml security views. *In Data and Applications Security*, 2006.

[12] N. Rassadko. Query rewriting algorithm evaluation for xml security views. *In VLDB Workshop*, 2007.

[13] J. Robie, D. Chamberlin, M. Dyck, D. Florescu, J. Melton, and J. Siméon. Xquery update facility 1.0. March 2011.

[14] B. ten Cate and C. Lutz. The complexity of query containment in expressive fragments of xpath 2.0. *In PODS*, 2007.