On Securely Manipulating XML Data

Houari Mahfoud and Abdessamad Imine

University of Lorraine and INRIA-LORIA
Nancy, France
{Houari.Mahfoud,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 and expressive 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, in practice, 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 DTDs (recursive or not) into a safe one in order to be evaluated only over the XML data which can be updated by the user. To our knowledge, this work is the first effort for securely updating XML in the presence of arbitrary DTDs, a rich class of update operations, and a significant fragment of XPath.

Keywords: XML Access control, XML Updating, Query Rewriting, XPath, XQuery.

1 Introduction

The XQuery Update Facility language [1] 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 [2–5]. Moreover, a few works have considered update rights [4, 6, 7].

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 that can be updated by the user. Addressing such

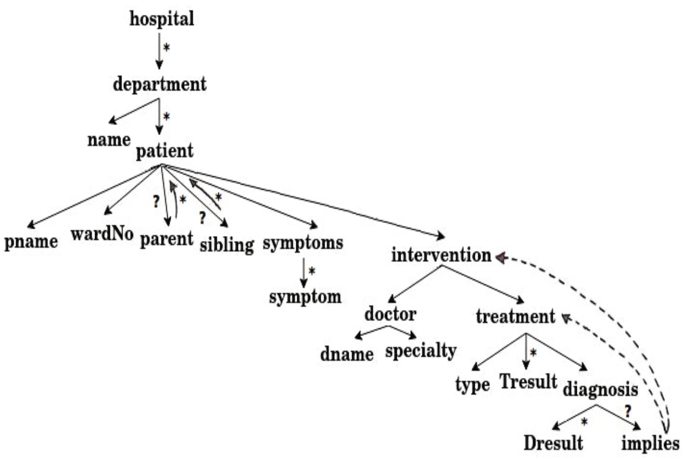

Fig. 1. Hospital DTD

concerns requires first a specification model to define update constraints and a flexible mechanism to enforce these constraints at update time.

We now discuss a motivating example for access control with updates. Consider the recursive DTD[1] depicted as a graph in Fig.1. We use '$*$' on an edge to indicate a list, '?' to indicate optional edge, while dashed edges represent disjunction. A hospital document conforming to this DTD consists of a list of departments (*dept*) defined by a *name*, and each department has a list of children representing patients currently residing in the hospital. For each *patient*, the hospital maintains her name (*pname*), ward number (*wardNo*), family medical history by means of the recursively defined *parent* and *sibling*, as well as list of *symptoms*. The hospitalization is marked by the *intervention* of one or many doctors depending on their specialty and the patient care requirement. For each intervention, the hospital also maintains the information of the responsible *doctor* (defined with name (*dname*) and *specialty*) and the *treatment* applied. A treatment is described by its *type*, a list of result (*Tresult*), and it is followed by a *diagnosis* phase. According to the results of the diagnosis (*Dresult*), the doctor may decide to do another treatment. However, if the required treatment is outside his area of expertise, then the current doctor would solicit the intervention of another doctor, specialist, or expert.

An instance of the hospital DTD is given in Fig. 2. Due to space limitation, this instance is split into two parts. Figure2 (a) represents a simple hospital document with *Cardiology* department, *Critical care* department, as well as some patients information of these departments[2]. Figure2 (b) depicts the three interventions done for $patient_1$: $intervention_1$, $intervention_2$, and $intervention_3$.

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

[2] We use the notation X_i to distinguish between different instances of element type X, like $patient_1$.

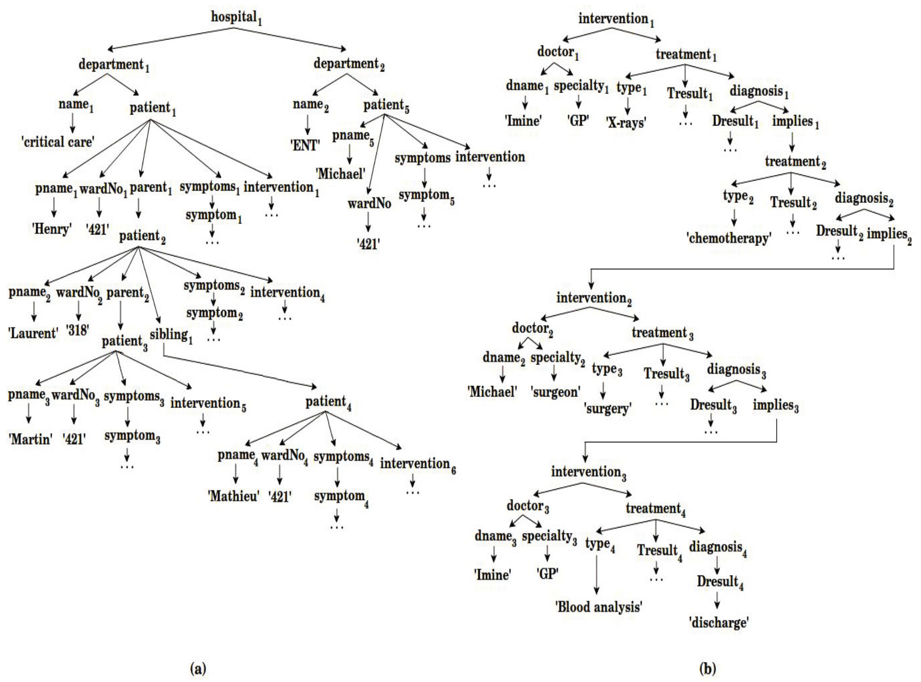

Fig. 2. Hospital Data: (a) patients information, (b) interventions done for patient$_1$

Example 1. (*Update Policy for doctors*) Suppose that the hospital wants to impose a security policy that authorizes each doctor to update only the information of treatments that she has done. For instance, the doctor *Imine* could update the data of $treatment_1$, $treatment_2$, and $treatment_4$ (like insert new *Dresult* sub-tree into the node $diagnosis_4$) but not $treatment_3$. We show in the following that this update policy, even simple, cannot be enforced by using some existing update specification languages. □

Problem 1. (*Expressiveness of Update Specification Languages*) In some case of recursive DTDs, the existing update access control models are unable to specify some update policies. In the model proposed by Damiani et al. [4], the update policy is defined by annotating the XML schema by security attributes. For instance, adding attribute `@insert=[`*test*`='`*Blood Analysis*`']` into element type *treatment* of the hospital DTD specifies that new sub-tree can be inserted to *treatment* nodes having '*Blood Analysis*' as *type*. However, only local annotations can be defined (i.e. the update constraint concerns only the node and not its descendants) which makes the proposed model restricted for non-recursive schema/DTD. For instance, the updates of doctor *Imine* cannot be discarded for the node $treatment_3$ as imposed by the update policy defined above. Specifically, adding attribute `@insert=[ancestor::`*intervention*`[`*doctor*`/`*dname*`='`*Imine*`']]` into *treatment* element type makes all treatment nodes updatable by *Imine*. Since the hospital DTD is recursive, this update policy cannot be specified by the model proposed in [4]. To specify the imposed update policy, a plausible solution may be done by using the *transitive closure operator* '*'. In this case, the adequate update constraint would be defined by adding the following attribute into *treatment* element type:

```
@insert=[(parent::implies/parent::diagnosis/parent::treatment)*/
parent::investigation[doctor/dname=$DNAME]]
```

Where $DNAME is treated as a constant parameter; i.e., when a concrete value, e.g., *Imine*, is substituted for $DNAME, the previous annotation defines the update right for doctor *Imine*. However, the *transitive closure operator* cannot be expressed in the standard XPath as outlined in [8].

Due to space constraints, we do not discuss about the limitation of the update access control model (called XACU) proposed in [6]. For more details, the reader is referred to our extended version available online [3].

To the best of our knowledge, no model exists for specifying update policies over recursive DTDs.

Problem 2. (*Query Rewriting Limitation*) For each update operation, an XPath expression is defined to specify the XML data at which the update is applied. To enforce 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 constraints into a safe one in order to be performed only over parts of the XML data that can be updated by the user who submitted the operation. 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. In case of recursive DTDs, it was shown that an XPath expression 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 (even by using the upward-axes: *parent*, *ancestor*, and *ancestor-or-self*).

To overcome this rewriting limitation, one can use the '*Regular XPath*' language [9], which includes the *transitive closure operator* and allows to express recursive paths. However, it remains a theoretical achievement since no tool exists to evaluate Regular XPath queries. 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 some parts of XML documents that conform to D. Each update right concerns one update operation (e.g., deny insertion of new nodes of type *Tresult* under

[3] http://hal.inria.fr/hal-00664975

treatment nodes). Our model supports *inheritance* and *overriding* of update privileges and overcomes expressivity limitations of existing models (see **Problem 1**). Our approach for enforcing such update policies is based on the notion of *query rewriting*. However, to overcome the rewriting limitation presented above as **Problem 2**, we investigate the extension of the downward fragment of XPath using upward-axes and position predicate. Based on this extension, our second contribution is a linear algorithm that rewrites any update operation defined in the downward fragment of XPath into another one defined in the extended fragment to be safely performed 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 [2–4]. There has been a few amount of work on securing XML data by considering the update rights. Damiani et al. [4] 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). Fundulaki et al. [6] propose an XML update access control model, called `XACU`, for 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. In the presence of non-recursive DTD only, the `XACU` rules can be translated into annotations over element types of the DTD to present an annotation-based model called $\texttt{XACU}^{annot}$.

The view-based access control for XML data has received an increased attention [2,5,10]. 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 [10,11] propose rewriting approaches based on the non-standard language, "Regular XPath" [9], 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. Thus, the need of a rewriting system of XPath queries (resp. update operations) over recursion remains an open issue.

Plan of the Paper. The paper is organized as follows. Section 2 reviews some basic notions tackled throughout the 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 Background

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:

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

where `str` denotes the text type `PCDATA`, ϵ is the empty word, B is an element type in *Ele*, α','α denotes concatenation, and α'|'α denotes disjunction. 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 *sub-element type* (or *child type*) of A and to A as a *super-element type* (or *parent type*) of B. The sub-elements structure can be specified using the operators '*' (set with zero or more elements), '+' (set with one or more elements), and '?' (optional set of elements). A DTD D is *recursive* if some element type A is defined in terms of itself directly or indirectly.

As depicted in Fig. 1, our DTD graph representation is specified with solid edges (which represent conjunction), dashed edges (which represent disjunction). These edges can be labeled with one of the operators '*', '+', or '?'. This simple graph representation suffices to depict our hospital DTD. However, for a complete representation of DTDs, a special *DTD graph* structure can be used, along the same lines as [3].

XML Trees. 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 [9]: T=$(N, R_\downarrow, R_\rightarrow, L)$, where $(N, R_\downarrow)$ is a finite rooted tree with child relation $R_\downarrow \subseteq N \times N$, $R_\rightarrow \subseteq N \times N$ 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.

An XML tree $T = (N, R_\downarrow, R_\rightarrow, L)$ conforms to a DTD $D = (Ele, Rg, r)$ if the following conditions hold: (*i*) the root of T is the unique node labeled with r; (*ii*) each node in T is labeled either with an *Ele* type A, called an A *element*, or with `str`, called a *text node*; (*ii*) 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 [12] queries, referred to as $\mathcal{X}$ and defined as follows:

$$\begin{array}{lll} p & := & \alpha::ntst \mid p[q] \mid p/p \mid p \cup p \\ q & := & p \mid p=\text{'}c\text{'} \mid q \wedge q \mid q \vee q \mid \neg\,(q) \\ \alpha & := & \varepsilon \mid \downarrow \mid \downarrow^+ \mid \downarrow^* \end{array}$$

where p denotes an XPath query and it is the start of the production, *ntst* is a node test that can be an element type, $*$ (that matches all types), or function *text()* (that tests whether a node is a text node), c is a string constant, and $\cup$, $\wedge$, $\vee$, $\neg$ denote *union*, *conjunction*, *disjunction*, and *negation* respectively; α stands for XPath axis relations and can be one of ε, $\downarrow$, $\downarrow^+$, or $\downarrow^*$ which denote *self*, *child*, *descendant*, and *descendant-or-self* axis respectively. Finally the expression q is called a *qualifier* or *predicate*. The result of the evaluation of an $\mathcal{X}$ query p at a *context node* n of an XML Tree T, is the set of nodes reachable via p from n, denoted by $n[\![p]\!]$. We denote by $n \vDash q$ a qualifier q that is valid at a node n.

Authors of [10] have shown that in the case of recursive security views, the fragment $\mathcal{X}$ (called *downward* fragment) is not *closed* under query rewriting. This means that is not always possible to rewrite XPath queries on views to be safely evaluated on the source. Consequently, it is also the problem of update operations rewriting since fragment $\mathcal{X}$ is the core of XQuery, XSLT and XML Schema. Our solution to make possible the update operations rewriting is based on the following extension:

$$
\begin{array}{lll}
p & := & \alpha::ntst \mid p[q] \mid p/p \mid p \cup p \mid p[n] \\
q & := & p \mid p='c' \mid q \wedge q \mid q \vee q \mid \neg\,(q) \\
\alpha & := & \varepsilon \mid \downarrow \mid \downarrow^+ \mid \downarrow^* \mid \uparrow \mid \uparrow^+ \mid \uparrow^*
\end{array}
$$

we enrich $\mathcal{X}$ by the *position* predicate and the upward-axes presented by *parent* axis ($\uparrow$), *ancestor* axis ($\uparrow^+$), and *ancestor-or-self* axis ($\uparrow^*$). The position predicate, defined with $[n]\,(n \in \mathrm{N})$, is used to return the n^{th} node from an ordered set of nodes. For instance, the query $\downarrow::*[2]$ at a node n of an ordered returns its second 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 security policies. While we will explain later how the fragment $\mathcal{X}^{\Uparrow}_{[n]}$ defined above can be used to avoid the rewriting limitation.

XML Update Operations. We review some update operations of the W3C XQuery Update Facility recommendation [1]. We study the use of the following operations: *insert*, *delete*, and *replace*. For each update operation, an XPath *target* expression is used to specify the set of XML node(s) in which the update is applied. In a *delete* operation, *target* specifies the XML nodes to be deleted (denoted *target-nodes*). For *insert*, and *replace* operations, *target* must specify a single node (denoted *target-node*); otherwise a dynamic error is raised. Moreover, the latter operations require a second argument *source* representing a sequence of XML nodes. The order defined between the nodes of *source* must be preserved during the insertion and replacement. In the following, names in brackets are abbreviations of the different operations.

Insert. We distinguish different types of insert operation depending on the position of the insertion:

• **insert** *source* **as first/last into** *target* [*insertAsFirst*/*insertAsLast*]: Here *target-node* must evaluate to a single element node; otherwise a dynamic error is raised. This operation inserts the nodes in *source* as first/last children of *target-node* respectively.

• **insert** *source* **before/after** *target* [*insertBefore*/*insertAfter*]: Inserts the nodes in *source* as preceding/following sibling nodes of *target-node* respectively. In this case, *target-node* must have a parent node; otherwise a dynamic error is raised.

• **insert** *source* **into** *target* [*insertInto*]: Inserts the nodes in *source* as children of the single element node *target-node* (otherwise a dynamic error is raised). Note that the positions of the inserted nodes among the children of *target-node* are implementation-dependent [4]. Thus, the effect of executing an *insertInto* operation on *target-node* can be that of *insertAsFirst*/*insertAsLast* executed on *target-node*, or that of *insertBefore*/*insertAfter* executed at children of *target-node*.

Delete. The operation "**delete** *target*" [*delete*] deletes all *target-nodes* along with their descendant nodes.

Replace. The operation "**replace** *target* **with** *source*" [*replace*] replaces *target-node* with the nodes in *source*. Here *target-node* must have a parent node; otherwise a dynamic error is raised. If *target-node* is an element or text node, then *source* must be a sequence of elements or text nodes respectively. The *target-node* is deleted along with its descendants and replaced by the nodes in *source* together with their descendants.

3 Update Access Control Model

This section describes our access control model for XML update.

3.1 Update Specifications

We follow the idea of *security annotations* presented in [2] and the *update access types* notion introduced in [13] to define a language for specifying expressive and fine-grained XML update policies in the presence of DTDs. An *update specification* S_{up} expressed in the language is a simple extension of the document DTD D associating element types with *update annotations* (XPath qualifiers), which specify for any XML tree T conforms to D, the parts of T that can be updated by the user through a specific update operation.

Definition 1. *Given a document DTD D, an* update type (ut) *defined over D is of the form* *insertInto*$[B_i]$, *insertAsFirst*$[B_i]$, *insertAsLast*$[B_i]$, *insertBefore*$[B_i,B_j]$, *insertAfter*$[B_i,B_j]$, *delete*$[B_i]$, *and* *replace*$[B_i,B_j]$, *where* B_i *and* B_j *are element types of D.* □

[4] For instance, in the DataDirect XQuery implementation, available at `http://www.cs.washington.edu/research/xmldatasets/`, *insertInto* operation has the same effect as *insertAsLast*.

Table 1. Semantics of the update annotations Y, N, and $[Q]$

Annotation	**Semantic**
$ann_{up}(A,\texttt{insertInto}[B_i]) = Y\|N\|[Q]$	for a node n of type A, one can (Y)/cannot (N)/can if $n \vDash Q$, insert nodes of type B_i in an arbitrary position children of n.
$ann_{up}(A,\texttt{insertAsFirst}[B_i]) = Y\|N\|[Q]$	for a node n of type A, one can (Y)/cannot (N)/can if $n \vDash Q$, insert nodes of type B_i as first children of n.
$ann_{up}(A,\texttt{insertBefore}[B_i,B_j]) = Y\|N\|[Q]$	for a node n of type A, one can (Y)/cannot (N)/can if $n \vDash Q$, insert nodes of type B_j as preceding sibling nodes of any child node of n whose type is B_i.
$ann_{up}(A,\texttt{delete}[B_i]) = Y\|N\|[Q]$	for a node n of type A, one can (Y)/cannot (N)/can if $n \vDash Q$, delete children of n whose type is B_i.
$ann_{up}(A,\texttt{replace}[B_i,B_j]) = Y\|N\|[Q]$	for a node n of type A, one can (Y)/cannot (N)/can if $n \vDash Q$, replace children of n of type B_i by some nodes of type B_j.

Intuitively, each update type *ut* represents an update operation that is restricted to be applied only for specific element types. For example, the update type $\texttt{replace}[B_i,B_j]$ 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.

Based on this notion of update type, we define our *update specifications* as follows:

Definition 2. *An* update specification S_{up} *is 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 defined over element types of* D*,* $ann_{up}(A, ut)$*, if explicitly 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 DTD D associating update annotations with element types of D. In a nutshell, a value of Y, N, or $[Q]$ for $ann_{up}(A,ut)$ indicates that, for A elements in an instantiation of D, the user is *authorized*, *unauthorized*, or *conditionally authorized* respectively, to perform update operations of type *ut* at A (case of *insertInto*, *insertAsFirst*, or *insertAsLast* operations) or at children of A (case of the remaining operations). Table 1 presents more specifically the semantics of the update annotations Y, N, and $[Q]$[5].

Our model supports *inheritance* and *overriding* of update annotations. If $ann_{up}(A,ut)$ is not explicitly defined, then an A element *inherits* from its parent node the update authorization 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.

[5] The semantics of annotations with the update types $ann_{up}(A,\texttt{insertAslast}\,[B_i])$ and $ann_{up}(A,\texttt{insertAfter}\,[B_i,B_j])$ are defined in a similar way as $ann_{up}(A,\texttt{insertAsFirst}\,[B_i])$ and $ann_{up}(A,\texttt{insertBefore}\,[B_i,B_j])$ respectively.

Table 2. Semantics of downward-closed annotations (ut can be any update type)

Downward-closed Annotation	**Semantic**
$ann_{up}(A,ut)$ = N_h	Same principle as $ann_{up}(A,ut)$ = N of Table 1. Moreover, for a node n of type A, all annotations of type ut defined over descendant types of A are discarded regardless their truth values.
$ann_{up}(A,ut)$ = $[Q]_h$	Same principle as $ann_{up}(A,ut)$ = $[Q]$ of Table 1. Moreover, for a node n of type A, if $n \nvDash Q$ then all annotations of type ut defined over descendant types of A are discarded regardless their truth values.

Finally, the semantics of the specification values N_h and $[Q]_h$ are given in Table 2. 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 any node of the subtree rooted at n, and no overriding of this authorization value is permitted for descendants of n. For instance, if n has a descendant node n' whose type is A', then an update operation with the same type ut cannot be performed at/under n' even though the annotation $ann_{up}(A',ut)=Y$ is explicitly defined (resp. $ann_{up}(A',ut)=[Q']$ with $n' \vDash Q'$). As for the annotation $ann_{up}(A,ut)=[Q]_h$, qualifier Q must be valid at A elements, otherwise no annotation with update type ut can override the *false* evaluation of Q. For instance, let n and n' be two nodes of type A and A' respectively, and let n' be a descendant node of n. The annotation $ann_{up}(A',ut)=[Q']$ indicates that an update operation of type ut can be performed at (children of) n' iff: $n' \vDash Q'$. Moreover, if the annotation $ann_{up}(A,ut)=[Q]_h$ is explicitly defined then the annotation $ann_{up}(A',ut)=[Q']$ takes effect at descendant node n' of n only if $n \vDash Q$. This means that an update operation of type ut can be performed at (children of) n' iff: $(n \vDash Q \wedge n' \vDash Q')$. We call annotation with value N_h or $[Q]_h$ as *downward-closed* annotation.

Example 2. Suppose that each nurse is attached to only one department and only one ward within this department (denoted \$NURSEDEPT and \$NURSEWARDNO resp.). Now, the hospital wants to impose an update policy that allows a nurse to update data of only patients having the same ward number as her (*Rule1*) and which are being treated at her department (*Rule2*). Moreover, all sibling data cannot be updated (*Rule3*). This policy can be specified by the following update annotations (ut denotes a general update type):

R_1: $ann_{up}(department,ut)=[\downarrow::name=\$\text{NURSEDEPT}]_h$
R_2: $ann_{up}(patient,ut)=[\downarrow::wardNo=\$\text{NURSEWARDNO}]$
R_3: $ann_{up}(sibling,ut)=N_h$

Consider the case of the nurse having the ward number *421* and working at *Critical care* department, and let ut be ***delete***$[symptom]$. This nurse can delete all symptoms of Fig. 2 except: $symptom_2$ (since $patient_2$ has ward number *318*), $symptom_4$ (representing part of sibling data), and $symptom_5$ (although $patient_5$

has ward number *421*, he is attached to *ENT* department). Notice that the annotations R_1 and R_3 must be defined as *downward-closed* to enforce the imposed policy, otherwise annotation R_2 overrides at nodes $patient_4$ and $patient_5$ the negative authorizations inherited respectively from the nodes $sibling_1$ and $department_2$, which violates the imposed policy and makes possible the deletion of the nodes $symptom_4$ and $symptom_5$. $\square$

3.2 Rewriting Problem

As will be seen shortly, in the case of recursive DTDs, update operations rewriting is already challenging for the small fragment $\mathcal{X}$ of XPath. Recall the update policy defined in Example 1. In our case, this policy can be specified by defining only the following update annotation:

$$ann_{up}(intervention,\ ut) = [\downarrow::doctor/\downarrow::dname{=}\$\text{DN{\scriptsize AME}}]$$

Where \$DName is a constant parameter representing doctor's name, and *ut* can be any update type relevant to the update rules of Example 1. Now, let \$DName be *Imine* and *ut* be $\texttt{delete}[Tresult]$. The update operation $\texttt{delete}\ \downarrow^{+}::treatment[\downarrow::type{=}'chemotherapy']/\downarrow::Tresult$ cannot be rewritten in $\mathcal{X}$ to be safe. Indeed, the *Tresult* nodes that doctor *Imine* is authorized to delete can be represented by an infinite set of paths. This latter can be captured by rewriting the previous update into the following one: $\texttt{delete}\ \downarrow^{+}::intervention[\downarrow::doctor/\downarrow::dname{=}\$\text{DN{\scriptsize AME}}]/(\downarrow::treatment/\downarrow::diagn\text{-}osis/\downarrow::implies)^{*}/\downarrow::treatment[\downarrow::type{=}'chemotherapy']/\downarrow::Tresult$, defined in Regular XPath and which, when evaluated on the XML tree of Fig. 2, delete only the node $Tresult_2$. However, the Kleene star cannot be expressed in XPath [9].

We explain in the next section 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 section 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 performed only at the nodes of T that can be updated by the user w.r.t. S_{up}. We assume that the XML tree 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} with the update type *ut* and by $|S_{ut}|$ the size of this set. Moreover, for an annotation function *ann*, 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, if a node n is updatable w.r.t. ***insertInto***$[B]$, then some nodes of type B can be inserted as children of n. Moreover, B_i children of n can be replaced with nodes of type B_j iff n is updatable w.r.t. ***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[6] *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′ (called 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:

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

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; checked by the predicate $\mathcal{S}[1][qual_2]$ (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 $\mathcal{U}_{ut} := \mathcal{U}^1_{ut} \wedge \mathcal{U}^2_{ut}$.* □

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 $\mathcal{U}_{ut} = false$.

[6] Note that an annotation $ann_{up}(A, ut)$=*value* is *valid* at a node n if this latter is of type A and either *value*=Y; or, *value*=$[Q]/[Q]_h$ and $n \vDash Q$.

Property 1. For an update specification $S_{up}{=}(D, ann_{up})$ and an update type ut, the updatability predicate $\mathcal{U}_{ut}$ can be constructed in at most $O(|ann_{up}|)$ time. Moreover, $|\mathcal{U}_{ut}|{=}O(|ann_{up}|)$. □

Example 3. Consider the three update annotations R_1, R_2, and R_3 defined in Example 2, and let the update type ut be $\texttt{delete}$[symptom]. According to these annotations, the predicate $\mathcal{U}_{ut} := \mathcal{U}^1_{ut} \wedge \mathcal{U}^2_{ut}$ is defined with:

$$\begin{aligned}\mathcal{U}^1_{\texttt{delete}[symptom]} := \ & \uparrow^*{::}{*}[\varepsilon{::}department \vee \varepsilon{::}patient \vee \varepsilon{::}sibling][1]\\ & [\varepsilon{::}department[\downarrow{::}name{=}\$\text{NURSEDEPT}]\\ & \vee\ \varepsilon{::}patient[\downarrow{::}wardNo{=}\$\text{NURSEWARDNO}]]\\ \mathcal{U}^2_{\texttt{delete}[symptom]} := \ & \text{not } (\uparrow^+{::}department[\text{not } (\downarrow{::}name{=}\$\text{NURSEDEPT})]) \wedge\\ & \text{not } (\uparrow^+{::}sibling)\end{aligned}$$

Consider the case of the nurse having the ward number *421* and working at *Critical care* department. The predicate $\uparrow^*{::}{*}[\varepsilon{::}department \vee \varepsilon{::}patient \vee \varepsilon{::}sibling]$ over the node $patient_3$ of Fig. 2 returns the ordered set $\mathcal{S}{=}\{patient_3, patient_2, patient_1, department_1\}$ of nodes (each one is concerned by an annotation of type $\texttt{delete}$[symptom]); $\mathcal{S}[1]$ returns $patient_3$ and the predicate $[\varepsilon{::}department[\downarrow{::}name{=}\text{'}Critical\ care\text{'}] \vee \varepsilon{::}patient[\downarrow{::}wardNo{=}\text{'}421\text{'}]]$ is valid at node $patient_3$ (i.e. $patient_3 \vDash \mathcal{U}^1_{\texttt{delete}[symptom]}$). Also, we can see that $patient_3 \vDash \mathcal{U}^2_{\texttt{delete}[symptom]}$. Consequently, the node $patient_3$ is updatable w.r.t. $\texttt{delete}[symptom]$ (i.e., $patient_3 \vDash \mathcal{U}_{\texttt{delete}[symptom]}$). This means that the nurse is granted to delete *symptom* elements of $patient_3$ (e.g. node $symptom_3$). However, for node $patient_5$ we can check that the predicate $\mathcal{U}^1_{\texttt{delete}[symptom]}$ is valid, while it is no longer the case for the predicate $\mathcal{U}^2_{\texttt{delete}[symptom]}$ ($patient_5$ has an ancestor node $department_2$ with $name \neq$ '*Critical care*'). Thus, the nurse is not allowed to delete the node $symptom_5$. □

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 T 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, where DTD $D = (Ele, Rg, root)$ and *source* is a sequence of nodes of type B_j.

- " **delete** *target* ": For any node n of type B_i referred to by *target*, parent node n' of n must be updatable w.r.t. $\texttt{delete}[B_i]$ (i.e., $n' \vDash \mathcal{U}_{\texttt{delete}[B_i]}$). To satisfy this, we rewrite *target* expressions of $\texttt{delete}$ operations into: $target[\vee_{B_i \in Ele}\ \varepsilon{::}B_i[\uparrow{::}{*}[\mathcal{U}_{\texttt{delete}[B_i]}]]]$.
- " **replace** *target* **with** *source* ": A node n of type B_i referred to by *target* can be replaced with the nodes in *source* iff its parent node n' is updatable w.r.t.

$\texttt{replace}[B_i,B_j]$ (i.e., $n' \vDash \mathcal{U}_{\texttt{replace}[B_i,B_j]}$). Thus, *target* expressions of $\texttt{replace}$ operations can be rewritten into:
$target[\vee_{B_i \in Ele}\ \varepsilon{::}B_i[\uparrow{::}*[\mathcal{U}_{\texttt{replace}[B_i,B_j]}]]]$.

• " **insert** *source* **before/after** *target* ": For any node n referred to by *target*, the user can insert nodes in *source* as preceding sibling nodes of n iff its parent node n' is updatable w.r.t. $\texttt{insertBefore}[B_i,B_j]$ (i.e., $n' \vDash \mathcal{U}_{\texttt{insertBefore}[B_i,B_j]}$). To satisfy this, *target* expressions of $\texttt{insertBefore}$ operations can be rewritten into: $target[\vee_{B_i \in Ele}\ \varepsilon{::}B_i[\uparrow{::}*[\mathcal{U}_{\texttt{insertBefore}[B_i,B_j]}]]]$. The same principle is applied for $\texttt{insertAfter}$ operations.

• " **insert** *source* **into** *target* ": For any node n referred to by *target*, the user can insert nodes in *source* as children of n (in an implementation-dependent position), provided that he holds the $\texttt{insertInto}[B_j]$ right on this node (i.e., $n \vDash \mathcal{U}_{\texttt{insertInto}[B_j]}$). To check this, *target* expressions of $\texttt{insertInto}$ operations can be simply rewritten into: $target[\mathcal{U}_{\texttt{insertInto}[B_j]}]$. The same principle is applied for $\texttt{insertAsFirst}$ and $\texttt{insertAsLast}$ operations.

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

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.

References

1. Robie, J., Chamberlin, D., Dyck, M., Florescu, D., Melton, J., Siméon, J.: Xquery update facility 1.0 (March 2011), `http://www.w3.org/TR/xquery-update-10/`
2. Fan, W., Chan, C.Y., Garofalakis, M.N.: Secure XML querying with security views. In: ACM SIGMOD (2004)
3. Kuper, G.M., Massacci, F., Rassadko, N.: Generalized XML security views. Int. J. Inf. Sec. 8(3), 173–203 (2009)
4. Damiani, E., Fansi, M., Gabillon, A., Marrara, S.: A general approach to securely querying XML. Computer Standards & Interfaces 30(6), 379–389 (2008)
5. Rassadko, N.: Policy Classes and Query Rewriting Algorithm for XML Security Views. In: Damiani, E., Liu, P. (eds.) Data and Applications Security 2006. LNCS, vol. 4127, pp. 104–118. Springer, Heidelberg (2006)
6. Fundulaki, I., Maneth, S.: Formalizing XML access control for update operations. In: ACM SACMAT (2007)

7. Duong, M., Zhang, Y.: An integrated access control for securely querying and updating XML data. In: Australasian Database Conference (2008)
8. ten Cate, B., Lutz, C.: The complexity of query containment in expressive fragments of xpath 2.0. In: Proceedings of the Twenty-Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (2007)
9. Marx, M.: XPath with Conditional Axis Relations. In: Bertino, E., Christodoulakis, S., Plexousakis, D., Christophides, V., Koubarakis, M., Böhm, K. (eds.) EDBT 2004. LNCS, vol. 2992, pp. 477–494. Springer, Heidelberg (2004)
10. Fan, W., Geerts, F., Jia, X., Kementsietsidis, A.: Rewriting regular xpath queries on XML views. In: ICDE (2007)
11. Groz, B., Staworko, S., Caron, A.-C., Roos, Y., Tison, S.: XML Security Views Revisited. In: Gardner, P., Geerts, F. (eds.) DBPL 2009. LNCS, vol. 5708, pp. 52–67. Springer, Heidelberg (2009)
12. Berglund, A., Boag, S., Chamberlin, D., Fernández, M.F., Kay, M., Robie, J., Siméon, J.: Xml path language (xpath) 2.0 (second edition). W3C Recommendation (December 2010), `http://www.w3.org/TR/2010/REC-xpath20-20101214/`
13. Bravo, L., Cheney, J., Fundulaki, I.: Repairing Inconsistent XML Write-Access Control Policies. In: Arenas, M. (ed.) DBPL 2007. LNCS, vol. 4797, pp. 97–111. Springer, Heidelberg (2007)