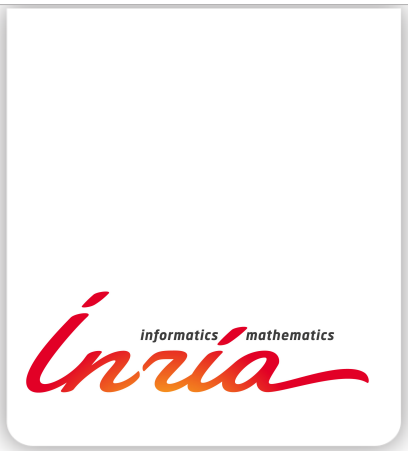
Secure Querying of Recursive XML Views: A Standard XPath-based Technique

Houari Mahfoud and Abdessamad Imine

RESEARCH REPORT
N° 7834
December 2011
Project-Teams CASSIS

ISSN 0249-6399 ISRN INRIA/RR--7834--FR+ENG



Secure Querying of Recursive XML Views: A Standard XPath-based Technique

Houari Mahfoud * and Abdessamad Imine†

Project-Teams CASSIS

Research Report n° 7834 — December 2011 — 30 pages

Abstract: Most state-of-the art approaches for securing XML documents allow users to access data only through authorized views defined by annotating an XML grammar (e.g. DTD) with a collection of XPath expressions. To prevent improper disclosure of confidential information, user queries posed on these views need to be *rewritten* into equivalent queries on the underlying documents. This rewriting enables us to avoid the overhead of view materialization and maintenance. A major concern here is that query rewriting for recursive XML views is still an *open* problem. To overcome this problem, some works have been proposed to translate XPath queries into non-standard ones, called Regular XPath queries. However, query rewriting under Regular XPath can be of exponential size as it relies on automaton model. Most importantly, Regular XPath remains a theoretical achievement. Indeed, it is not commonly used in practice as translation and evaluation tools are not available. In this paper, we show that query rewriting is always possible for recursive XML views using only the expressive power of the standard XPath. We investigate the extension of the downward class of XPath, composed only by *child* and *descendant* axes, with some axes and operators and we propose a general approach to rewrite queries under recursive XML views. Unlike Regular XPath-based works, we provide a rewriting algorithm which processes the query only over the annotated DTD grammar and which can run in linear time in the size of the query. An experimental evaluation demonstrates that our algorithm is efficient and scales well.

Key-words: Queries Rewriting, XML Access control, XML views, XPath.

* Univ. Nancy 2, UMR 7503 & INRIA Nancy Grand Est (Houari.Mahfoud@inria.fr).
† Univ. Nancy 2, UMR 7503 & INRIA Nancy Grand Est (Abdessamad.Imine@inria.fr).

RESEARCH CENTRE NANCY – GRAND EST

615 rue du Jardin Botanique
CS20101
54603 Villers-lès-Nancy Cedex

Interrogation Sécurisée des Vues XML Récursives: Une Technique basée sur le Standard XPath

Résumé : La plupart des travaux existant autour du contrôle d'accès des documents XML se basent sur la définition d'une vue pour chaque utilisateur qui représente les parties des données dont il est autorisé à lire et/ou modifier. Cette vue est le résultat de l'annotation de la grammaire associée au document XML (par exemple une DTD) par différentes conditions d'accès exprimées sous forme d'expressions XPath. Pour empêcher l'accès à des données confidentielles – cachées par la vue –, chaque requête posée par l'utilisateur sur la vue doit être réécrite pour qu'elle soit évaluée en toute sécurité sur le document original. Cette réécriture permet d'éviter le coût de la matérialisation et de la maintenance de la vue. Cependant, la réécriture des requêtes XPath dans le cas des vues XML récursives reste un probleme ouvert. Pour pallier à ce problème, certains travaux ont proposé de travailler avec un langage de requêtes non-standard, appelé "Regular XPath". Néanmoins, le langage "Regular XPath" reste au stade théorique car aucun outil d'évaluation n'est disponible en pratique. Une implémentation de ce langage est basée sur les automates, ce qui peut engendrer une complexité de réécriture exponentielle.

Dans ce papier, nous montrons que la réécriture des requêtes XPath dans le cas des vues XML récursives est possible sans passer par des transformations vers d'autres langages (tel que "Regular XPath"), et peut être faite en temps linéaire. Nous étudions l'extension du fragment XPath, appelé en anglais *downward class* (composé seulement par les axes *child* et *descendant*), par certains axes et opérateurs XPath. En nous basant sur cette extension, nous proposons un modèle général pour réécrire des requêtes XPath pour des vues XML arbitraires, récursives ou non. Une phase d'expérimentation montre bien l'efficacité ainsi que le passage à l'échelle de notre algorithme de réécriture.

Mots-clés : Réécriture des requêtes, Contrôle d'accès pour documents XML, Vues XML, XPath.

Contents

1 Introduction

XML has become the standard of representation and exchange of data across the web. With this emergence, a challenge is raised with regards to the security of XML documents whose content is available to one or more users based on their access privileges. First access control models for securing XML have been proposed in [7, 10, 16]. However, these models suffer from various limitations. They can cause leakage of sensitive information [7], focus on the annotation of the entire XML data to deal with the static analysis limitations [10], or based on costly schemes for rewriting user queries [16].

To avoid these problems, the notion of XML security views was studied by Fan et al. [4]. We briefly review the main principle of the XML security view-based approaches. We start by the fragment of XPath, called *downward* class, where the axes are limited to *child* and *descendant* axes. We use this fragment since it is commonly used in practice. A conform XML document T w.r.t a DTD D (i.e. T is an instance of D), can be queried simultaneously by different users. For each class of users a security view is defined by annotating D with some access conditions to specify the (in)accessible element types of the DTD. The annotated version of D is later sanitized by removing the inaccessible element types which results in a DTD view D_v. Then the security view is defined as $V{=}(D_v, \sigma)$ where D_v is given to the users, which describes accessible data they are able to see, and σ is a function used to extract for each XML document T conforms to D, its view T_v representing only authorized data of the users. Each query over the view T_v is translated into an equivalent one in order to be evaluated over the original data T.

Problem Statement. The problem of XPath queries rewriting studied in this paper is defined as follows:

> Given a DTD D, an XML security view $V{=}(D_v, \sigma)$, and an XPath query Q over D_v. The rewriting problem consists in defining a rewriting function $\mathcal{R}$ that computes another XPath query $\mathcal{R}(Q)$ over the original document D such that: for any instance T of D and its view T_v computed w.r.t V, the evaluation of Q on T_v yields the same result as the evaluation of $\mathcal{R}(Q)$ on T.

Most of the security view-based approaches of XPath queries rewriting deal only with non-recursive DTDs [4, 9, 11]. A DTD is recursive iff at least one of its elements is defined (directly or indirectly) in terms of itself. Note that recursive DTDs often arise when specifying medicals data and the problem of query rewriting is more intriguing in this case. A security view is recursive, if its view D_v is recursive.

For each pair of element types A and B in the DTD, the σ function is an XPath expression denoting the set of paths to reach an element B from an element A in the DTD view (where some element types are hidden between A and B and generated by σ). However, in the case of recursive DTDs, the σ function is not computable since there may be an infinite set of paths from A to B and the notion of security view (as defined above) cannot be used in queries rewriting. That is why some authors [5, 8] resort to the use of Regular XPath to avoid this problem. However, Regular XPath remains of theoretical use since no evaluation tools have been provided for practical use of this langage.

To the best of our knowledge, no practical approach exists for answering queries under recursive XML security views. Accordingly, the XPath query rewriting remains an open issue.

Contribution. Our main contribution is making possible the query rewriting for recursive XML security views using only the expressive power of the standard XPath. We show that extending the downward class of XPath queries with some axes and operators is sufficient to deal with the query rewriting under recursion (without the need of the *Kleene star* or the translation from XPath to Regular XPath).

Intuitively, for a query Q based on the downward class of XPath, our rewriting solution consists in computing another query $Q' = \mathcal{R}(Q)$ using an extended fragment of XPath in such a way for any instance T of D and its view T_v (computed w.r.t D_v) the evaluation of Q on T_v gives the same result than evaluating Q' on T.

We provide a linear rewriting algorithm for arbitrary views (recursive or not) which, unlike Regular XPath-based works (relying on Mixed Finite Automata [5]), consists only in processing the query over the annotated DTD to produce the equivalent query on any valid instance of the original DTD. We validate our solution with a performance evaluation which shows that our rewriting algorithm is efficient and scales well. Lastly, we show how our proposed solution can be extended to deal with a large fragment of XPath (including upward-axes) and to go beyond some limitations of existing access control specification languages.

Related Work. We briefly discuss two approaches of access control policy enforcement for XML documents with or without XML grammar.

In [7] authors propose a formal model to specify access control for XML documents independently of the DTD. The policy rules definition is based on XPath and each query is rewritten by adding a predicate access (which represents all accessible data) to that query. We use the same predicate access principle in our rewriting approach. However, inference of sensitive information can be detected since only the last subquery is controlled among all subqueries parsed by the query. To overcome this problem, we improve the method given in [7] by attaching a predicate access to each entity (element/attribute) parsed by the query.

Vercammen [16] proposes a new method based on the intersection and union of XPath queries to avoid the problem of information leakage. The policy rules are translated to a single query which stands for all accessible data; this query is incorporated by intersection with each query requested over the user XML document view. However, this approach yields the same performance than the materialization of this view.

Other access control approaches are based on the notion of security views and the query rewriting principle. Fan et al. [4] propose the notion of security view by the annotation of a regular non-recursive DTD. The use of only downward class of XPath queries allowed them to achieve more precise query rewriting, i.e. computing all possible paths connecting each two adjacent elements in the query, which provides practically performance gains for the query evaluation. A view derivation algorithm is proposed to compute the DTD view, w.r.t. the access conditions, and an optimization step is also done over the rewritten query.

However, to keep the DTD view regular, an inaccessible element may be replaced with anonymous element *dummy* which can be source of security breaches. In [9, 11], authors refine the Fan's model by eliminating *dummies*, extending the class of XPath queries with *upward*-axes and with a novel notion of security views. Different types of policies are also discussed. These works can deal only with non-recursive DTDs. They are inapplicable to recursive DTDs because the description of paths connecting two element types may be infinite.

Unlike the XPath query rewriting over non-recursive DTDs, the problem posed by the recursion has not received a more attention. Authors of [5] extend the principle proposed in [4] with a translation of XPath queries to Regular XPath and propose a first algorithm for evaluating Regular XPath over XML data. In [8], a more generalized rewriting approach has been studied by dealing with restrictions on the class of queries and DTD types. The defined accessibility function is based on the *Kleene star*. It should be noted that the *Kleene star* cannot be expressed in the standard XPath.

Although the query formulation and rewriting on Regular XPath is more expressive than the standard XPath, we cannot find any practical system for both proposed approaches[1]. Consequently, the need of a rewriting system of XPath queries over recursive XML security views remains an open issue.

Plan of the paper. The rest of the paper is organized as follows. Section 2 presents formally the query rewriting problem for recursive views, and sketches our solution to deal with this problem. In Section 3, we give the ingredients of our access control specification. Our rewriting approach is detailed in Section 4. Section 5 presents how our approach can be extended to consider a large fragment of XPath and used to overcome some limitations of existing access control approaches. An implementation issue is presented in Section 6. Finally, we conclude this paper in Section 7.

2 Formal Problem Statement

In this section we present the query rewriting problem for recursive views, and sketch our solution to deal with this problem.

2.1 Preliminaries

We briefly review some notions of Document Type Definitions (DTDs) and the class of XPath Queries most used in practice.

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

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

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,

[1] According to [12] the SMOQE system proposed in [6] has been removed because of conduction of future researches.

and the Kleene closure respectively. We refer to $A \rightarrow P(A)$ as the *production* of A. For each element type B occurring in *P(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. If an element type A is defined in term of an element type B directly (B is subelement type of A) or indirectly, then A is an *ancestor type* of B and B is a *descendant type* of A. The DTD is said *recursive* if some element type A is defined in terms of itself directly or indirectly.

We use the graph representation to depict our DTDs where dashed edges represent disjunction.

XML Documents. We model an XML document with an unranked ordered finite node-labeled tree, also called *XML Tree*. Let Σ be a finite set of node labels, an XML tree T over Σ is a structure defined as [15]: $T = (N, R_{\downarrow}, R_{\rightarrow}, L)$ where N is the set of nodes, $R_{\downarrow} \subseteq N \times N$ is the parent-child relation, $R_{\rightarrow} \subseteq N \times N$ is a successor relation on (ordered) siblings, and $L : N \rightarrow \Sigma$ assigns a label to each node.

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 element type A, called A *element*, or with *str*, called *text node*; (iii) for each node n of type A and with k ordered children $n_1, ..., n_k$, the word $L(n_1), ..., L(n_k)$ belongs to the regular language defined by *P(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. In the DTD instances depicted in our figures, we use X^i to distinguish between elements of the same type X.

XPath Queries. We introduce the *downward* class of XPath queries referred to as $\mathcal{X}$ and defined as follows:

```
path   :=  axis::label | path'['qual']'
         | path'/'path | path'∪'path
qual   :=  path | path = c
         | qual and qual | qual or qual
         | not qual | '('qual')'
axis   :=  ↓ | ↓+
```

where *label* refers to element type in *Ele* or `*` (that matches all labels), $\cup$ stands for union, c denotes *text constant*, *axis* is the XPath axis relation, and $\downarrow, \downarrow^+$ denote *child* and *descendant* axis respectively, and finally *qual* is called an XPath *qualifier* (*predicate* or *filter*) which can be a text content comparison, an XPath query, or a boolean expression (using boolean operators such as: *and*, *or*, *not*).

Let n be a node in an XML tree T. The evaluation of an XPath query p at node n, called *context node* n, results in a set of nodes which are reachable from n with p, denoted by $n[\![p]\!]$. A qualifier q is said valid at node context n, denoted by $n \vDash q$, iff one of the following conditions holds: (i) q is given by $p{=}c$ and there is at least one element reachable from n with p which has c as text content; (ii) q is an XPath query and $n[\![q]\!]$ is nonempty; (iii) q is a boolean expression (e.g. *not(p)*) and it is evaluated to true at n.

2.2 XML Access Control Model

We define below some concepts of security specifications and XML security views as initially presented in [4, 9].

Security Specifications. Given an XML document T conforms to a DTD D. For each class of users some access privileges may be defined to restrict access to sensitive information on T. Thus, an access-control specification language is defined to specify what elements in T the users are granted, denied, or conditionally granted access to. An *access specification* in this language is defined as follows:

Definition 2.1 *An access specification* S *is a pair* (D,`ann`) *consisting of a DTD* D *and a partial mapping* `ann` *such that, for each production* $A \rightarrow P(A)$ *and each element type* B *in* P(A)*,* `ann(`A`,`B`)` *, if explicitly defined, is an annotation of the form:*

$$\texttt{ann(}A\texttt{,}B\texttt{)} \texttt{ := } Y \texttt{ | } N \texttt{ | } \texttt{[}Q\texttt{]}$$

where `[`Q`]` *is a qualifier in our XPath fragment* $\mathcal{X}$*. A special case is the root of* D *for which we define* `ann(`$root$`)`$=Y$ *by default.* □

The specification values Y, N, and $[Q]$ indicate that the B children of A elements in an XML document conforms to the DTD D are *accessible*, *inaccessible*, or *conditionally accessible* respectively. If `ann(`A`,`B`)` is not explicitly defined, then B inherits the accessibility of A. On the other hand, if `ann(`A`,`B`)` is explicitly defined then B may *override* the accessibility inherited from A. A text node is accessible only if its parent element is accessible. For an element node n of type B with parent node of type A, we say that n is concerned by an annotation if `ann(`A`,`B`)`$=$*value* exists, moreover, this annotation is valid at n if *value*$=Y$, or *value*$=[Q]$ and $n \vDash Q$.

Security Views. To enforce an access specification, a *security view* is defined to compute for each document T conforms to a DTD D: (i) an instance view T_v containing only accessible data; and, (ii) a DTD view D_v which describes schema of all accessible data. Both documents T_v and D_v are seen only by authorized users.

More formally, let $S=(D, \texttt{ann})$ be an access specification. A security view for S is defined as a pair $V=(D_v, \sigma)$ where: (i) D_v is a view of D computed by eliminating inaccessible element types[2] from D, according to annotations given in S; (ii) σ is a function used to extract accessible data in such a way that for each pair of types A and B where B occurs in $P(A)$ in D_v, the $\sigma(A, B)$ is an XPath query (described in fragment $\mathcal{X}$) defining paths to reach element nodes B from an element node A in the original document T conforms to D. It should be noted that function σ is hidden from the users. A security view $V=(D_v, \sigma)$ is said *recursive* if its D_v is recursive.

Example 2.1 Consider the DTD D depicted in Figure 1(a) where the labels of the edges represent the following access specification: `ann(`$root$`,`A`)`$=[q]$,

[2] An element type B is inaccessible if for each parent type A, either (i) the annotation `ann(`A`,`B`)`$=N$ exists or (ii) A is an inaccessible element type.

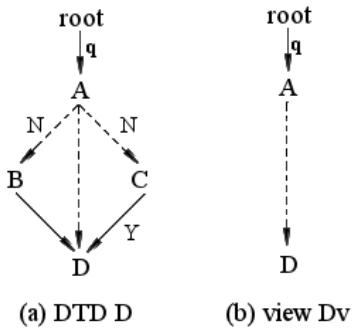


Figure 1: Simple non-recursive DTD.

$\texttt{ann}(A,B)=N$, $\texttt{ann}(A,C)=N$, and $\texttt{ann}(C,D)=Y$. We define the security view $V=(D_v,\sigma)$ as follows:[3]

D_v : $root \rightarrow A$, $A \rightarrow D|\epsilon$, $D \rightarrow \epsilon$.
root $\rightarrow$ **A**: we have $\sigma(root,A)=A[q]$.
A $\rightarrow$ **D**$|\epsilon$: we get $\sigma(A,D)=(C \cup \epsilon)/D$.

Figure 1(b) depicts the resulting view D_v. □

2.3 DTD Recursion Problem

Most existing approaches [4, 9, 11] for securing access to XML documents are based on the notion of security view. Given a security view $V=(D_v, \sigma)$, the query rewriting principle is applied to translate each XPath query p over D_v to another one p_t over the original DTD D, such that for any instance T of D (T_v of D_v resp.), p_t over T yields the same answer as p over D_v (i.e. $p_t(T) = p(T_v)$)[4]. Thanks to query rewriting we do not need to materialize view T_v and its major problem namely the view maintenance. However, only non-recursive DTDs are considered. The security view as specified before cannot be applied in the case of recursive DTDs. To illustrate this problem we give the following example:

Example 2.2 For the query $\downarrow^+::H$ over the DTD given in Figure 2(a), we should enumerate all the paths from the *root* which give an accessible element H (as done in [4]): $/root/A[q]/(B \cup D/E/G)/H$. However, the task is complicated in the case of recursion. With the same query over the DTD in Figure 2(b), the function σ used to extract accessible data, cannot be defined, e.g. $\sigma(D, E)$ can be E, $F/G/D/E$, or $F/G/D/F/G/D/E$ etc. Then $\sigma(D, E)$ leads to infinitely many paths and cannot be defined in $\mathcal{X}$. Moreover, the rewriting of the query $\downarrow^+::H$ is equivalent to the following regular expression:

$/root/A[q]/B/H \cup /root/A[q]/q_1/(q_1)^*/H \cup$
$/root/A[q]/(q_2)^*/D/E/G/(D/G)^*/H$

where: q_1 = $D/G \cup D/E/G$, and q_2 = $D/G \cup D/E/G \cup D/F/G$. □

[3] Note that ϵ denotes the empty path.
[4] We denote by $p(T)$ the result of evaluating query p over document T.

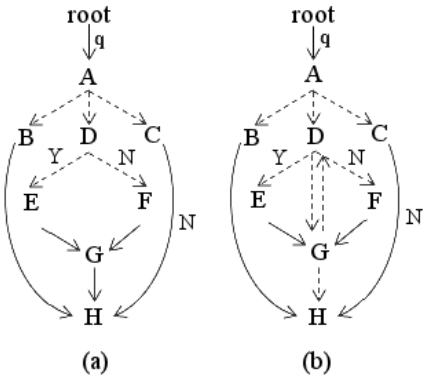


Figure 2: DTD Recursion Problems.

Since the *Kleene Star* (denoted by $*$) is not part of the standard XPath and cannot be expressed as outlined in [14,15], the rewriting of XPath queries is not always possible. We refer to this problem as the *non-closure* of XPath fragment under the query rewriting. The *closure* property is defined as follows:

Definition 2.2 *A class C of XPath queries is* closed *under query rewriting if there is a function* `Rewrite`*: $C \rightarrow C$ that, for any security view V=(D_v, σ) and any query Q in C over D_v, computes Q_t=*`Rewrite`*(Q) in C such that for any instance T conforms to the original DTD D and its view T_v w.r.t V, we have $Q(T_v) = Q_t(T)$.* □

It has been shown in [5] that the *downward* class (i.e. fragment $\mathcal{X}$) of XPath queries is not *closed* under query rewriting.

Theorem 1 *For recursive XML security views, fragment $\mathcal{X}$ is not closed under query rewriting [5].* □

2.4 Our Proposed Solution

We show that the expressive power of the standard XPath [2] is sufficient to overcome the query rewriting problem over recursive views. We propose to redefine function `Rewrite` given in Definition 2.2 into `Rewrite`: $C_1 \rightarrow C_2$ where C_2 is the fragment C_1 extended by adding some axes and operators. Using this extension, for any access specification S=$(D, \texttt{ann})$ and any query Q in C_1 over D_v (view of D computed w.r.t S), we can compute $Q_t =$ `Rewrite`(Q) in C_2 such that for any instance T of D and its view T_v w.r.t S, we have $Q(T_v) = Q_t(T)$.

The input fragment C_1 in our case is fragment $\mathcal{X}$ (namely the downward class) defined in Section 2.1, which is used only to formulate user queries and to define access specifications, while C_2 is an extended fragment of $\mathcal{X}$ defined as follows:

```
path    :=  axis::label | path'['qual']' | path'['n']'
          | path'/'path | path'∪'path
qual    :=  path | path = c | path = ε::label
          | qual and qual | qual or qual
          | not qual | '('qual')'
axis    :=  ε | ↓ | ↑ | ↓+ | ↑+ | ↑*
```

we enrich $\mathcal{X}$ by the *self*-axis (ε), the upward-axes *parent* ($\uparrow$), *ancestor* ($\uparrow^+$), and *ancestor-or-self* ($\uparrow^*$), the *position* and the *node comparison* predicates. 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 the XML document with an ordered tree, the query $\downarrow::*[1]$ at an element node n returns its first child element, while $\uparrow^+::*[B =' topo'][1]$ returns its first ancestor element which has an child element B with text content *'topo'*. The *node comparison* predicate `[`$target_1$`=`$target_2$`]` is true only if the evaluation of the right and left sides result in exactly the same single node. For example the predicate in the following query $\downarrow^+::A \downarrow::/B[\uparrow::* = \uparrow^+::*[1]]$ is valid for any B element child of an A element.

We summarize the augmented fragment of $\mathcal{X}$ by the following subsets $\mathcal{X}^{\Uparrow}$ ($\mathcal{X}$ with *self* and *upward*-axes), $\mathcal{X}^{\Uparrow}_{[n]}$ ($\mathcal{X}^{\Uparrow}$ with *position* predicate), and the final fragment $\mathcal{X}^{\Uparrow}_{[n,=]}$ ($\mathcal{X}^{\Uparrow}_{[n]}$ with *node comparison* predicate).

It should be noted that for a given query Q, a rewriting technique must ensure the following conditions: (i) each subquery[5] of Q refers only to accessible element nodes; and (ii) each relationship defined between two subqueries of Q is respected. For instance, the query $\downarrow^+::D/\downarrow::*$ over the access specification depicted in Figure 2(b) must returns only accessible element nodes of type G or E which have an accessible D element as parent.

We define the accessibility problem as: "*When does an element node of a given type is accessible*?". It is clear that the function σ cannot solve this problem because of infinitely many possible paths involved by recursive views (see Example 2.2). We show in the next that the accessibility of a given element node w.r.t a given recursive view cannot be defined in the fragment $\mathcal{X}$ (even in $\mathcal{X}^{\Uparrow}$). We investigate the use of the augmented fragment $\mathcal{X}^{\Uparrow}_{[n]}$ to solve the accessibility problem in particular, and the fragment $\mathcal{X}^{\Uparrow}_{[n,=]}$ as a solution to avoid the non-closure of XPath fragment $\mathcal{X}$ in general.

2.5 Notations

Given an access specification $S{=}(D, \texttt{ann})$, and a document T conforms to D. We define two predicates $\mathcal{A}_1^{acc}$ and $\mathcal{A}_2^{acc}$ as follows[6]:

$$\mathcal{A}_1^{acc} := \uparrow^*::*[\varepsilon::root \bigvee_{ann(A',A)\in ann} \varepsilon::A/\uparrow::A'][1] \; [\varepsilon::root \bigvee_{(ann(A',A)=Y|[Q])\in ann} \varepsilon::A.\sigma(A',A)/\uparrow::A']$$

$$\mathcal{A}_2^{acc} := \bigwedge_{(ann(A',A)=[Q])\in ann} \texttt{not}\ (\uparrow^+::A[\texttt{not}\ (Q)]/\uparrow::A')$$

[5] For example, the query $\downarrow^+::B[\downarrow::C]$ contains two subqueries $\downarrow^+::B$ and $\downarrow::C$ with a parent/child relation defined between C and B, and an ancestor/descendant relation defined between B and the node context at which the query is posed.

[6] Note that $A.\sigma(A', A)$ gives $A[Q]$ if $\texttt{ann}(A',A){=}[Q]$ and A otherwise.

Where $\bigwedge$ and $\bigvee$ denote *conjunction* and *disjunction* respectively. The predicate $\mathcal{A}_1^{acc}$ has the form $\uparrow^*\texttt{::*[}qual_1\texttt{][1][}qual_2\texttt{]}$. Applying $\uparrow^*\texttt{::*[}qual_1\texttt{]}$ on an element node n of T returns an ordered set $\mathcal{S}$ of element nodes (n and/or some of its ancestor elements) such that for each one an annotation is defined. Thus, with $\mathcal{S}[qual_2]$ ($n \vDash \mathcal{A}_1^{acc}$) we ensure that the first element node in $\mathcal{S}$ is concerned by a valid annotation. With the second predicate, we use $n \vDash \mathcal{A}_2^{acc}$ to ensure that all qualifiers defined over ancestor elements of n are valid (we discuss this restriction in Section 5.2). These predicates are "powerful tools" to solve the accessibility problem as we will see in the next section.

Since the σ function is not computable in case of recursion, the parent/child relation defined between two element types in the query (e.g. query $\downarrow^+\texttt{::}A/\downarrow\texttt{::}B$ defines parent/child relation between A and B) cannot be rewritten in $\mathcal{X}$. Accordingly, we define the two predicates $\mathcal{A}^+$ and $\mathcal{A}^B$ to rewrite parent/child relation:

$$\mathcal{A}^+ \texttt{ := } \uparrow^+\texttt{::*[}\mathcal{A}_1^{acc}\texttt{]}$$

$$\mathcal{A}^B \texttt{ := } \mathcal{A}^+\texttt{[1]/}\varepsilon\texttt{::}B$$

For an element node n in T, $n[\![\mathcal{A}^+]\!]$ returns the set of all accessible ancestor elements of n. The element node n has an accessible B element as parent if and only if $n \vDash \mathcal{A}^B$.

We use these four predicates throughout the paper to formalize our solution.

3 Access Control with Recursive DTDs

Our access control framework is presented in Figure 3. For each class of users, the administrator defines an access specification $S{=}(D, \texttt{ann})$ over the DTD D. The DTD view D_v is derived first and given to the users to formulate their queries. For each instance T of D, we compute a virtual[7] view T_v of T to show only accessible data. Each $\mathcal{X}$ query Q over T_v is efficiently rewritten, using the security view V (defined below in Section 3.1), to an equivalent $\mathcal{X}^{\Uparrow}_{[n,=]}$ query Q_t over T, in order to return only accessible data.

3.1 Recursive Security Views

We redefine the security view over an access specification $S{=}(D, \texttt{ann})$ to be $V{=}(D_v, \texttt{ann})$, where D_v is the view of D, computed by algorithm `DeriveView` illustrated in Figure 4, and used by the users to formulate their queries.

We use first a DTD parser [8] to explore the DTD D into an expressive indexed structure in such a way, for each element type A in D, the set of its children types and descendant types are returned, also the content model *P(A)* is represented as a tree where all sub-expressions composing *P(A)* are detected as we explain in the next.

We define the recursive function `Exp`(*A,access*) that, according to a given access specification $S{=}(D,\texttt{ann})$, extracts the content model for the element type A and for all its descendant types in D. For each element type A parsed by

[7] The views of T are never materialized.

[8] Available at: `http://www.rpbourret.com/dtdparser/index.htm`.

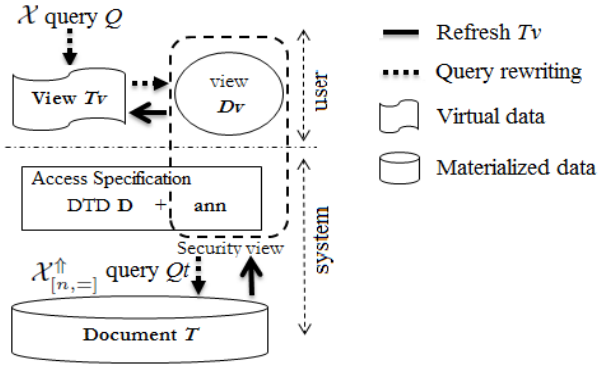


Figure 3: XML Access Control Framework.

`Exp`, we eliminate all inaccessible subelement types B_i of A (`if ann`(A, B_i)=N exists) to compute the new content model $P_v(A)$. The value *access* represents the inherited accessibility. For a given content model "$A \longrightarrow B_1, (B_2|B_3)$", we refer by G_1 op_A G_2 to the sub-expressions B_1 and $B_2|B_3$ respectively, separated by $op_A =$ ",". The list `Parsed` is used (i) to store the extracted content model of each element type in D; and (ii) to avoid more than one parsing of the same element type. By invoking `Exp`(*root,true*) in algorithm `DeriveView`, the content models of all element types of D are computed. The value of `Parsed`(A,*true*)=ϕ indicates that the element type A is not accessible, while the value `Parsed`(A,*true*)=ϵ indicates that the content model of A is an empty word. The output of algorithm `DeriveView` is a DTD view $D_v = (Ele_v, P_v, root)$ where Ele_v is computed by eliminating inaccessible element types from D, and P_v returns the content model of each (accessible) element type in D_v.

The complexity of our DTD view derivation algorithm, `DeriveView`, is given by the following theorem:

Theorem 2 *Let S=(D, `ann`) be an access specification, and P' be the largest production in D, then the view D_v of D can be derived w.r.t S in at most $O(|D| * |P'|)$ time.* □

Proof. For an element type A in D, we denote by $|P(A)|$ the number of all subelement types and operators (“,” or “|”) defining *P(A)*. The procedure `Exp` of Figure 4 works over the hierarchical, parse-tree representation of the regular expression *P(A)*. This tree is given using the DTD parser cited above, where its intermediate nodes represent operators and the leaves are the subelement types of A. Each operator links two or more element types/sub-expressions. To compute the new content model of A, we parse all the element types B_i (i.e. the leaves) of the *P(A)* tree to eliminate each inaccessible B_i (i.e. `ann`(A, B_i)=N exists). Next, each node operator with no children nodes is eliminated. Finally, the new resulting tree is translated into a regular expression which represents the content model $P_v(A)$. Thus, these steps are done by parsing all the nodes of the *P(A)* tree in $O(|P(A)|)$ time. If we consider that P' is the largest production in D then $O(|P(A)|)$ is bounded by $O(|P'|)$ and the content models of all element types of D are computed in at most $O(|D| * |P'|)$ time. □

Algorithm: *DeriveView*

input : an access specification S=(D, ann) with D=($Ele, P, root$).
output: a DTD view D_v.

```
1  {Ele_v,P_v} := {φ,φ};
2  Exp(root,true);
3  foreach element type A ∈ D do
4      if Parsed(A,true)≠ φ then
5          Ele_v := Ele_v ∪ A;
6          P_v(A) := Parsed(A,true);
7  D_v := (Ele_v, P_v, root);
8  return D_v;
```

Procedure: *Exp*(A, *access*)
Input: an element type A, inherited accessibility *access*.
Output: content model of A.

```
1  if Parsed(A,access)≠ null then
2      return Parsed(A,access);
3  exp := φ;
4  case P(A) is str or ϵ
5      if access then
6          exp := str (ϵ resp);
7  case P(A) is G_1 op_A ... op_A G_n
       // op_A is "|" or ","
8      foreach subexpression G_i do
9          if G_i = B then // case of single element type
10             if ann(A,B) ∉ ann then
11                 if access then
12                     exp := exp op_A B; Exp(B,true);
13                 else if Exp(B,false)≠ φ then
14                     exp := exp op_A Exp(B,false);
15             else if ann(A,B) = Y then
16                 exp := exp op_A B; Exp(B,true);
17             else if ann(A,B) = N and Exp(B,false)≠ φ then
18                 exp := exp op_A Exp(B,false);
19             else /* ann(A,B) = [Q]                                  */
20                 exp := exp op_A (B|ϵ);
21                 Exp(B,true);
22         else if G_i = B* then
23             similar to the previous case except that in exp, B is replaced with
               B* (also for Exp(B,true) and Exp(B,false));
24         else // G_i is composition of element types/subexpressions
25             define A' in D as temporary element type;
26             define content model P(A') := G_i;
27             if Exp(A',access)≠ φ then
28                 exp := exp op_A Exp(A',access);
29             delete A' from D and the Parsed(A',access) entry;
30 if (exp = φ and access) then exp := ϵ;
31 Parsed(A,access):=exp;
32 return exp;
```

Figure 4: DTD View Derivation Algorithm.

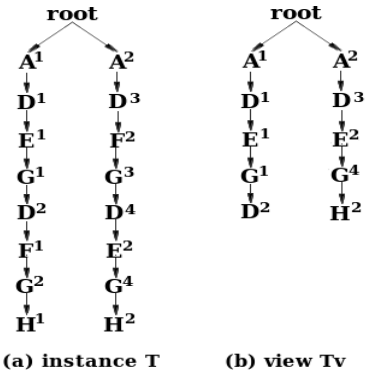


Figure 5: Example of Instance view.

3.2 Accessibility

Now we define the element node accessibility based on the use of recursive views:

Definition 3.1 *Given a security view* V=(D_v, ann) *and a document* T *conforms to the original DTD* D*, then an element node* n *on* T *of type* B *with parent node of type* A *is accessible (shown in the view* T_v *of* T*), if and only if the following conditions hold:*

i) The element node n *is concerned by a valid annotation, or* ann(A, B) *does not exist and there is an annotation defined over ancestor element* n' *of* n *where:* n' *is the first ancestor element of* n *concerned by an annotation, and this annotation is valid at* n'*.*

ii) For each ancestor element n' *of* n *concerned by an annotation with value* $[Q']$*,* $n' \vDash Q'$ *must be verified.* □

Example 3.1 Consider the DTD depicted in Figure 2(b) where annotation q is $\downarrow::D$. For an element node n of type H within the instantiation of this DTD, if its parent element is of type C then n is not accessible. Otherwise, the first ancestor element of n which is concerned by an annotation can be either of type F (i.e. ann(D,F)=N), of type E (i.e. ann(D,E)=Y), or of type A (i.e. ann(*root*,A)=$[q]$). This means that n may be accessible if its first ancestor element is of type E or A, and it has no ancestor element n' of type A with $n' \nvDash q$.

Note that the element node accessibility over recursive XML views cannot be defined in $\mathcal{X}^{\Uparrow}$. We consider the access specification S=(D, ann) composed by the DTD of Figure 2(b) and the annotations depicted in the edges. Figure 5 represents (a) an instance T of D and (b) its view T_v computed according to S. The query $\downarrow^+::H$ over T must be rewritten to return only the node H^2, which is accessible w.r.t S. However $\downarrow^+::H[\uparrow^+::E$ *or* $\uparrow^+::A[q]]$ returns both the nodes H^1 and H^2, and $\downarrow^+::H[(\uparrow^+::E$ *or* $\uparrow^+::A[q])$ *and not* $(\uparrow^+::F)]$ rejects the accessible node H^2 shown in T_v. □

We use the predicates $\mathcal{A}_1^{acc}$ and $\mathcal{A}_2^{acc}$ defined in Section 2.5 to satisfy the accessibility conditions ***(i)*** and ***(ii)*** respectively of Definition 3.1.

Definition 3.2 *For any security view* V$=(D_v, \texttt{ann})$ *and any instance* T *conforms to the original DTD* D*, we define the accessibility predicate* $\mathcal{A}^{acc}$ *which refers to an* $\mathcal{X}^{\Uparrow}_{[n]}$ *qualifier such that, an element node* n *on* T *is accessible iff* $n \vDash \mathcal{A}^{acc}$*, with* $\mathcal{A}^{acc} := \mathcal{A}^{acc}_1 \wedge \mathcal{A}^{acc}_2$*.* □

For an element type B in a DTD D, $\downarrow^+::B[\mathcal{A}^{acc}]$ stands for all accessible B elements in an instance of D.

Example 3.2 Consider the access specification depicted in Figure 2(b), the predicates $\mathcal{A}^{acc}_1$ and $\mathcal{A}^{acc}_2$ are defined as follows:

$\mathcal{A}^{acc}_1$:= $\uparrow^*::*[\varepsilon::root \vee \varepsilon::A/\uparrow::root \vee \varepsilon::E/\uparrow::D \vee \varepsilon::F/\uparrow::D \vee \varepsilon::H/\uparrow::C][1][\varepsilon::root \vee \varepsilon::A[q]/\uparrow::root \vee \varepsilon::E/\uparrow::D]$

$\mathcal{A}^{acc}_2$:= not $(\uparrow^+::A[\text{not } (q)]/\uparrow::root)$

Consider the element node H^1 of the XML document illustrated in Figure 5(a). Then, $H^1[\![\uparrow^*::*[\varepsilon::A/\uparrow::root \vee \varepsilon::E/\uparrow::D \vee \varepsilon::F/\uparrow::D \vee \varepsilon::H/\uparrow::C \vee \varepsilon::root]]\!]$ returns the set $\mathcal{S}=\{F^1,E^1,A^1,root\}$ of ordered element nodes (element node H^1 and/or some of its ancestor elements) where for each one an annotation exists (e.g. $\texttt{ann}(D,F)=N$ for element node F^1). Note that $\mathcal{S}[1]$ returns the ancestor element F^1 and the final predicate $\mathcal{A}^{acc}_1$ over the element node H^1 of Figure 5(a) is not satisfied (i.e. $H^1 \nvDash \mathcal{A}^{acc}_1$) since the first ancestor element concerned by an annotation in $\mathcal{S}$ is not accessible (F^1's annotation is not valid). The query $\downarrow^+::H[\mathcal{A}^{acc}]$ over the instance T of Figure 5(a) returns only the accessible element H^2 (shown in the view T_v of Figure 5(b)). □

Property 1. For any security view $V=(D_v, \texttt{ann})$, the accessibility predicate $\mathcal{A}^{acc}$ can be constructed in $O(|\texttt{ann}|)$ time. □

Proof. For any security view $V=(D_v, \texttt{ann})$, the construction of $\mathcal{A}^{acc}_1$ and $\mathcal{A}^{acc}_2$ depends only on the parsing of all annotations ***ann*** of V which is done in $O(|\texttt{ann}|)$ time. □

4 Query Rewriting over Recursive XML Views

In this section we describe our XPath-based query rewriting algorithm. Given an access specification $S=(D, \texttt{ann})$, the security view $V=(D_v, \texttt{ann})$ of S, an instance T conforms to D, and its virtual view T_v computed w.r.t V. Then, for any query p over T_v, the goal of query rewriting is to find a rewriting function that we define as:

$\mathcal{X} \longrightarrow \mathcal{X}^{\Uparrow}_{[n,=]}$
$p \longrightarrow \texttt{Rewrite}(p)$ such that $p(T_v) = \texttt{Rewrite}(p)(T)$

Our rewriting function `Rewrite` ensures that only accessible element nodes are referred to by the subqueries of p, which is ensured by the accessibility predicate of Definition 3.2. Moreover, the relationships defined between each two subqueries of p must be respected.

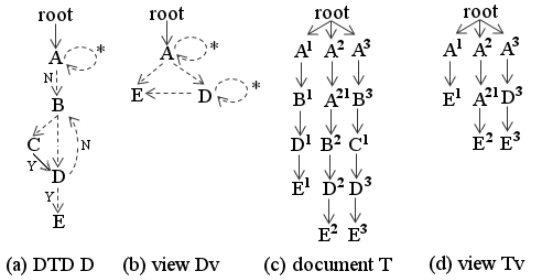


Figure 6: Query Rewriting Problems.

Notice that, for a given security view V=(D_v, ann), we compute first the predicates $\mathcal{A}^{acc}$ and $\mathcal{A}^{+}$ w.r.t V in $O(|\texttt{ann}|)$ time (Properties 1 and 2 resp.). Also, for each element type A in D_v we compute the lists of its children types and descendant types denoted by Reach($\downarrow$,A) and Reach($\downarrow^+$,A) respectively. Each list is computed in $O(|D_v|)$ time. The lists of all element types of D_v are computed in $O(|D_v|^2)$ time. This preprocessing step is done only one time after the security view V is defined and it provides performance gains during the query rewriting step.

In this section, we consider the DTD view D_v shown in Figure 6(b) which represents the derivation of the DTD D of Figure 6(a) with respect to the access specification depicted in the edges. Figure 6(c) represents a valid instance T of D and its derived view T_v is depicted in Figure 6(d).

4.1 Queries Without Predicates

For a DTD D=(*Ele,P,root*), we discuss the rewriting of queries without predicates with the form $axis_1::E_1/\ldots/axis_n::E_n$ where $E_i \in \{Ele,*\}$ and $axis_i$ is an XPath axis in fragment $\mathcal{X}$. Given the query $\downarrow^+::E_i/\downarrow^+::E_j$, it is clear that the rewritten query can be $\downarrow^+::E_i[\mathcal{A}^{acc}]/\downarrow^+::E_j[\mathcal{A}^{acc}]$ to return accessible E_j elements which have at least one accessible ancestor element of type E_i. However, it is not so simple in the case of *child*-axis.

Example 4.1 The query $\downarrow^+::A/\downarrow::E$ over the view T_v of Figure 6(d) returns E elements having an A element as parent. So the elements E^1 and E^2 are returned. Using the kleene closure, this query can be rewritten over the instance T of Figure 6(c) into: $\downarrow^+::A/\downarrow::B/\downarrow::D/(\downarrow::B/\downarrow::D)*/\downarrow::E$. However, using the standard XPath, a cycle in the DTD cannot be replaced by '$\downarrow^+$'. For instance the query $\downarrow^+::A[\mathcal{A}^{acc}]/\downarrow^+::E[\mathcal{A}^{acc}]$ returns the elements E^1, E^2, and E^3, while E^3 does not have a parent A. □

We use the predicate $\mathcal{A}^B$ (B can be any element type) defined in Section 2.5 to rewrite the parent/child relation. For instance, the query $\downarrow^+::A/\downarrow::E$ of the previous example can be rewritten into $\downarrow^+::E[\mathcal{A}^{acc}][\mathcal{A}^A]$ to return only accessible E elements (verified with $\mathcal{A}^{acc}$) which have an accessible parent of type A (verified with $\mathcal{A}^A$).

Example 4.2 Given the access specification of Figure 6(a), we define the predicate $\mathcal{A}^+$ as follows:

$\mathcal{A}^+$:= $\uparrow^+$::*[$\uparrow^*$::*[ε::E/$\uparrow$::D $\vee$ ε::B/$\uparrow$::D $\vee$ ε::D/$\uparrow$::C $\vee$ ε::B/$\uparrow$::A $\vee$ ε::$root$][1][ε::E/$\uparrow$::D $\vee$ ε::D/$\uparrow$::C $\vee$ ε::$root$]].

At element node E^3 of Figure 6(d), $\mathcal{A}^+$ returns the set of its ordered accessible ancestor elements $\{D^3,A^3,root\}$, that we denote $E^3[\![\mathcal{A}^+]\!]$. The predicate $\mathcal{A}^A$ (i.e. $\mathcal{A}^+[1]/\varepsilon::A$) does not hold at element node E^3 (i.e. $E^3 \not\models \mathcal{A}^A$) since the first accessible ancestor element of E^3 is not of type A (i.e. $\mathcal{A}^+[1]$ at E^3 returns its ancestor element D^3). However, $\mathcal{A}^A$ holds at element nodes E^1 and E^2. □

Property 2. For any security view V=(D_v, ann) and any element type B in D_v, the predicates $\mathcal{A}^+$ and $\mathcal{A}^B$ can be constructed in $O(|\texttt{ann}|)$ time. □

Proof. The same principle as the proof of Property 1. □

Finally, given a (recursive) security view V=(D_v, ann), we define the rewriting function Rewrite : $\mathcal{X} \times Ele \rightarrow \chi^{\Uparrow}_{[n]}$ that we use to rewrite an $\mathcal{X}$ query p=$p_1/\ldots/p_n$ (where each subquery p_i is given with $axis_i$::E_i) over a node context of type E in D_v, to an equivalent one Rewrite(p,E) defined in $\mathcal{X}^{\Uparrow}_{[n]}$ over the original DTD as:

Rewrite(p,E) := $\downarrow^+$::E_n[$\mathcal{A}^{acc}$][$prefix^{-1}(p_1/\ldots/p_n)$]

Where the qualifier $prefix^{-1}(p_1/\ldots/p_n)$ is recursively defined over the descending list of subqueries of p. For each subquery p_i, $prefix^{-1}(p_1/\ldots/p_{i-1})$ is already computed and used to compute $prefix^{-1}(p_1/\ldots/p_i)$ as follows:

- $axis_i = \downarrow$: $prefix^{-1}(p_1/.../p_i) := \mathcal{A}^{E_{i-1}}[prefix^{-1}(p_1/.../p_{i-1})]$
- $axis_i = \downarrow^+$: $prefix^{-1}(p_1/.../p_i) := \uparrow^+$::$E_{i-1}[\mathcal{A}^{acc}][prefix^{-1}(p_1/.../p_{i-1})]$

Recall that E is the type of context node at which the query is evaluated. As a special case we have: $prefix^{-1}(\downarrow::E_1)$=$\mathcal{A}^E$, and $prefix^{-1}(\downarrow^+::E_1)$=$\uparrow^+::E[\mathcal{A}^{acc}]$.

Example 4.3 Consider the query $\downarrow::A/\downarrow::E$ over the node context *root* of the view T_v of Figure 6(d). Using our algorithm Rewrite we obtain:

Rewrite($\downarrow::A/\downarrow::E$, $root$) = $\downarrow^+::E[\mathcal{A}^{acc}][\mathcal{A}^A[\mathcal{A}^{root}]]$=
$\downarrow^+::E[\mathcal{A}^{acc}][\mathcal{A}^+[1]/\varepsilon::A[\mathcal{A}^+[1]/\varepsilon::root]]$

$\mathcal{A}^+$ is given in Example 4.2. The evaluation of $\mathcal{A}^+$ returns $\{A^1,root\}$ at element node E^1, $\{A^{21},A^2,root\}$ at E^2, and $\{D^3,A^3,root\}$ at E^3. With $\mathcal{A}^A$ we ensure that for an element node E referred to by the query, its first accessible ancestor element is of type A which is verified for E^1, E^2 and not for E^3 in T_v of Figure 6(d). Moreover, with $\mathcal{A}^{root}$ we ensure that the A element returned by $\mathcal{A}^A$, which can be A^1 or A^{21} for element nodes E^1 and E^2 respectively, must have *root* as the first accessible ancestor which is verified only for A^1. Thus the query Rewrite($\downarrow::A/\downarrow::E$, $root$) at *root* of Figure 6(d) returns $\{E^1\}$. □

Algorithm: *Rewrite*

input : A query p, and an element type A for which query rewriting is carried.
output: a rewritten query p_t w.r.t A.

```
1  if p = p1∪...∪pn then
2      return ∪_{≤i≤n} Rewrite(pi, A);
3  reach := {A};
4  compute the descending list L of subqueries of p;
5  each subquery pi = axisi::Ei[fi];
6  pt := ϵ; filters := ϵ;
   // compute prefix^{-1}(p) to be pt
7  foreach pi in the order of L do
       // axes rewriting
8      case axisi =↓
9          if (filters = ϵ) then
10             pt := A^{fs(reach,ε)}[pt];
11         else
12             pt := A^{fs(reach,ε)}[filters][pt];
13     case axisi =↓+
14         if (filters = ϵ) then
15             pt := fs(reach, ↑+)[A^{acc}][pt];
16         else
17             pt := fs(reach, ↑+)[A^{acc}][filters][pt];
       // ∗-label elimination
18     if (Ei = ∗) then
19         reach := ∪_{E∈reach} Reach(axisi, E);
20     else if (∃ E ∈ reach s.t Ei ∈ Reach(axisi, E)) then
21         reach := {Ei};
22     else
23         reach := {};
24     if (reach={}) then return ϕ;
       // rewriting of predicate fi over reach elements
25     filters := RW_Pred(fi, reach);
26     if (filters = false) then // invalid predicate
27         return ϕ;
28     else if (filters = true) then // omitte fi from pi
29         filters := ϵ;
   // rewritten query pt of p w.r.t A
30 if (filters = ϵ) then
31     pt := fs(reach, ↓+)[A^{acc}][pt];
32 else
33     pt := fs(reach, ↓+)[A^{acc}][filters][pt];
34 return pt;
```

Figure 7: Algorithm for XPath Queries Rewriting.

The detail of Function `Rewrite` is given in Figure 7. After computing the descending list L of p's subqueries, we parse them to generate the $prefix^{-1}$ of the rewritten query as explained above. The node context A can be initialized to *root* of the DTD for rewriting p over the entire document. If p_i is $axis_i$::$*$, then the $*$-label is replaced by the set of children/descendant types of E_{i-1} ($axis_i$ is $\downarrow$ or $\downarrow^+$ resp.). Then, the rewriting of p_i over p_{i-1} can result in a set of element types denoted by `reach`. Moreover, if $E_i \neq *$, then it must exist at least one element type E in `reach` (result of the rewriting of p_{i-1}) where E_i is a child

type of E if $axis_i = \downarrow$ or one of its descendant types if $axis_i = \downarrow^+$ (lines 18-23). If the rewriting of p_i over p_{i-1} stands for an empty set `reach` then the query is rejected (line 24). By *fs* we refer to the function fusion, e.g. $fs(\{E_1, ..., E_n\}, \uparrow^+) = (\uparrow^+ :: E_1 \cup .. \cup \uparrow^+ :: E_n)$.

Function `RW_Pred`, called in the algorithm `Rewrite`, represents the predicates rewriting, which is the subject of the next section. As shown above, *upward*-axes and the *position* predicate are necessary for rewriting simple queries (without predicates). We prove below that fragment $\mathcal{X}^{\Uparrow}_{[n]}$ is not closed under query rewriting. Extending this fragment with the *node comparaison* operator (which results in the final fragment $\mathcal{X}^{\Uparrow}_{[n,=]}$) turns out sufficient to rewrite any query in $\mathcal{X}$.

Theorem 3 *For recursive XML security views, the XPath fragment $\mathcal{X}^{\Uparrow}_{[n]}$ is not closed under query rewriting.* □

Proof. (*by contradiction*) We consider query with the form E`[q]` which represents the rewriting limitation of the XPath fragment $\mathcal{X}^{\Uparrow}_{[n]}$. Assume that the query rewriting can be done in $\mathcal{X}^{\Uparrow}_{[n]}$. The query $\downarrow^+ :: A[\downarrow :: E]$ over the view instance T_v depicted in Figure 6(d) cannot be correctly rewritten, using the previous definition of algorithm `Rewrite`, into `Rewrite`$(\downarrow^+ :: A, root)$[`Rewrite`$(\downarrow :: E, A)$] equivalent to $\downarrow^+ :: A[\mathcal{A}^{acc}][\downarrow^+ :: E[\mathcal{A}^{acc}][\mathcal{A}^A]]$. Indeed, the resulting query returns $\{A^1, A^2, A^{21}\}$, but A^2 does not have an immediate child E. The limitation is due to the fact that predicate $[\downarrow :: E]$ must return all descendant elements E having as the first accessible ancestor the node context A at which the predicate is evaluated (i.e. the element node returned by $\downarrow :: E/\mathcal{A}^+[1]$ must be the same element node of type A at which the predicate is evaluated). This cannot be expressed in $\mathcal{X}^{\Uparrow}_{[n]}$ and can be done only by introducing the node set comparison (e.g. $\mathcal{X}^{\Uparrow}_{[n,=]}$) as we will present in the following. □

4.2 Predicates Rewriting

We explain the rewriting of predicates to complete the definition of our rewriting algorithm `Rewrite`. For a given query $axis_1 :: E_1[q_1]/.../axis_n :: E_n[q_n]$, we rewrite each predicate q_i over the element type E_i at which q_i is defined. Given a security view $V = (D_v, ann)$ and a subquery E`[q]` (we take a simple predicate $q = q_1/.../q_n$ where $q_i = axis_i :: E_i$ for more comprehension). We define the function `RW_Pred` : $\mathcal{X} \times$ `Ele` $\rightarrow \mathcal{X}^{\Uparrow}_{[n,=]}$ to rewrite the predicate q in $\mathcal{X}$ over element type E in D_v, to an equivalent one `RW_Pred`(q, E) in $\mathcal{X}^{\Uparrow}_{[n,=]}$, recursively defined over the descending list of sub-predicates of q as follows:

- $axis_i = \downarrow$: `RW_Pred`$(q_i/.../q_n, E_{i-1})$:=
  $\downarrow^+ :: E_i$[$\mathcal{A}^{acc}$][`RW_Pred`$(q_{i+1}/.../q_n, E_i)$]/$\mathcal{A}^+$[1] = $\varepsilon :: E_{i-1}$
- $axis_i = \downarrow^+$: `RW_Pred`$(q_i/.../q_n, E_{i-1})$:=
  $\downarrow^+ :: E_i$[$\mathcal{A}^{acc}$][`RW_Pred`$(q_{i+1}/.../q_n, E_i)$]

Given a query $axis_i :: E_i[axis_j :: E_j = "c"]$ (text-content comparison), we have the following rewriting:

$\texttt{RW\_Pred}(\downarrow::E_j\texttt{="}c\texttt{"},E_i)\ :=\ \downarrow^+::E_j\texttt{[}\mathcal{A}^{acc}\texttt{]}\texttt{[}\varepsilon\texttt{::*="}c\texttt{"]}/\mathcal{A}^+\texttt{[1]}\ =\ \varepsilon::E_i$

$\texttt{RW\_Pred}(\downarrow^+::E_j\texttt{="}c\texttt{"},E_i)\ :=\downarrow^+::E_j\texttt{[}\mathcal{A}^{acc}\texttt{]}\texttt{[}\varepsilon\texttt{::*="}c\texttt{"]}$

Example 4.4 The query $\downarrow^+::A\texttt{[}\downarrow::E\texttt{]}$, given in the proof of Theorem 3, is rewritten into:

$\texttt{Rewrite}(\downarrow^+::A[\downarrow::E],\ root) = \downarrow^+::A[\mathcal{A}^{acc}][\uparrow^+::root[\mathcal{A}^{acc}]][\texttt{RW\_Pred}(\downarrow::E,\ A)]$
$\equiv \downarrow^+::A[\mathcal{A}^{acc}][\downarrow^+::E[\mathcal{A}^{acc}]/\mathcal{A}^+[1]{=}\varepsilon::A]$

where $\uparrow^+::root[\mathcal{A}^{acc}]$ is omitted since *root* is always accessible. The rewritten query over T_v of Figure 6(d), returns the element nodes A^1 and A^{21}. □

The detail of Function `RW_Pred` is given in Figure 8. We have seen in `Rewrite` algorithm that the rewriting of subquery $axis_{i+1}\texttt{::*}$ over E_i results in a set of element types (`reach`) reachable from E_i. Then, the predicate q in $axis_i\texttt{::}E_i\texttt{[}q\texttt{]}$ is rewritten over element type E_i ($E_i \neq$*) as explained in the above definition of `RW_Pred`. While the predicate q in $axis_i\texttt{::}E_i/axis_{i+1}\texttt{::*[}q\texttt{]}$ is rewritten over the set of element types resulting by the rewriting of $axis_{i+1}\texttt{::*}$ over E_i. We denote this set by L. For a given predicate $q_1/\ldots/q_n$ over element type E, L :=$\texttt{reach(}q_1\texttt{,}\{E\}\texttt{)}$ denotes the result of rewriting sub-predicate q_1 over E (element types reachable from E with q_1), sub-predicate q_2 is rewritten over L resulting in a new set L :=$\texttt{reach(}q_2\texttt{,}L\texttt{)}$ (i.e. L :=$\texttt{reach(}q_2\texttt{,reach(}q_1\texttt{,}\{E\}\texttt{)))}$, and so on until rewriting q_n over L:=$\texttt{reach(}q_{n-1}\texttt{,}L\texttt{)}$. Each sub-predicate q_i can contain other sub-predicates (case of $axis_i::E_i[f_i]$). The *-labels in the sub-predicates are eliminated with the same principle explained in algorithm `Rewrite` using the precomputed lists of children and descendant types (`Reach`). The rewriting result of predicate q can be *false* if some element types in q are inaccessible (they do not appear in D_v) or some relationships are not respected (e.g. the rewriting of predicate $\downarrow::E_i$ over element type E_{i-1} is false if E_i is not a subelement type of E_{i-1}, such that $E_i \notin \texttt{Reach}(\downarrow,E_{i-1})$). Rewriting result can be *true* (the predicate is omitted from the query) in case of *not(q)* with non valid q.

Example 4.5 Consider the query $\downarrow^+::A[\downarrow::\texttt{*}/\downarrow::D]$ over the Figure 6(b). Using our algorithm `RW_Pred`, the predicate $\downarrow::\texttt{*}/\downarrow::D$ is rewritten over element type A as follows:

$[\texttt{RW\_Pred}(\downarrow::\texttt{*}/\downarrow::D,A)] =$
$[(\downarrow^+::A\cup\downarrow^+::D\cup\downarrow^+::E)[\mathcal{A}^{acc}][\texttt{RW\_Pred}(\downarrow::D,\{A,\ D,\ E\})]/\mathcal{A}^+[1]{=}\varepsilon::A] =$
$[(\downarrow^+::A\cup\downarrow^+::D\cup\downarrow^+::E)[\mathcal{A}^{acc}][\downarrow^+::D[\mathcal{A}^{acc}]/\mathcal{A}^+[1]{=}\varepsilon::\texttt{*}]/\mathcal{A}^+[1]{=}\varepsilon::A].$ □

Now, we generalize the formal definition of the algorithm `Rewrite` given in the previous section to handle predicates. Given the query $p_1/\ldots/p_n$ where $p_i = axis_i\texttt{::}E_i\texttt{[}f_i\texttt{]}$ and f_i ([f_i] is optional) is a predicate defined over element type E_i. We rewrite this query over node context of type E as follows:

$\texttt{Rewrite}(p,E)\ :=\ \downarrow^+::E_n\texttt{[}\mathcal{A}^{acc}\texttt{]}\texttt{[}f'_n\texttt{]}\texttt{[}prefix^{-1}\texttt{(}p_1/\ldots/p_n\texttt{)]}$

where an intermediate step, $prefix^{-1}\texttt{(}p_1/\ldots/p_i\texttt{)}$ is recursively defined with:

- $axis_i = \downarrow$: $prefix^{-1}(p_1/.../p_i) := \mathcal{A}^{E_{i-1}}[f'_{i-1}][prefix^{-1}(p_1/.../p_{i-1})]$
- $axis_i = \downarrow^+$: $prefix^{-1}(p_1/.../p_i) :=$
  $\uparrow^+::E_{i-1}[\mathcal{A}^{acc}][f'_{i-1}][prefix^{-1}(p_1/.../p_{i-1})]$
  where $f'_{i-1}\ :=\ \texttt{RW\_Pred(}f_{i-1},E_{i-1}\texttt{)}$.

Algorithm: *RW_ Pred*

input : a predicate q and a list L of element types.
output: a rewritten predicate q_t w.r.t element types in L.

1 q_t := $false$, $f' := true$;
/* *content-test is optional, if it does not exists then [ε::E=c] below is omitted* */
2 **if** *(q is a single predicate $axis::E[f]= c$)* **then** // [f] is optional
3 | **if** *($E = *$)* **then**
4 | | reach(q,L) := $\cup_{E' \in L}$Reach($axis_i, vE'$);
5 | **else if** *($\exists\ E' \in L$ s.t $E \in$ Reach($axis_i, E'$))* **then**
6 | | reach(q,L) := $\{E\}$;
7 | **else**
8 | | reach(q,L) := {};
9 | **if** *(*reach(q,L)=$\{\}$*)* **then return** $false$;
| | // rewriting of f' if [f] exists
10 | f' := *RW_Pred*(f,E);
11 | **if** *([f] exists and $f' = false$)* **then**
12 | | **return** $false$;
| | // [f'] *below is omitted if* f'=*true*
13 | **if** *($axis =\downarrow$)* **then**
14 | | q_t := $fs(\downarrow^+, reach(q, L))$[$\mathcal{A}^{acc}$][$f'$][$\varepsilon::E$=$c$]/$\mathcal{A}^+$[1]=$\varepsilon::*$;
15 | **else**
16 | | q_t := $fs(\downarrow^+, reach(q, L))$[$\mathcal{A}^{acc}$][$f'$][$\varepsilon::E$=$c$];
17 **else if** *(q is q_f/q_r where q_f = $axis_1::E[f]$ and q_r is the remaining steps)* **then**
18 | *rewrite q_f as done in lines 3-16*;
19 | q'_r := *RW_Pred*(q_r,reach(q_f,L));
20 | **if** *(q'_r=$false$)* **then return** $false$;
21 | **if** *($axis_1 =\downarrow$)* **then**
22 | | q_t := $fs(\downarrow^+, reach(q_f, L))$[$\mathcal{A}^{acc}$][$f'$][$q'_r$]/$\mathcal{A}^+$[1]=$\varepsilon::*$;
23 | **else**
24 | | q_t := $fs(\downarrow^+, reach(q_f, L))$[$\mathcal{A}^{acc}$][$f'$][$q'_r$];
25 **else if** *(q is $q_1 \wedge \ldots \wedge q_n$)* **then**
26 | **if** *($\exists\ q_i$ s.t RW_Pred(q_i,L)= $false$)* **then**
27 | | q_t := $false$;
28 | **else if** *(RW_Pred(q_i,L)= $true$ for each q_i)* **then**
29 | | q_t := $true$;
30 | **else**
31 | | q_t := $\bigwedge_{RW\_Pred(q_i,L) \neq true}$*RW_Pred*($q_i$,$L$);
32 **else if** *(q is $q_1 \vee \ldots \vee q_n$ or $q_1 \cup \ldots \cup q_n$)* **then**
33 | **if** *($\exists\ q_i$ s.t RW_Pred(q_i,L)= $true$)* **then**
34 | | q_t := $true$;
35 | **else**
36 | | q_t := $\bigvee_{RW\_Pred(q_i,L) \neq false}$*RW_Pred*($q_i$,$L$);
37 **else if** *case of not (q)* **then**
38 | **if** *RW_Pred(q,L)= $false$* **then**
39 | | q_t := $true$;
40 | **else if** *RW_Pred(q,L)≠ $true$* **then**
41 | | q_t := not (*RW_Pred*(q,L));
42 **else if** *case of* ϵ **then**
43 | q_t := $true$;
44 **return** q_t;

Figure 8: Predicate Rewriting.

4.3 Complexity Analysis

Given a security specification S=(D,ann), we extract first the security view V=(D_v, ann) corresponding to S, where D_v is derived using our algorithm `DeriveView` of Figure 4. The user is allowed to request its authorized data represented with the DTD view D_v. For each query Q in $\mathcal{X}$, our algorithm `Rewrite` translates this query to an equivalent one Q_t in $\mathcal{X}^{\Uparrow}_{[n,=]}$ such that, for any instance T conforms to D, its virtual view T_v conforms to D_v, we have $Q(T_v)$=`Rewrite(`Q`)`(T).

The overall complexity time of our rewriting algorithm `Rewrite` is stated as follows:

Theorem 4 *For any security view* V=(D_v, ann) *and any* $\mathcal{X}$ *query* Q *over the DTD view* D_v*, the algorithm* `Rewrite` *computes an equivalent query* Q_t *in* $\mathcal{X}^{\Uparrow}_{[n,=]}$ *over the original DTD in at most* $O(|Q| * |D_v|^2)$ *time.* □

Proof. Given an $\mathcal{X}$ query p=$axis_1$::$E_1[q_1]/.../axis_n$::$E_n[q_n]$, we denote by $|p|$ the number of subqueries and sub-predicates of p, e.g. $|{\downarrow}::E_1[\text{not}({\downarrow^+}::*)]|$=2. Each subquery (or sub-predicate) p_i=$axis_i$::$E_i[q_i]$ of p must be rewritten over p_{i-1}=$axis_{i-1}$::$E_{i-1}[q_{i-1}]$ to check the accessibility of E_i and to preserve the relationship defined between E_i and E_{i-1}. This is done in a constant time by using the precomputed predicates $\mathcal{A}^{acc}$ and $\mathcal{A}^+$ as in lines 8-17 of algorithm `Rewrite` and lines 13-16 of algorithm `RW_Pred`. The $*$-label of each subquery (or sub-predicate) is eliminated as done in the lines 18-23 of algorithm `Rewrite`, and lines 3-8 of algorithm `RW_Pred`, which causes an additional cost $|D_v|^2$. For instance, to rewrite the query ↓::$*$ over the set `reach`, the elimination of the $*$-label amounts to parse each element type E in `reach` and compute the union of its children types given by the precomputed list `Reach`(↓,E) (|`Reach`(↓,E)|=$O|D_v|$). Next, the $*$-label of the query ↓::$*$ is replaced by the union of children types of element types in `reach` which is done in at most $O|D_v|^2$ time. Thus, a given query p can be rewritten in at most $O(|p| * |D_v|^2)$ time. □

4.4 Query Rewriting Improvements

We discuss in this section some possible implementations of our rewriting algorithm `Rewrite` to improve the overall complexity of Theorem 4.

The first optimization can be done by avoiding the $*$-label elimination step discussed above. For a security view V=(D_v, ann), an $\mathcal{X}$ query Q over the DTD view D_v can be rewritten in a linear time $O(|Q|)$. Using the precomputed predicates $\mathcal{A}^{acc}$ and $\mathcal{A}^+$, each subquery p_i of the query p can be rewritten over p_{i-1} in a *constant time* by adding the predicate $\mathcal{A}^{E_{i-1}}$ (i.e. $\mathcal{A}^+[1]/\varepsilon::E_{i-1}$) or $\mathcal{A}^{acc}$, case of $axis_i$=↓ and $axis_i$=$\downarrow^+$ respectively. For instance, the query ↓::$*$/↓::B over context node of type A can be rewritten into $\downarrow^+::B[\mathcal{A}^{acc}][\mathcal{A}^+[1]/\varepsilon::*/\mathcal{A}^+[1]/\varepsilon::A]$. In the same way, each sub-predicate q_i=$axis_i$::E_i can be rewritten over q_{i-1} in a *constant time* into $\downarrow^+::E_i[\mathcal{A}^{acc}]/\mathcal{A}^+[1]=\varepsilon::E_{i-1}$ in case of $axis_i$=↓, or into $\downarrow^+::E_i[\mathcal{A}^{acc}]$ otherwise. For instance, the predicate [↓::$*$/↓::C] over element type B can be rewritten into: $[\downarrow^+::*[\mathcal{A}^{acc}][\downarrow^+::C[\mathcal{A}^{acc}]/\mathcal{A}^+[1]=\varepsilon::*]/\mathcal{A}^+[1]=\varepsilon::B]$. Thus, the rewriting of the query p depends on the parsing of all its subqueries and sub-predicates which is done in $O(|p|)$ time.

Since the query answering time concerns the rewriting time and the evaluation time of the rewritten query, then the existence of the $*$-label in the rewritten query can induce for poor performance when some inaccessible elements are parsed by the rewritten query. For instance, given the query $\downarrow::E$ over context node n of type A with $E{=}*$. Without eliminating the $*$-label, this query is rewritten into $\downarrow^+::E[\mathcal{A}^{acc}][\mathcal{A}^A]$, and then for each descendant node of n of any element type, the predicates $\mathcal{A}^{acc}$ and $\mathcal{A}^A$ are evaluated. To improve the query evaluation time, the elimination of the $*$-label in the query is indispensable and ensures that two defined predicates are evaluated only at accessible descendant nodes of n (i.e. whose types appear in D_v). This elimination can give good performance since the size of D_v (number of accessible element types) is too small than the size of the original DTD D in practice. For this reason, the precomputed lists `Reach` can be sorted in such a way the union of the children/descendant types of two element types of D_v (e.g. `Reach`($\downarrow$,E_1) $\cup$ `Reach`($\downarrow$,E_2)) can be linear on the size of D_v. Accordingly, the $*$-label elimination phase, done in lines 18-23 of algorithm `Rewrite` and lines 3-8 of algorithm `RW_Pred`, can be efficiently improved to take at most $|D_v|$ time and any query p can be rewritten in this case in at most $O(|p| * |D_v|)$ time.

5 Extensions

We discuss some extensions of our proposed rewriting approach to deal with a large fragment of XPath queries, and to overcome some limitations of existing access specifications languages.

5.1 Upward-axes Rewriting

For the rewriting of upward-axes ($\uparrow$ and $\uparrow^+$), we extend the algorithms `Rewrite` and `RW_Pred` without increasing the complexity of the global rewriting (as explained in Theorem 4).
In `Rewrite`, $prefix^{-1}(p_1/\ldots/p_i)$ is defined over upward-axes as follows:

- $axis_i = \uparrow : prefix^{-1}(p_1/\ldots/p_i)$ `:=`
  $\downarrow^+::E_{i-1}[\mathcal{A}^{acc}][f'_{i-1}][prefix^{-1}(p_1/\ldots/p_{i-1})]/\mathcal{A}^+[1]{=}\varepsilon::E_i$

- $axis_i = \uparrow^+ : prefix^{-1}(p_1/\ldots/p_i)$ `:=`
  $\downarrow^+::E_{i-1}[\mathcal{A}^{acc}][f'_{i-1}][prefix^{-1}(p_1/\ldots/p_{i-1})]$
  where f'_i `:= RW_Pred(`f_i, E_i`).`

In `RW_Pred`, an intermediate predicate $q_i/../q_n$ ($q_i{=}axis_i::E_i[f_i]$) is rewritten over element type E_{i-1} in case of upward-axes as follows:

- $axis_i = \uparrow$: `RW_Pred(`$q_i/\ldots/q_n, E_{i-1}$`) :=`
  $\mathcal{A}^{E_i}[f'_i]$`[RW_Pred(`$q_{i+1}/\ldots/q_n, E_i$`)]`

- $axis_i = \uparrow^+$: `RW_Pred(`$q_i/\ldots/q_n, E_{i-1}$`) :=`
  $\uparrow^+::E_i[\mathcal{A}^{acc}][f'_i]$`[RW_Pred(`$q_{i+1}/\ldots/q_n, E_i$`)]`
  where f'_i `:= RW_Pred(`f_i, E_i`).`

5.2 Revision of Access Specifications

The node accessibility w.r.t the specification value `[Q]` has been defined with two different meanings [4, 9]. Like in [4], we have assumed in the definition of the element node accessibility that for each element node n concerned by an annotation of value $[Q]$, if Q is not valid at n (i.e. $n \not\models Q$) then n and all its descendant elements are not accessible, even if there is some valid annotations defined over these descendants (condition (ii) of Definition 3.1). However, in [9] the element node n can be not accessible (i.e. $n \not\models Q$), but one of its descendant element B' can be accessible if it is concerned by a valid annotation.

We assume that both meanings are useful and an access control specification language must provide the definition of each of them. For this reason, we redefine the access specification of Definition 2.1 as follows:

Definition 5.1 *An access specification* S *is a pair* $(D, \texttt{ann})$ *consisting of a DTD* D *and a partial mapping* `ann` *such that, for each production* $A \rightarrow P(A)$ *and each element type* B *in* P(A), `ann`(A,B), *if explicitly defined, is an annotation of the form:*

$$\texttt{ann}(A,B) \texttt{ := } Y \texttt{ | } N \texttt{ | } [Q] \texttt{ | } N_h \texttt{ | } [Q]_h$$

□

Given an element node n of type B with parent node of type A, then with the specification values N and $[Q]$, accessibility overwriting is allowed under n even though `ann`(A,B)=N or `ann`(A,B)=$[Q]$ and $n \not\models Q$. The semantics of the new specification values N_h and $[Q]_h$ are given as follows. If the element node n is concerned by an annotation with value N_h, then no overwriting of this value is permitted to descendant elements of n, i.e. if n' is a descendant element of n, then n' is not accessible even if it is concerned by a valid annotation. While if n is concerned by an annotation with value $[Q]_h$, then the annotations defined under n (i.e. under B element type) take effect only if $n \models Q$. For instance, if a descendant element n' of n is concerned by an annotation of value $[Q']$, then n' is accessible only if $n' \models Q'$ and $n \models Q$. We call the annotation with value N_h or $[Q]_h$, *downward-closed* annotation.

Example 5.1 We consider the hospital DTD of Figure 9(a) and we give the following access specification:

- ***A patient can access only to its own*** *diagnosis* ***information:***
  `ann`$(department, patient)$ `=` $[pname$=$\$name]_h$
  `ann`$(patient, parent)$ `=` `ann`$(patient, sibling)$=N_h
  `ann`$(patient, visit)$ `=` N`,` `ann`$(medication, diagnosis)$ `=` Y
- ***A research institute can access to patients whose have*** *"disease1"***:**
  `ann`$(department, patient)$ `=` `ann`$(parent, patient)$ `=` `ann`$(sibling, patient)$
  `=` $[visit/treatment/medication[diagnosis$=$\texttt{'disease1'}]]$

For the first policy, $\$name$ is a variable system denoting patient's name. For a given patient with name "*Bob*", if $p \not\models pname$="*Bob*" then all the fragment rooted at p is hidden from the patient "*Bob*" and any annotation under element node p can be applied since it is the medical data of another patient. Also, the annotation `ann`$(patient, parent)$=N_h cannot be defined with

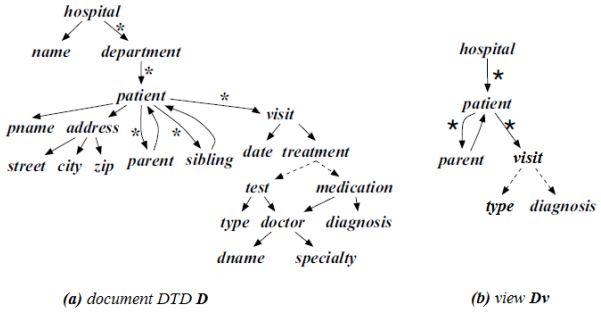


Figure 9: Hospital DTD.

$\texttt{ann}(patient, parent)$=$N$ otherwise $\texttt{ann}(medication, diagnosis)$=$Y$ makes diagnosis information of parent accessible to patient "*Bob*" as its own information. The same principle is applied with the annotation $\texttt{ann}(patient, sibling)$=$N_h$. For the second policy, it is clear that a false evaluation of the predicate does not imply the inaccessibility of the sub patients, i.e. a given patient may not have "*disease1*", while its parent/sibling can be affected by this disease and must be shown to the research institute. □

The new access specification defined can be taken into account simply by applying the following changes in our rewriting approach. The predicate $\mathcal{A}_1^{acc}$ is redefined with:[9]

$\mathcal{A}_1^{acc}$:= $\uparrow^*$::*[ε::$root$ $\vee_{ann(A',A)\in ann}$ ε::A/$\uparrow$::A'][1]
[ε::$root$ $\vee_{(ann(A',A)=Y|[Q]|[Q]_h)\in ann}$ ε::$A.\sigma(A',A)$/$\uparrow$::A']

While The predicate $\mathcal{A}_2^{acc}$ is redefined with:

$\mathcal{A}_2^{acc}$:= $\bigwedge_{(ann(A',A)=N_h)\in ann}$ not ($\uparrow^+$::A/$\uparrow$::A')
$\bigwedge_{(ann(A',A)=[Q]_h)\in ann}$ not ($\uparrow^+$::A[not (Q)]/$\uparrow$::A')

6 Experimental Results

We have developed a prototype to improve effectiveness of our rewriting approach. The performance study is done using a real-life recursive DTD and a various forms of XPath queries. The experimental results show the efficiency of our XPath query rewriting approach w.r.t the answering approach based on the materialization of the view. Notice that we cannot do comparison between our approach and the two existing approaches which deal with queries rewriting under recursive views [5, 8], since they are based on the non-standard language "regular XPath", and no practical tool is present to evaluate regular XPath queries. The experiments were conducted using Ubuntu system, with a dual Core 2.53 GHz and 1 GB of memory.

XML Documents. Using `ToXGene` generator [1], we generated set of XML documents that conform to the hospital DTD of Figure 9 and with sizes ranging from 10MB to 100MB.

[9] Note that $A.\sigma(A', A)$ gives $A[Q]$ if $\texttt{ann}(A',A)$=$[Q]$|$[Q]_h$ and A otherwise.

Security Specification. Figure 9(b) represents the hospital DTD view D_v of a research institute studying inherited patterns of some diseases. This view shows only patients having one or more disease from {*disease1, disease2, disease3*} with their parent hierarchy, and denies access to their *name*, *address*, *test* and *doctor* data. Formally, we define this view with the following annotations:

1. ann(*hospital*)=*Y*
2. ann(*hospital*,*name*)=*N*
3. ann(*hospital*,*department*)=*N*
4. ann(*department*,*patient*)= [↓::*visit*/↓::*treatment*/↓::*medication*[↓::*diagnosis*='disease1' or ↓::*diagnosis*='disease2' or ↓::*diagnosis*='disease3']]$_h$
5. ann(*patient*,*pname*)=*N*
6. ann(*patient*,*address*)=*N*
7. ann(*patient*,*sibling*)=N_h
8. ann(*visit*,*date*)=*N*
9. ann(*visit*,*treatment*)=*N*
10. ann(*medication*,*diagnosis*)=*Y*
11. ann(*test*,*type*)=*Y*

The annotation 7 must be downward-closed, otherwise the annotations 10 and 11 can overwrite some sibling data (*diagnosis* and *type* of visit) to be accessible.

XPath Queries. We define the following set of XPath queries :

1. Q_1. ↓::*patient*[↓$^+$::*visit*[↓::*diagnosis*='disease1' or ↓::*diagnosis*='disease2' or ↓::*diagnosis*='disease3']].
2. Q_2. ↓$^+$::*patient*[↓::*visit*[↓::*diagnosis*='disease1' or ↓::*diagnosis*='disease2' or ↓::*diagnosis*='disease3'] and not (↓$^+$::*patient*/↓::*visit*[↓::*diagnosis*='disease1' or ↓::*diagnosis*='disease2' or ↓::*diagnosis*='disease3'])].
3. Q_3. ↓$^+$::*diagnosis*[↑::*visit*/↑::*/↑::*/↑::*/↑::*hospital*]

The first query returns patients whose some of its ancestors also had the sames diseases. The Second query Q_2 returns the first generation where the discussed diseases appeared for the first time, and Q_3 represents the diagnosis of the second generation of infected patients. Each query Q_i is rewritten over the root node (*hospital*) of each document into Rewrite(Q_i,*hospital*), and this by using the security view V=(D_v, ann) defined with the DTD view of Figure 9(b) and the annotations defined above.

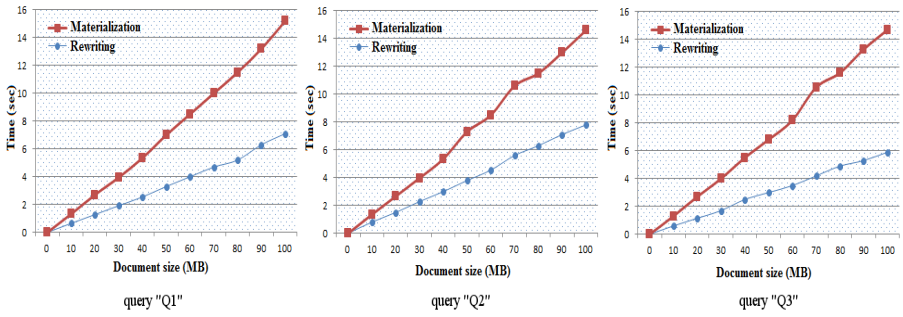


Figure 10: XPath Queries Evaluation Time.

Approaches. A comparison is done between our rewriting approach and the *materialization* approach. Given a security view $V{=}(D_v, \texttt{ann})$, the *materialization* consists in incorporating an accessibility label (+/-) to each element node in the document which is concerned by an annotation of V. Each element node n, not yet labeled (i.e. no invalid downward-closed annotation is defined over its ancestor elements), is labeled with "+" if it is concerned by an annotation with value Y or with value $[Q]|[Q]_h$ and $n \vDash Q$ (resp. n is labeled with "-" in case of annotation with value $N|N_h$ or with value $[Q]|[Q]_h$ and $n \nvDash Q$). In case of an element node n concerned by an invalid downward-closed annotation (with value N_h or with value $[Q]_h$ and $n \nvDash Q$), the n and all its descendant elements are labeled with "-". After applying all the annotations of the security view V over the document, each unlabeled element node is annotated by inheritance from its nearest labeled ancestor element. The obtained document is called *fully annotated document*. Finally, the *materialized view* of the original document is computed by deleting all inaccessible element nodes (labeled with "-") from the fully annotated document and user queries are evaluated directly over this view. Thus, we compare the answering time of the materialization approach (defined as the *view materialization* time and *query evaluation* time over the materialized view) with that of our rewriting approach (defined as the rewriting time of the query and the evaluation time of the rewritten query over the original document).

Performance Results. The experimental results are shown in Figure 10 where the answering time of each query is evaluated using our rewriting algorithm and the materialization approach. The size of the answer ranges from few hundred to a few thousand of nodes. Figure 10 shows clearly that our algorithm remains more efficient than the materialization approach.

We observe first that the translation of XPath queries from $\mathcal{X}$ to $\mathcal{X}^{\Uparrow}_{[n,=]}$ does not induce for a poor performance and the average of the answering time of our rewriting approach remains in general less than 8 seconds for a large XML document. Second, a query containing time-consuming elements like $*$-labels or `parent` axes does not degrade the rewriting performance as shown with the query Q_3.

7 Conclusions and Future Work

The proposed approach yields the first practical solution to rewrite XPath queries over recursive XML views using only the expressive power of the standard XPath. The extension of the downward class of XPath queries with some axes and operators has been investigated in order to make queries rewriting possible under recursion.

The conducted experimentation shows the efficiency of our approach by comparison with the materialization approach. Most importantly, the translation of queries from $\mathcal{X}$ to $\mathcal{X}^{\Uparrow}_{[n,=]}$ does not impact the performance of the queries answering. We have discussed how our approach can be extended to deal with the upward-axes without additional cost. Lastly, a revision of the access specification language is presented to go beyond some limitations in the definition of some access privileges.

As future work, we plan first to provide an optimized version of our approach and also to use the same principle to secure XML updating.

References

[1] D. Barbosa, A. Mendelzon, J. Keenleyside, and K. Lyons. Toxgene: An extensible template-based data generator for xml. *In WebDB 2002, pp. 49-54.*

[2] 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 14 december 2010. http://www.w3.org/tr/2010/rec-xpath20-20101214/.

[3] E. Bertino and E. Ferrari. Secure and selective dissemination of xml documents. *In TISSEC*, 5(3):290–331, 2002.

[4] W. Fan, C.-Y. Chan, and M. Garofalakis. Secure xml querying with security views. *In SIGMOD 2004, pp. 587-598.*

[5] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Rewriting regular xpath queries on xml views. *In ICDE 2007, pp.666-675.*

[6] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Smoqe: A system for providing secure access to xml. *In VLDB 2006, pp. 1227-1230.*

[7] I. Fundulaki and M. Marx. Specifying access control policies for xml documents with xpath. *In SACMAT 2004, pp. 61-69.*

[8] B. Groz, S. Staworko, A.-C. Caron, Y. Roos, and S. Tison. Xml security views revisited. *In DBPL 2009, pp, 52-67.*

[9] G. Kuper, F. Massacci, and N. Rassadko. Generalized xml security views. *In SACMAT 2005, pp. 77-84.*

[10] M. Murata, A. Tozawa, and M. Kudo. Xml access control using static analysis. *In CCS 2003, pp. 73-84.*

[11] N. Rassadko. Policy classes and query rewriting algorithm for xml security views. *In Data and Applications Security 2006, pp. 104-118.*

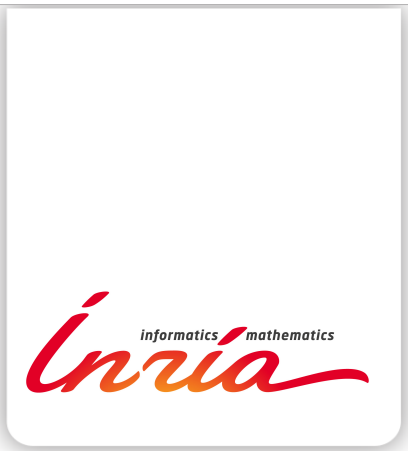
[12] N. Rassadko. Query rewriting algorithm evaluation for xml security views. *In Secure Data Management 2007, pp. 64-80.*

[13] A. Stoica and C. Farkas. Secure xml views. *In the IFIP WG 11.3 Sixteenth International Conference on Data and Applications Security 2002, pp. 133-146.*

[14] B. ten Cate. The expressivity of xpath with transitive closure. *In PODS 2006, pp. 328-337.*

[15] B. ten Cate and C. Lutz. The complexity of query containment in expressive fragments of xpath 2.0. *Journal of the ACM 2009, pp. 31-48.*

[16] R. Vercammen, J. Hidders, and J. Paredaens. Query translation for xpath-based security views. *In EDBT 2006, pp. 250-263.*



RESEARCH CENTRE
NANCY – GRAND EST

615 rue du Jardin Botanique
CS20101
54603 Villers-lès-Nancy Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399